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

(10) **DE 696 36 599 T2** 2007.08.23

(12) # Übersetzung der europäischen Patentschrift

(97) **EP 0 850 462 B1**
(21) Deutsches Aktenzeichen: **696 36 599.5**
(86) PCT-Aktenzeichen: **PCT/US96/12780**
(96) Europäisches Aktenzeichen: **96 928 062.7**
(87) PCT-Veröffentlichungs-Nr.: **WO 1997/006512**
(86) PCT-Anmeldetag: **02.08.1996**
(87) Veröffentlichungstag der PCT-Anmeldung: **20.02.1997**
(97) Erstveröffentlichung durch das EPA: **01.07.1998**
(97) Veröffentlichungstag der Patenterteilung beim EPA: **04.10.2006**
(47) Veröffentlichungstag im Patentblatt: **23.08.2007**

(51) Int Cl.[8]: ***G06T 15/00*** (2006.01)

(30) Unionspriorität:

511553	**04.08.1995**	**US**
560114	**17.11.1995**	**US**
671412	**27.06.1996**	**US**
671506	**27.06.1996**	**US**
672347	**27.06.1996**	**US**
672425	**27.06.1996**	**US**
672694	**27.06.1996**	**US**

(73) Patentinhaber:
Microsoft Corp., Redmond, Wash., US

(74) Vertreter:
BOEHMERT & BOEHMERT, 28209 Bremen

(84) Benannte Vertragsstaaten:
DE, FR, GB, IE

(72) Erfinder:
MYHRVOLD, P., Nathan, Bellevue, WA 98005, US; KAJIYA, T., James, Duvall, WS 98019, US; TORBORG, G., John, Redmond, WA 98052, US; KENWORTHY, L., Mark, Duvall, WA 98019-7806, US; TOELLE, Allen, Michael, Bellevue, WA 98008, US; GRIFFIN, E., Kent, Bellevue, WA 98008, US; LENGYEL, Edward, Jerome, Seattle, WA 98108, US; GABRIEL, A., Steven, Redmond, WA 98052-3256, US; VERES, E., James, Woodinville, WA 98072, US; CHAUVIN, W., Joseph, Issaquah, WA 98029, US; GOOD, Howard, Seattle, WA 98105, US; POWELL, Chambers, William, Seattle, WA 98122, US

(54) Bezeichnung: **VERFAHREN UND SYSTEM ZUR WIEDERGABE VON GRAFISCHEN OBJEKTEN DURCH TEILUNG IN BILDSTÜCKE UND ZUSAMMENSETZEN VON BILDLAGEN ZU EINEM WIEDERGABEBILD**

Beschreibung

TECHNISCHES GEBIET DER ERFINDUNG

[0001] Die Erfindung betrifft allgemein die Wiedergabe von Grafiken und betrifft besonders verbesserte Verfahren und Systeme zur Wiedergabe von grafischen Objekten.

ALLGEMEINER STAND DER TECHNIK

[0002] Bei der umfangreichen Anwendung von Computern in allen Bereichen des modernen Lebens besteht ein verstärktes Bedürfnis, die Mensch-Maschine-Schnittstelle durch die Verwendung von optischen Informationen zu verbessern. Bei grafischer Software und Handware sind durch Fortschritte bereits drastische Verbesserungen der Mensch-Maschine-Schnittstelle erreicht worden. Interaktive Grafiken, wie zum Beispiel Fensterumgebungen für Desktop-Computer, haben zum Beispiel die leichte Anwendung und Interaktivität von Computern deutlich verbessert und sind heutzutage allgemein üblich. Da das Preis-Leistungs-Verhältnis von Hardware fällt, greift die Verwendung von computererzeugten Grafiken und von Animation immer mehr um sich. Leider haben die Kosten der Herstellung von echt interaktiven und realistischen Effekten ihre Anwendung eingeschränkt. Es gibt daher ein Bedürfnis nach neuen Grafik verarbeitenden Verfahren und Architekturen, die interaktivere und realistischere Effekte zu niedrigeren Kosten liefern.

[0003] Obwohl es zahlreiche Wege gibt, die Grafikverarbeitung zu klassifizieren, besteht ein häufiger Ansatz darin, ein Bild im Hinblick auf die Dimensionen des Objektes zu beschreiben, die es repräsentieren soll. Ein Grafiksystem kann zum Beispiel Objekte in zwei Dimensionen (z.B. mit x- und y-Koordinaten) repräsentieren, in welchem Fall man sagt, dass die Grafik „zweidimensional" ist, oder drei Dimensionen (z.B. mit x-, y- und z-Koordinaten), in welchem Fall man sagt, dass die Grafik „dreidimensional" („3D") ist.

[0004] Da Anzeigegeräte, wie zum Beispiel Bildschirmröhren (CRTs) zweidimensional sind („2D"), sind Bilder, die von Computergrafiksystemen angezeigt werden, im allgemeinen 2D. Wie unten detaillierter diskutiert wird, kann jedoch der Computer das angezeigte Bild ändern, um eine andere Perspektive des Objektes im 3D-Raum zu zeigen, wenn der Computer ein grafisches Modell bewahrt, das das abgebildete Objekt im dreidimensionalen Raum repräsentiert. Obwohl ein 2D-Grafikbild vor der Anzeige transformiert werden kann (z.B. skaliert, verschoben oder gedreht), kann im Gegensatz dazu der Computer nicht ohne weiteres das Aussehen des Objektes aus einer anderen Perspektive im 3D-Raum abbilden.

[0005] Die wachsenden Fähigkeiten moderner Computer, mit 2D- und besonders 3D-Grafiken effizient umzugehen, haben zu einer wachsenden Vielfalt von Anwendungen für Computer sowie zu grundlegenden Änderungen in der Schnittstelle (UI) zwischen Computern und ihren Nutzern geführt. Die Verfügbarkeit von 3D-Grafiken gewinnt zunehmend an Bedeutung für das Wachstum von Anwendungen im Unterhaltungsbereich, einschließlich der Herstellung von Qualitätsfilmanimationswerkzeugen sowie Spielen und Multimediaprodukten mit geringerer Auflösung für den Heimbereich. Zu den vielen anderen Bereichen, die in Verbindung mit 3D-Grafiken stehen, gehören Bildung, Videokonferenzen, Videobearbeitung, interaktive Nutzerschnittstellen, computergestütztes Konstruieren und computergestützte Produktion (CAD/CAM), wissenschaftliche und medizinische Bildgebung, Geschäftsanwendungen und das elektronische Publizieren.

[0006] Man kann sich vorstellen, dass ein Grafikverarbeitungssystem ein Anwendungsmodell, Anwendungsprogramm, Grafiksubsystem sowie die konventionellen Hardware- und Softwarekomponenten eines Computers und seine Peripheriegeräte umfasst.

[0007] Das Anwendungsmodell repräsentiert die Daten oder Objekte, die angezeigt werden sollen, wobei man natürlich annimmt, dass die Bildverarbeitung auf einem Modell beruht. Das Modell umfasst Informationen, die die Grundelemente betreffen, wie zum Beispiel Punkte, Linien und Polygone, die die Formen der Objekte sowie die Attribute der Objekte (z.B. Farbe) definieren.

[0008] Das Anwendungsprogramm steuert die Eingaben in und die Ausgaben aus dem Anwendungsmodell, wobei es effektiv als Übersetzer zwischen dem Anwendungsmodell und dem Grafiksubsystem fungiert. Und schließlich ist das Grafiksubsystem für die Weiterleitung der Nutzereingaben an das Anwendungsmodell verantwortlich und ist für die Erzeugung des Bildes aus den detaillierten Beschreibungen verantwortlich, die vom Anwendungsmodell gespeichert werden.

[0009] Das typische Grafikverarbeitungssystem umfasst ein physisches Ausgabegerät, das für die Ausgabe

oder Anzeige der Bilder verantwortlich ist. Obwohl andere Formen von Anzeigegeräten entwickelt wurden, wird die heute vorherrschende Technologie als Rastergrafik bezeichnet. Ein Rasterbildschirm umfasst ein Feld von einzelnen Punkten oder Bildelementen (d.h. Pixeln), die in Zeilen und Spalten angeordnet sind, um das Bild zu erzeugen. In einer Bildschirmröhre entsprechen diese Pixel einem Leuchtstoffpunktfeld, das auf dem Schirmträger der Bildschirmröhre bereitgestellt wird. Die Emission von Licht aus jedem Leuchtstoffpunkt im Feld wird unabhängig von einem Elektronenstrahl gesteuert, der das Feld nacheinander, jeweils eine Zeile zu einem Zeitpunkt, als Reaktion auf gespeicherte Informationen abtastet, die repräsentativ für jedes Pixel im Bild sind. Die verschachtelte Abtastung alternierender Zeilen des Feldes ist auch ein häufiges Verfahren, zum Beispiel im Fernsehbereich. Das Feld der Pixelwerte, die auf den Bildschirm abgebildet werden, wird oft als Bitmap oder Pixmap bezeichnet.

[0010] Ein Problem, das mit Rastergrafikgeräten verbunden ist, ist der Speicher, der zum Speichern der Bitmap benötigt wird, selbst für ein einziges Bild. Das System kann zum Beispiel 3,75 MByte (MB) des RAM benötigen, um eine Anzeigeauflösung von 1280 × 1024 (d.h. Zahl Pixelspalten und -reihen) und 24 Bit Farbinformationen pro Pixel zu unterstützen. Diese Informationen, die wieder das Bild eines einzigen Bildschirms repräsentieren, werden in einem Abschnitt des Anzeigespeichers des Computers gespeichert, der als Bildpufferspeicher bekannt ist.

[0011] Ein weiteres Problem bei konventionellen Rastergrafikgeräten, wie zum Beispiel Bildschirmröhren, ist das relativ schnelle Abklingen des Lichts, das von dem Gerät abgegeben wird. In Folge dessen muss die Anzeige normalerweise mit einer Rate, die bei etwa 60 Hz oder mehr liegt, wiederholt werden (d.h. das Raster neu abgetastet werden), um ein „Flackern" des Bildes zu vermeiden. Dies stellt eine strenge Forderung für das Bilderzeugungssystem dar, um Bilddaten mit einer festen Rate zu liefern. Einige Systeme gehen dieses Problem durch Einsatz von zwei Bildpufferspeichern an, wobei einer der Pufferspeicher mit Pixmap-Informationen aktualisiert wird, die dem nachfolgenden Bild entsprechen, während der andere Pufferspeicher zum Wiederanzeigen (refresh) des Bildschirms mit dem Pixmap für das aktuelle Bild verwendet wird.

[0012] Die Forderungen, die an das System gestellt werden, werden weiter durch die Komplexität der Informationen verschärft, die oft verarbeitet werden müssen, um ein Bild des Objektes, das vom Anwendungsmodell gespeichert wird, wiederzugeben. Zum Beispiel ist das Modellieren einer dreidimensionalen Oberfläche an sich schon eine komplexe Aufgabe. Die Oberflächenmodellierung wird durch das Anwendungsmodell ausgeführt und kann die Verwendung von Polygongittern, parametrischen Oberflächen oder quadratischen Oberflächen beinhalten. Während eine gekrümmte Fläche durch ein Gitter von ebenen Polygonen repräsentiert werden kann, hängt die „Glattheit" ihres Aussehens im wiedergegebenen Bild sowohl von der Auflösung der Anzeige als auch von der Zahl der einzelnen Polygone ab, die zum Modellieren der Fläche verwendet werden. Die Berechnungen, die mit der hochauflösenden Modellierung von komplexen Oberflächen auf der Basis von Polygongittern verknüpft sind, können äußerst ressourcenintensiv sein.

[0013] Wie oben angedeutet, gibt es ein Bedürfnis, realistischere und interaktivere Bilder zu erzeugen. Der Begriff „Echtzeit" wird häufig dazu verwendet, interaktive und realistische Bildverarbeitungssysteme zu beschreiben. In einem „Echtzeit"-System sollte der Nutzer eine kontinuierliche Bewegung von Objekten in einer Szene wahrnehmen. In einem Videospiel, das Echtzeitfähigkeiten besitzt, sollten die handelnden Personen und der Blickpunkt mit minimaler Verzögerung auf die Eingaben eines Nutzers reagieren und sollten sich flüssig bewegen.

[0014] Um solche Echtzeiteffekte zu erzeugen, muss ein Bildwiedergabesystem ein neues Bild mit einer ausreichend hohen Rate generieren, so dass der Nutzer eine kontinuierliche Bewegung von Objekten in einer Szene wahrnimmt. Die Rate, mit der ein neues Bild für die Anzeige berechnet wird, wird als „rechnerische" Rate oder als „rechnerische Bildrate" bezeichnet. Die rechnerische Rate, die benötigt wird, realistische Effekte zu erreichen, variiert je nachdem, wie schnell sich Objekte durch die Szene bewegen und wie schnell sich die Sichtperspektive ändert. Für eine typische Anwendung berechnet ein Echtzeit-Grafiksystem ein neues Bild mindestens zwölf mal pro Sekunde neu, um eine Reihe von Bildern zu erzeugen, die eine kontinuierliche Bewegung simulieren. Für hochwertige Animationsanwendungen muss jedoch die rechnerische Rate beträchtlich höher sein.

[0015] Ein weiteres kritisches Problem für Echtzeitsysteme ist die Transportverzögerung. Die Transportverzögerung ist die Zeit, die zum Berechnen und Anzeigen eines Bildes als Reaktion auf die Eingabe vom Nutzer, d.h. Bewegung eines Joysticks, um eine Person in einer Szene zu bewegen, benötigt wird. In dem Maße, in dem die Transportverzögerungszeit für einen Nutzer merklich ist, gibt es eine Beeinträchtigung der „Echtzeit"-Interaktivität. Im idealen Fall sollte der Nutzer keine Transportverzögerung wahrnehmen. In der Praxis je-

doch gibt es immer eine gewisse Verzögerung, die auf die Wiedergabe von Objekten in einer Szene als Reaktion auf neue Eingaben und die Erzeugung eines Anzeigebildes zurückzuführen ist. Verbesserungen der Echtzeitinteraktivität, ohne Daten zu verwerfen, was die Bildqualität stören kann, sind sehr erwünscht.

[0016] Wie oben angeführt, umfassen konventionelle Grafiksysteme normalerweise einen Bildpufferspeicher. Zum Erzeugen eines Bildes gibt das Grafiksystem alle Objekte einer Szene wieder und speichert das resultierende Bild in diesem Bildpufferspeicher. Das System überträgt dann die wiedergegebenen Bilddaten auf eine Anzeige. In einer konventionellen Grafikarchitektur wird der gesamte Bildpufferspeicher gelöscht und die Szene wird neu wiedergegeben, um ein nächstes Bild zu erzeugen. Bei dieser Art von System muss jedes Objekt für jedes Bild neu gezeichnet werden, weil der Bildpufferspeicher zwischen den Bildern gelöscht wird. Daher wird jedes Objekt mit derselben Rate aktualisiert, ungeachtet seiner tatsächlichen Bewegung in der Szene oder seiner Bedeutung für die spezielle Anwendung.

[0017] Diese konventionelle Architektur stellt mehrere Hürden für die Erzeugung sehr realistischer und interaktiver Grafiken auf. Erstens wird jedes Objekt in einer Szene für ein bestimmtes Bild mit derselben Priorität bei derselben Aktualisierungsrate wiedergegeben. Objekte im Hintergrund, die als solche wenige Details haben und sich nicht bewegen, werden mit derselben Rate wiedergegeben wie Objekt im Vordergrund, die sich schnell bewegen und mehr Oberflächendetails aufweisen. Im Ergebnis dessen werden Verarbeitungs- und Speicherressourcen beim erneuten Wiedergeben von Hintergrundobjekten verbraucht, selbst wenn diese Hintergrundobjekte sich nicht wesentlich von Bild zu Bild unterscheiden.

[0018] Ein weiterer Nachteil bei dieser konventionellen Architektur ist, dass jedes Objekt in der Szene mit derselben Auflösung wiedergegeben wird. Effektiv stehen die Wiedergaberessourcen, die bei dieser Art von Ansatz verbraucht werden, in Beziehung zur Größe des Bildschirmfläche, die das Objekt belegt, und nicht mit der Bedeutung des Objektes für die Gesamtszene. Mit einem Beispiel lässt sich dieses Problem besser illustrieren. In einem typischen Videospiel gibt es handelnde Personen im Vordergrund, die sich bei jedem Bild ändern können, und einen Hintergrund, der sich selten von Bild zu Bild ändert. Die Kosten für das Erzeugen des Hintergrundes in Form von Speichernutzung sind viel größer als für das Erzeugen der handelnden Personen, weil der Hintergrund viel mehr Fläche auf dem Bildschirm einnimmt. Bilddaten müssen für jede Pixelstelle gespeichert werden, die Hintergrundobjekte belegen. Für die kleineren handelnden Personen jedoch werden Pixeldaten nur für die Pixel erzeugt und gespeichert, die von den kleineren Personen belegt werden. In Folge dessen belegt der Hintergrund mehr Speicher, obwohl er eine geringere Bedeutung in der Szene besitzt. Außerdem muss in einer konventionellen Architektur der gesamte Hintergrund neu für jedes Teilbild wiedergegeben werden, was wertvolle Verarbeitungsressourcen verbraucht.

[0019] Eine wesentliche Stärke des Ansatzes mit dem Bildpufferspeichers ist, dass er zum Aufbau eines beliebigen Bildes auf einem Ausgabegerät mit einer beliebigen Zahl von Grundobjekten verwendet werden kann, die nur der Begrenzung durch räumliche und Intensitätsauflösung des Ausgabegerätes unterliegt. Es gibt jedoch mehrere Schwachpunkte bei einem Grafiksystem, das einen Bildpufferspeicher verwendet.

[0020] Ein Bildpufferspeicher verwendet eine große Menge von teurem Speicher (z.B. 64-128 MB). Normaler Direktzugriffsspeicher (RAM) ist für Bildpufferspeicher wegen seiner niedrigen Zugriffsgeschwindigkeit nicht geeignet. Zum Beispiel dauert das Löschen der Millionen Pixel auf einem 1024 × 1024 großen Bildschirm etwa ¼ Sekunde, wenn man annimmt, dass jeder Speicherzyklus 250 Nanosekunden dauert. Daher wird normalerweise schnellerer und teurerer Video-RAM (VRAM) oder dynamischer RAM (DRAM) für Bildpufferspeicher verwendet. Hochleistungssysteme enthalten oft zwei teure Bildpufferspeicher: Ein Bildpufferspeicher wird zur Anzeige des aktuellen Teilbildes verwendet, während der andere zum Wiedergeben des nächsten Teilbildes verwendet wird. Diese große Menge an speziellem Speicher erhöht die Kosten des Grafiksystems dramatisch.

[0021] Speicherbandbreite für Bildpufferspeicher ist ebenfalls ein Problem. Die Unterstützung der Verarbeitung eine Grafikbildes mit Strukturierungs-, Farb- und Tiefeninformationen, die für jedes Pixel gespeichert sind, erfordert eine Bandbreite von etwa 1,7 Gigabyte pro Sekunde für die Verarbeitung eines Bildes bei 30 Hz. Da ein typischer DRAM nur eine Bandbreite von 50 MB/s hat, muss ein Bildpufferspeicher aus einer großen Zahl von DRAMs aufgebaut werden, die mit Parallelverarbeitungsverfahren verarbeitet werden, um die gewünschte Bandbreite zu erreichen.

[0022] Um interaktive Echtzeiteffekte zu erreichen, verwenden High-End-Grafiksysteme parallele Rendering-Engines. Es sind drei grundlegende Parallelstrategien entwickelt worden, um die Probleme mit großem Bildpufferspeicher zu lösen: (1) Pipelineverarbeitung des Wiedergabeprozesses über mehrere Prozessoren; (2) Aufteilen des Bildpufferspeichers in Gruppen von Speicherchips, jede mit ihrem eigenen Prozessor, und (3)

Kombinieren der Verarbeitungsschaltkreise auf den Bildpufferspeicherchips mit dichten Speicherschaltkreisen. Diese Verfahren haben die Verarbeitung von Grafiksystemen unter Verwendung von großen Bildpufferspeichern verbessert, haben aber auch die Kosten dieser Systeme dramatisch ansteigen lassen.

[0023] Selbst mit teuren Parallelverarbeitungsverfahren ist es sehr schwierig, das ausgeklügelte Anti-Aliasing-Verfahren zu unterstützen. Anti-Aliasing betrifft Prozesse zum Reduzieren von Artefakten in einem wiedergegebenen Bild, das durch die Darstellung kontinuierlicher Flächen mit diskreten Pixeln verursacht wird. In typischen Bildpufferspeicherarchitekturen werden Pixelwerte für ein ganzes Teilbild in beliebiger Reihenfolge berechnet. Zur Ausführung des ausgeklügelten Anti-Aliasing müssen daher Pixeldaten für das ganze Teilbild erzeugt werden, bevor das Anti-Aliasing beginnen kann. In einem Echtzeitsystem gibt es nicht ausreichend Zeit, das Anti-Aliasing auf die Pixeldaten anzuwenden, ohne eine zusätzliche Transportverzögerung zu erhalten. Außerdem erfordert das Anti-Aliasing zusätzlichen Speicher zur Speicherung von Pixelfragmenten. Da ein Bildpufferspeicher eine große Menge an teurem Speicher umfasst, wird durch den zusätzlichen Spezialspeicher, der zur Unterstützung des Anti-Aliasing benötigt wird, das Bildpufferspeichersystem noch teurer.

[0024] Bildkomprimierungsverfahren können ebenfalls nicht ohne weiteres in einem Grafiksystem unter Verwendung eines Bildpufferspeichers während der Bildverarbeitung eingesetzt werden. Durch die Verwendung von Parallelverarbeitungsverfahren zur Beschleunigung der Verarbeitung in einem Grafiksystem mit einem Bildpufferspeicher werden Hürden für den Einsatz von Komprimierungsverfahren aufgebaut. Während der Parallelverarbeitung kann auf jeden Teil des Bildpufferspeichers direkt zu jeder Zeit zugegriffen werden. Die meisten Bildkomprimierungsverfahren erfordern, dass Bilddaten sich während der Komprimierungsverarbeitung nicht ändern, so dass die Bilddaten später dekomprimiert werden können.

[0025] In Bildpufferspeicherarchitekturen wird der teure Speicher und die Parallelverarbeitungshardware immer zu wenig ausgenutzt, weil nur ein kleiner Bruchteil des Bildpufferspeichers oder der Parallelverarbeitungseinheiten zu einem beliebigen Zeitpunkt aktiv eingesetzt wird. Daher wird, obwohl eine Bildpufferspeicherarchitektur eine große Menge an teurem Speicher und Verarbeitungshandware umfasst, diese Hardware nicht vollständig ausgenutzt.

KURZDARSTELLUNG DER ERFINDUNG

[0026] Die Erfindung stellt ein Verfahren und System zur Wiedergabe von grafischen Daten bereit, wie zum Beispiel geometrische Grundelemente zur Erzeugung von Anzeigebildern. Die Erfindung ist besonders gut für die Wiedergabe von 3D-Grafiken in Echtzeit geeignet, kann aber auch auf andere Grafik- und Bildverarbeitungsanwendungen angewendet werden.

[0027] In einer Implementierung des Grafikwiedergabesystems gibt das System getrennt grafische Objekte in Bildschichten, die Gsprites genannt werden, wieder und setzt dann die Gsprites zu einem Anzeigebild zusammen. Genauer gesagt, weist das System Gsprites Objekten zu und rendert dann jedes Objekt oder Objekte in ein entsprechendes Gsprite. Um ein Gsprite zu rendern, gibt das System Bildregionen oder Chunks des Gsprites nacheinander wieder. Das System unterteilt Gsprites in Chunks, sortiert die Objektgeometrie in diesen Chunks und gibt dann diese Chunks nacheinander wieder. Das System setzt die Gsprites zu einem Anzeigebild zusammen.

[0028] Ein Aspekt der Erfindung ist die Art, wie Gsprites transformiert werden können, um die Bewegung eines 3D-Objekts zu simulieren und den zusätzlichen Aufwand für die Wiedergabe zu reduzieren. In einer Implementierung rendert das System Objekte in einer Szene in separate Gsprites. Nach dem Rendern eines Objektes in ein Gsprite kann das System den Gsprite für nachfolgende Teilbilder wieder verwenden, statt das Objekt neu zu rendern. Um dies zu erreichen, berechnet das System eine affine Transformation, die die Bewegung des 3D-Objektes simuliert, das der Gsprite repräsentiert. Das System führt eine affine Transformation am Gsprite aus und setzt diesen Gsprite mit anderen Gsprites zusammen, um ein Anzeigebild zu erzeugen.

[0029] Ein weiterer Aspekt der Erfindung ist die Art, wie das System Pixelfragmente für Chunks von Bilddaten verarbeitet. Das System rastert Grundelemente für einen Chunk, um Pixeldaten für Pixelorte zu erzeugen, die von einem Grundelement entweder vollständig oder teilweise bedeckt werden. In Fällen, in denen ein Grundelement einen Pixelort teilweise bedeckt oder Lichtdurchlässigkeit besitzt, erzeugt das System ein Pixelfragment und speichert das Fragment in einem Fragmentpuffer. In Fällen, in denen ein Grundelement einen Pixelort vollständig bedeckt und lichtundurchlässig ist, speichert das System seine Farbdaten in einem Pixelpuffer. Das System rastert Grundelemente für einen Chunk und löst dann die Pixeldaten für den Chunk in einem Nachverarbeitungsschritt auf. Die Architektur zum Rendern von Chunks ermöglicht die Ausführung eines aus-

geklügelten Anti-Aliasing an den Pixeldaten, während gleichzeitig Anzeigebilder in Echtzeitraten erzeugt werden.

[0030] Ein weiterer Aspekt der Erfindung ist die Art, wie das Rasterprogramm im System Fragmentspeicher dadurch sparen kann, dass versucht wird, ein erzeugtes Pixelfragment mit einem Fragment zu vereinen, das im Fragmentpuffer gespeichert ist. Wenn ein gespeichertes Fragment innerhalb einer vorgegebenen Tiefen- und Farbtoleranz des erzeugten Fragmentes liegt, vereint eine Pixelengine im System die Fragmente. Die Pixelengine vereint die Fragmente zum Teil durch Kombinieren der Bedeckungsdaten (z.B. eine Bedeckungsmaske) des erzeugten und des gespeicherten Fragments. Wenn das vereinte Pixelfragment voll abgedeckt und lichtundurchlässig ist, kann die Pixelengine es zu einem entsprechenden Pixelpuffereintrag verschieben und den Fragmentsatz aus dem Fragmentspeicher lösen.

[0031] Noch ein weiterer Aspekt der Erfindung ist die Art, wie das Fragmentauflösungssubsystem Listen von Fragmentsätzen auflöst. Bei einem Ansatz hat ein Fragmentauflösungssubsystem separate Farb- und Alphazwischenspeicher für jeden einzelnen Subpixelort eines Pixels, und es speichert die Farbe an jedem Subpixelort separat. Das Subsystem umfasst die Logik zum Kombinieren der zwischengespeicherten Farbe von jedem Subpixelort, um ein endgültiges Ausgabepixel zu berechnen. Bei einem anderen Ansatz hält sich das Fragmentauflösungssubsystem über die Subpixelregionen auf dem laufenden, die einen gemeinsamen zwischengespeicherten Alphawert haben, wenn jeder Fragmentsatz in einer nach der Tiefe sortierten Liste der Fragmente aufgelöst wird. Dieses Fragmentauflösungssubsystem berechnet die zwischengespeicherte Farbe für die Regionen innerhalb eines Pixels (Pixelregionen), die ein gemeinsames zwischengespeichertes Alpha besitzen. Nach dem Auflösen jedes Fragmentes in einer Liste ist die Ausgabe beider Ansätze eine Ausgabepixel, das einen einzelnen Satz von Farbwerten (RGB) und möglicherweise einen Alphawert besitzt. Für jeden Pixelort kombiniert das Fragmentauflösungssubsystem die Farbwerte im Pixelpuffer mit allen Fragmentsätzen in einer zugehörigen Fragmentliste, um einen aufgelösten Pixelwert, einschließlich zum Beispiel von RGB-Farbwerten und einem Alphawert, zu berechnen.

[0032] Ein weiterer Aspekt der Erfindung ist ein Verfahren zur Ausführung einer anisotropen Filterung. Bei der Texturabbildung im allgemeinen bildet ein grafisches Renderingsystem eine Texturabbildung auf die Fläche eines geometrischen Grundelementes ab. Bei diesem speziellen Verfahren beginnt das System damit festzustellen, wie ein Punkt an einem Pixelort im Betrachtungsraum sich auf die Texturkarte abbildet. Konzeptionell bestimmt das System, wie sich ein Filterabdruck auf die Texturkarte abbildet. Für eine perspektivische Abbildung hat ein isotroper Filterabdruck, der in die Texturkarte abgebildet ist, eine verformte Gestalt in Richtung der Anisotropie. Daher ist das Filtern der Textur mit einem anisotropen Filter nicht ausreichend, um hochwertige Ergebnisse zu erhalten. In einer speziellen Ausführungsform bestimmt das System, wie ein Filterabdruck sich in die Textur abbildet, indem die inverse Jacobi-Matrix für einen Pixelort in Betrachtungsraumkoordinaten (z.B. Bildschirmkoordinaten), die auf Texturkoordinaten abgebildet sind, berechnet wird.

[0033] Das System bestimmt dann eine Anisotropielinie von dem abgebildeten Filterabdruck und bestimmt speziell in dieser einen Ausführungsform die Anisotropielinie von der inversen Jacobi-Matrix. Die Anisotropielinie ist begrifflich eine Linie, die durch die Koordinaten des Punktes läuft, der aus dem Betrachtungsraum in den Texturraum abgebildet wird und in Richtung der maximalen Ausdehnung des abgebildeten Filterabdrucks ausgerichtet ist. Das System wendet den Filter wiederholt entlang der Anisotropielinie an, um Werte aus der Texturkarte zu prüfen. Die Ausgabewerte dieses wiederholten Filtrationsschritts werden gefiltert und gesammelt, um endgültige Texturwerte zu berechnen. Es gibt eine Reihe von Variationen zu dieser Vorgehensweise. In einer speziellen Implementierung führt das System eine trilineare Interpolation entlang der Anisotropielinie aus. Die Ausgabe des trilinearen Filters wird dann kombiniert, um einen einzelnen Satz von Farbwerten für einen Pixelort zu berechnen. Bei dieser Implementierung wendet eine Texturfilter-Engine einen eindimensionalen Filter, zum Beispiel in Form eines Dreiecks oder Trapezoids, auf die Ausgabewerte der dreilinigen Interpolation entlang der Anisotropielinie an. Bei Verwendung dieses Verfahrens ist jedoch eine Reihe von Variationen an den Filtern, die entlang der Anisotropielinie angewendet werden, möglich.

[0034] Ein weiterer Aspekt der Erfindung ist die Art, wie das System kleine Teile eines Bildes rendern kann, falls der Fragmentspeicher überläuft. In einer Implementierung verfolgt das System die Verwendung des Fragmentspeichers und kann eine Bildregion in kleinere Teile aufteilen, wenn die Zahl der verwendeten Fragmenteinträge einen vorgegebenen Wert erreicht. Wenn das System Pixelfragmente erzeugt, verfolgt es die Zahl der Einträge in den Fragmentpuffer. Wenn die Zahl der Einträge einen vorgegebenen Wert erreicht, wird die Bildregion in kleinere Regionen unterteilt und rendert die kleineren Regionen jeweils eine mit einem Mal, so dass ausreichend Fragmentspeicher zur Verfügung steht, um jede Teilregion zu rendern. Das System kann eine Teilregion in noch kleinere Bildregionen aufteilen, wenn die Zahl der Fragmenteinträge den vorgegebenen

Wert erreicht. Im Ergebnis dessen kann das System die Bildregion, die gerade gerendert wird, unterteilen, um sicherzustellen, dass der Fragmentspeicher nicht überschritten wird. Dies ermöglicht dem System, einen kleineren Fragmentspeicher einzusetzen, ohne Fragmente zu verwerfen, wo anderenfalls der Fragmentspeicher überlaufen würde.

[0035] Ein weiterer Aspekt der Erfindung ist die Art, wie das System Texturabrufoperationen in Umgebungen mit hoher Latenz ausführt. Zum Beispiel gibt es bei Texturabbildung, Schattenbildung (shadowing) oder Mehrfachdurchlauf-Renderingoperationen oft eine hohe Verzögerungszeit bzw. Latenz beim Abrufen der Texturdaten, um die Operation auszuführen. Diese Verzögerungszeit kann wegen der Verzögerung, die beim Lesen von Daten aus dem Speicher auftritt, der Verzögerung, die beim Dekomprimieren von Texturdaten auftritt, oder wegen beidem entstehen.

[0036] In einer Implementierung werden geometrische Grundelemente in einem Eingabedatenstrom in einer Grundelementeschlange gespeichert, die ausreichend lang ist, um die Verzögerungszeit des Abrufens eines Blocks von Texturdaten aus dem Speicher aufzufangen. Eine Vorrastervorrichtung wandelt die geometrischen Grundelemente in der Grundelementewarteschlange in Texturblockreferenzen um, die in einer zweiten Schlange gespeichert werden. Die Texturblöcke, auf die in dieser zweiten Schlange verwiesen wird, werden aus dem Speicher abgerufen und in einen Texturcache gebracht. Nacheinander rastert eine Postrastervorrichtung jedes Grundelement in der Schlange. Wenn jedes Grundelement gerastert ist, werden Texturdaten aus dem Texturcache nach Bedarf abgerufen, um die Ausgabepixel für das aktuelle Grundelement zu berechnen. Die Grundelemente werden aus der Schlange entfernt, nachdem sie gerastert sind.

[0037] In einer zweiten Implementierung werden Grundelemente gerastert und die resultierenden Pixeldaten werden in eine Schlange gebracht, die ausreichend lang ist, um die Verzögerungszeit des Abrufens eines Texturblocks aufzufangen. In einer speziellen Implementierung umfassen die Einträge in der Schlange eine Pixeladresse, Farbdaten für diese Adresse und eine Texturanforderung, die aus dem Mittelpunkt einer Texturprobe in den Koordinaten einer Texturkarte besteht. Die Texturanforderungen werden in Texturblockadressen umgewandelt und die Texturblöcke werden abgerufen und in einen Texturcache gebracht. Die Einträge in der Schlange werden aus der Schlange abgerufen und zugehörige Texturdaten, die sich nun im Texturcache befinden, werden dazu verwendet, Ausgabepixel zu berechnen. Beide Vorgehensweisen erzeugen zwei Sätze von Textanforderungen, wobei jeder Satz gegenüber dem anderen verzögert ist. Der erste Satz wird dazu verwendet, die Texturdaten tatsächlich abzurufen und möglicherweise zu dekomprimieren, und der zweite Satz wird dazu verwendet, Texturdaten aus einem Texturcache zu holen.

[0038] Weitere Merkmale und Vorteile der Erfindung werden bei Bezugnahme auf die folgende ausführliche Beschreibung und die begleitenden Zeichnungen erkennbar.

KURZBESCHREIBUNG DER ZEICHNUNGEN

[0039] Fig. 1 ist ein Schema eines Bildverarbeitungssystems.

[0040] Fig. 2 ist ein Schema der Systemumgebung für eine Ausführungsform der Erfindung.

[0041] Fig. 3 ist ein Schema der Systemarchitektur für eine Ausführungsform.

[0042] Fig. 4A ist ein Schema der Bildverarbeitungshardware für eine Ausführungsform.

[0043] Fig. 4B ist ein Schema, das Teile eines Bildprozessors zum Rendern der geometrischen Grundelemente in einer Ausführungsform zeigt.

[0044] Die Fig. 5A und Fig. 5B sind Flussdiagramme, die einen Überblick über den Renderingprozess in einer Ausführungsform illustrieren.

[0045] Fig. 6 ist ein Flussdiagramm, das einen Überblick über den Anzeigeerzeugungsprozess einer Ausführungsform illustriert.

[0046] Fig. 7 ist ein Diagramm, das einen Aspekt der Anzeigeerzeugung bezüglich der Teilbildperioden in einer Ausführungsform illustriert.

[0047] Fig. 8 ist ein Schema eines Digitalen Signalprozessors (DSP) in einer Ausführungsform.

[0048] Die Fig. 9A-C sind Schemata, die alternative Ausführungsformen eines Tilers illustrieren.

[0049] Fig. 10 ist ein Schema, das ein System für den Zugriff auf Texturdaten aus dem Speicher illustriert.

[0050] Fig. 11 ist ein Schema, das ein System für den Zugriff auf Texturdaten aus dem Speicher illustriert.

[0051] Die Fig. 12A-B sind Schemata alternativer Implementierungen einer Gsprite-Engine.

[0052] Fig. 13 ist ein Schema eines Zusammensetzungspuffers in einer Ausführungsform.

[0053] Fig. 14 ist ein Schema eines Digital-Analog-Converters (DAC) in einer Ausführungsform.

[0054] Die Fig. 15A-C sind Diagramme eines Beispiels, das einen Aspekt des Chunking illustriert.

[0055] Die Fig. 16A-B sind Schemata, die Aspekte des Chunking in einer Ausführungsform illustrieren.

[0056] Die Fig. 17A-B sind Flussdiagramme, die Aspekte des Chunking in einer Ausführungsform illustrieren.

[0057] Die Fig. 18A-B sind Schemata, die Aspekte des Chunking in einer Ausführungsform illustrieren.

[0058] Die Fig. 19A-B sind Schemata, die Aspekte des Chunking in einer Ausführungsform illustrieren.

[0059] Fig. 20 ist ein Schema, das die Bildkompression in einer Ausführungsform illustriert.

[0060] Die Fig. 21A-B sind Flussdiagramme, die die Verarbeitung von Gsprites in einer Ausführungsform illustrieren.

[0061] Fig. 22 ist ein Flussdiagramm, das einen Aspekt eines Verfahrens zur Ausführung von Gsprite-Transformationen in einer Ausführungsform illustriert.

[0062] Fig. 23 ist ein Diagramm, das illustriert, wie Gsprite-Transformationen die Transportverzögerung in einer Ausführungsform reduzieren können.

[0063] Fig. 24 ist ein Schema von Gsprite-Datenstrukturen in einer Ausführungsform.

[0064] Fig. 25 ist ein Diagramm, das ein Beispiel eines Gsprites, der auf Ausgabegerätekoordinaten abgebildet wird, in einer Ausführungsform illustriert.

[0065] Fig. 26 ist ein Flussdiagramm, das einen Aspekt der Anzeigeerzeugung in einer Ausführungsform illustriert.

[0066] Fig. 27 ist ein Flussdiagramm, das die Anzeigeerzeugung von Fig. 26 in Bezug auf die Bandperioden illustriert.

[0067] Die Fig. 28A-F sind Flussdiagramme, die Aspekte der Pixel- und Fragmenterzeugung in drei alternativen Ausführungsformen illustrieren.

[0068] Fig. 29 ist ein Flussdiagramm eines Verfahrens zum Verschmelzen von Fragmenten in einer Ausführungsform der Erfindung.

[0069] Fig. 30 ist ein Schema, das eine Implementierung von Fragmentverschmelzungsschaltkreisen in einer Ausführungsform der Erfindung illustriert.

[0070] Fig. 31 ist ein Schema, das eine Implementierung eines Verschmelzungstestmoduls in den Fragmentverschmelzungsschaltkreisen illustriert, die in Fig. 30 gezeigt werden.

[0071] Fig. 32 ist ein Diagramm, das einen Teil des Pixel- und Fragmentpuffers illustriert.

[0072] Fig. 33 ist ein Diagramm, das diese hierarchische Zerlegung abbildet.

[0073] Die Fig. 34A-B ist ein Flussdiagramm, das ein Verfahren zur Pufferzerlegung im Tiler illustriert.

[0074] Fig. 35 ist ein Schema, das eine Implementierung eines Fragmentauflösungs-Teilsystems illustriert.

[0075] Fig. 36 ist ein Schema, das eine weitere Implementierung eines Fragmentauflösungs-Teilsystems illustriert.

[0076] Fig. 37 ist ein Diagramm, das die Texturabbildung illustriert.

[0077] Die **Fig.** 38A-D sind Schemata, die ein Verfahren zur anisotropen Filterung in einer Ausführungsform illustrieren.

[0078] Fig. 39 ist ein Schema, das eine Implementierung eines Textur- und Schattenfilters illustriert.

[0079] Fig. 40 ist ein Schema, das eine Implementierung des Keygenerators in Fig. 39 illustriert.

[0080] Fig. 41 ist ein Schema, das eine Implementierung des Farbinterpolators in Fig. 39 illustriert.

[0081] Fig. 42 ist ein Schema, das eine Implementierung des Schattenfilterakkumulators in Fig. 39 illustriert.

[0082] Fig. 43 ist ein Schema, das eine Implementierung des Akkumulators und Postprozessors in Fig. 39 illustriert.

AUSFÜHRLICHE BESCHREIBUNG

Systemübersicht

[0083] In der folgenden detaillierten Beschreibung beschreiben wir mehrere Ausführungsformen unter Bezugnahme auf ein Bildverarbeitungssystem.

[0084] Das Bildverarbeitungssystem unterstützt Echtzeit-Bild-Rendering und -Erzeugung sowohl für Grafik- als auch Videoverarbeitung. Auf Grund der neuartigen Architektur und Bildverarbeitungsverfahren, die im System eingesetzt werden, kann es hoch entwickelte Echtzeit-3D-Animationen bei beträchtlichen Kosteneinsparungen gegenüber derzeitigen Grafiksystemen erzeugen. Neben der Grafikverarbeitung unterstützt das System die Videoverarbeitung, wie zum Beispiel Videoeditieranwendungen, und kann auch Video und Grafik kombinieren. Das System kann zum Beispiel dazu benutzt werden, Video auf grafische Objekte anzuwenden, oder umgekehrt kann es eingesetzt werden, um grafische Objekte zu Videodaten hinzuzufügen.

[0085] Das System unterstützt eine breite Palette von interaktiven Anwendungen. Mit seiner Fähigkeit, fortgeschrittene Echtzeitanimation zu unterstützen, ist es gut für Spiele, Erziehungsanwendungen und viele interaktive Anwendungen geeignet. Das System unterstützt hochentwickelte Nutzerschnittstellen, einschließlich 3D-Grafik oder eine Kombination von Grafik und Video.

[0086] Durch Verbesserung der begrenzten Grafikfähigkeiten der heutigen Fensterdarstellungsumgebungen für Personal Computer kann das System verbesserte grafische 3D-Nutzerschnittstellen für Anwendungen unterstützen, die von der Büroinformationsverarbeitung auf Desktop-Computern bis zu interaktiven Fernsehanwendungen in einer Set-Top-Box reichen. Das System nutzt Speicher- und Prozessorzeit sehr effizient und kann daher eine beeindruckende Bildverarbeitung und -anzeige liefern, ohne die Ausführung der Anwendung oder die Reaktionsfähigkeit der Nutzerschnittstelle auf Nutzeraktionen ungebührlich zu behindern.

[0087] Fig. 1 ist ein Schema des Bildverarbeitungssystems **100**. Das Bildverarbeitungssystem umfasst eine Bilddatenquelle und -speicher **102**, einen Bildpräprozessor **104**, einen Bildprozessor **106** und eine Anzeigevorrichtung **108**, wenn eine sofortige Anzeige von gerenderten Bildern gewünscht wird. Die Elemente im System kommunizieren durch eine Systemschnittstelle **110**. Die Bilddatenquelle und -speicher **102** liefert Bilddaten an das System und speichert Bilddaten und Befehle. Der Bildpräprozessor **104** ist für die Handhabung der Bilddaten verantwortlich, durch die sie auf das Rendern vorbereitet werden. Beispiele für Präprozessorfunktionen sind u.a.: Definieren von Objekten im Hinblick auf geometrische Modelle, Definieren von Beleuchtungs- und Schattenbildungsmodelle, Bestimmen von Objektorten, Bestimmen des Orts eines Betrachtungspunktes und von Lichtquellen und Geometrieverarbeitung.

[0088] Der Bildprozessor **106** rendert die Bilder und erzeugt ein Anzeigebild, das auf der Anzeigevorrichtung **108** angezeigt werden soll. Rendern bezieht sich auf den Prozess der Erzeugung von Bildern aus Modellen und umfasst solche Funktionen wie Geometrieverarbeitung (man beachte, dass die Geometrieverarbeitung auch eine Vorverarbeitungsfunktion sein kann), Bestimmung sichtbarer Flächen, Scankonversion und Beleuchtung, um nur ein paar zu nennen. Nach dem Rendern eines Bildes oder von Teilen eines Bildes überträgt der Bildprozessor **106** die gerenderten Bilddaten auf die Anzeigevorrichtung zur Anzeige.

[0089] Unten beschreiben wir mehrere Merkmale des Bildverarbeitungssystems **100** im Detail unter Bezugnahme auf spezielle Hardware- und Software-Architektur. Es ist jedoch wichtig zu bemerken, dass die Bildverarbeitung, die unten beschrieben wird, in einer Reihe von alternativen Architekturen implementiert werden kann.

[0090] Das Bildverarbeitungssystem **100** erreicht eine riesige Verbesserung des Preis-Leistungsverhältnisses gegenüber hochwertigen 3D-Grafiksystemen, die den Erfindern bekannt sind. Eine Reihe von Fortschritten in der Computergrafik trägt zu dieser Verbesserung bei. Zu diesen Fortschritten gehören: zusammengesetzte Bildschichten, Bildkompression, Chunking und Mehrfachrendering. Wir stellen diese Fortschritte hier vor und beschreiben detaillierter diese und andere Fortschritte unten.

Zusammengesetzte Bildschichten (Gsprites)

[0091] In unserem System können mehrere unabhängige Bildschichten bei Videoraten zusammengesetzt werden, um das Ausgabevideosignal zu erzeugen. In diese Bildschichten, die wir verallgemeinerte Sprites oder Gsprites nennen, kann gerendert werden, und sie können unabhängig voneinander manipuliert werden. Das System verwendet im allgemeinen einen unabhängigen Gsprite für jedes andere nicht durchdringende Objekt in der Szene. Dies ermöglicht es, jedes Objekt unabhängig zu aktualisieren, so dass die Objektaktualisierungsrate auf der Basis von Prioritäten in der Szene optimiert werden kann. Ein Objekt, zum Beispiel, das sich im fernen Hintergrund bewegt, braucht nicht so oft oder mit solcher Genauigkeit wie ein Objekt im Vordergrund aktualisiert zu werden.

[0092] Gsprites können eine beliebige Größe und Form besitzen. In einer Ausführungsform verwenden wir rechteckige Gsprites. Pixel im Gsprite haben Farb- und Alpha-Informationen, die mit ihnen verbunden sind, so dass mehrere Gsprites zusammengesetzt werden können, um die Gesamtszene zu erzeugen.

[0093] An Gsprites können mehrere verschiedene Operationen bei Videoraten ausgeführt werden, einschließlich Skalieren, Rotation, Subpixelpositionierung und Transformationen zur Imitation von Bewegung, wie zum Beispiel affine Warps. Während also Gsprite-Aktualisierungsraten variabel sind, können Gsprite-Transformationen (Bewegung usw.) bei vollen Videoraten auftreten, was zu einer viel flüssigeren Dynamik führt, als durch ein konventionelles 3D-Grafiksystem erreicht werden könnte, das keine Aktualisierungsratengarantien besitzt.

[0094] Viele 3D-Transformationen können durch 2D-Bildoperationen simuliert werden. Ein sich zurückziehendes Objekt kann durch Skalieren der Größe des Gsprites simuliert werden. Durch die Nutzung von 2D-Transformationen für vorher gerenderte Bilder für intermediäre Teilbilder werden die Gesamtverarbeitungsanforderungen beträchtlich reduziert, und die 3D-Renderingleistung kann dort angewendet werden, wo sie benötigt wird, um Ergebnisse mit höchster Qualität zu erreichen. Dies ist eine Form von zeitlicher Ebene des Detailmanagements.

[0095] Durch die Anwendung der Gsprite-Skalierung kann das Niveau der räumlichen Detailliertheit ebenfalls so angepasst werden, dass es den Prioritäten der Szene entspricht. Hintergrundobjekte, bewölkter Himmel usw. können zum Beispiel in einen kleinen Gsprite (niedrige Auflösung) gerendert werden, der dann auf die richtige Größe für die Anzeige skaliert wird. Durch die Nutzung von hochwertiger Filterung sind die typischen Artefakte bei niedriger Auflösung nicht wahrnehmbar.

[0096] Eine typische 3D-Grafikanwendung (besonders ein interaktives Spiel) verzichtet auf ein hohes Niveau an geometrischen Details, um höhere Animationsraten zu erreichen. Gsprites ermöglichen dem System, zwei zusätzliche Szenenparameter zu nutzen – das zeitliche Detailniveau und das räumliche Detailniveau – um die effektive Ausführung zu optimieren, wie sie vom Nutzer gesehen wird. Die räumliche Auflösung, bei der das Bild eines Objektes gerendert wird, braucht nicht mit der Bildschirmauflösung übereinzustimmen, mit der es gerendert wird. Ferner kann das System diese Abstriche automatisch handhaben, ohne Unterstützung durch die Anwendung zu erfordern.

Bildkompression

[0097] Der vielleicht wichtigste Faktor bei der Bestimmung von Systemkosten und -leistung ist der Speicher. Ein traditionelles Highend-3D-Grafiksystem zum Beispiel hat über 30 MByte Speicher, einschließlich Bildpufferspeicher (doppelt gepuffert), eines Tiefenpuffers, eines Texturpuffers und eines Anti-Aliasing-Puffers. Und der größte Teil davon ist spezieller Speicher, der deutlich teurer ist als DRAM. Die Speicherbandbreite ist immer ein kritischer Flaschenhals. Die Kosten für Hochleistungssysteme werden oft durch die Notwendigkeit angetrieben, zahlreiche Bänke von verschachteltem Speicher bereitzustellen, um so für eine angemessene Bandbreite für Zugriffe auf Pixel und Texturdaten zu sorgen.

[0098] Das System wendet weitgehend Bildkompressionstechnologie an, um diese Probleme zu lösen. Traditionell ist die Bildkompression wegen der rechnerischen Komplexität, die für eine hohe Qualität erforderlich ist, und weil sie sich nicht leicht in eine konventionelle Grafikarchitektur einfügen lässt, nicht in Grafiksystemen verwendet worden. Durch die Verwendung eines Konzepts, das wir Chunking nennen (wird unten beschrieben), sind wir in der Lage, die Kompression wirksam auf Bilder und Texturen anzuwenden, wodurch wir eine beträchtliche Verbesserung des Preis-Leistungsverhältnisses erreichen.

[0099] In einer Hinsicht haben Grafiksysteme die Kompression auf Bildpufferspeicher doch angewendet. Spitzensysteme verwenden acht Bits für jede der drei Farbkomponenten und enthalten oft auch einen Achtbit-Alphawert. Preiswerte Systeme komprimieren diese 32 Bit pro Pixel in nur vier Bit, wobei Informationen verworfen werden und/oder eine Farbpalette verwendet wird, um die Zahl gleichzeitig anzeigbarer Farben zu reduzieren. Diese Kompression führt zu stark sichtbaren Artefakten, erreicht keine deutliche Reduzierung in den Datenanforderungen und zwingt Anwendungen und/oder Treiber dazu, sich mit einer breiten Palette von Pixelformaten abzugeben.

[0100] Die Kompression, die in unserem System verwendet wird, kann eine sehr hohe Bildqualität erreichen und doch Kompressionsverhältnisse von 10:1 oder besser bereitstellen. Ein weiterer Vorteil unserer Herangehensweise ist, dass ein einziges hochqualitatives Bildformat für alle Anwendungen verwendet werden kann, das sich von der Standard-PC-Grafikarchitektur unterscheidet, welche Kompromisse zwischen räumlicher Auflösung und Farbtiefe erfordert.

Chunking

[0101] Ein weiterer wichtiger Fortschritt in unserem System wird als Chunking bezeichnet. Auf ein traditionelles 3D-Grafiksystem (oder eigentlich jeden Bildpufferspeicher) kann direkt zugegriffen werden (und wird zugegriffen). Auf beliebige Pixel auf dem Bildschirm kann direkt zugegriffen werden. Da Kompressionsalgorithmen sich darauf verlassen, dass sie Zugriff auf eine recht große Zahl von benachbarten Pixeln haben (um Nutzen aus der räumlichen Kohärenz zu ziehen), kann die Kompression erst angewendet werden, nachdem alle Pixelaktualisierungen vorgenommen wurden, was auf die Direktzugriffsmuster zurückzuführen ist, die von den Grafikalgorithmen verwendet werden. Dadurch wird die Anwendung der Kompressionstechnologie auf Anzeigepuffer undurchführbar.

[0102] Dieses Direktzugriffsmuster bedeutet auch, dass die pixelweise Entfernung von verborgenen Flächen und die Anti-Aliasing-Algorithmen zusätzliche Informationen für jedes Pixel auf dem Bildschirm enthalten müssen. Dadurch erhöhen sich die Speichergrößenanforderungen drastisch, und es entsteht ein weiterer Leistungsflaschenhals.

[0103] Unser System zeigt eine andere Herangehensweise. Eine Szene, oder Teile einer Szene, kann in Pixelbereiche (32×32 Pixel in einer speziellen Ausführungsform) unterteilt werden, die Chunks genannt werden. In einer Ausführungsform unterteilt das System die Geometrie, die den Gsprites zugewiesen wird, in Chunks, eine andere Ausführungsform könnte das Chunking ohne Gsprites ausführen. Die Geometrie wird in Bins vorsortiert, die darauf beruhen, in welchen Chunk die Geometrie gerendert wird. Dieser Prozess wird als Chunking bezeichnet. Geometrie, die eine Chunkgrenze überlappt, wird vorzugsweise in jedem Chunk referenziert, in dem sie sichtbar ist. Wenn die Szene animiert ist, wird die Datenstruktur zur Anpassung an die Geometrie modifiziert, die sich von einem Chunk zum anderen bewegt.

[0104] Chunking bietet mehrere große Vorteile. Die Verwendung von Chunking sorgt für eine effektive Form der Kompression. Da die ganze Geometrie in einem Chunk gerendert wird, bevor zum nächsten Schritt fortgeschritten wird, braucht der Tiefenpuffer nur so groß wie ein einzelner Chunk zu sein. Durch die Verwendung einer relativ geringen Chunkgröße, wie zum Beispiel 32×32 Pixel, kann der Tiefenpuffer direkt auf dem Gafik-

renderingchip implementiert werden. Dadurch wird eine beträchtliche Speichermenge eliminiert und ermöglicht auch, dass der Tiefenpuffer unter Verwendung einer speziellen Speicherarchitektur implementiert werden kann, auf die mit sehr hoher Bandbreite zugegriffen werden kann und der während der Doppelpufferoperationen gelöscht werden kann, was den zusätzlichen Aufwand bei der traditionellen Bildpufferspeicherlöschung zwischen den Teilbildern beseitigt.

[0105] Das Anti-Aliasing ist auch wesentlich einfacher, da jeder Chunk unabhängig behandelt werden kann. Die meisten Z-gepufferten Spitzengrafiksysteme, die das Anti-Aliasing implementieren, nutzen einen großen Teil von zusätzlichem Speicher und führen immer noch eine relativ einfache Filterung aus. Beim Chunking wird jedoch die Menge der benötigten Daten beträchtlich reduziert (um einen Faktor 1000), was die praktische Umsetzung eines viel höher entwickelten Anti-Aliasing-Algorithmus ermöglicht.

[0106] Zusätzlich zur Z-Pufferung und zum Anti-Aliasing kann das System gleichzeitig auch die Lichtdurchlässigkeit korrekt und nahtlos unterstützen. Während ein Chunk gerade aufgebaut wird, kann das System sowohl Anti-Aliasing- als auch Lichtdurchlässigkeitsberechnungen an einem weiteren Chunk ausführen. Mit anderen Worten, während der Zeit, die zum Aufbauen eines Chunks benötigt wird, kann das System Anti-Aliasing und Lichtdurchlässigkeit an einem weiteren Chunk verarbeiten. Das System kann zwischen Chunks hin- und herpendeln und so hoch entwickelte Verarbeitungsvorgänge ohne Verzögerung bei der Verarbeitung eines Bildes für Echtzeitanwendungen ausführen.

[0107] Ein weiterer Vorteil ist, dass das Chunking die blockorientierte Bildkompression ermöglicht. Sobald ein Chunk gerendert (und mit Anti-Aliasing behandelt) wurde, kann er mit einem auf Blocktransformation beruhenden Kompressionsalgorithmus komprimiert werden. Daher unterstützt das Chunking zusätzlich zur Kompression, die aus dem separaten Rendering der Chunks erreicht wird, höher entwickelte und besser anpassbare Kompressionsmethoden.

Mehrfachrendering

[0108] Ein weiterer Vorteil der Architektur unseres Systems ist die Möglichkeit, dass interaktive 3D-Anwendungen den Look der späten 70er Jahre von CAD-Grafiksystemen verlassen können: langweilige lambertsche Gouraud-schattierte Polygone mit Phong-Beleuchtung. Die Texturabbildung von Farbe verbessert das Aussehen, erzeugt aber ein weiteres charakteristisches Aussehen bei Anwendungen. In den 1980er Jahren eröffnete die Idee von programmierbaren Schattierern (Shader) und prozeduralen Texturabbildungen eine neue Flexibilität für den Renderingprozess. Diese Ideen gingen durch die Offline-Rendering-Welt und erzeugten die hochwertigen Bilder, die wir heute in Spezialeffekten von Filmen sehen.

[0109] Die starren Rendering-Pipelines und festen Renderingarten der heutigen typischen 3D-Spitzengrafikrechner machen es unmöglich, solche Effekte ohne drastische Abstriche bei der Echtzeitleistung zu implementieren. Im Ergebnis dessen müssen Nutzer, die eine Echtzeitanzeige fordern, sich mit der begrenzten Renderingflexibilität abfinden.

[0110] Durch Reduzieren der Bandbreitenanforderungen unter Verwendung der Verfahren, die oben umrissen wurden, kann das System der vorliegenden Erfindung ein einziges gemeinsam genutztes Speichersystem für alle Speicheranforderungen, einschließlich der Speicherung von komprimierter Textur und der Speicherung von komprimierten Gsprites, verwenden. Diese Architektur ermöglicht es, Daten, die durch den Renderingprozess erzeugt wurden, zurück durch den Texturprozessor zu führen, um sie als Daten beim Rendering eines neuen Gsprites zu verwenden. Wegen dieser Unterstützung der Rückführung kann das System ein effizientes Mehrfachrendering ausführen.

[0111] Durch Kopplung des effizienten Mehrfachrendering mit einer Reihe von Zusammensetzungsmodi und einer flexiblen Schattierungssprache kann das System eine Vielzahl von Renderingeffekten in Echtzeit bereitstellen, die vorher die Domäne der Offline-Software-Renderer waren. Dazu gehört die Unterstützung von Funktionen, wie zum Beispiel Schatten (einschließlich Schatten von mehreren Lichtquellen), umgebungsabgebildete reflektierende Objekte, Punktleuchtquellen, Bodennebel, realistische Unterwassersimulation usw.

[0112] In einer Ausführungsform umfasst das Bildverarbeitungssystem (**100**) eine Kombination von Hardware und Software. Im folgenden Abschnitt beschreiben wir die Systemumgebung unten unter Bezugnahme auf eine Hardware- und Software-Architektur. Wo möglich, beschreiben wir alternative Architekturen. Jedoch können die Hardware- und Software-Architekturen variieren und sind daher nicht auf die speziellen Beispiel begrenzt, die unten angeführt werden.

[0113] Das Bildverarbeitungssystem, oder Teile davon, kann auf einer Reihe von unterschiedlichen Plattformen implementiert werden, einschließlich Desktop-Computern, Set-Top-Boxen und Spielesystemen.

[0114] Fig. 2 ist ein Schema eines Computersystems **130**, in dem das Bildverarbeitungssystem implementiert werden kann. Das Computersystem **130** umfasst einen Prozessor **132**, Hauptspeicher **134**, Speichersteuerung **136**, Sekundärspeicher **138**, Eingabevorrichtung(en) **140**, Anzeigevorrichtung **142** und Bildverarbeitungshardware **144**. Die Speichersteuerung **136** dient als Schnittstelle zwischen dem Prozessor **132** und dem Hauptspeicher **134**; sie fungiert auch als Schnittstelle für Prozessor **132** und Hauptspeicher mit dem Bus **146**.

[0115] Eine Reihe von Computersystemen hat dieselbe oder eine ähnliche Architektur wie die in Fig. 2 illustrierte. In solchen Systemen können verschiedene Prozessoren verwendet werden. Außerdem umfassen einige Computersysteme mehr als eine Verarbeitungseinheit. Um ein paar zu benennen, kann der Prozessor ein Pentium- oder Pentium Pro-Prozessor von der Intel Corporation, ein Mikroprozessor aus der MIPS-Familie von Silicon Graphics, Inc., oder der PowerPC von Motorola sein.

[0116] Der Hauptspeicher **134** ist ein Hochgeschwindigkeitsspeicher und wird in den meisten herkömmlichen Computersystemen mit Direktzugriffsspeicher (RAM) implementiert. Der Hauptspeicher lässt sich mit dem Prozessor und Bus mit einer Reihe von bekannten Verfahren verbinden. Hauptspeicher **134** speichert Programme, wie zum Beispiel das Betriebssystem eines Computers und aktuell laufende Anwendungsprogramme. Unten beschreiben wir Erscheinungsformen einer Ausführungsform mit Bezug auf die symbolischen Repräsentationen von Anweisungen, die durch das Computersystem ausgeführt werden. Diese Anweisungen werden manchmal als computer-ausgeführt bezeichnet. Diese Aspekte der Ausführungsform können in einem Programm oder Programmen implementiert werden, die eine Reihe von Anweisungen umfassen, welche auf einem computer-lesbaren Medium gespeichert sind. Das computer-lesbare Medium kann eine der Vorrichtungen oder eine Kombination der Vorrichtungen, die hierin beschrieben werden, in Verbindung mit einem Hauptspeicher oder einem Ergänzungsspeicher sein.

[0117] Der Bus **146** verbindet die Speichersteuerung **136**, den Ergänzungsspeicher **138** und die Bildverarbeitungshardware **144** miteinander. In einer Ausführungsform ist der Bus zum Beispiel ein PCI-Bus. Der PCI-Standard ist gut bekannt, und mehrere Computersystemplatinen unterstützen diesen Standard. Computersysteme, die andere Busarchitekturen besitzen, können ebenfalls das Bildverarbeitungssystem unterstützen. Beispiele sind u.a. ISA-Bus, EISA-Bus, lokaler VESA-Bus und der NuBus.

[0118] Die Anzeigevorrichtung **142** ist eine Farbanzeige mit kontinuierlicher Auffrischung zur Anzeige eines Bildes. Die Anzeigevorrichtung ist in einer Ausführungsform eine Bildröhre (CRT), sie kann aber auch eine Flüssigkeitsanzeige (LCD) oder eine andere Form von Anzeigevorrichtung sein.

[0119] Die Ergänzungsspeichervorrichtung **138** kann eine Reihe von Speichermedien umfassen. Die Ergänzungsspeichervorrichtung kann zum Beispiel Disketten, Festplatten, Band, CD-ROM usw. und andere Vorrichtungen umfassen, die elektrisches, magnetisches, optisches oder anderes Aufzeichnungsmaterial verwenden.

[0120] Die Eingabevorrichtung(en) **140** kann eine Tastatur, Cursorpositioniervorrichtung, wie zum Beispiel eine Maus, Joysticks, sowie eine Reihe von handelsüblichen Eingabevorrichtungen umfassen.

[0121] In einer Ausführungsform, die unten detailliert beschrieben wird, wird die Bildverarbeitungshardware **144** auf der Leiterplatte implementiert, die sich mit dem Computersystem über den PCI-Bus verbindet. In einer alternativen Implementierung kann die Bildverarbeitungshardware auf einer Systemplatine zusammen mit einem Prozessor oder anderer Bildverarbeitungshardware und Speicher angeordnet sein. In einem Spielesystem zum Beispiel ist die Bildverarbeitungshardware normalerweise auf der Hauptplatine angeordnet. Analog kann die Bildverarbeitungshardware in einer Set-Top-Box ebenfalls auf der Hauptplatine angeordnet sein.

[0122] Obwohl wir die Architektur eines Computersystems umrissen haben, ist nicht beabsichtigt, unsere Erfindung auf die Systemarchitektur, die in Fig. 2 illustriert ist, zu begrenzen. Unser Bildverarbeitungssystem kann in Spielesystemen, Set-Top-Boxen, Videoeditiervorrichtungen usw. implementiert werden. Wir beschreiben unten eine Ausführungsform eines Bildverarbeitungssystems in der Umgebung der Systemarchitektur, die in Fig. 2 gezeigt wird. Wir beschreiben alternative Implementierungen in der ganzen folgenden Beschreibung, es ist jedoch nicht beabsichtigt, mit der Beschreibung von Alternativen eine vollständige Auflistung anderer möglicher Implementierungen zu liefern. Auf der Basis unserer detaillierten Beschreibung unten kann der Fachmann auf diesem Gebiet unser Bildverarbeitungssystem oder Erscheinungsformen desselben auf anderen Plattformen implementieren.

[0123] Fig. 3 ist ein Schema, das die Beziehung zwischen der Software und der Hardware in einer Ausführungsform illustriert. In dieser Ausführungsform wird das Bildverarbeitungssystem unter Verwendung von Verarbeitungsressourcen des Prozessors des Hostcomputers und der Bildverarbeitungshardware **144** implementiert. Die Bildverarbeitungshardware **144** wird auf einer Erweiterungsplatine **164** implementiert, die einen Prozessor (z.B. einen Digitalen Signalprozessor) **166** und Bildverarbeitungsschaltkreise **168** umfasst. Die Prozessoren des Hostcomputers **130** und der Bildverarbeitungsplatine **164** teilen sich die Bildverarbeitungsaufgaben. Unten umreißen wir allgemein die Funktionen, die vom Hostcomputer **130** und der Bildverarbeitungsplatine **174** ausgeführt werden.

[0124] Grafikunterstützungssoftware **160** wird auf dem Hostcomputersystem **130** ausgeführt und kommuniziert mit der Bildverarbeitungsplatine **164** über die Hardwareabstraktionsschicht (HAL) **162**. Die Bildverarbeitungsplatine **164** umfasst einen programmierbaren digitalen Signalprozessor **166**, der DSP genannt wird, und zusätzliche Bildverarbeitungshardware **168**, die unten im Detail beschrieben wird.

[0125] Die Grafikunterstützungssoftware **160** kann Funktionen zur Unterstützung des Speichermanagements, Betrachtungsvolumen-Culling, Tiefensortierung, Chunking sowie Gsprite-Zuweisung, Transformation und Detailgrad umfassen. Die Grafikunterstützungssoftware kann eine Bibliothek von Grafikfunktionen umfassen, die durch Grafikanwendungen zugänglich sind, um die Funktionen, die hier aufgelistet sind, auszuführen.

[0126] Die Grafikunterstützungssoftware **160** umfasst Funktionen, die das Gsprite-Paradigma, das oben eingeführt wird, unterstützen. Wie oben angegeben, werden Gsprites unabhängig gerendert und brauchen nicht auf jedem Teilbild gerendert zu werden. Statt dessen können Änderungen in der Position eines Gsprites mit affinen oder anderen Transformationen angenähert werden. Die Grafikunterstützungssoftware **160** stellt Funktionen bereit, die helfen, ein Objekt oder Objekte einem Gsprite zuzuweisen und die Bewegungsdaten zu verfolgen, die die Position und Bewegung des Gsprites beschreiben. Die Grafikunterstützungssoftware stellt auch Funktionen bereit, um festzustellen, wann ein gerenderter Gsprite aktualisiert werden muss. Die Notwendigkeit, einen Gsprite zu aktualisieren, kann je nach der Objektbewegung, Bewegung des Betrachtungspunktes, Beleuchtungsänderungen und Objektkollisionen variieren.

[0127] Wir stellen weitere Details bezüglich der Funktionen der Grafikunterstützungssoftware unten bereit. Die Bildverarbeitungsplatine **164** führt eine Geometrieverarbeitung auf niedriger Ebene aus, einschließlich Beleuchtung und Schattierung, Texturierung, Anti-Aliasing, Lichtdurchlässigkeit usw. In einer Ausführungsform ist der DSP **166** verantwortlich für die Front-End-Geometrieverarbeitungs- und Beleuchtungsberechnungen. Eine Reihe von diesen Funktionen kann auch durch den Prozessor **132** des Hosts ausgeführt werden.

Überblick über die Bildverarbeitungsplatine

[0128] Fig. 4A ist ein Schema, das die Bildverarbeitungsplatine **174** illustriert. Die Bildverarbeitungsplatine **174** kommuniziert mit dem Hostcomputer über den Bus **146**. Sie umfasst einen DSP **176**, Tiler **200**, gemeinsam genutzten Speicher **216**, die Gsprite-Engine **204**, Zusammensetzungspuffer **210** und einen Digital-Analog-Konverter (DAC) **212**. Der Bus **146** (Fig. 2) überträgt Befehle und Daten zwischen dem Wirt und dem DSP **176**. Als Reaktion auf Befehle vom Wirt rendert die Bildverarbeitungsplatine **174** Bilder und überträgt Anzeigebilder auf eine Anzeigevorrichtung **142** (Fig. 2) durch den DAC **212**.

[0129] In der Ausführungsform, die in den Fig. 2-Fig. 4A illustriert ist, teilen sich der Hostprozessor und der DSP die Funktionen des Bild-Präprozessors von Fig. 1. Der Bildprozessor umfasst Tiler **200**, Gsprite-Engine **204**, Zusammensetzungspuffer **210** und DAC **212**. Unten stellen wir mehr Details zu diesen Elementen bereit. Es sollte jedoch nicht vergessen werden, dass die Implementierung des Bildverarbeitungssystems variieren kann.

[0130] Der gemeinsam genutzte Speicher **202** speichert Bilddaten und Bildverarbeitungsbefehle auf der Bildverarbeitungsplatine **174**. In einer Ausführungsform wird der gemeinsam genutzte Speicher dazu verwendet, Gsprite- und Texturdaten in komprimierter Form, DSP-Code und -Daten zu speichern, und verschiedene Puffer dienen zur Übertragung von Daten zwischen verschiedenen Verarbeitungsteilsystemen.

[0131] Der DSP **176** ist für die Videokompression/-dekompression und die Grafikvorverarbeitung (Transformationen, Beleuchtung usw.) verantwortlich. Der DSP sollte vorzugsweise Gleitpunkt- und Ganzzahlrechnungen mit mehr als 1000 MFLOPS/MOPS unterstützen.

[0132] Der Tiler **200** ist ein VLSI-Chip, der die Scan-Conversion, Schattierung, Texturierung, das Entfernen

verborgener Flächen, Anti-Aliasing, Lichtdurchlässigkeit, Schattenabbildung und Mischung für das Mehrfachrendering ausführt. Die resultierenden gerenderten Gsprite-Chunks werden dann komprimiert und in komprimierter Form im gemeinsam genutzten Speicher abgelegt. Der Tiler führt zusätzlich die Dekompression und die Wiederkompression von Gsprite-Daten als Unterstützung der Video- und Fensteroperationen aus.

[0133] Die Gsprite-Engine **204** arbeitet bei Videoraten, um die Gsprite-Chunkdaten zu adressieren und zu dekomprimieren und die notwendige Bildverarbeitung für allgemeine affine Transformationen (die das Skalieren, Translation mit Subpixelgenauigkeit, Rotation, Reflektion und Shearing) auszuführen. Nach dem Filtern werden die resultierenden Pixel (mit Alpha) an die Zusammensetzungspuffer geschickt, wo die Anzeigepixeldaten berechnet werden.

[0134] Gsprite-Chunkdaten werden mit einer Geschwindigkeit von ein paar Scanreihen auf einmal zur Anzeige verarbeitet. In einer Ausführungsform werden die Chunkdaten mit jeweils 32 Scanreihen auf einmal verarbeitet. Der Zusammensetzungspuffer (**210**) umfasst zwei 32-Scanreihen-Farbpuffer, die zwischen Anzeige und Zusammensetzungsaktivität umgeschaltet werden. Der Zusammensetzungspuffer umfasst auch einen 32 Scanreihen-Alphapuffer, der zum Akkumulieren von Alpha für jedes Pixel verwendet wird.

[0135] Der DAC **212** umfasst einen RGB-Video-DAC und den entsprechenden Videoport **214** zu Videoeditiergeräten. Einzelne Komponenten können zum Implementieren der DAC-Funktionalität verwendet werden.

Systembetrieb

[0136] Die Fig. 5A und Fig. 5B sind Flussdiagramme, die Schritte beim Rendern eines Bildes im Bildverarbeitungssystem illustrieren. Bevor der Bildverarbeitungsprozessor **106** mit dem Rendern eines Bildes für den Betrachtungsraum beginnt, bestimmt der Bildpräprozessor **104** Objekt- und Betrachtungspunktorte (**240**). In der Ausführungsform, die in den Fig. 2 und Fig. 3 gezeigt wird, stellt die Grafikunterstützungssoftware **160**, die im Hostcomputersystem läuft, die Objekt- und Betrachtungspunktorte von Daten fest, die von einer Grafikanwendung bereitgestellt werden. Die Grafikanwendung, die auf dem Hostprozessor läuft, definiert Modelle, die die relevanten Objekte repräsentieren, und liefert eine Modelliertransformation, die zum Platzieren des Objektes mit den anderen Objekten in „Weltkoordinaten" verwendet wird.

[0137] Als Nächstes wählt der Bildpräprozessor **104** möglicherweise sichtbare Objekte (**242**) aus. Er stellt möglicherweise sichtbare Objekte auf der Basis des Betrachtungsvolumens fest. Das Betrachtungsvolumen ist ein dreidimensionaler Raum in Weltkoordinaten, der die Grenzen für eine Szene bereitstellt. Der Präprozessor wählt möglicherweise sichtbare Objekte durch Verschieben von Objekten und Bestimmen, ob ihre Grenzen das Betrachtungsvolumen schneiden. Objekte, die das Betrachtungsvolumen schneiden, sind im geometrischen oder räumlichen Sinn möglicherweise sichtbar.

[0138] In einigen Fällen ist es nützlich, „temporär" möglicherweise sichtbare Objekte außerhalb des aktuellen Betrachtungsvolumens festzustellen, um zukünftige Änderungen in der Szene zu berücksichtigen. Dies ermöglicht dem System, sich auf schnelle Änderungen im Betrachtungsvolumen einzustellen. In typischen 3D-Grafiksystemen besteht die einzige Möglichkeit, auf diese schnellen Änderungen zu reagieren, darin, eine vollkommen neue Szene zu erzeugen, die auf der geänderten Eingabe beruht, wobei eine beträchtliche Transportverzögerung dazwischengeschaltet ist. Solch eine lange Verzögerung hat negative Auswirkungen auf den Nutzer, was Probleme verursacht, wie zum Beispiel Überziehen und Übelkeit. Zur Verringerung dieser Verzögerung kann der Bildpräprozessor der vorliegenden Erfindung den Ort on Objekten berechnen, die in einem erweiterten Bereich außerhalb des sichtbaren Bereichs gelegen sind, und der Bildprozessor kann Bilder innerhalb dieses erweiterten Bereichs rendern und speichern. Unter Verwendung der Fähigkeit des Systems zur affinen Transformation kann die Eingabe des Betrachtungspunktes für ein nachfolgendes Teilbild dazu verwendet werden, die Gsprites aus diesem erweiterten Bereich neu zu anzuordnen und so die Systemtransportverzögerung auf weniger als 2 rechnerische Teilbilder zu reduzieren. Solch eine kurze Transportverzögerung ist mit aktuellen 3D-Grafikhardwaresystemen, die den Erfindern bekannt sind, nicht zu erreichen und ermöglicht Simulationen viel höherer Qualität mit viel stärkerem Vertieftsein des Nutzers.

[0139] Der Bildpräprozessor bestimmt die Konfiguration der Gsprites für das Bild (**244**). Dieser Schritt beinhaltet das Feststellen, wie man möglicherweise sichtbare Objekte auf Gsprites abbildet. Als Teil dieses Prozesses ordnet der Bildpräprozessor **104** Gsprites zu, was das Erzeugen einer Gspritedatenstruktur zum Speichern von Bilddaten umfasst, die einem oder mehreren möglicherweise sichtbaren Objekten entsprechen. Wenn es die Verarbeitungsressourcen ermöglichen, wird jedes nicht völlig durchdringende Objekt in der Szene einem unabhängigen Gsprite zugewiesen. Völlig durchdringende oder selbst verdeckende Objekte können als

einzelner Gsprite verarbeitet werden.

[0140] Der Bildpräprozessor **104** kann Gsprites aggregieren, wenn der Bildprozessor nicht die Leistungsfähigkeit besitzt, die Gsprites bei der gewünschten rechnerischen Teilbildrate zusammenzusetzen oder wenn der Systemspeicher unzureichend ist, um die Gsprites zu speichern. Das Rendern in getrennte Gsprites ist rechnerisch immer effizienter; wenn also das System den Speicher und die Leistungsfähigkeit zum Zusammensetzen besitzt, sollten sich nicht überschneidende Objekte in getrennte Gsprites gerendert werden. Wenn das System nicht in der Lage ist, zu speichern oder ein Anzeigebild basierend auf einer aktuellen Zuweisung von Gsprites zu erzeugen, können einige Gsprites aggregiert werden, um dieses Problem abzuschwächen.

[0141] Nachdem ein Objekt oder Objekte Gsprites zugewiesen wurden, unterteilt der Bildprozessor die Gsprites in Bildregionen, die „Chunks" (**248**) genannt werden. Der Bildpräprozessor durchläuft die Gsprites zyklisch und unterteilt die Gsprites in Chunks (**246**, **248**). In einer Ausführungsform umfasst dieser Prozess das Transformieren der Grenzvolumina von Objekten in den Betrachtungsraum und das Finden rechteckiger Bildregionen, die die transformierten Grenzvolumina einschließen. Diese Bildregionen definieren die Dimensionen des Gsprites hinsichtlich des zweidimensionalen Raums, in den das Objekt oder die Objekte des Gsprites gerendert werden. Der Gsprite wird in Chunks unterteilt, indem die rechteckige Bildregion in Chunks geteilt und diese Chunks mit der Gsprite-Datenstruktur verknüpft werden.

[0142] Als Optimierung kann das transformierte Grenzvolumen skaliert und/oder gedreht werden, so dass die Zahl der Chunks, die zum Rendern des Gsprites benötigt werden, minimiert wird. Wegen dieser zusätzlichen Transformation (Skalierung oder Drehung) ist der Raum, in den die Objekte, welche den Gsprites zugewiesen werden, gerendert werden, nicht unbedingt der Bildschirmraum. Dieser Raum wird als Gsprite-Raum bezeichnet. Beim Prozess der Erzeugung eines Anzeigebildes muss der Gsprite auf den Bildschirmraum zurücktransformiert werden.

[0143] Der nächste Schritt besteht darin festzustellen, wie man die Objektgeometrie unter den Chunks (**250**) aufteilt. Der Bildpräprozessor bestimmt, wie die geometrischen Grundelemente (z.B. Polygone) unter den Chunks aufgeteilt werden müssen, indem er die Polygone auf den 2D-Raum (**252**) transformiert und feststellt, in welchen Chunk oder Chunks sich die Polygone projizieren. Auf Grund der Kosten der Beschneidung von Polygonen ist der bevorzugte Ansatz, die Polygone, die am Rand eines Chunks liegen, nicht zu beschneiden. Statt dessen umfasst ein Chunk Polygone, die seine Kante überlappen. Wenn sich ein Polygon über die Grenze von zwei Chunks erstreckt, werden zum Beispiel bei diesem Ansatz die Eckpunkte in jedem Chunk eingeschlossen.

[0144] Der Bildpräprozessor stellt dann die Chunkdaten zum Tiling in die Schlange. Tiling bezeichnet den Prozess der Bestimmung von Pixelwerten, wie zum Beispiel Farbe und Alpha für Pixelorte, die von einem oder mehreren Polygonen bedeckt oder teilweise bedeckt werden.

[0145] Der Entscheidungsschritt (**254**) (Fig. 5B) und der Schritt (**256**), der darauf folgt, repräsentieren den Prozess des Tiling der Polygone innerhalb des Chunks. Während der Bildprozessor die Polygone eingeschlossen hat, die die Grenzen des aktuellen Chunks überlappen, erzeugt er nur Pixel, die innerhalb des Chunks liegen. Die erzeugten Pixel umfassen Informationen für das Anti-Aliasing (Fragmentdatensätzen), die gespeichert werden, bis alle Pixel erzeugt sind.

[0146] Nach der Beendigung des Tiling von Polygonen in einem Chunk löst der Bildprozessor die Anti-Aliasing-Daten (wie zum Beispiel Fragmentdatensätze) für die Pixel (**258**) auf. In einer Ausführungsform verwendet der Tiler **200** die Doppelpufferung, um einen vorherigen Chunk aufzulösen, während der nächste getilet wird. Alternativ kann der Tiler einen gewöhnlichen Puffer mit einer freien Liste verwenden. Die freie Liste stellt den freien Speicher im gewöhnlichen Puffer dar, der zugewiesen wird, wenn neue Fragmentdatensätze erzeugt werden, und hinzugefügt wird, wenn Fragmentdatensätze aufgelöst werden. Eine Kombination von Doppelpufferung und gemeinsamem Speicher kann ebenfalls verwendet werden.

[0147] Der Bildprozessor komprimiert den aufgelösten Chunk unter Verwendung einer Kompressionsmethode, die weiter unten beschrieben wird (**260**). Während der Bildprozessor einen Block von Pixeln auflöst, kann er einen anderen Block komprimieren. Der Bildprozessor speichert den komprimierten Chunk im gemeinsam genutzten Speicher (**262**).

[0148] Fig. 6 ist ein Flussdiagramm, das die Schritte illustriert, die zur Anzeige eines Bildes ausgeführt werden. Auf der Bildverarbeitungsplatine **174**, die oben beschrieben wird, werden Bilder aus dem gemeinsam ge-

nutzten Speicher **216** gelesen, in physische Ausgabevorrichtungskoordinaten durch die Gsprite-Engine **204** transformiert, im Zusammensetzungspuffer **210** zusammengesetzt, in den DAC **212** übertragen und dann auf eine Ausgabevorrichtung übertragen.

[0149] Während des Anzeigeprozesses greift der Bildprozessor auf eine Liste von Gsprites, die angezeigt werden sollen, für das aktuelle Teilbild zu. Im Prozess der Bestimmung der Gsprite-Konfiguration bestimmt der Bildpräprozessor die Tiefenordnung von Gsprites (**280**). Wie oben bemerkt, wird ein Objekt vorzugsweise einem Gsprite zugeordnet. Der Bildpräprozessor kann jedoch mehr als ein Objekt einem Gsprite zuordnen, um zum Beispiel Verarbeitungsbegrenzungen eines bestimmten Bildprozessors, der im System verwendet wird, zu berücksichtigen. Der Bildpräprozessor sortiert Objekte in Z-Reihenfolge, d.h. nach dem Abstand vom Betrachtungspunkt. Neben dem Sortieren der Objekte sortiert er auch Gsprites nach Tiefenordnung und speichert diese Tiefendaten in den Gsprite-Datenstrukturen.

[0150] Der Entscheidungsschritt (**282**) in Fig. 6 stellt einen zyklischen Durchlauf durch die Gsprites im Anzeigeprozess dar. Die Schritte innerhalb dieses zyklischen Durchlaufs können 1) das Berechnen einer Transformation für einen gerenderten Gsprite und 2) das Aufstellen einer Gsprite-Anzeigeliste umfassen, um zu steuern, wie die Gsprite angezeigt werden. Diese Schritte werden unten beschrieben.

[0151] Der Bildprozessor berechnet die Gsprite-Transformationen für Gsprites im möglicherweise sichtbaren Bereich. Eine Gsprite-Transformation bezeichnet eine Transformation an einem gerenderten 2D-Gsprite. In einer Ausführungsform kann der Bildprozessor eine Transformation an einem Gsprite ausführen, um den Rendering-Zusatzaufwand zu reduzieren. Statt jedes Objekt für jedes Teilbild zu rendern, reduziert der Bildprozessor den Rendering-Zusatzaufwand durch Wiederverwendung eines gerenderten Gsprites.

[0152] Es ist nicht notwendig, eine Gsprite-Transformation für jedes Teilbild der Bilddaten zu berechnen. Wenn zum Beispiel ein Gsprite für das aktuelle Teilbild der Bilddaten gerendert wird, braucht er möglicherweise nicht transformiert zu werden, es sei denn, dass zum Beispiel der Gsprite transformiert wurde, um besser zum begrenzenden Kasten für das Objekt zu passen.

[0153] Außerdem brauchen möglicherweise einige Gsprites nicht neu gerendert oder transformiert zu werden, weil das Objekt oder die Objekte, die ihnen zugeordnet wurden, sich nicht geändert haben oder sich nicht bewegen. Dementsprechend ist der Schritt zum Transformieren eines Gsprites optional.

[0154] Der Gsprite kann mit der Einheitsmatrix in Fällen multipliziert werden, wo sich die Position des Gsprites nicht geändert hat. Dies kann zum Beispiel in Fällen gelten, wo der Bildprozessor den Gsprite für das aktuelle Teilbild gerendert hat oder wo die Gsprite-Position sich nicht geändert hat, seit er ursprünglich gerendert wurde.

[0155] Um festzulegen, wie Gsprites angezeigt werden sollen, erzeugt der Bildprozessor eine Gsprite-Anzeigeliste. Die Anzeigeliste bezeichnet eine Liste oder Listen, die bestimmen, welche Gsprites auf dem Anzeigebildschirm angezeigt werden sollen. Dieses Konzept der Anzeigeliste kann auch für andere Ausgabevorrichtungen zum Präsentieren eines Teilbildes von Bilddaten gelten. Der Bildprozessor verwendet die Anzeigeliste beim Abbilden und Zusammensetzen gerenderter Gsprites auf die physischen Gerätekoordinaten. Obwohl der Schritt des Ausbauens einer Anzeigeliste als Teil eines zyklischen Durchlaufs durch Gsprites illustriert wird, ist es nicht notwendig, dass die Liste oder Listen speziell innerhalb dieses zyklischen Durchlaufs erzeugt wird.

[0156] Die Anzeigeliste kann eine Liste oder Listen oder eine Liste von Gsprites pro Band bezeichnen. Ein „Band" ist ein horizontaler Scanzeilenbereich eines Anzeigebildschirms. In einer Ausführungsform ist zum Beispiel ein Band 32 Scanzeilen hoch mal 1344 Pixel breit. Die Anzeigeliste kann eine getrennte Liste von Gsprites für jedes Band umfassen, in diesem Fall beschreiben die Bandlisten die Gsprites, die auf die jeweiligen Bänder entfallen. Alternativ kann die Anzeigeliste aus einer einzigen Liste bestehen, die durch Kennzeichnen der Gsprites implementiert wird, um festzustellen, auf welche Bänder die Gsprites fallen.

[0157] Die Anzeigeliste in der erläuterten Ausführungsform ist doppelgepuffert. Doppelpufferung ermöglicht es dem System, eine Anzeigeliste zu erzeugen, während es eine weitere liest. Während das System die Gsprite-Transformationen berechnet und die Anzeigeliste für ein Teilbild aufbaut, liest es die Anzeigeliste für ein anderes Teilbild und zeigt die Bilddaten in dieser Liste an.

[0158] Wegen der Doppelpufferung überlappen sich die Schritte, die in Fig. 6 gezeigt sind: Der Bildpräprozessor führt Schritte (**280**-**286**) für ein Teilbild aus, während der Bildprozessor Schritte (**290**-**298**) für ein ande-

res Teilbild ausführt.

[0159] Fig. 7 ist ein Schema, das den Zeitablauf dieser Schritte illustriert. Nachdem das System die Schritte (**280**-**286**) (Fig. 6) für ein Teilbild **310** vollendet hat, wartet es auf ein Teilbildsynchronisierungssignal (vertikaler Rücklauf) und führt dann den Puffertausch aus. Die Anzeigeliste, die es gerade erzeugt hat, wird dann zur Bestimmung der Gsprites verwendet, die im aktuellen Teilbild **312** angezeigt werden sollen. Während diese Anzeigeliste verarbeitet wird **312**, werden Gsprite-Transformationen berechnet und eine Anzeigeliste wird für das nächste Teilbild aufgebaut **314**. Im nächsten Teilbild werden dann die Gsprite-Transformationen und die Anzeigeliste, die im vorherigen Teilbild **314** erzeugt wurden, dazu verwendet, das Anzeigebild **316** zu erzeugen.

[0160] Der Bildprozessor konvertiert Gsprites auf Ausgabegerätekoordinaten auf der Basis der Liste der Gsprites in der Anzeigeliste. Der Bildprozessor liest Gsprite-Daten aus dem gemeinsamen Speicher, einschließlich Farbe, Alpha und Daten, die die Position des Gsprites kennzeichnen. Auf der Basis dieser Daten bestimmt der Bildprozessor die Farbe und Alpha für Pixel, die vom Gsprite bedeckt werden.

[0161] In einer Ausführungsform durchläuft der Bildprozessor zyklisch jedes Band, wobei er Gsprites transformiert, die gemäß der Gsprite-Anzeigeliste auf dieses Band fallen. Wir werden diesen Anzeigeprozess detaillierter unten beschreiben.

[0162] Nach dem Transformieren der Gsprite-Daten setzt der Bildprozessor die resultierenden Pixeldaten zusammen. Dies umfasst das Berechnen von Farbe und Alpha für Pixel in Ausgabegerätekoordinaten auf der Basis der Gsprite-Transformationen. Der Bildprozessor transformiert die Pixeldaten für Gsprites in der Anzeigeliste und setzt dann die transformierten Pixeldaten zusammen. Der Prozess beinhaltet das Bestimmen von Farbe und Alpha an einer Pixelstelle auf der Basis eines Beitrags von einem oder mehreren Pixelwerten von Gsprites, die diese Pixelstelle bedecken.

[0163] In einer Ausführungsform durchläuft der Bildprozessor zyklisch die Bänder und setzt Pixeldaten für jedes Band zusammen. Der Bildprozessor puffert Pixeldaten doppelt: Er transformiert und setzt Gsprite-Daten für ein Band in einem Puffer zusammen, während er zusammengesetzte Pixeldaten für ein anderes Band anzeigt.

[0164] Nach dem Zusammensetzen der Pixeldaten überträgt der Bildprozessor dann die zusammengesetzten Pixeldaten an eine Ausgabevorrichtung. Die typischste Ausgabevorrichtung, die in Verbindung mit diesem System verwendet wird, ist natürlich eine Anzeige. Zur Anzeige der Pixeldaten werden sie in ein Format umgewandelt, das mit der Anzeige verträglich ist.

[0165] Nachdem wir den Systembetrieb einer Ausführungsform beschrieben haben, stellen wir nun mehr Details bezüglich der Bildverarbeitungsplatine bereit.

Bildverarbeitungsplatine

[0166] In der einen Ausführungsform umfasst der gemeinsam genutzte Speicher **216** 4 MB RAM. Er wird unter Verwendung von zwei 8-Bit-RAM-Buskanälen implementiert. Die Menge und Art des Speichers kann jedoch variieren.

[0167] Fig. 8 ist ein Schema, das den DSP **336** auf der Bildverarbeitungsplatine **174** illustriert. Der DSP **336** ist für das Parsen des Befehlsstroms aus dem Hostprozessor und die Ausführung eines Teils der Videoverarbeitung und der Front-End-Geometrieverarbeitung verantwortlich Der DSP führt die Front-End-Geometrieverarbeitung und die Beleuchtungsberechnungen aus, die für die 3D-Grafik verwendet werden. Dies umfasst Modell- und Betrachtungstransformationen, Beschneiden und Beleuchtung. Teile des Gsprite-Animationsmanagement werden ebenfalls im DSP gehandhabt, wie zum Beispiel die Gsprite-Bewegungsextrapolation.

[0168] Rendering-Befehle werden in Hauptspeicherpuffern gespeichert und über den PCI-Bus und durch den PCI-Buscontroller **342** auf die Bildverarbeitungsplatine **174** geDMAt. Diese Befehle werden dann im gemeinsam genutzten Speicher **216** auf der Platine gepuffert, bis sie vom DSP **336** benötigt werden (Fig. 8).

[0169] Der DSP-Kern **338** umfasst einen Prozessor zur Ausführung der Bildverarbeitungsberechnungen, die oben beschrieben werden. Außerdem führt der DSP-Kern die Zeitplanung und das Ressourcenmanagement aus.

[0170] Die Speicherschnittstelle **340** unterstützt Hochgeschwindigkeitsdatenübertragungen, z.B. 64 Bit bei 80 MHz. Sie ist so gestaltet, dass sie eine Schnittstelle zu konventionellen DRAM- und SDRAM-Einrichtungen bildet. Der Tiler **200** ist so ausgelegt, dass er direkt an diesen Bus angeschlossen werden kann, womit die für den DSP erforderliche Speicherzeitsteuerung simuliert wird.

[0171] Der Datenformatierer und -konverter **346** im DSP formatiert die Rendering-Befehle für den Tiler. Dieser Block konvertiert Gleitkommafarbkomponenten in ganze Zahlen und packt sie in die Tiler-spezifischen Datenstrukturen. Er puffert auch einen kompletten Befehl und DMAt ihn direkt in einem Speicherpuffer in einem gemeinsam genutzten Speicher. Diese Rendering-Befehle werden später vom Tiler gelesen, wenn er zur Ausführung der Operationen bereit ist.

[0172] Unter seinen Formatierungsaufgaben formatiert der Datenformatierer und -konverter **346** Dreiecksbefehlsdaten für den Tiler. R G B α (Alpha)-Daten, die vom DSP (**336**) in Gleitpunkt berechnet werden, werden in 8-Bit-Ganzzahlen umgewandelt. Koordinateninformationen werden vom Gleitkomma in 12.4-Festkommazahlen umgewandelt. Die Daten werden in 64-Bit-Worte gepackt und in einem zusammenhängenden Block in den gemeinsamen Speicher übertragen, um die Bandbreite zu optimieren.

[0173] Die Anzeigespeichermanagementeinheit (MMU) **344** wird für den Desktop-Anzeigespeicher verwendet. Sie fängt PCI-Zugriffe in einem linearen Adressbereich ab, der als Desktop-Anzeigespeicher zugewiesen wird. Sie bildet dann diese Zugriffe auf Bildblöcke ab, die im gemeinsamen Speicher gespeichert sind.

[0174] Die Architektur der Bildverarbeitungsplatine (Fig. 4A, **174**) ist relativ unabhängig vom speziellen DSP. Jedoch sollte der DSP vorzugsweise eine beträchtliche Gleitkomma-Rechenleistung besitzen. Geeignete DSPs umfassen den MSP-1 von Samsung Semiconductor und TriMedia von Philips Semiconductor. Diese speziellen DSPs sind zwei Beispiele für DSPs, die eine ausreichende Gleitkomma-Rechenleistung bieten.

[0175] Fig. 9A ist ein Schema des Tilers **200** auf der Bildverarbeitungsplatine **174**. Der Tiler ist für 2D- und 3D-Grafikbeschleunigung und für die Steuerung des gemeinsamen Speichers verantwortlich. Wie im Schema der Bildverarbeitungsplatine gezeigt, schließt sich der Tiler direkt an den DSP (**176**, **Fig.** 4), die Gsprite-Engine **204** und das gemeinsame Speichersystem **216** an.

[0176] Die Funktionsblöcke, die im Schema oben gezeigt werden, werden in diesem Abschnitt beschrieben.

[0177] Der Tiler **378** umfasst eine Reihe von Komponenten zum Rendern von Grundelementen. Die Befehls- und Speichersteuerung **380** umfasst eine Schnittstelle zum gemeinsamen Speicher **216**, der Gsprite-Engine **204** und dem DSP **176**. Zugriffe auf den Speicher vom Tiler, DSP und der Gsprite-Engine wurden durch diesen Block arbitriert. Eine Schlange wird zum Puffern von Lesezugriffen bereitgestellt.

[0178] Der Setup-Block **382** berechnet die linearen Gleichungen, die die Kanten-, Farb- und Texturkoordinateninterpolation über die Fläche des Dreiecks bestimmen. Diese Gleichungen werden ebenfalls dazu verwendet, um festzustellen, welche Texturblöcke benötigt werden, um das Dreieck zu rendern. Die Kantengleichungen werden auch in den Scanumwandlungsblock **394** geschoben und in den Grundelementeregistern **396** gespeichert, bis sie von der Scanumwandlungs-Engine **398** benötigt werden.

[0179] Der Setup-Block **382** umfasst drei Komponenten: den Eckeneingabeprozessor **384**, die Ecken- und Kontrollregister **386** und die Setup-Engine **388**. Der Eckeneingabeprozessor **384** parst den Befehlsstrom vom DSP. Die Ecken- und Kontrollregister **386** speichern Informationen, die für die Verarbeitung von Polygonen und anderen geometrischen Grundelementen notwendig sind. Die Dreiecksverarbeitung wird in dieser speziellen Ausführungsform verwendet, und der Tiler **200** umfasst Register für sechs Ecken (drei für jedes Dreieck), was die Doppelpufferung der Dreiecksverarbeitung ermöglicht. Die Setup-Engine **388** berechnet die Differenzen für die Farb-, Tiefen-, Kanten- und Texturkoordinateninterpolation über die Fläche des Dreiecks. Diese Gleichungen werden ebenfalls dazu verwendet, festzustellen, welche Texturblöcke zum Rendern des Dreiecks verwendet werden. Die Setup-Engine ruft auch Texturchunks im voraus ab, so dass sie verfügbar sind, wenn sie von der Scanumwandlungs-Engine **398** benötigt werden.

[0180] Die Setup-Engine **388** kommuniziert auch mit der Texturleseschlange **390** und einem Texturadressgenerator **392**. Die Texturleseschlange **390** puffert Leseanforderungen für Texturblöcke aus gemeinsamem Speicher. Obwohl wir den Begriff „Textur" bei der Bezugnahme auf die Abschnitte des Tilers verwenden, die zum Abrufen von Bilddatenblöcken aus dem Speicher verwendet werden, versteht es sich, dass dieser Begriff sich auf Texturabbildungen, Schattenabbildungen und andere Bilddaten beziehen kann, die bei Mehrfachrendering-

operationen verwendet werden. Der Texturadressgenerator **392** bestimmt die Adresse im Speicher der angeforderten Chunks und sendet Texturleseanforderungen an die Befehls- und Speichersteuerung **380**. Der Texturadressgenerator **392** umfasst eine Speichermanagementeinheit, die das Schreiben von Bilddaten in den Texturcache steuert.

[0181] Der Scanumwandlungsblock **394** empfängt Differenzen und andere Eckpunktdaten vom Setup-Block und erzeugt Pixel-Daten. Der Scanumwandlungsblock **394** umfasst Grundelementeregister **396** und die Scanumwandlungs-Engine **398**. Die Grundelementeregister **396** speichern die Gleichungsparameter für jeden Dreieckparameter. Die Grundelementeregister umfassen Register zum Speichern mehrerer Sätze von Gleichungen, so dass die Scanumwandlungs-Engine beim Warten auf Texturdaten nicht stecken bleibt.

[0182] Die Scanumwandlungs-Engine **398** scannt und wandelt Polygone um, die in diesem Fall Dreiecke sind. Der Scanumwandlungsblock **394** umfasst die Interpolatoren für sich bewegende Kanten und die Bewertung von Farben, Tiefen usw. Die Pixeladresse, zusammen mit Farbe und Tiefe, und Anti-Aliasing-Bedeckungsinformationen werden an die Pixelengine zur Verarbeitung geschickt.

[0183] Die Scanumwandlungs-Engine **398** sendet Texturadressen an die Texturfilter-Engine **400**, die die Texturdaten berechnet. Die Texturfilter-Engine **400** berechnet Pixelfarbe und Alphadaten für Polygone, die gerade gerendert werden. Die illustrierte Texturfilter-Engine berechnet einen Filterkern, der auf der Z-Neigung und -Orientierung des Dreiecks, das gerendert wird, und auf der Mitte der Texturanforderung beruht (S- und T-Koordinaten eines Punktes, der in die Textur abgebildet wird). Das Filtern wird in zwei Durchläufen in einer Pipeline-Weise ausgeführt, so dass in jedem Zyklus ein neues Pixel erzeugt wird. Der Filterkern kann ein anisotropes Filter oder ein isotropes Filter sein. Wenn keine Anisotropie benötigt wird, kann der Filterkern negative Keulen verwenden, was viel schärfere Texturen ermöglicht, als mit dreiliniger Interpolation möglich ist. Die Texturfilter-Engine **400** handhabt auch Z-Vergleichsoperationen zum Berechnen der Effekte bei Schatten.

[0184] Der Texturcache **402** speichert Blöcke von dekomprimierten Bilddaten. In einer Implementierung speichert der Texturcache **402** Texturdaten für sechzehn 8×8-Pixelblöcke. Die Daten sind so organisiert, dass bei jedem Taktzyklus auf 16 Texturelemente zugegriffen werden kann.

[0185] Die Dekompressionsengine **404** dekomprimiert Texturdaten und überträgt sie zum Texturcache **402**. In dieser Ausführungsform umfasst die Dekompressionsengine zwei Dekompressoren, einen, der eine diskrete Kosinustransformation (DCT) implementiert, die auf dem Algorithmus für kontinuierliche Halbtonbilder, wie zum Beispiel Texturen, beruht, und den anderen, der einen verlustlosen Algorithmus für Desktop-Pixeldaten implementiert. Der DCT-basierende Algorithmus wird durch zwei parallele Dekompressionsblöcke implementiert, die jeweils acht Pixelelemente (d.h. zwei Pixel) pro Taktzyklus erzeugen können.

[0186] Der komprimierte Cache **416** kann zum Puffern von komprimierten Daten verwendet werden, bevor die Dekompressionsengine **404** sie dekomprimiert und sie in den Texturcache **402** überträgt.

[0187] Die Scanumwandlungs-Engine **398** überträgt Pixeldaten in die Pixelengine **406**. Die Pixelengine **406** führt Berechnungen auf Pixelniveau aus, einschließlich Mischen und Tiefenpufferung. Die Pixelengine handhabt auch die Z-Vergleichsoperationen, die für Schatten benötigt werden. Um eine optimale Leistung zu erhalten, sollte die Pixelengine vorzugsweise bei einem Pixel pro Takt arbeiten.

[0188] Die Pixelengine **406** steuert die Übertragungen von Pixeldaten an einen Rasterungspuffer. Der Rasterungspuffer umfasst in der erläuterten Ausführungsform Pixelpuffer **408** und Fragmentpuffer **410**. Die Pixelpuffer **408** umfassen zwei Puffer zur Unterstützung der Doppelpufferung. Bei dieser Implementierung der Pixelpuffer speichert jeder Pixeleintrag acht Bit pro Farbkomponente (R G B), acht Bit für die Alpha-Komponente, 24 Bit für den Z-Puffer, 8 Bit für den Schablonenpuffer und einen Neun-Bit-Zeiger im Fragmentpuffer. Das macht insgesamt 73 Bit pro Pixel. Ein Pixelpuffer wird von der Pixelengine **406** verwendet, während der andere durch die Anti-Aliasing-Engine **412** verwendet wird. Dann werden die Puffer getauscht.

[0189] Die Fragmentpuffer **410** speichern Fragmente für teilweise bedeckte Pixel, Pixelfragmente genannt, die aus Pixeln von Polygonen resultieren, deren Kanten ein gegebenes Pixel schneiden oder die lichtdurchlässig sind. Der Fragmentpuffer ist ein einzelner Puffer in der Implementierung, die in Fig. 9A gezeigt wird. Eine freie Liste von Fragmenten wird unterhalten, so dass beim Auflösen von Fragmenten diese zur freien Liste hinzugefügt werden und beim Erzeugen von Fragmenten diese Einträge aus der freien Liste verwenden. Alternativ könnte der Fragmentpuffer doppelt gepuffert sein, so dass ein Fragmentpuffer von der Anti-Aliasing-Engine aufgelöst werden könnte, während der andere parallel dazu von der Pixelengine gefüllt wird.

[0190] In einer Ausführungsform umfasst ein Fragmentdatensatz dieselben Daten wie in den Pixelpuffereinträgen plus eine 4×4-Maske. Der Neun-Bit-Zeiger wird dazu verwendet, eine verlinkte Liste von Einträgen mit einem reservierten Wert zu bilden, der das Ende der Liste anzeigt. In dieser Ausführungsform umfassen die Fragmentpuffer **410** insgesamt **512** Einträge, die Größe kann aber variieren.

[0191] Die Anti-Aliasing-Engine **412** berechnet die Farb- und Alphakomponente für Pixel, die von mehr als einem Polygon betroffen sind, was passiert, wenn Polygone nur teilweise die Pixelfläche abdecken (d.h. die Polygonkanten schneiden die Pixel) oder wenn Polygone Lichtdurchlässigkeit besitzen. Die Anti-Aliasing-Engine **412** überträgt aufgelöste Pixeldaten zur Kompressions-Engine **414**. In dieser Ausführungsform umfasst die Kompressions-Engine **414** zwei Kompressoren, einen DCT-basierten für Halbtonbilder und einen verlustlosen für Desktop-Pixeldaten. Der DCT-basierte Algorithmus wird unter Verwendung eines Kompressors implementiert, der acht Pixelelemente pro Taktzyklus komprimieren kann. Die Kompressions-Engine **414** komprimiert die sich ergebenden gerenderten Gsprites und sendet die komprimierten Daten an den Befehlsspeicher und -steuerung **380** zur Speicherung im gemeinsamen Speicher **216** (**Fig.** 4). Der Tiler besitzt ebenfalls einen komprimierten Cache **416** zum Cachen von komprimierten Daten.

[0192] Die Fig. 10 und Fig. 11 illustrieren zwei alternative Implementierungen für den Zugriff auf Bilddaten aus dem Speicher während des Pixelerzeugungsprozesses. Es gibt eine Reihe von Fällen, bei denen auf Bilddaten aus dem Speicher während der Pixelerzeugung zugegriffen werden muss. Dazu gehört zum Beispiel der Zugriff auf eine Texturabbildung während einer Texturabbildungsoperation, Zugriff auf eine Schattenabbildung während einer Schattierungsoperation und Zugriff auf Farb- und/oder Alphadaten während Mehrfachmischungsoperationen Der Einfachheit halber bezeichnen wir die Bilddaten im Speicher als „Texturen" oder „Texturdaten". Es sollte sich jedoch verstehen, dass die Verfahren und Systeme, die hier beschrieben werden, auch auf andere Arten von Bilddaten angewendet werden können, auf die vom Speicher während der Pixelerzeugung zugegriffen wird.

[0193] Die Implementierungen, die in den Fig. 10 und Fig. 11 illustriert werden, bieten alternative Herangehensweisen, um einen Texturcache auf dem Tiler effizient zu laden und zu nutzen. Ein wesentlicher Vorteil dieser Ansätze ist, dass Texturdaten in Speichern mit hoher Verzögerungszeit und selbst in einem komprimierten Format gespeichert werden können, ohne die Leistungsfähigkeit übermäßig zu beeinträchtigen. Im Ergebnis dessen kann weniger spezieller und preiswerterer Speicher verwendet werden, um Hochleistungs-Rendering-Hardware zu implementieren.

[0194] Auf Texturdaten aus dem Speicher wird zugegriffen, und diese werden in Einheiten gespeichert, die "Blöcke" genannt werden, welche normalerweise eine kleine rechteckige Region darstellen, die sich zum effizienten Abrufen und Cachen eignen. Eine typische Blockgröße hat eine Größe von etwa 8×8 Abfragewerten. Für Texturabbildungen ist ein typischer Block zum Beispiel 8×8 Texel groß.

[0195] Fig. 10 ist ein Funktionsschema, das eine Ausführungsform zum Zugriff auf diese Blöcke von Texturdaten illustriert. Diese Ausführungsform löst das Verzögerungszeitproblem durch Puffern von Pixeldaten aus dem Rastergenerator **417**, einschließlich der Texturdatenanforderungen, in einer Texturreferenzdatenschlange **418**. Die Schlange umfasst genügend Einträge, um die Verzögerungszeit aufzunehmen, die anderenfalls beim Zugriff (und möglicherweise Dekomprimieren) eines Texturblocks auftreten würde, so dass der Renderingprozess mit voller Geschwindigkeit ablaufen kann. Wenn zum Beispiel 100 Zyklen erforderlich sind, um einen Texturblock abzurufen, und der Tiler in der Lage ist, ein Pixel pro Takt zu erzeugen, dann umfasst die Texturreferenzdatenschlange mindestens 100 Einträge.

[0196] Der Datenfluss im System, der in Fig. 10 illustriert wird, läuft folgendermaßen ab. Zuerst werden die geometrischen Grundelemente zum Rastern eingerichtet, wie in Block **416** gezeigt. Die Einrichtungsverarbeitung umfasst zum Beispiel das Lesen der Eckpunkte für ein geometrisches Grundelement, wie zum Beispiel ein Dreieck, und die Berechnung von Differenzen für Farbe, Tiefe und Kanten über die Fläche des Dreiecks. Die Parameter, die sich aus diesen Berechnungen ergeben, werden dann in den Rastergenerator **417** eingespeist.

[0197] Der Rastergenerator **417** liest die Gleichungsparameterdaten für jedes Grundelement und erzeugt Pixeldaten. Der Rastergenerator erzeugt Pixeldaten, einschließlich Texturkoordinaten und Filterdaten, und puffert diese Daten in der Texturreferenzdatenschlange **418**. Der Texturabrufblock **420** liest die Texturreferenzdaten, die in der Schlange **418** gespeichert sind, und ruft die entsprechenden Texturblöcke aus dem Speicher **419** ab.

[0198] Die Pixeldaten, die in der Texturreferenzdatenschlange **418** in dieser Ausführungsform gespeichert sind, umfassen: eine Zieladresse für das Pixel (X, Y), das gerade berechnet wird; Tiefendaten (Z); eine Bedeckungsmaske; Farb- und Lichtdurchlässigkeitsdaten; die Koordinaten der Mitte für die Texturanforderung (S, T), und Texturfilterdaten. Die Tiefen- und Bedeckungsdaten werden in der Texturreferenzdatenschlange nur benötigt, wenn ein hochwertiges Anti-Aliasing von Pixeln gewünscht wird. Alternativ kann das Entfernen verborgener Flächen und das Anti-Aliasing im Rastergenerator **417** ausgeführt werden. Wenn das Entfernen verborgener Flächen und das Anti-Aliasing im Rastergenerator ausgeführt wird, brauchen Tiefendaten und Bedeckungsdaten nicht in der Datenschlange **418** gespeichert zu werden. Die Texturfilterdaten können zum Beispiel einen Parameter für die Detailliertheit für die MIP-Abbildung umfassen oder können anisotrope Filterdaten für eine höherwertige Texturfilterung umfassen.

[0199] Die Texturblockabrufung **420** liest die Texturreferenzdaten, die in der Datenschlange gespeichert sind, und ruft die entsprechenden Texturdaten aus dem Speicher **419** ab. Im Fall von Texturabbildungzugriffen konvertiert die Texturblockabrufeinheit die (S,T)-Mitte der Texturanforderung und die Texturfilterdaten in die Adressen der Blöcke, die benötigt werden, um die Texturfilteroperation ausführen zu können. Die Blöcke, die bei diesem Prozess festgestellt werden, werden dann in den Cache geholt, wobei sie andere Blöcke bei Bedarf ersetzen. Bilddatenblöcke können unter Verwendung eines Algorithmus zum Ersetzen eines zuletzt verwendeten (LRU) oder eines anderen geeigneten Cache-Ersetzungsalgorithmus abgerufen werden. Um Speicherzugriffe zu reduzieren, verfolgt die Texturblockabrufeinheit die Texturblöcke, die bereits im Texturcache **421** gespeichert sind, und vermeidet das mehr als einmalige Abrufen desselben Blocks. Diese Fähigkeit reduziert beträchtlich die Speicherbandbreite, die zur hochqualitativen Texturfilterung benötigt wird, weil die Verzögerungszeit beim Abrufen eines Texturblocks nur einmal beim Berechnen eines Bildes auftritt.

[0200] Die Texturblockabrufeinheit umfasst einen Sperrmechanismus, um das Überschreiben von Texturblöcken zu verhüten, die noch in der Texturfiltereinheit im Tiler benötigt werden. Eine Möglichkeit, einen solchen Sperrmechanismus zu implementieren, besteht darin, einen Referenzzählwert mit jedem Texturblock zu verknüpfen, um zu verfolgen, ob der Texturfilter einen bestimmten Texturblock verwendet hat. Dieser Referenzzählwert wird beim Erhalt einer Texturanforderung für einen Block durch die Texturabrufeinheit erhöht und als Reaktion auf seine Verwendung durch die Texturfiltereinheit verringert. Die Texturblockabrufeinheit ersetzt dann nur Blöcke, die einen entsprechenden Referenzzählwert von null besitzen.

[0201] Eine andere Möglichkeit, den Sperrmechanismus zu implementieren, ist das Zuweisen eines Puffers für die zeitweilige Speicherung der Texturblockausgabe durch die Texturabrufeinheit. Bei diesem Ansatz wird der Bildblock zuerst in den temporären Speicherpuffer geschrieben. Nachdem die Texturabrufeinheit das Schreiben des Bildblocks in den temporären Speicherpuffer beendet hat, kann er dann in den Texturcache übertragen werden. Bildblöcke werden in den Texturcache geswappt, wenn sie zum ersten Mal von der Texturfiltereinheit **422** benötigt werden.

[0202] Im Fall von Texturabbildungsoperationen liest der Texturfilterblock **422** Texturproben aus dem Texturcache **421** und die Pixeldaten, die in der Texturreferenzdatenschlange **418** gespeichert sind, und berechnet Pixelfarb- und möglicherweise Alphawerte aus den Texturprobedaten.

[0203] Zusätzlich zu den Texturabbildungsoperationen kann dieser Ansatz auch auf das Schattierungs- und Mehrfachmischoperationen angewendet werden. Zum Beispiel kann die Texturreferenzdatenschlange dazu verwendet werden, eine Schattentiefenabbildung abzurufen, die im Speicher liegt. Alternativ kann die Texturreferenzdatenschlange dazu genutzt werden, Farb- und/oder Alphadaten abzurufen, die in Mehrfachbeleuchtungs- und Schattierungsoperationen verwendet werden. Weitere Details bezüglich Texturabbildungs-, Schattierungs- und Mehrfachoperationen werden unten angeführt.

[0204] Es gibt eine Reihe von Vorteilen für das Puffern von Pixeldaten in der oben beschriebenen Weise. Ein wesentlicher Vorteil ist, dass Bilddaten in weniger speziellem Speicher (mit höherer Zugriffszeit) gespeichert werden können, was die Kosten für das Gesamtsystem reduziert. Außerdem können Bilddaten, die Texturen enthalten, in komprimiertem Format gespeichert werden, und es kann darauf immer noch mit ausreichend schnellen Raten zugegriffen werden, um anspruchsvolle Pixeloperationen, wie die Texturfilterung, auszuführen. Im Ergebnis ist das System in der Lage, eine verbesserte Leistungsfähigkeit bei niedrigeren Kosten im Vergleich zu bekannten Verfahren für den Zugriff auf Texturdaten zu erreichen.

[0205] Ein weiterer Vorteil für diesen Ansatz ist, dass die Texturreferenzdatenschlange genau vorhersagen kann, auf welche Bildblöcke aus dem Speicher zugegriffen werden muss. Im Ergebnis dessen erfährt das System eine Verzögerungszeit für Speicherzugriffe, die nicht größer als notwendig ist. Sobald die Bilddatenblöcke

im Texturcache sind, kann die Texturfiltereinheit mit der vollen Geschwindigkeit des Rastergenerators laufen, solange es ausreichend Speicherbandbreite und Texturabrufdurchsatz gibt, um die angeforderten Bildblöcke in den Texturcache zu schreiben.

[0206] Mit dem Einstellen von Texturreferenzen in die Schlange mit der Texturanforderungsmitte und dem Filtern der Daten kann die Schlange viel kleiner sein, als wenn die Texel mit ihren entsprechenden Texturfiltergewichten in die Schlange gestellt werden.

[0207] Fig. 11 ist ein Funktionsschema, das eine alternative Ausführungsform für den Zugriff auf Bilddaten aus dem Speicher illustriert. Bei diesem Ansatz werden geometrische Grundelemente in die Schlange gestellt und dann in einem Vorrastergenerator verarbeitet, um die Verzögerungszeit des Texturblockabrufs während des Pixelerzeugungsprozesses zu verbergen. Mit einem Beispiel lässt sich das Konzept besser illustrieren. Wenn ein durchschnittliches Grundelement 25 Zyklen zum Rastern benötigt und es 100 Zyklen dauert, einen Texturblock aus dem Speicher abzurufen, sollte die Grundelementeschlange mindestens vier Grundelemente lang sein. Eine vereinfachte Version des Postrastergenerators, des Vorrastergenerators umfasst Schaltkreise zur Bestimmung der Bilddatenblöcke, auf die im Speicher zugegriffen werden muss. Sobald die Texturdaten abgerufen sind, kann der Nachrastergenerator Pixeldaten unter Verwendung von Texturdaten erzeugen, ohne der Verzögerungszeit ausgesetzt zu sein, die beim Abruf von Blöcken aus dem Speicher auftritt.

[0208] Der Datenfluss durch diese Implementierung verläuft folgendermaßen. Wie bei der Implementierung, die oben beschrieben wird, werden geometrische Grundelemente in einem Setup-Block **425** zur Rasterung verarbeitet. Bei dieser speziellen Implementierung umfasst der Setup-Block **425** jedoch eine größere Grundelementeschlange, um mehr Grundelemente zu puffern. Der Vorrastergenerator **426** konvertiert die Grundelement schnell in der Reihenfolge, wie die Blöcke vom Nachrastergenerator **427** benötigt werden, in eine Liste von Texturblöcken, die benötigt werden, um die Texturfilterungsbedürfnisse für alle Pixel zu erfüllen, die vom Grundelement abgedeckt werden. Der Vorrastergenerator ist eine vereinfachte Version des Nachrastergenerator **427** oder des Rastergenerators **417** in der alternativen Implementierung. Bei diesem Ansatz braucht der Vorrastergenerator nur Texturdatenadressen zu berechnen und Texturanforderungen zu bestimmen.

[0209] Der Vorrastergenerator enthält auch ein Modell des Texturblockcaches und führt den Cacheersetzungsalgorithmus, wie zum Beispiel zuletzt verwendet (LRU) aus, um das Überschreiten der Größe des Texturblockcaches zu vermeiden. Als Teil des Cacheersetzungsalgorithmus komprimiert der Vorrastergenerator wiederholte Anforderungen an einen einzelnen Texturblock auf nur eine Anforderung an die Texturblockabrufeinheit **429**.

[0210] Die Texturblockabrufschlange **428** umfasst Einträge zum Speichern von Texturblockanforderungen. Die Texturblockabrufeinheit **429** liest die Texturabrufe aus der Texturblockabrufschlange und ruft die entsprechenden Blöcke aus dem Speicher **430** ab.

[0211] Der Nachrastergenerator rastert Grundelemente, die sich im Setup-Block **425** in der Schlange befinden, um Pixeldaten für einen Pixelort zu erzeugen. Wenn auf Bilddaten aus dem Speicher während des Pixelerzeugungsprozesses zugegriffen werden muss, rastert der Nachrastergenerator die Grundelemente so schnell wie die notwendigen Texturblöcke in den Texturblockcache **431** übertragen werden können. Wenn der Nachrastergenerator das Rastern eines Grundelementes, das sich im Setup-Block in der Schlange befindet, abschließt, wird das Grundelement entfernt und durch eine weiteres Grundelement aus dem Eingabedatenstrom ersetzt. Der Setup-Block ist dafür verantwortlich, dass die Schlange mit Grundelementen gefüllt gehalten wird, so dass der Vorrastergenerator und der Nachrastergenerator nicht im Pixelerzeugungsprozess stehen bleiben.

[0212] Wie die alternative Ausführungsform, die oben beschrieben wird, sollte die Texturblockabrufung vorzugsweise einen Sperrmechanismus umfassen, um das Überschreiben der Texturblöcke zu verhüten, die noch vom Nachrastergenerator benötigt werden. Die zwei Sperrmechanismen, die oben beschrieben sind, können auch in dieser Implementierung verwendet werden. Speziell kann ein Referenzzählwert dazu verwendet werden zu verfolgen, wann ein Bildblock angefordert und dann verwendet wurde. In diesem Fall würde ein Referenzkonto beim Erhalt einer Texturanforderung für einen Block durch den Vorrastergenerator erhöht und bei Verwendung durch den Nachrastergenerator verringert werden. Die Texturblockabrufeinheit ersetzt dann nur Blöcke im Texturcache, wenn ihr entsprechender Referenzzählwert null ist.

[0213] Alternativ kann ein Puffer zur temporären Speicherung von Texturblockausgaben durch den Texturabrufblock zugewiesen werden. Wenn der Texturabrufblock das Schreiben eines Blocks in diesen temporären

Puffer abgeschlossen hat, kann er dann in den Texturblockcache **431** übertragen werden, wenn er vom Nachrastergenerator **427** angefordert wird. Wenn der Nachrastergenerator **427** zuerst Daten in einem Texturblock im temporären Puffer anfordert, wird der Block dann in den Texturblockcache **431** übertragen.

[0214] Es gibt eine Reihe von Vorteilen für diese Vorgehensweise. Erstens können Texturdaten in weniger speziellem Speicher gespeichert werden, und darauf kann trotzdem mit Raten zugegriffen werden, die benötigt werden, um eine anspruchsvolle Texturfilterung zu unterstützen. Ein wichtiger, damit verwandter Vorteil ist, dass Texturdaten in einem komprimierten Format gespeichert und dann zur Verwendung im Pixelerzeugungsprozess dekomprimiert werden können.

[0215] Ein weiterer Vorteil dieses Ansatzes ist, dass Anforderungen an Speicher vorhergesagt werden können, so dass die Verzögerungszeit für den Speicherzugriff nur einmal für jeden Texturblock zum Rendern einer Szene auftritt. Sobald die ersten Texturblöcke im Texturcache sind, kann der Nachrastergenerator mit voller Geschwindigkeit laufen, solange es ausreichend Speicherbandbreite und Texturabrufdurchsatz gibt, um den Cache aktuell zu halten.

[0216] Fig. 9B illustriert eine detailliertere Implementierung des Systems, das in Fig. 10 illustriert wird. Der Setup-Block **381** in Fig. 9B entspricht dem Setup-Block **416** in Fig. 10. Im Gegensatz zum Setup-Block **382** von Fig. 9A, erzeugt der Setup-Block **381** in dieser alternativen Implementierung keine Texturleseanforderungen. Statt dessen erzeugt der Scanumwandlungsblock **395** Pixeldaten, einschließlich Texturreferenzdaten, die in der Texturreferenzdatenschlange **399** gespeichert werden.

[0217] Der Scanumwandlungsblock **395** von Fig. 9B ist eine spezielle Implementierung des Rastergenerators **417** in Fig. 10. Er berechnet einen Z-Wert, eine Bedeckungsmaske, Farb- und Lichtdurchlässigkeitswerte und die Mitte der Texturanforderung in Texturkoordinaten. Für einige Texturabbildungoperationen berechnet er auch Detailliertheitsdaten oder anisotrope Filterdaten. Die Texturfilter-Engine **401** liest die Texturanforderung und möglicherweise Texturfilterdaten, die in der Texturreferenzdatenschlange **399** gepuffert sind, und greift auf die entsprechenden Texturproben im Texturcache zu. Aus diesen Texturdaten berechnet die Texturfilter-Engine den Beitrag der Textur zu den Pixelfarb- und -Alphawerten. Die Texturfilter-Engine kombiniert Farbe und Alpha in der Texturreferenzdatenschlange **399** mit dem Beitrag aus der Textur, um Pixelwerte zu erzeugen, die an die Pixelengine **406** gesendet werden.

[0218] Die Texturcachesteuerung **391**, Texturleseschlange **393**, Befehls- und Speichersteuerung **380** sind spezielle Implementierungen des Texturblockabrufs **420** in Fig. 10. Außerdem sind für komprimierte Texturblöcke der komprimierte Cache **416** und die Dekompressionsengine **404** auch Teil des Texturblockabrufs **420**.

[0219] Fig. 9C illustriert eine detailliertere Implementierung des Systems, das in Fig. 11 illustriert wird. In dieser Implementierung wird die Funktionalität, die in Verbindung mit den Blöcken **425** und **426** von Fig. 11 beschrieben wird, innerhalb des Setup-Blocks **383** implementiert. Speziell umfasst der Setup-Block **383** den Vorrastergenerator **426**. Der Setup-Block **383** umfasst auch zusätzliche Eckpunktkontrollregister **387** zum Puffern zusätzlicher Grundelemente, so dass der Vorrastergenerator die Grundelemente schnell konvertieren kann, um Texturdatenanforderungen in Gang zu setzen. Die Setup-Engine und der Vorrastergenerator **383** senden Anforderungen für Texturblöcke an die Texturcachesteuerung **391**, die in Fig. 9C gezeigt wird.

[0220] Die Texturcachesteuerung **391** stellt sicher, dass die benötigten Texturblöcke sich im Texturcache **402** befinden, wenn sie benötigt werden. Die Texturleseschlange puffert Leseanforderungen für Texturdatenblöcke im gemeinsamen Speichersystem. Die Befehls- und Speichersteuerung **380** arbitriert den Zugriff auf das gemeinsame Speichersystem, und sie umfasst einen Puffer zur Pufferung von Daten aus dem Speicher. Die Texturcachesteuerung **391**, Texturleseschlange **393** und die Befehls- und Speichersteuerung **380** sind spezielle Implementierungen des Texturblockabrufs **429** in Fig. 11. Für komprimierte Texturblöcke sind der komprimierte Cache **416** und die Dekompressionsengine **404** auch Teil des Texturblockabrufs **429**. Die Texturcachesteuerung **391** leitet den Strom von Texturblöcken aus dem komprimierten Cache **416** durch die Dekompressionsengine **404** in den Texturcache **402**.

[0221] Der Scanumwandlungsblock **397** und die Texturfilter-Engine **403** sind eine spezielle Implementierung des Nachrastergenerators **427** in Fig. 11. Der Scanumwandlungsblock **397** und die Texturfilter-Engine **403** arbeiten ähnlich wie ihre Gegenstücke, die in Fig. 9A illustriert und oben beschrieben werden.

Texturcachesteuerung

[0222] Oben haben wir zwei Ansätze zum Rastern in Umgebungen mit hoher Verzögerungszeit für Texturabrufoperationen beschrieben. Wir geben nun Erscheinungsformen der Texturcachesteuerung detaillierter an.

[0223] Die Texturcachesteuerungsmethode ermöglicht es dem Rastergenerator, während der Texturabbildung trotz einer hohen Verzögerungszeit für Texturabbildungsabrufoperationen mit voller Geschwindigkeit zu arbeiten. Im Tiler ist diese Verzögerungszeit das Ergebnis der Zeit, die zum Lesen nicht komprimierter Texturdaten aus dem gemeinsamen Speicher (z.B. RAMBUS) benötigt wird, plus der Zeit, die zum Dekomprimieren von Blöcken der Texturabbildung benötigt wird. Das Verfahren gilt auch für die Gsprite-Engine, die Gsprite-Blöcke aus dem gemeinsamen Speicher abruft, sie möglicherweise dekomprimiert und Pixeldaten im Gsprite-Raum in den Betrachtungsraum (oder spezieller in Bildschirmkoordinaten) konvertiert.

[0224] Die Grundvoraussetzung für das Texturcachesteuerungsverfahren ist die Erzeugung zweier identischer Ströme von Texel-(oder Gsprite-Pixel)-Anforderungen, die zeitlich versetzt sind. Der erste (frühere) Strom ist eine Vorabrufanforderung, für die keine Texturdaten zurückgegeben werden, während der zweite (spätere) Strom eine tatsächliche Anforderung ist, die wirklich Texeldaten zurückliefert. Die Zeitdifferenz zwischen diesen zwei Strömen wird dazu verwendet, die Verzögerungszeit des Lesens und Dekomprimierens der Texturdaten zu verbergen.

[0225] Zwei Ansätze zum Erzeugen dieser zeitlich getrennten Anforderungen, die oben beschrieben werden, sind: (1) Doppelte Rastergeneratoren, die beide von einem einzigen Grundelemente-FIFO (**Fig. 11** und **Fig. 9C**) lesen, und (2) ein einziger Rastergenerator, gefolgt von einem Pixel-FIFO (**Fig. 10** und **Fig. 9B**).

[0226] Bei Ansatz (1) schaut der erste Rastergenerator sich die Grundelemente aus Positionen an der oder in der Nähe der Eingabeseite des Grundelemente-FIFOs an und rastert die Grundelemente, wobei Texturanforderungen gemacht werden, aber keine Texel zurückgeliefert werden und keine Pixel erzeugt werden. Der zweite Rastergenerator entfernt Grundelemente aus dem FIFO-Ausgang und erstellt zu einem späteren Zeitpunkt identische Anforderungen, empfängt die Texel aus der Texturcachesteuerung und erzeugt die Pixel. Die Tiefe der Grundelementeschlange, kombiniert mit der Zahl der Pixel pro Grundelement, bestimmt die potenzielle zeitliche Differenz zwischen den zwei Anforderungsströmen.

[0227] Bei Ansatz (2) verarbeitet der einzige Rastergenerator Grundelemente und erstellt Texturanforderungen und gibt teilweise vollständige Pixeldaten in einen Pixel-FIFO aus. Diese Teilpixeldaten umfassen alle Daten, die notwendig sind, um die Berechnung des Pixels abzuschließen, sobald die Texturanforderungen erfüllt wurden. Auf der Ausgabeseite des Pixel-FIFO wird das Teilpixel abgeschlossen, was den identischen Strom von Texturanforderungen erzeugt, die Texel empfängt und fertig gestellte Pixel erzeugt. Die Tiefe der Pixelschlange bestimmt die potenzielle zeitliche Differenz zwischen den zwei Anforderungsströmen.

Texturcachesteuerung

[0228] Die Texturcachesteuerung umfaßt zwei konzeptionelle Caches: den virtuellen Cache und den physischen Cache. Der virtuelle Cache ist mit dem ersten (Vorabruf-)-Anforderungsstrom verknüpft und umfaßt keine Daten, die direkt die Cache-Einträge begleiten (Anforderungen an diesen Cache geben keine Daten zurück). Der physische Cache ist mit dem zweiten (tatsächlichen) Anforderungsstrom verknüpft und umfaßt echte Texturdaten, die jeden Cache-Eintrag begleiten (und gibt daher Daten an den Anfordernden zurück). Diese Caches haben dieselbe Zahl von Einträgen.

[0229] Der virtuelle Cache steuert und verfolgt den zukünftigen Inhalt des physischen Caches; daher hat er an einer beliebigen Position in seinem Anforderungsstrom einen Satz von Cache-Key- und Eintragsverknüpfungen, die der physische Cache an derselben relativen Position in seinem Anforderungsstrom (zu einem zukünftigen Zeitpunkt) haben wird.

[0230] Beim Erhalt einer Anforderung (eines neuen ‚Keys') führt der virtuelle Cache den Vergleich mit seinem aktuellen Satz von Keys aus. Wenn der angeforderte Key nicht im virtuellen Cache ist, dann wird eine Cache-Ersetzungsoperation ausgeführt. Die virtuelle Cache-Ersetzung umfasst 1) Auswählen eines Eintrags zum Auswechseln (über LRU oder einen anderen Algorithmus), 2) Ersetzen des Keys für diesen Eintrag und 3) Aufrufen des (Speicher- und) Dekompressionsteilsystems, um mit dem Prozess des Abrufens und Dekomprimierens der Daten zu beginnen, die mit diesem Key verbunden sind. In den speziellen Implementierungen, die in den **Fig. 9B** und **Fig. 9C** gezeigt werden, umfasst das Dekompressionsteilsystem die Befehls- und Speicher-

steuerung **380**, den komprimierten Cache **416** und die Dekompressionsengine **404**.

[0231] Die Ausgabe des Dekompressionsteilsystems ist ein Block von Texturdaten, der dann in einen Eintrag im physischen Cache (den Texturcache **402** zum Beispiel) gebracht wird. Im Tiler, der in den Fig. 9B und C gezeigt wird, wird die Verarbeitung, die vom Dekompressionsteilsystem ausgeführt wird, in einer Mehreintrags-Pipeline realisiert, in der eine Reihenanordnung eingehalten wird.

[0232] Man beachte, dass bei Vorhandensein des angeforderten Keys im virtuellen Cache keine Aktion erforderlich ist, weil sich die damit verknüpften Daten zu dem Zeitpunkt, zudem sie vom zweiten Anforderungsstrom angefordert werden, im physischen Cache befinden.

[0233] Anforderungen an den physischen Cache führen zu einem ähnlichen Key-Vergleich, um festzustellen, ob die angeforderten Daten bereits im Cache sind. Wenn ein passender Key gefunden wird, werden die damit verknüpften Daten zurückgegeben. Wenn nichts Passendes gefunden wird, dann ist garantiert, dass die nächste Datenausgabe durch das Dekompressionsteilsystem die gewünschten Daten enthält. Man beachte, dass der physische Cache keine Auswahlverarbeitung von Ersetzungseinträgen ausführt – der Eintrag im physischen Cache, der durch die neuen Daten ersetzt wird, wird durch den virtuellen Cache über einen Cache-Eintragszielindex diktiert, der durch den virtuellen Cachecontroller berechnet wird und durch das Dekompressionsteilsystem mit den gewünschten Daten geschickt wird.

[0234] Das korrekte Funktionieren des Verfahrens erfordert, dass die Strömungskontrolle auf die Schnittstelle zwischen dem Dekompressionsteilsystem und dem physischen Cache angewendet wird. Wenn dekomprimierte Daten ihren Zieleintrag im physischen Cache überschreiben dürfen, sobald sie verfügbar sind, ist es möglich, dass möglicherweise nicht alle Referenzen auf den vorherigen Inhalt dieses Cache-Eintrags vervollständigt sind. (Man beachte, dass der Controller des physischen Cashes möglicherweise ebenfalls auf Daten warten muss, die vom Dekompressionsteilsystem auszugeben sind.) Diese Flusssteuerung wird durch Warten erreicht, bis der neue Eintrag angefordert wird, bevor der Inhalt des vorherigen Eintrags überschrieben wird. Das Eintragen neuer Daten in den Texturcache wird also immer bis zum letzten Moment, wenn sie benötigt werden, verzögert.

[0235] Da diese Ersetzung verzögert wird, bis sie erforderlich ist, kann jede Zeit, die benötigt wird, um die Daten in den physischen Cache zu bringen, eine Verzögerungszeit in den Prozess einführen, der den zweiten Anforderungsstrom antreibt. Die zwei Verfahren zur Abschwächung dieser Verzögerungszeit sind folgende.

[0236] Das erste Verfahren besteht darin, die Daten im physischen Cache doppelt zu puffern. Das ermöglicht es dem Dekompressionsteilsystem, die Daten jedes Eintrags sofort auf seine Seite des Doppelpuffers zu schreiben, und der Controller des physischen Caches kann einen (vermutlich schnellen) Puffertausch vornehmen, um die Daten auf seiner Seite des Caches abzubilden. Das Dekompressionsteilsystem braucht nur zu warten, ob der zu füllende Eintrag bereits voll ist und noch nicht getauscht wurde. Man beachte, dass der Cache-Ersetzungsalgorithmus, der vom Controller des virtuellen Caches verwendet wird, nicht die Neigung besitzt, denselben Eintrag wiederholt zu überschreiben und so das Schreiben auf die Cache-Einträge auszudehnen.

[0237] Das zweite Verfahren bestehet darin, für den physischen Cache ein oder mehrere ‚Extra'-Einträge zusätzlich zu der Zahl der „keyed" bzw. verschlüsselten Einträge vorzusehen. Die Zahl der ‚keyed'-Einträge ist die Zahl, für die Cache-Keys existieren, und entspricht der Zahl der Einträge im virtuellen Cache. Die Zahl der Extraeinträge stellt die Zahl der Einträge dar, die nicht abgebildet sind (d.h. die aktuell nicht verschlüsselt sind). Die Summe dieser ist die Gesamtzahl von Dateneinträgen im physischen Cache.

[0238] Im zweiten Verfahren können alle Cache-Einträge zwischen nicht abgebildet und abgebildet (mit einem Key verknüpft) wechseln. Die Menge der nicht abgebildeten Einträge bildet einen FIFO von Einträgen, in den das Dekompressionsteilsystem fertig gestellte Blöcke von Daten schreibt. Eine separate FIFO-Struktur wird für die Hauptzielindizes geführt, die mit diesen nicht abgebildeten Einträgen verknüpft sind. Wenn eine Anforderung an den physischen Cache erstellt wird, für die kein passender Key vorhanden ist, wird der erste Eintrag in der Schlange der nicht abgebildeten Einträge in den Zielindex abgebildet und mit diesem Key verknüpft. Der ersetzte Eintrag wird nicht abgebildet und (leer) an das Ende der nicht abgebildeten Schlange gesetzt.

Cache-Key-Erzeugung

[0239] Die Grundvoraussetzung des Verfahrens ist, dass zwei identische Ströme von Anforderungen erzeugt werden. Es ist jedoch nicht erforderlich, dass die speziellen Keys, die mit diesen Anforderungen verknüpft werden, identisch sind.

[0240] Die Cache-Keys, die den ersten (frühen) Strom von Anforderungen bilden, werden zur Steuerung des Lesens und der nachfolgenden Dekompression der Texturdaten verwendet. Diese Keys müssen eine gewisse direkte Relevanz für die angeforderten Daten besitzen (wie zum Beispiel eine Speicheradresse).

[0241] Die Cache-Keys, die den zweiten (späteren) Strom von Anforderungen bilden, brauchen nicht genau dem Inhalt des ersten Stroms zu entsprechen – es ist nur eine Forderung, dass es eine eindeutige 1:1-Abbildung zwischen den beiden gibt. Dies ist auf die Tatsache zurückzuführen, dass die Keys für den zweiten Strom nur zum Abstimmen vorhandener Cache-Einträge verwendet werden, nicht für Datenabrufoperationen. Der kritische Fakt ist hier, dass die Verknüpfung zwischen dem Key des physischen Caches und einem Cache-Eintrag hergestellt wird, wenn die neuen Daten auf den physischen Cache abgebildet werden und der Index des zugehörigen Eintrags vom virtuellen Cache berechnet und durch das Dekompressionsteilsystem geschickt wird.

[0242] Diese Tatsache kann ausgenutzt werden, um die Steuerelemente für den Prozess zu vereinfachen, der gerade die Keys für den zweiten Anforderungsstrom erzeugt, da die Keys für den Strom nur eindeutig und nicht ganz ‚korrekt' sein müssen.

[0243] Fig. 12A ist ein Schema, das die Gsprite-Engine **436** auf der Bildverarbeitungsplatine **174** illustriert. Die Gsprite-Engine **436** ist für das Erzeugen der Grafikausgabe aus einer Sammlung von Gsprites verantwortlich. Sie hat eine Schnittstelle zur Tiler-Speicherschnittstelleneinheit für den Zugriff auf die Gsprite-Datenstrukturen im gemeinsamen Speicher. Gsprites werden von der Gsprite-Engine transformiert (gedreht, skaliert usw.) und zum Zusammensetzungspuffer weitergeleitet, wo sie mit Pixeln zusammengesetzt werden, die von anderen Gsprites bedeckt werden.

[0244] Die Schnittstellensteuerung **438** wird zum Verbinden der Gsprite-Engine mit dem gemeinsamen Speichersystem über den Tiler verwendet. Dieser Block umfasst einen FIFO, um Zugriffe vom Speicher zu puffern, bevor sie durch die Gsprite-Engine verteilt werden.

[0245] Der Anzeigesteuerprozessor **440** wird zur Steuerung der Videoanzeigeaktualisierungen verwendet. Er umfasst einen Videotaktgenerator, der die Videoanzeigeauffrischung steuert und die Taktsignale erzeugt, die zur Steuerung der Gsprite-Zugriffe notwendig sind. Dieser Block durchläuft auch die Gsprite-Anzeigedatenstrukturen zur Feststellung, welche Gsprites für ein gegebenes 32-Scanzeilen-Band gelesen werden sollen.

[0246] Die Gsprite-Header **442**-Register speichern Gsprite-Headerdaten, die vom Bildprozessoradressgenerator **454** und der Gsprite-Filterengine **456** dazu verwendet werden, die Transformationen an jedem Gsprite festzustellen. Sie werden auch vom Gsprite-Headerdecoder **444** zur Bestimmung der Blöcke (in diesem Fall der 8×8-Kompressionsblöcke) verwendet, die benötigt werden, um den Gsprite in jedem Band zu rendern.

[0247] Der Gsprite-Headerdecoder **444** bestimmt, welche Blöcke aus jedem Gsprite im 32-Scanzeilen-Band sichtbar sind, und erzeugt Blockleseanforderungen, die zur Gsprite-Leseschlange **446** übertragen werden. Dieser Block beschneidet auch den Gsprite unter Verwendung der Gsprite-Kantengleichungsparameter auf das aktuelle Band. Dieser Prozess wird detaillierter unten beschrieben.

[0248] Die Gsprite-Leseschlange **446** puffert Leseanforderungen für Gsprite-Blöcke. Diese Schlange speichert in dieser Ausführungsform Anforderungen für sechzehn Blöcke

[0249] Der Gspritedaten-Adressgenerator bestimmt die Adresse im Speicher der angeforderten Gsprite-Blöcke und sendet Gsprite-Leseanforderungen an den Schnittstellensteuerungsblock. Der Gspritedaten-Adressgenerator **448** umfasst eine Speichermanagementeinheit.

[0250] Komprimierte Daten, die aus dem gemeinsamen Speicher **216** (Fig. 4A) abgerufen werden, können temporär im komprimierten Cache **458** gespeichert werden.

[0251] Die Dekompressionsengine **450** umfasst zwei Dekompressoren, einen, der einen DCT-basierten Algorithmus für Halbtonbilder, wie zum Beispiel 3D-Gsprites, implementiert, und den anderen, der einen verlust-

losen Algorithmus für Desktop-Pixeldaten implementiert. Der DCT-basierte Algorithmus wird durch zwei parallele Dekompressionsblöcke implementiert, die jeweils acht Pixelelemente (d.h. zwei Pixel) pro Taktzyklus erzeugen können.

[0252] Der Gsprite-Cache **452** speichert dekomprimierte Gsprite-Daten (R G B α) für sechzehn 8×8-Blöcke. Die Daten sind so organisiert, dass bei jedem Taktzyklus auf 16 Gspritepixel zugegriffen werden kann.

[0253] Der Bildprozessoradressgenerator **454** wird dazu verwendet, jeden Gsprite auf der Basis der festgelegten affinen Transformation abzuscannen und die Filterparameter für jedes Pixel zu berechnen. Gsprite-Cacheadressen werden erzeugt, um auf Gsprite-Daten im Gsprite-Cache **452** zuzugreifen und sie in die Gsprite-Filterengine zuzuführen. Der Bildprozessoradressgenerator **454** steuert auch den Zusammensetzungspuffer.

[0254] Die Gsprite-Filterengine **456** berechnet die Pixelfarbe und Alphadaten für Pixelorte auf der Basis der Filterparameter. Diese Daten werden in die Zusammensetzungspuffer zum Zusammensetzen übertragen. Dieser Block **456** berechnet einen 4- oder 16-Pixel-Filterkern auf der Basis der Gsprite-s- und -t-Koordinaten an einem Pixelort. Das Filter kann zum Beispiel entweder bilinear oder eine anspruchsvollere Kosinussummenfunktion sein. Der 16-Pixel-Filterkern kann negative Keulen haben, was eine viel schärfere Filterung ermöglicht, als mit bilinearer Interpolation möglich ist. Die Gsprite-Filterengine **456** erzeugt vier neue Pixel, die bei jedem Taktzyklus zusammengesetzt werden. Diese Pixel sind in einem Muster von zwei mal zwei ausgerichtet.

[0255] Die Gsprite-Engine **436** ist mit dem Tiler **200** und dem Zusammensetzungspuffer **210** verbunden. Steuersignale steuern die Videozeitabläufe und den Datentransfer zum DAC **212**.

[0256] Fig. 12B ist ein Schema einer alternativen Implementierung der Gsprite-Engine **437**. Diese spezielle Implementierung umfasst sowohl einen Vorrastergenerator **449** als auch den Rastergenerator **454**, so dass die Gsprite-Engine Gsprite-Pixeldaten aus dem Gsprite-Raum in den Bildschirmraum konvertieren kann, ohne eine Verzögerungszeit beim Abrufen und Dekomprimieren von Blöcken der Gsprite-Pixeldaten zu erfahren. Der Ansatz mit zwei Rastergeneratoren, der in dieser Implementierung verwendet wird, wird oben in Verbindung mit Fig. 11 und Fig. 9C beschrieben.

[0257] Der Betrieb der Blöcke in der Gsprite-Engine **437** ist im allgemeinen so wie oben für Fig. 12A beschrieben, außer dass diese Implementierung das Doppel-Rastergenerator-Verfahren zum Abrufen von Blöcken von Texturdaten verwendet. Bei dieser Implementierung (Fig. 12B) liest der Gsprite-Headerdecoder **444** das Gsprite-Headerregister **442**, beschneidet den Gsprite auf das aktuelle Anzeigeband und setzt den Gsprite zum Rastern in die Gsprite-Schlange **447**. Der Datenadressgenerator oder „Vorrastergenerator" **449** scannt jeden Gsprite auf der Basis der festgelegten affinen Transformation im Gsprite-Header und erzeugt Leseanforderungen für die Gsprite-Cachesteuerung **451**. Unter Verwendung eines Verfahrens, das oben in Verbindung mit der Texturcachesteuerung beschrieben wird, stellt die Gsprite-Cachesteuerung **451** sicher, dass die erforderlichen Gsprite-Datenblöcke in der Gsprite-Engine **437** und speziell im Gsprite-Cache **452** sind, wenn der Bildprozessorblock **455** sie benötigt. Sie steuert den Strom von Gspritedatenblöcken aus dem komprimierten Cache **458** durch die Dekompressions-Engine **450** in den Gspritecache **452**. Die Leseschlange **453** puffert Anforderungen für Gsprite-Datenblöcke zum gemeinsamen Speichersystem, und die Schnittstellensteuerung **438** liest die Anforderungen in der Leseschlange **453**, steuert Zugriffe auf den gemeinsamen Speicher und setzt Blöcke von Gsprite-Daten in den komprimierten Cache **458**.

[0258] Das Dekompressionsteilsystem in der Gsprite-Engine umfasst den komprimierten Cache **458** und die Dekompressions-Engine **450**. Die Cachesteuerung **451** steuert den Strom von Gsprite-Blöcken durch das Dekompressionsteilsystem, wie oben in Verbindung mit der Texturcachesteuerung beschrieben.

[0259] Der Bildprozessoradressgenerator (Rastergenerator) **454** scannt jeden Gsprite auf der Basis der festgelegten affinen Transformation im Gsprite-Header und berechnet die Filterparameter für jedes Pixel. Er erzeugt auch Gsprite-Cacheadressen von Gsprite-Daten, die er an eine Cache-Adressenkarte im Gsprite-Cache zur Verwendung durch die Gsprite-Filterengine **456** sendet. In einer speziellen Implementierung des Caches wählt die Cache-Adressenkarte aus, welche 14 Pixelblöcke aktiv sind und welche zwei Blöcke aus der Dekompressions-Engine gefüllt werden.

[0260] Die Gsprite-Filterengine **456** bildet Farb- und Alphawerte an Pixelorten im Gsprite-Raum auf den Bildschirmraum ab. Bei dieser Implementierung wendet sie entweder einen 2×2- oder einen 4×4-Filterkern an, um Pixelwerte (Farbe oder Farbe und Alpha) an Pixelorten im Bildschirmraum zu berechnen. Die Steuerung **457**

des Zusammensetzungspuffers sendet Pixelwerte, in diesem Fall vier Pixel pro Taktzyklus, an den Zusammensetzungspuffer. Die Steuerung **457** des Zusammensetzungspuffers überwacht die Bereitschaftsleitung vom Zusammensetzungspuffer, um sicherzustellen, dass die Gsprite-Engine **437** nicht den Zusammensetzungspuffer überlaufen lässt. Der Rastergenerator **454** steuert die Steuerung des Zusammensetzungspuffers.

[0261] Fig. 13 ist ein Schema, das den Zusammensetzungspuffer **480** auf der Bildverarbeitungsplatine **174** illustriert. Der Zusammensetzungspuffer **480** ist eine spezielle Speichervorrichtung, die zum Zusammensetzen der Gsprite-Daten aus der Gsprite-Engine und zur Erzeugung digitaler Videodaten zur Übertragung zum DAC **212** verwendet wird. Der Zusammensetzungspuffer arbeitet jeweils mit 32 Scanzeilen auf einmal – wobei er Gsprites für ein 32-Scanzeilenband zusammensetzt, während gleichzeitig die vorherigen 32 Scanzeilen angezeigt werden.

[0262] Die Zusammensetzungslogik **482** ist für die Berechnung der Pixelwerte verantwortlich, wenn sie in den Scanzeilenpuffer geschrieben werden. Dies wird durch Ausführen einer Mischoperation zwischen dem Pixelwert, der aktuell im Scanzeilenpuffer gespeichert ist, und dem, der gerade in den Zusammensetzungspuffer geschrieben wird, erreicht. Diese Operation wird detaillierter unten beschrieben. In einer Implementierung führt die Zusammensetzungslogik vier parallele Pixeloperationen pro Taktzyklus aus.

[0263] Die Speichersteuerung **484** wird zur Steuerung der Adresse und des zyklischen Durchlaufens der Speicherbänke verwendet. Adresseninformationen werden in einem Zeilenspaltenformat wie bei normalen DRAMS weitergeleitet.

[0264] Die Alphapuffer **486** umfassen einen Achtbitwert für jedes von 1344×32 Pixel. Der Speicher ist so organisiert, dass bei jedem Taktzyklus vier zusammenhängende Pixel gelesen und geschrieben werden können. Der Alphapuffer besitzt auch einen schnellen Löschmechanismus, um den Puffer schnell beim Umschalten zwischen den 32-Scanzeilenbändern zu löschen.

[0265] Es werden zwei unabhängige Scanzeilenpuffer **488** bereitgestellt. Die Scanzeilenpuffer umfassen drei Achtbitfarbwerte für jedes von 1344×32 Pixeln. Der Speicher ist so organisiert, dass bei jedem Taktzyklus vier zusammenhängende Pixel gelesen und geschrieben werden können. Ein Puffer wird dazu verwendet, die Pixeldaten für ein Band zum DAC zu übertragen, während der andere zum Zusammensetzen der Pixel für das nächste Band dient. Sobald das Band abgeschlossen ist, wechseln ihre Funktionen.

[0266] Ein Multiplexer dient dazu, Daten aus einem der zwei Scanzeilenpuffer **488** auszuwählen und die Pixelanzeigedaten an den DAC zu senden. Der Multiplexer schaltet nach jeweils 32 Scanzeilen zwischen den Puffern um.

[0267] Der Zusammensetzungspuffer **480** ist mit der Gsprite-Engine **204** verbunden und überträgt Bilddaten zum DAC **212**.

[0268] Fig. 14 ist ein Schema, das den DAC **514** auf der Bildverarbeitungsplatine **174** illustriert. Der DAC **514** implementiert die Grundfunktionen, die bei den meisten RAMDACs, die heute auf dem Markt sind, vorhanden sind. Der DAC umfasst Logik zum Lesen und Schreiben interner Steuerbefehlsregister und zur Pipelineverarbeitung der Videosteuersignale. Weitere Funktionsblöcke werden unten beschrieben.

[0269] Der Pixeldatenroutingblock **516** wird zur Steuerung der Weiterleitung von Pixeldaten aus den Zusammensetzungspuffern verwendet. Im normalen Betriebsmodus werden diese Daten mit Pixelraten zu den Farb-LUTs **518** für jeden der drei Kanäle weitergeleitet. Dieser Block ermöglicht auch, dass die Daten für diagnostische Zwecke zum DSP zurückgelesen werden.

[0270] Der Stereobildsplitter **520** unterstützt zwei getrennte Videosignale für die stereoskopische Anzeige unter Verwendung eines kopfmontierten Anzeigesystems. Bei diesem Modus werden zwei Videokanäle (**522**, **524**) vom Zusammensetzungspuffer verschachtelt und müssen vom DAC **514** getrennt werden. Der Stereobildsplitter **520** führt diese Funktion am DAC **514** aus. Im normalen Einkanalmodus werden die LUT-Daten direkt zu den Primären DACs weitergeleitet.

[0271] Alternativ kann der DAC **514** so ausgelegt werden, dass er eine einzelne Videoausgabe erzeugt. Bei einer einzelnen Videoausgabe kann der DAC eine stereoskopische Anzeige unter Verwendung eines linienverschachtelten Formats erzeugen, wo auf eine Scanzeile für ein Auge die Scanzeile für das andere Auge folgt. Der resultierende Videostrom hat ein Format, wie zum Beispiel 640×960, das zwei Bildern von 640×480

entspricht.

[0272] Der Taktimpulsgenerator **526** wird zum Erzeugen der Video- und Audiotaktimpulse verwendet. Diese Taktimpulse werden durch zwei phasensynchronisierte Taktimpulsgeneratoren erzeugt, um die Synchronisationsdrift zu beseitigen. Der Taktimpulsgenerator kann auch mit einem Steuersignal aus dem Medienkanal fremdsynchronisiert werden, was ermöglicht, die Bildverarbeitungsplatine mit einer externen Synchronisationsquelle zu synchronisieren.

[0273] Nachdem wir die Struktur und den Betrieb des Bildverarbeitungssystems oben beschrieben haben, beschreiben wir nun verschiedene Komponenten und Merkmale des Systems mit mehr Details. Wir beginnen mit einer Einführung in die Datenstrukturen, die im System zur Implementierung der Konzepte, die oben vorgestellt wurden, verwendet werden können.

Chunking

[0274] Im Gegensatz zu herkömmlichen Grafiksystemen, die einen großen Bildpufferspeicher und Z-Puffer in RAM verwenden, um Farb-, Tiefen- und andere Informationen für jedes Pixel zu speichern, unterteilt unser System Objekte in einer Szene in Bildregionen, die „Chunks" genannt werden, und rendert Objektgeometrien getrennt in diese Chunks. In einer Ausführungsform werden die Objekte in Gsprites gerendert. Die Gsprites werden in Chunks unterteilt, und die Chunks werden getrennt gerendert. Obwohl unsere Beschreibung sich auf mehrere spezielle Ausführungsformen bezieht, versteht es sich, dass das Chunking auf vielerlei Weise angewendet werden kann, ohne den Geltungsbereich der Erfindung zu verlassen.

[0275] Mit ein paar Beispielen lässt sich das Konzept des Chunking besser illustrieren. Wie in Fig. 15A gezeigt, wird ein Objekt **546** in einer Grafikszene von einem Kasten umschlossen, der als Begrenzungskasten **548** bezeichnet wird. Wendet man sich **Fig.** 15H zu, so kann ein Objekt **550** in der Grafikszene, das durch einen Begrenzungskasten eingeschlossen ist, in eine Bildregion gerendert werden, die ein Gsprite **552** genannt wird. Der Begrenzungskasten kann gedreht, skaliert, gedehnt oder auf andere Weise transformiert (z.B. affin transformiert) werden, um einen Gsprite im Bildschirmraum zu erzeugen. Nachdem der Begrenzungskasten erzeugt wurde, wird der Begrenzungskasten sowohl in X- als auch in Y-Richtung um das Objekt herum gedehnt, so dass er ein ganzzahliges Vielfaches der 32-Pixel-Chunkgröße wird, wenn der Begrenzungskasten nicht auf eine 32-Pixel-Grenze (d.h. die Chunkgrenze) **554** fällt. Wie aus dem Objekt **550** in Fig. 15B zu erkennen ist, wird ein Begrenzungskasten, der um das Objekt **546**, in Fig. 15A gezeigt, gezeichnet wurde, auf die 32×32-Pixelgrenzen in Fig. 15B erweitert. Der Gsprite wird dann vor dem Rendern in 32×32-Pixel"chunks" **556** unterteilt. Es könnten jedoch auch andere kleinere oder größere Chunkgrößen und alternativ geformte Chunks verwendet werden. Rechteckige und die bevorzugten quadratischen Chunks werden jedoch erläutert.

[0276] Wie in Fig. 15C gezeigt, enthält eine Grafikszene **558** eine Reihe von überlappenden Objekten (**560**, **562**). Diese Objekte sind in Begrenzungskästen eingeschlossen und werden Gsprites (**564**, **566**) zugewiesen. Die Begrenzungskästen, die in Fig. 15C gezeigt werden, sind bereits auf das Mehrfache von 32 Pixeln gedehnt (und gedreht, skaliert und anderweitig transformiert) worden, so dass 32×32-Chunks **568** erzeugt werden können. Wie man jedoch aus Fig. 15C erkennen kann, richten sich die Gsprites und ihre entsprechenden 32×32-Pixel-Chunkgrenzen **570** normalerweise nicht exakt auf die 32-Pixel-Bildschirmgrenzen **572** aus. Daher sind zusätzliche Gsprite-Manipulationen während des Chunkings erforderlich, so dass der Gsprite in den Bildschirmraum verschoben werden kann.

[0277] Ein Ansatz zum Erzeugen von Gsprites, die unter Verwendung des Chunking gerendert werden, besteht darin, eine Reihe von Objekten zu kombinieren, um einen größeren zusammengesetzten Gsprite zu erzeugen, statt eine Reihe von kleineren individuellen Gsprites zu erzeugen und zu rendern, die die geometrische Elemente der einzelnen Objekte enthalten. Diese Kombination von Gsprites erspart Verarbeitungszeit während des Renderns und ist oft erwünscht, wenn die kombinierten Objekte sich nicht sehr oft innerhalb einer Grafikszene ändern. Ein weiterer Ansatz zur Erzeugung von Gsprites ist, sich Komponenten eines Objekts mit komplexen Geometrien vorzunehmen und dann diese komplexen Geometriekomponenten in eine Reihe von Gsprites aufzuteilen. Diese Aufteilung kann zusätzliche Verarbeitungszeit erfordern, wird aber verwendet, um die Ausgabeauflösung eines bestimmten komplexen Objekts, das sich häufig ändert, zu verbessern. An einigen Objekten kann auch eine Kombination dieser Verfahren verwendet werden.

[0278] Man betrachte zum Beispiel eine Person in einem Videospiel, deren Arme mit einer Reihe von Stacheln unterschiedlicher Größe bedeckt ist und deren Arme sich häufig bewegen. Der Körper und Kopf und andere Teile der Person können kombiniert werden und einen größeren zusammengesetzten Gsprite bilden, da

diese Teile des Objektes sich nicht häufig ändern. Die Arme der Person jedoch, die mit Stacheln besetzt sind und komplexe Geometrien darstellen und sich häufig ändern, werden in eine Reihe von Gsprites unterteilt, um die Ausgabeauflösung zu verbessern. In diesem Fall wird sowohl die Kombination als auch die Unterteilung verwendet. Da es nicht einfach oder praktisch ist, solch eine Person zu zeichnen, wird zum Zweck der Erläuterung ein viel einfacheres Objekt, eine „Kaffeetasse" statt dessen verwendet, um die Kombination und die Unterteilung zu illustrieren.

[0279] Fig. 16A zeigt eine „Kaffeetasse". Diese „Kaffeetasse" ist tatsächlich aus eine Reihe von separaten Objekten zusammengesetzt. Man kann zum Beispiel die „Kaffeetasse" als aus einem Tassenbehälter, einem Tassengriff, einer Untertasse und Dämpfen, die aus der Tasse kommen, bestehend betrachten. Ein Ansatz wäre, diese einzelnen Objekte zu einem großen Gsprite (d.h. zu einer „Kaffeetasse") zu kombinieren, wie in Fig. 16A gezeigt. Ein weiterer Ansatz wäre, die „Kaffeetasse" in eine Reihe von kleineren Objekten (z.B. Tassenbehälter, Tassengriff, Untertasse und Dämpfe) zu unterteilen und kleinere einzelne Gsprites zu erzeugen, wie in Fig. 16B gezeigt. Fig. 16B erläutert auch, wie ein Objekt mit komplexen Geometrien unterteilt werden kann.

[0280] Wenn man die „Kaffeetasse" **574** als ein einfaches Objekt behandelt, wie in Fig. 16A gezeigt, können die einzelnen Komponenten (z.B. Tassenbehälter, Tassengriff, Untertasse, Dämpfe) des Objekts kombiniert werden, um einen großen Gsprite zu erzeugen. In diesem Fall würde ein Begrenzungskasten **576** um das Objekt gezeichnet werden, um das Objekt in den Bildschirmraum zu transformieren und einen großen Gsprite zu erzeugen. Der Begrenzungskasten kann gedreht, skaliert, gedehnt oder auf andere Weise manipuliert werden, um einen Gsprite zu erzeugen, der auf die 32×32-Pixelgrenzen im Bildschirmraum fällt. Der Gsprite wird dann in eine Reihe von 32×32-Pixelchunks **578** unterteilt.

[0281] Eine Möglichkeit, einen Gsprite in Chunks zu teilen, ist, durch die ganze Geometrie, die in den Objekten vorhanden ist, zu laufen und die geometrischen Elemente in Chunks zu legen. Ein weiterer Ansatz durchläuft die Chunks und zeichnet dabei alle Geometrien auf, die den betreffenden Chunk berühren. Die illustrierte Ausführungsform verwendet den zweiten Ansatz, jedoch kann der erste Ansatz und können weitere Ansätze ebenfalls verwendet werden. Wie aus Fig. 16A zu ersehen ist, sind einige Chunks leer (d.h. werden nicht von Objektgeometrien berührt). Diese Chunks können beim Rendern ignoriert werden, wie unten erklärt wird.

[0282] Wenn man nun die „Kaffeetasse" als komplexes Objekt behandelt, so wird das Objekt in kleinere Objektkomponenten unterteilt, die verarbeitet werden, um eine Reihe von kleineren Gsprites zu erzeugen, wie in Fig. 16B gezeigt. Zum Beispiel umfasst das Objekt „Kaffeetasse" die Teilobjekte Tassenbehälter ohne den Griff **579**, den Tassengriff **580**, die Untertasse **581** und die Dämpfe **582**. Jedes dieser Teilobjekte wird von Begrenzungskästen, die durch **583**-**586** gezeigt werden, eingeschlossen, um vier einzelne Gsprites zu erzeugen. Die „Kaffeetasse", die die vier einzelnen Gsprites umfasst, wird ebenfalls von einem einschließenden Begrenzungskasten eingeschlossen, wie durch **587** gezeigt. Jeder der Begrenzungskästen kann gedreht, skaliert, gedehnt oder auf andere Weise transformiert (z.B. affin transformiert) werden, um einen Gsprite zu erzeugen, der auf die 32×32-Pixelgrenzen im Bildschirmraum fällt. Jeder einzelne Gsprite wird dann in eine Reihe von 32×32-Pixelchunks unterteilt. Der umschließende Begrenzungskasten **587** wird dann in Chunks geteilt und enthält Bereiche von leeren Chunks **588**, die beim Rendern ignoriert werden. Die Chunks des umschließenden Begrenzungskastens werden jedoch in Fig. 16B nicht gezeigt.

[0283] Als Ergebnis des Chunkings wird das Grafikbild nicht als einzelnes Teilbild gerendert, sondern wird als Folge von Chunks gerendert, die später zu einem Teilbild oder Betrachtungsraum zusammengeführt werden. Nur Objekte innerhalb eines einzelnen Gsprites, die den 32×32-Pixelchunk des Bildes, das gerade gezeichnet wird, schneiden, werden gerendert. Das Chunking ermöglicht es, dass der Bildpufferspeicher und der Z-Puffer von geringer physischer Größe im Speicher sind (d.h. deutlich weniger Speicher als in den traditionellen Grafiksystemen belegen, die oben beschrieben werden) und einen hohen Ausnutzungsgrad des Speichers, der belegt ist, sowie eine steigende Speicherbandbreite erreichen. Die geringe Chunkgröße ermöglicht auch die Verwendung anspruchsvollerer Renderingverfahren, Verfahren, die bei großen Bildpufferspeichern und Z-Puffern nicht effektiv angewendet werden könnten.

[0284] Das Rendern von Chunks wird im Tiler ausgeführt. Das Rendern könnte jedoch auch in anderen Hardwarekomponenten oder unter Verwendung von Software ausgeführt werden. VLSI-Speicher auf dem Tiler-Chip wird zum Speichern der kleinen Chunks (32×32 Pixel) des Teilbildes, das gerade gerendert wird, verwendet. Der chipintegrierte VLSI-Speicher ist viel schneller und hat eine viel größere Speicherbandbreite als ein externer RAM. Wegen des Chunkingprozesses wird jedoch eine große Menge an Speicher zum Speichern des gesamten Bildpufferspeichers und des Z-Puffers für den Renderingprozess nicht mehr benötigt. Der inter-

ne Speicher innerhalb des Tilers wird nur zur Verarbeitung des aktuellen Chunks verwendet und wird dann für jeden nachfolgenden Chunk, der verarbeitet wird, immer wieder neu verwendet. Im Ergebnis dessen wird der verfügbare interne Speicher während des Grafikrenderns gut ausgenutzt.

[0285] Die Verwendung von internem VLSI-Speicher beseitigt auch Pinansteuerungsverzögerungen, die normalerweise auf Grund der chipexternen Kommunikation sowie des zusätzlichen Aufwandes, der mit der Ausführung von LESE- und SCHREIB-Operationen auf großem externem Speicher verbunden ist, welcher für konventionellen Bildpufferspeicher und Z-Puffer benötigt wird, auftreten. Außerdem ermöglicht die kleine Chunkgröße anspruchsvollere Anti-Aliasing- (z.B. Fragmentpuffer) und Texturierungsmethoden, die am Chunk auszuführen sind, als an einem großen Bildpufferspeicher und Z-Puffer, der in einer großen Menge an externem Speicher untergebracht ist, ausgeführt werden könnte, weil ein ganzer 32×32-Pixelchunk vollständig in der erläuterten Ausführungsform gerendert werden kann, bevor der nächste Chunk berechnet wird. Die geringe Chunkgröße bietet sich auch für Bildkompressionsverfahren an, die detailliert unten beschrieben werden.

[0286] Nachdem alle sich schneidenden Polygone in den Chunk gezeichnet wurden und die Fragmente aufgelöst sind, werden die Pixeldaten, einschließlich Farbe und Opazität, im Tiler-Chip komprimiert und dann zum externen Speicher bewegt.

[0287] Das Flussdiagramm in den Fig. 17A und Fig. 17B zeigt einen Überblick auf hoher Ebene, wie man eine Grafikszene in Chunks aufteilt. Zuerst wird eine oder werden mehrere Begrenzungskästen für jedes Objekt erzeugt. (**592**) (Fig. 17A). Wenn das Objekt eine komplexe Geometrie besitzt (z.B. mosaikartig fein unterteilt usw.), dann wird eine Reihe von Begrenzungskästen erzeugt, um jede komplexe Komponente des Objekts einzuschließen (mehrere Gsprites zu erzeugen) (**596**). Wenn die Objektgeometrie nicht komplex ist, dann kann ein einziger Begrenzungskasten zum Einschließen des Objektes und Erzeugen eines Gsprites (**598**) verwendet werden. Wenn das Objekt jedoch komplex ist, dann umschließt ein einzelner Begrenzungskasten auch die mehreren Begrenzungskästen, die zum Umschließen der komplexen Komponenten des Objekts erzeugt wurden. Wenn der Begrenzungskasten oder die -kästen kein ganzzahliges Vielfaches von 32 Pixeln (**600**) sind, wird/werden der Begrenzungskasten oder die -kästen symmetrisch in X- oder Y-Richtung (oder in beiden Richtungen) gedehnt, so dass sich ein ganzzahliges Vielfaches von 32 Pixeln ergibt. Das Objekt (oder die Objektkomponenten, wenn die Geometrie komplex ist) wird werden dann im Begrenzungskasten (**602**) zentriert. Dies wird durch die Gsprites illustriert, die in den Fig. 15B und Fig. 15C gezeigt werden. Die symmetrische Ausdehnung ist vorzuziehen, obwohl nicht erforderlich, da sie das beste Gleichgewicht für die Verarbeitung zwischen Chunks in einem einzelnen Gsprite bietet.

[0288] Wieder mit Bezugnahme auf **Fig.** 17, werden die Gsprites dann in 32×32-Pixelchunks (**604**) (Fig. 17B) unterteilt. Wie erkennbar ist, befinden sich diese Chunks nicht an festen Orten im Betrachtungsraum, sondern sind an adressierbaren und variablen Orten je nach dem Ort des gechunkten Objektes. Nach dem Teilen der Gsprites in Chunks, werden die Chunks verarbeitet. Wenn das Rendern der Chunks abgeschlossen ist (**606**), endet der Prozess. Wenn das Rendern der Chunks nicht abgeschlossen ist, wird die Verarbeitung des nächsten Chunks gestartet, nachdem zuerst untersucht wurde, ob er leer ist (**608**). Wenn der Chunk leer ist, wird er nicht verarbeitet, und der nächste Chunk wird untersucht. Wenn der Chunk nicht leer ist, dann setzt sich das Rendern (**610**) des Chunks im Tiler fort, bis alle Objekte, die auf dem Chunk liegen, verarbeitet sind. Dieser Prozess setzt sich fort, bis alle Chunks in jedem Gsprite und alle Gsprites verarbeitet sind.

[0289] Die Größe der Gsprites kann als Prozentsatz der Gesamtbildschirmfläche ausgedrückt werden. Hintergrund-Gsprites sind ziemlich groß, jedoch andere Komponenten der Szene sind normalerweise ziemlich klein im Vergleich zur Gesamtbildschirmfläche. Die Leistungsfähigkeit jeder Chunkingmethode, die verwendet wird, schwankt mit der Bildschirmraumgröße der Grundelemente in den Gsprites. Im Ergebnis dessen ist es notwendig, den Objektdateneingabestrom, der zur Erzeugung der Gsprites verwendet wird, richtig zu regulieren (z.B. in eine Schlange zu setzen). Die richtige Regulierung des Objektdateneingabestroms ermöglicht es, die Objektverarbeitung mit einer größeren Bandbreite abzuschließen und den Systemdurchsatz zu erhöhen.

[0290] Unser System verwendet einen Befehlsstromcache, um den Objektdateneingabestrom zu cachen. Der Befehlsstromcache kann zum Cachen des gesamten Inhalts eines Gsprites und dann zum Iterieren über jeden Chunk und seine zugehörigen Geometrien im Gsprite, die im Cache gespeichert sind, verwendet werden.

[0291] Der Cache kann auch zum selektiven Cachen verwendet werden. Zum Beispiel wenn man einen Schwellwert definiert, so dass geometrische Grundelemente automatisch gecacht werden, wenn sie eine bestimmte Zahl von Chunks berühren. Wenn ein Cache verfügbar ist, kann das virtuelle Chunking ausgeführt werden. Beim virtuellen Chunking wird ein Chunk-Speicherbereich erzeugt, der den Regionen von N × M

Chunks entspricht, wobei jede Region ein virtueller Chunk ist. Das virtuelle Chunking ermöglicht es, die virtuellen Chunks dem Inhalt und der Größe der Geometrie, die bearbeitet wird, entsprechend adaptiv in der Größe zu bemessen.

[0292] Eine weitere Verwendung für den Cache ist das modifizierte Cachen von Szenengraphen. Statt zu cachen und sich auf statische Abschnitte der Szene zu beziehen, wird gecacht, und auf dynamische Abschnitte wird durch Anspielung verwiesen. Nehmen wir zum Beispiel einen Gsprite an, der in einer Kuckucksuhr mit recht komplexen Geometrien enthalten ist. Die Uhr selbst ist recht komplex, aber die einzigen sich bewegenden Teile sind ein Vogel, zwei Türen und zwei Uhrzeiger. Ferner ist jede dieser Geometrien starr und ändert sich nicht. Daher beinhaltet das Rendern der Uhr sechs statische Bäume und sechs Transformationen (d.h. einen für Uhr, Vogel, 2 Türen und 2 Uhrzeiger). Wenn der Cache groß genug ist, wird der ganze Szenengraph in einen Befehlsstrom transformiert. Beim Rendern werden die aktuellen Transformationen über den gecachten Befehlsstrom gepatcht, und der resultierende Befehlsstrom wird für alle Chunks im Gsprite eingeleitet. Die gepatchten Teile des Befehlsstroms haben dann über alle Renderingoperationen hinweg dieselbe Größe. Ein flexiblerer Ansatz besteht darin, einen Aufrufbefehl in den gecachten statischen Szenengraphen einzuführen. Beim Rendern werden die dynamischen Teile in Speicher veränderlicher Größe geschrieben und gecacht. Adressen dieser dynamischen Abschnitte werden dann in den zugehörigen Aufrufbefehl im statischen Befehlsstrom gepatcht. Dieser Ansatz ist flexibler, da die Größe des dynamischen Befehls von Rendering zu Rendering variieren kann. Dieser Ansatz ist also vom Effekt her ein speicher-gecachter Rückrufansatz. Im Fall der Kuckucksuhr würde dies das Schreiben von sechs Transformationen und möglicherweise einen Rückruf für die Vogelgeometrie bedeuten, so dass er leer sein könnte, wenn die Türen geschlossen sind. Dieser Ansatz ist äußerst kompakt in Bezug auf Busbandbreite und bietet sich für den schnellen, gerichteten Durchlauf des Szenengraphen an.

[0293] Auch wenn der Cachespeicher begrenzt ist, können einige geometrische Strukturen oder Attribute über mehrere Renderings im Cache bleiben. Zum Beispiel würde bei einem Autorennspiel das Cachen einer Autokarosseriegeometrie zu einer beträchtlichen Gesamteinsparung über die Renderings führen. In gleicher Weise könnten häufige Attributzustände (oder Teilzustände) über viele Gsprites oder das Rendering eines einzelnen Gsprites wieder verwendet werden. Wie gerade beschrieben, kann das Verwenden eines Caches in einer Chunkingmethode zu beträchtlichen Zeiteinsparungen führen. Jedoch könnte eine angemessene Chunkingleistungsfähigkeit auch ohne den Befehlsstromcache dadurch erreicht werden, dass ein Befehlsstrom fliegend für jeden berührten Chunk im Gsprite erzeugt wird.

[0294] Bei der Implementierung des Tilers, die in den Fig. 9A-Fig. 9C gezeigt wird, werden Chunks sequenziell verwendet, um ein ganzes Teilbild auf einem Prozessor zu rendern, statt mehrere gleichzeitige Chunks auf parallelen Prozessoren zu verwenden, um die rechnerische Last zu teilen. Obwohl dies in geringerem Maße bevorzugt wird, könnte auch eine Kombination von serieller und paralleler Verarbeitung von Chunks verwendet werden. Bei einer vollständig parallelen Verarbeitungsimplementierung von Chunks würde ein Objekt, das sich über den Bildschirm bewegt, notwendigerweise konstante Chunkingoperationen erfordert, wenn es über den Bildschirm bewegt wird. In der erläuterten Ausführungsform der Erfindung kann jedoch ein Objekt wegen der seriellen Verarbeitung von Chunks an den Chunkgrenzen in einem Gsprite festgemacht werden und daher KEIN Chunking erfordern, wenn das Objekt über den Bildschirm bewegt wird. Das Rendern von Chunks bei Parallelverarbeitung ermöglicht auch keine Anwendung von anspruchsvollen Anti-Aliasing- und Texturierungsmethoden auf einzelne Chunks, wie dies für das serielle Rendern von Chunks der Fall ist. Die Chunkgröße und das sequenzielle Rendern sind für Bildkomprimierungsverfahren sehr wertvoll, da ein ganzer 32×32-Pixelchunk vor dem Berechnen des nächsten Chunks gerendert wird und daher sofort komprimiert werden kann.

[0295] Der Zweck der Bildkomprimierung besteht darin, Bilder mit weniger Daten darzustellen, um Speicherkosten und/oder Übertragungszeit und -kosten zu sparen. Je weniger Daten zum Darstellen eines Bildes erforderlich sind, desto besser, vorausgesetzt, das Bild kann angemessen wiederhergestellt werden. Die effektivste Kompression wird durch Annähern des Originalbildes anstelle von einer exakten Reproduktion erreicht. Je größer die Kompression, umsomehr wird das Endbild eine Näherung („verlustbehaftete Kompression") sein.

[0296] Der Prozess des Chunkings ist selbst ein Komprimierungsverfahren. Objekte werden mit einem oder mehreren Gsprites angenähert, die wiederum aus einer Reihe von 32×32-Pixelchunks erzeugt werden. Das tatsächliche Objekt wird mit Gsprites angenähert und aus gerenderten Gsprites rekonstruiert. Die Rekonstruktion des Originalobjekts hängt davon ab, wie effektiv das Objekt durch seine Aufteilung in Gsprites und anschließendes Chunking angenähert wurde (z.B. unter Verwendung komplexer Objektgeometrieteilungsverfahren, die oben beschrieben werden).

[0297] Die einzelnen 32×32-Chunks werden ebenfalls unter Verwendung von Bildkomprimierungsverfahren komprimiert. Ein komprimierter 32×32-Pixelchunk nimmt weniger Raum in dem verfügbaren kleinen Umfang an internem Speicher ein. Die 32×32-Pixelchunks können in sechzehn 8×8-Pixelchunks aufgeteilt werden, welches die Größe ist, die häufig in Bildkomprimierungsverfahren, die diskrete Kosinustransformationen (DCT) einsetzen, verwendet wird.

[0298] In einer Implementierung unterstützen die Kompressions- und Dekompressionsengines im Tiler und die Dekompressionsengine in der Gsprite-Engine sowohl verlustbehaftete als auch verlustfreie Formen der Kompression/Dekompression. Die verlustbehaftete Form umfasst eine verlustfreie Farbtransformation von RGB nach YUV, eine DCT, gleichförmige oder wahrnehmbare Quantisierung und Entropiecodierung (Lauflängen- und Huffman-Codierung). Die verlustfreie Form umfasst eine Farbtransformation von RGB nach YUV, eine Vorhersagestufe und Entropiecodierung, wie sie in der verlustbehafteten Form ausgeführt wird.

[0299] Um die Speicheranforderungen zur Verarbeitung von Grafikbildern unter Verwendung des Chunkings drastisch zu reduzieren, wird in der erläuterten Ausführungsform ein kleiner Z-Puffer (z.B. etwa 4 Kilobyte (kB)) verwendet. Speziell ist der Z-Puffer in dieser Implementierung etwas kleiner als 4 kB (1024×26), jedoch kann die Zahl der Präzisionsbits variieren. Es könnten jedoch auch ein kleinerer oder größerer Z-Puffer verwendet werden. Durch die Verwendung eines kleinen 4-kB-Z-Puffers können jeweils nur 1024 Pixel Z-Puffer-gerendert werden. Um Szenen (z.B. Szenen, die aus Gsprites bestehen) mit beliebiger Größe unter Verwendung eines 4-kB-Z-Puffers zu rendern, wird die Szene in Chunks mit der Größe von 32×32 Pixeln aufgeteilt (es gibt normalerweise mehrere Gsprites in einer Szene, jedoch wird jeder Gsprite in Chunks aufgeteilt). Bei dieser Methode sendet der Bildpräprozessor die entsprechende Geometrie an jeden Chunk in einem Gsprite zum Z-Puffer-Rendern.

[0300] Als Beispiel dafür, wie das Chunking arbeitet, betrachte man die acht Objekte und ihre zugehörigen geometrischen Strukturen, die in Fig. 18A gezeigt werden. Der Einfachheit halber werden die acht Objekte **612**-**619** durch ein einziges Attribut **620** (z.B. Farbe) definiert, das einen von vier Werten A-D haben kann. Die acht Objekte werden dann in einer Grafikszene überlagert, wie in Fig. 18B gezeigt. Wenn man individuelle Gsprites und ihre Erzeugung ignoriert, sich aber statt dessen für Zwecke der Erläuterung auf vier isolierte Chunks konzentriert, werden die vier isolierten Chunks **621**-**624** in Fig. 18B gezeigt. Die vier isolierten Chunks **621**-**624** (Fig. 18B) werden von den geometrischen Strukturen 1-8 und den Attributen A-D berührt, wie in Fig. 19A gezeigt. Chunk 1 **630** (Fig. 19A) wird von den geometrischen Strukturen 1, 2 und 5 und dem Attribut B, Chunk 2 **639** wird von keinen geometrischen Strukturen und den Attributen A-D berührt, Chunk 3 **632** wird von den geometrischen Strukturen 2, 4, 7 und 8 und den Attributen A, B, D berührt, und Chunk 4 **634** wird von den geometrischen Strukturen 4 und 6 und den Attributen A, C berührt. Ein Beispiel für einen Teilszenengraphen, der von der Bildvorverarbeitung (unter Verwendung der Chunks, die in den Fig. 18B und Fig. 19A gezeigt werden) aufgebaut wird, wird in Fig. 19B gezeigt. Die Attribute jedes Chunks (z.B. Farbe usw., die durch A-D, X gezeigt werden) werden als Kreise **638** und die geometrischen Strukturen (z.B. die verschiedenen Formen, gezeigt durch 1-8) werden als Quadrate **640** gezeigt. Der Buchstabe X bezeichnet einen Vorgabewert für ein Attribut. Die Zwischenknoten enthalten Attributoperationen zur Anwendung auf die geometrischen Strukturen der Grundelemente. Die Blattknoten im Szenengraphen enthalten Grundelementestrukturen, die auf die Chunks angewendet werden sollen, und können auch Begrenzungsvolumina um die Strukturen herum enthalten, die die Chunks beschreiben (Blattknoten mit Begrenzungsvolumina werden unten beschrieben).

[0301] Ein Ansatz für eine Chunkingmethode besteht darin, über jeden Chunk zu iterieren und jedes Mal die volle geometrische Struktur zu senden. Ein anderer, besserer Ansatz besteht darin, nur geometrische Strukturen zu senden, die im aktuellen Chunk sichtbar sind (Man beachte, dass im optimalen Fall geometrische Strukturen übersprungen werden, die verdeckt oder anderweitig unsichtbar sind). Das Verfahren zum Chunken eines Gsprites in einem 32×32-Pixelblock, das tatsächlich in unserem System verwendet wird, liegt zwischen den zwei Extremen und wird Bucket Chunking genannt. Jedoch können auch andere Methoden, die auf oder zwischen die zwei Extreme fallen, verwendet werden, um Chunks für eine Chunkingmethode zu erzeugen.

[0302] Die Bucket Chunking-Methode besteht aus zwei Durchläufen. Der erste Durchlauf durchläuft den Szenengraphen, während gleichzeitig die aktuelle Transformation in den Betrachtungsraum mit dem Ziel aufrechterhalten wird, einen beschreibenden Befehlsstrom für jeden Chunk im Betrachtungsraum aufzubauen. Der Betrachtungsraum wird in N × M Chunk Buckets (Speicherbereiche) aufgeteilt, die am Ende jeweils eine Liste der geometrischen Strukturen enthalten, die auf den entsprechenden Chunk fallen. Wenn ein Geometrie-Grundelement-Knoten auftritt, wird die aktuelle Transformation auf das Begrenzungsvolumen angewendet, um eine 2D-„Fußspur" im Betrachtungsraum zu erhalten. Für jeden Chunk, der von der Fußspur berührt wird, wird die geometrische Struktur (und der akkumulierte Attributzustand) zum entsprechenden Bucket hinzugefügt. Am

Abschluss dieses ersten Durchlaufs enthält jeder Bucket die notwendigen Daten, um den entsprechenden Chunk zu rendern. Man beachte, dass diese Chunkingmethode empfindlich auf die Qualität der berechneten Fußspur reagiert – eine lose Grenze am Objekt ergibt eine größere Fußspur und trifft Chunks, die nicht von der umschlossenen geometrischen Struktur berührt werden. Eine enge Grenze am Objekt ergibt eine kleinere Fußspur und trifft die meisten Chunks, die von den umschlossenen geometrischen Strukturen berührt werden.

[0303] Als Beispiel für den ersten Durchlauf betrachte man eine Teilmenge von vier Chunks, die sich überlagernde Objekte enthalten, die von den geometrischen Strukturen 1-8 und den Attributen A-D, X, in **Fig. 19A** gezeigt, beschrieben werden. Ein Ansatz für das Durchlaufen des Szenegraphen in Durchlauf eins besteht darin, den aktuellen Zustand für jeden Chunk beizubehalten und dann die geometrische Struktur zu überspringen, die nicht in einen gegebenen Chunk fällt. Damit wird sichergestellt, dass der Attributkontext für jede geometrische Struktur in jedem Chunk aktuell ist. Die Anwendung dieses Ansatzes auf den Szenegraphen in **Fig. 19B** ergibt den folgenden Befehlsstrom in den Chunk Buckets nach Durchlauf eins:
Chunk 1 Bucket: X, A, B, 1, 2, 5, A, X, C, D, C, X
Chunk 2 Bucket: X, A, B, A, X, C, D, C, X
Chunk 3 Bucket: X, A, B, 2, 7, 8, A, 4, X, D, 3, C, X
Chunk 4 Bucket: X, A, B, A, 4, X, C, 6, D, C, X

[0304] Ein weiterer Ansatz besteht im Beibehalten des aktuellen Attributzustandes und Senden des vorherigen Zustandes vor dem Senden jeder akzeptierten geometrischen Struktur. Dies führt zu den folgenden Befehlsströmen in den Chunk Buckets:
Chunk 1 Bucket: B, 1, B, 2, B, 5
Chunk 2 Bucket: <leer>
Chunk 3 Bucket: B, 2, B, 7, B, 8, A, 4, D, 3
Chunk 4 Bucket: A, 4, C, 6

[0305] Der zweite Ansatz ist eine Verbesserung gegenüber dem ersten Ansatz. Man beachte, dass das Attribut B vor den geometrischen Strukturen 2 und 5 ein zweites und ein unnötiges drittes Mal angegeben wird. Dieses Verhalten zeigt sich auch in Chunk 3 für B für die geometrischen Strukturen 7 und 8. In der Realität ist die Situation schlimmer als hier dargestellt, weil ein Speicherauszug des aktuellen Attributzustandes bedeutet, dass jedes einzelne Attribut für jede geometrische Struktur neu angegeben wird. Mit anderen Worten: Selbst wenn die Texturtransformationsmatrix invariant für den ganzen Szenegraphen ist, wird sie immer noch vor jeder einzelnen geometrischen Struktur in jedem Chunk gesendet.

[0306] Daher widmet sich dieser spezielle Ansatz der Attributbeibehaltung für vorrangige Attribute und statt dessen dem für das Zusammensetzen der Attribute. Diffuse Farbe ist ein vorrangiges Attribut. Wie durch den Bildpräprozessor definiert (z.B. Bildvorverarbeitungssoftware, die zum Beispiel auf dem Bildpräprozessor 24 usw. läuft), der den Szenegraphen erzeugt, führen Attribute, die auf rot(blau(Würfel)) angewendet werden, zu einem roten Würfel. Dies steht im Gegensatz zu anderen Bildpräprozessorgrafikschnittstellen, die das nächstgelegene Attribut an das Objekt binden. Binden des nahegelegensten Attributes an das Objekt für rot(blau(Würfel)) würde zu einem blauen Würfel führen.

[0307] Die Verwendung des am weitesten außen gelegenen Attributs als vorrangiges Attribut vereinfacht die Attributbeibehaltung für Attribute in großem Maße. Während des Durchlaufs durch den Szenegraphen kann man beim Treffen eines Attributknotens alle Knoten dieser Attributart unterhalb desselben im Szenegraphen ignorieren, da das am weitesten oben gelegene Attribut alle anderen nichtig macht.

[0308] Eine lokale Transformation ist ein Zusammensetzungsattribut. Der aktuelle Wert wird also durch den vorherigen Wert und den neuen Wert bestimmt. Das Zusammensetzungsattribut erfordert eine gewisse Art von Stack, während der Szenegraph durchlaufen wird, um vorherige Werte zu speichern.

[0309] Die Bucket Chunking-Methode verwendet die folgenden Strukturen:
- Den Attributknoten, der den aktuellen Wert enthält,
- Den Durchlaufkontext. Das ist eine Struktur, die für jedes vorrangige Attribut einen Zeiger auf den aktuellen Attributwert enthält.
- Ein Gitter von Buckets, wovon jeder einen Befehlsstrompuffer und eine Bucketkontextstruktur derselben Art wie der globale Durchlaufkontext enthält.
- Eine Liste von Vorgabeattributwerten, wobei auf jeden von ihnen vom Durchlaufkontext verwiesen werden kann.

[0310] Zur Initialisierung wird der Kontext in den Vorgabezustand versetzt, so dass alle Attribute sich auf den Vorgabekontext beziehen. Vorgabewerte werden langsam geladen und nicht en masse vor dem Senden der Renderingbefehle abgeladen.

Attributerhaltung Initialisieren:

```
für jedes Attribut: attr
    für jeden Bucket: bucket
        Bucket.Kontext(attr) <-- nil // Kontext für jeden Bucket löschen
    end
    Kontext[attr] <-- Vorgabe[attr]      //Auf Vorgabewerte initialisieren
    end
```

[0311] Das Folgende diktiert, wie ein gegebener Attributknoten zu verarbeiten ist:

Attribut verarbeiten:

```
if Kontext[attr] ≠ Vorgabe[attr]
    ProcessGeom()      //Attr bereits gesetzt, nachfolgenden Wert ignorieren.
else
    Kontext[attr] ⇦ SetAttr(attr, Wert) //Auf neuen Wert setzen
    ProcessGeom()
    Kontext[attr] ⇦ SetAttr(attr, Vorgabe[attr])
endif
```

[0312] Der Prozess zur Handhabung von Geometrieknoten synchronisiert den aktuellen Durchlaufstatus mit den Attributzuständen jedes Buckets:

Geometrie verarbeiten:

```
geomCommand ⇦ ConvertGeometry (geom) // In Befehlsstrom konvertieren.
für jeden berührten Bucket: bucket
    für jedes Attribut: attr
    if(Bucket.Kontext(attr) ≠ Kontext[attr])
        Bucket.Kontext[attr] ⇦ Kontext[attr]
        append (Bucket.Kontext(attr))
    endif
end
append (Bucket.geomCommand)
end
```

[0313] Zusammensetzungsattribute fungieren in einer ähnlichen Weise gegenüber vorrangigen Attributen, mit der Ausnahme, dass ein Stack während des Durchlaufs beibehalten wird. Dies wird durch die Verwendung von Knoten für die Speicherung der Stackwerte erreicht. Dieses Verfahren erfordert die folgenden Strukturen:

- Der aktuelle Attributknoten, der die Zusammensetzung der vorherigen Werte mit dem neuen Wert enthält.
- Den Durchlaufkontext. Das ist eine Struktur, die für jedes Zusammensetzungsattribut einen Zeiger auf den aktuellen Attributknoten enthält.

• Eine Liste von Vorgabeattributwerten, wobei auf jeden von ihnen vom Durchlaufkontext verwiesen werden kann.
• Ein Gitter von Buckets, wovon jeder einen Befehlsstrompuffer und eine Bucketkontextstruktur derselben Art wie der globale Durchlaufkontext enthält.

[0314] Die Initialisierung zur Zusammensetzung von Attributen sieht so wie für vorrangige Attribute aus:

Attributerhaltung Initialisieren:

```
für jedes Attribut: attr
    für jeden Bucket: bucket
        Bucket.Kontext(attr) ⇦ nil// Kontext für jeden Bucket löschen
        end
        Kontext[attr] ⇦ Vorgabe[attr] //Auf Vorgabewerte initialisieren
    end
```

[0315] Das Verarbeiten eines zusammensetzenden Attributknotens beinhaltet die Zusammensetzung des neuen Wertes mit allen Werten vor dem aktuellen Knoten im Durchlauf. Man beachte, dass der vorherige Wert gespeichert und wieder abgerufen werden muss, um einen Stack von Werten zu implementieren.

Attribut verarbeiten:

```
Knoten.ZusammengesetzterWert ⇦ Zusammensetzen(Kontext[attr], Knoten.Wert)
SavePtr ⇦Kontext[attr] //vorherigen zusammengesetzten Wert speichern
Kontext[attr]⇦Knoten
ProcessGeom()
Kontext[attr]⇦SavePtr //den vorherigen zusammengesetzten Wert wiederherstellen
```

[0316] Das Geometrie-Handhabungsprogramm ist identisch mit dem Fall der vorrangigen Attribute:

Process Geometrie:

```
geomCommand⇦ConvertGeometry(geom) //In Befehlsstrom konvertieren.
für jeden berührten Bucket: bucket
    für jedes Attribut: attr
        if(bucket.Kontext(attr)≠Kontext(attr)
            bucket.Kontext(attr)⇦Kontext(attr)
            anhängen(bucket.Kontext(attr))
        endif
    end

anhängen(bucket,geomCommand)
end
```

[0317] Der zweite Durchlauf der Bucket Chunking-Methode iteriert über das Gitter von Buckets und gibt den entsprechenden Befehlsstrom aus. Für jeden nicht leeren Bucket wird der entsprechende Chunk aus den Informationen, die in diesem Bucket gespeichert sind, gerendert. Man beachte, dass es leere Buckets in der Szene geben kann, was bedeutet, dass nicht jeder Chunk im Gsprite gerendert werden muss. Für die aktivsten

Gsprites, die aus einem lichtundurchlässigen Objekt auf einem transparenten Hintergrund bestehen, sollte ein guter Teil der Chunks leer sein.

[0318] Der Ansatz zur Aufrechterhaltung des Attributzustandes, der oben beschrieben wird, ist zum Rendern von Geometrie in einer gechunkten Weise besonders geeignet. Das Chunking bewirkt, dass Sätze von geometrischen Strukturen in anderer Reihenfolge, als ursprünglich angegeben, gerendert werden. Beim Rendern eines Chunks, zum Beispiel, überspringt das Renderingsystem Gruppen von geometrischen Strukturen, die den Chunk nicht schneiden. Daher sollten auf dem unteren Niveau des gechunkten Geometrierenderns höchstens zwei Zustandsniveaus aufrechterhalten werden: 1) ein globaler Zustand in einem Format, das mit dem Tiler oder alternativer Rendering-Hardware kompatibel ist, um das Rendern der geometrischen Strukturen zu ermöglichen; und 2) kleine Statusüberlagerungen innerhalb eines Satzes von geometrischen Strukturen, die nur für diesen Satz von geometrischen Strukturen gelten. Mit dieser Herangehensweise kann jeder Satz von geometrischen Strukturen unabhängig voneinander gerendert werden, und das Rendern eines Satzes von geometrischen Strukturen kann als frei von Nebeneffekten angesehen werden.

Bildkompression

[0319] Wie oben beschrieben wurde, ist die Chunkgröße und das sequenzielle Rendern für Bildkomprimierungsverfahren sehr wertvoll, da ein ganzer 32×32-Pixelchunk vor dem Berechnen des nächsten Chunks gerendert wird und daher sofort komprimiert werden kann. Der Tiler unterstützt verlustbehaftete und verlustfreie Formen von Kompression, um Chunks zu komprimieren. Sowohl die verlustbehaftete als auch verlustfreie Kompressionsform komprimiert Chunks in unabhängigen Blöcken von 8×8 Pixeln. Also würde jeder 32×32-Pixelchunk aus 16 solcher komprimierter Blöcke bestehen.

[0320] Die Kompression von Bildern ermöglicht einen kleineren Speichergrößenbedarf und einen ungemein reduzierten Speicherbandbreitenbedarf. Die Konstruktion verwendet eine Kombination von Caching, Strategien zum vorherigen Abrufen sowie Chunking, um die Verzögerungszeiten und den zusätzlichen Aufwand auf Grund der Kompression des Blockzugriffs zu reduzieren. Da das gesamte Bild in einem 32×32-Pixelpuffer berechnet wird, wird die Gsprite-Bildkompression mit minimalem zusätzlichem Aufwand erreicht. Der Gesamtentwurf der Kompressionsarchitektur wird in **Fig. 20** gezeigt.

[0321] Die Transformationsengine **660** (**Fig. 20**) berechnet Modell- und Betrachtungstransformationen, Beschneiden (Clipping), Beleuchtung usw. und leitet diese Informationen an den Tiler **662** weiter. Während der Tiler Transformationsinformationen verarbeitet, liest er Texturdaten aus dem Texturspeicher **664**. Die Texturdaten werden in einem komprimierten Format gespeichert, daher werden die Texturblöcke, wenn sie benötigt werden, von der Tiler-Dekompressionsengine **666** dekomprimiert und in einem chipinternen Texturcache im Tiler gecacht. Wenn der Tiler Pixeldaten auflöst, transferiert er die aufgelösten Daten zur Tiler-Kompressionsengine **668**, die die aufgelösten Daten komprimiert und die komprimierten Daten im Gsprite-Speicher **670** speichert. Wenn die Gsprite-Engine **672** die komprimierten Gsprite-Daten benötigt, verwendet sie die Gsprite-Dekompressionsengine **674**, um die Gsprite-Daten aus dem Gsprite-Speicher **667** zu dekomprimieren und die Daten in einem chipinternen Gsprite-Cache zu cachen. In der tatsächlichen Hardware sind der Texturspeicher **664** und der Gsprite-Speicher **670** identisch (d.h. die komprimierten Daten werden in einem Speicher gespeichert, der von den verschiedenen Engines gemeinsam genutzt wird). Gemeinsam genutzter Speicher wird nicht benötigt, solange die verwendeten Kompressions- und Dekompressionsverfahren kompatibel sind. Die Gsprite-Daten können auch aus einer Datenbank oder einer anderen Bildquelle **676** genommen und im Texturspeicher **664** oder dem Gsprite-Speicher **670** gespeichert werden.

[0322] Eine Implementierung der Erfindung unterstützt die verlustbehaftete und die verlustfreie Kompression und Dekompression von Pixelblöcken

[0323] Die verlustbehaftete Form der Bildkompression besitzt zwei Stufen: eine verlustbehaftete erste Stufe und eine verlustfreie zweite Stufe. Die verlustbehaftete Form der Kompression beginnt mit einer optionalen Farbraumumwandlung von roten, grünen, blauen (R, G, B) Intensitätswerten in Leuchtdichte-(Y) und Chrominanz-(U und V, auch als Cr und Cb bezeichnet) Werte. Die verlustbehaftete Stufe umfasst eine direkte Kosinustransformation (DCT) und eine Quantisierung, die die Genauigkeit bestimmter Frequenzkomponenten reduziert.

[0324] Die zweite Stufe ist eine verlustfreie Form der Kompression, die die Huffman-Codierung und die Lauflängencodierung (RLE) umfasst. Alternative Codierverfahren, wie zum Beispiel das arithmetische Codieren, können an Stelle der Huffman-Codierung verwendet werden.

[0325] Die Dekompression für das verlustbehaftete Verfahren umfasst eine Decodierstufe, eine Dequantisierung der komprimierten Daten, eine inverse DCT und eine optionale Farbraum-Konversion von YUV nach RGB.

[0326] Die verlustfreie Form der Kompression umfasst eine optionale Farbraum-Konversion von RGB nach YUV, eine Vorhersagestufe und eine verlustfreie Codierungsstufe. Diese Codierungsstufe kann identisch mit der Entropiecodierungsstufe in der verlustbehafteten Form der Kompression sein. Die Dekompression für dieses verlustfreie Verfahren umfasst eine Decodierstufe, eine inverse Vorhersagestufe für jede Farbkomponente und eine optionale Farbraum-Konversion von YUV nach RGB.

Verlustbehaftete Kompression/Dekompression

[0327] Eine spezielle Implementierung des verlustbehafteten Kompressionsverfahrens in der Kompressions-Engine **414** (Fig. 9A-C) des Tilers geschieht in vier oder fünf Schritten:

1. Konvertiere die RGB-Dateneingabe in ein YUV-ähnliches Leuchtdichte-Chrominanz-System (optional).
2. Führe eine vorwärts gerichtete zweidimensionale diskrete Kosinustransformation (DCT) individuell an jeder Farbkomponente aus.
3. Ordne die zweidimensionalen DCT-Koeffizienten annähernd in einer monoton steigenden Frequenzordnung an.
4. Quantisiere die DCT-Koeffizienten: Teile entweder durch einen einheitlichen Teiler oder einen frequenzabhängigen Teiler.
5. Codiere die sich ergebenden Koeffizienten unter Verwendung der Huffman-Codierung mit festen Codetabellen.

[0328] Die verlustbehaftete Dekompression läuft in vier oder fünf Schritten ab:

1. Decodiere die komprimierte Dateneingabe unter Verwendung der Huffman-Decodierung mit festen Codetabellen.
2. Dequantisiere die komprimierten Daten: Multipliziere mit einem einheitlichen Multiplikator oder mit dem frequenzabhängigen Multiplikator, der beim Quantisierungsschritt der Kompression verwendet wurde.
3. Ordne das lineare Feld von Daten in die richtige zweidimensionale Ordnung für die DCT-Koeffizienten um.
4. Führe eine inverse zweidimensionale DCT individuell an jeder einzelnen Farbkomponente aus.
5. Konvertiere die Farben in dem YUV-ähnlichen Leuchtdichte-Chrominanz-System in RGB-Farben, wenn der Kompressionsschritt den entsprechenden optionalen Schritt umfasste.

Farbraumumwandlung

[0329] Die Farbraumumwandlung transformiert die RGB-Farben in ein Helligkeits-Farbsystem mit der Helligkeitskoordinate Y und den Farbkoordinaten U und V. Dieses Leuchtdichte-Chrominanz-System ist kein Standardfarbraum. Die Verwendung dieses Systems verbessert den Grad der Kompression, weil die Farbkoordinaten nur einen kleinen Bruchteil der Bits benötigen, die zum Komprimieren der Helligkeit benötigt werden. Die verlustlose, reversible Umwandlung wird auf jedes Pixel unabhängig angewandt und verändert den Wert von Alpha nicht.

RGB nach YUV (zur Kompression

[0330] Die Umwandlung von ganzzahligen RGB-Werten in ganzzahlige YUV-Werte verwendet die folgende Transformation:

$$Y = (4R + 4G + 4B)/3 – 512$$

$$U = R – G$$

$$V = 14B – 2R – 2G)/3$$

YUV nach RGB (zur Dekompression)

[0331] Die Umwandlung von ganzzahligen YUV-Werten in ganzzahlige RGB-Werte verwendet die folgende Transformation:

R = (((Y + 512) – V)/2 + U + 1)/2

G = (((Y + 512) – V)/2 – U + 1)/2

B = ((Y + 512)/2 + V + 1)/2

Diskrete Kosinustransformation

[0332] Bilder und Texturen sind Pixel, die die Amplituden für drei Farben und die Amplitude für Lichtundurchlässigkeit (Opazität) enthalten. Die Pixelpositionen entsprechen einem räumlichen Ort in einem Bild oder einer Texturabbildung. Ein Bild oder eine Textur in dieser Form liegt in der räumlichen Domäne. Für Bilder und Texturen berechnet die diskrete Kosinustransformation (DCT) Koeffizienten, die mit den Basisfunktionen des DCT multipliziert werden. Die Anwendung der DCT auf ein Bild oder eine Textur ergibt einen Satz von Koeffizienten, die das Bild oder die Textur äquivalent repräsentieren. Ein Bild oder eine Textur in dieser Form liegt in der Frequenzdomäne.

[0333] Die DCT bildet die Amplitude der Farben und Opazität eines 8×8-Pixelblocks zwischen der räumlichen Domäne und der Frequenzdomäne ab. In der Frequenzdomäne sind benachbarte Koeffizienten weniger stark korreliert, und der Kompressionsprozess kann jeden Koeffizienten unabhängig behandeln, ohne die Kompressionseffizienz zu reduzieren.

[0334] Die Vorwärts-DCT bildet die räumliche Domäne auf die Frequenzdomäne ab, und umgekehrt bildet die inverse DCT die Frequenzdomäne auf die räumliche Domäne ab. Ein geeigneter Ansatz für die Vorwärts- und die inverse DCT ist der Ansatz, der in den **Fig.** A.1.1 und A.1.2 in Discrete Cosine Transform Rao, K. R, und P. Yip. San Diego: Academic Press, Inc., 1990, beschrieben wird.

[0335] Die zweidimensionale DCT erzeugt ein zweidimensionales Feld von Koeffizienten für die Frequenzdomänendarstellung jeder Farbkomponente. Die Zickzackordnung ordnet die Koeffizienten so neu an, dass die niedrigen DCT-Frequenzen hauptsächlich an niedrigen Positionen eines linearen Feldes auftreten. Bei dieser Ordnung ist die Wahrscheinlichkeit, dass ein Koeffizient null ist, annähernd eine monoton steigende Funktion der Position im linearen Feld (wie sie durch den linearen Index gegeben ist). Dieses Ordnen vereinfacht die Wahrnehmungsquantisierung und LOD-Filterung und verbessert auch beträchtlich die Leistungsfähigkeit der Lauflängencodierung (RLE).

Quantisierung

[0336] Quantisierung verringert durch Teilen der Koeffizienten durch eine ganze Zahl die Zahl unterschiedlicher Werte, die die zickzackgeordneten DCT-Koeffizienten haben können. Je nach dem Wert des Parameters der Kompressionsart kann die Quantisierung entweder gleichförmig oder perzeptiv bzw. wahrnehmend erfolgen. Keiner der beiden Fälle modifiziert den DC-Frequenzkoeffizienten (Index = 0), sondern leitet ihn unverändert weiter.

[0337] Der Quantisierungsprozess beginnt mit der Festlegung des Quantisierungsfaktors für ein Bild oder einen Teil eines Bildes. Bei dieser Implementierung wird ein Quantisierungsfaktor für einen 32×32-Pixelchunk festgelegt. Ein Quantisierungsindex (QIndex) gibt einen entsprechenden Quantisierungsfaktor (QFaktor) zur Verwendung für den Chunk an. Die folgende Tabelle zeigt die Beziehung zwischen QIndex und QFaktor.

Quantisierungsfaktor

QIndex	QFaktor	QIndex	QFaktor
0	2	8	32
1	3	9	48
2	4	10	64
3	6	11	96
4	8	12	128
5	12	13	192
6	16	14	256
7	24	15	4096

[0338] Jede Farbebene hat einen anderen Wert für den Chunk-QIndex. Ein QIndex von 15 wählt einen QFaktor von 4096, was bei der Quantisierung und inversen Quantisierung Nullen produziert. Der Quantisierungsprozess teilt jeden Koeffizienten in einem Block durch einen QFaktor und rundet ihn auf eine ganze Zahl. Der inverse Quantisierungsprozess multipliziert jeden Koeffizienten mit einem QFaktor. Quantisierung und inverse Quantisierung ändern die DC-Frequenzkomponente nicht.

Blockquantisierungsfaktor

[0339] Der QIndex, und damit der QFaktor, kann von Block zu Block (8×8 Pixel) variieren. Der QIndex für einen Block ergibt sich aus der Inkrementierung des QIndex für den Chunk mit einem Wert, der in die Blockkompressionsart eingebettet ist:

Block-QIndex = Chunk-QIndex + (Blockkompressionsart – 3)

[0340] Dadurch wird der Chunk-QIndex um eins, zwei, drei oder vier inkrementiert. Weil der größtmögliche QIndexwert 15 ist, wird jeder inkrementierte Wert größer als 15 auf 15 gesetzt.

[0341] Der QIndex, und damit der QFaktor, kann auch von Koeffizient zu Koeffizient (von Feldindex zu Feldindex) variieren, wenn die Quantisierungsart perzeptiv ist.

[0342] Für eine gleichförmige Quantisierung ist der Koeffizient QIndex gleich dem Block-QIndex, daher multipliziert (inverse Quantisierung) oder dividiert (Quantisierung) der entsprechende QFaktor jeden Koeffizienten im Block.

[0343] Für die perzeptive Quantisierung hängt der Koeffizient QIndex vom Wert (0...63) des Index im linearen Feld ab. Die folgende Tabelle führt den resultierenden Koeffizienten QIndex als Funktion des Feldindexwertes an.
Koeffizient QIndex Feldindex
Block-QIndex Index < 12
Block-QIndex +1 12 ≤ Index < 28
Block-QIndex + 2 28 ≤ Index < 52
Block-QIndex + 3 52 ≤ Index

Entropiecodierung

[0344] Die Huffman/RLE-Codierung verarbeitet das lineare Feld quantisierter DCT-Koeffizienten durch:
1. Unabhängiges Codieren von nichtverschwindenden Koeffizienten mit möglichst wenigen Bits (weil die DCT-Koeffizienten nicht korreliert sind).
2. Optimales Codieren kontinuierlicher „Durchgänge" von Koeffizienten mit Nullwerten – besonders am Ende des linearen Feldes (wegen der Zickzackordnung).

[0345] Ein geeigneter Ansatz für den Huffrnan/RLE-Codierungsprozess ist der Huffman/RLE-Codierungsprozess, der für die AC-Koeffizienten im wohlbekannten JPEG-Standbildkompressionsstandard verwendet wird.

[0346] Um einen direkten Zugriff auf Blöcke zu ermöglichen, codiert dieser spezielle Ansatz den DC-Frequenzkoeffizienten (Index = 0) nicht, sondern leitet ihn unverändert weiter.

[0347] Der Algorithmus berechnet eine Reihe von Codewörtern mit variabler Länge, von denen jedes beschreibt:

1. Die Länge, von null bis 15, einer Folge von Nullen, die dem nächsten nichtverschwindenden Koeffizienten vorangehen.
2. Die Zahl zusätzlicher Bits, die zum Angeben des Vorzeichens und der Mantisse des nächsten nichtverschwindenden Koeffizienten erforderlich ist.

[0348] Das Vorzeichen und die Mantisse des nichtverschwindenden Koeffizienten folgen dem Codewort. Ein reserviertes Codewort gibt an, dass die restlichen Koeffizienten in einem Block alles Nullen sind.

Codierung

[0349] Die Codierung aller Blöcke verwendet die typischen Huffman-Tabellen für AC-Koeffizienten aus Anhang K Abschnitt K.3.2 des Internationalen ISO-Standards 10918. Dies umfasst die Tabelle K.5 für die Luminanz (Y)-AC-Koeffizienten und Tabelle K.6 für die Chrominanz (U und V)-AC-Koeffizienten.

Decodierung

[0350] Die Decodierung aller Blöcke verwendet dieselben festen Tabellen wie der Codierungsprozess. Daher ist es niemals notwendig, die Huffmnan-Tabellen mit den Daten zu speichern oder weiterzugeben.

Verlustfreie Kompression/Dekompression

[0351] Die verlustfreie Kompression läuft in der Kompressions-Engine **414** im Tiler in zwei oder drei Schritten ab:

1. Konvertiere die einlaufenden RGB-Daten in ein YUV-ähnliches Luminanz-Chrominanz-System (optional).
2. Führe eine Differenzenvorhersageberechnung an jeder Farbkomponente aus. Codiere die sich ergebenden Koeffizienten unter Verwendung der Huffman-Codierung mit festen Codetabellen.

[0352] Die verlustfreie Dekompression läuft in den Dekompressionsengines **404**, **450** im Tiler und der Gsprite-Engine in zwei oder drei Schritten ab:

1. Decodiere die einlaufenden komprimierten Daten unter Verwendung der Huffman-Decodierung mit festen Codetabellen.
2. Führe eine inverse Differenzenvorhersage (Rekonstruktion) an jeder Farbkomponente aus.
3. Konvertiere die Farben im YUV-ähnlichen Luminanz-Chrominanz-System in RGB-Farben, wenn der Kompressionsschritt diesen entsprechenden optionalen Schritt umfasste.

Farbraumumwandlung

[0353] Die Farbraumumwandlung transformiert die RGB-Farben reversibel in ein Helligkeits-Farbsystem mit der Helligkeitskoordinate Y und den Farbkoordinaten U und V. Dies ist ein eindeutiger Farbraum, der den Grad der Kompression stärker selbst als das YUV-System verbessert, da die Zahlen, die in den Huffman/RLE-Kodierer einlaufen, kleiner und damit kompressibler sind. Die Farbraumumwandlung wird auf jedes Pixel unabhängig angewandt und verändert den Wert von Alpha nicht.

RGB nach YUV (zur Kompression)

[0354] Die Umwandlung von ganzzahligen RGB-Werten in ganzzahlige YUV-Werte verwendet die folgende Transformation:

$Y = G$

$U = R - G$

$V = B - G$

YUV nach RGB (zur Dekompression)

[0355] Die Umwandlung von ganzzahligen YUV-Werten in ganzzahlige RGB-Werte verwendet die folgende Transformation:

R = Y + U

G = Y

B = Y + V

[0356] Die Alphainformationen werden während der Farbraumumwandlung nicht geändert.

[0357] Die Farbraumumwandlung kann umgangen werden. Die Dekompressionshardware wird in den Fällen, in denen die Farbtransformation umgangen wird, durch ein Kennzeichen (Flag) in der Gsprite-Steuerdatenstruktur informiert.

[0358] Die Vorhersagestufe tritt nach der Farbraumtransformation auf. Die Vorhersage ist ein verlustfrei umkehrbarer Schritt, der die Entropie der meisten Quellbilder reduziert, insbesondere Bilder mit viel leerem Raum und horizontalen und vertikalen Linien.

[0359] In der Vorhersagestufe der Kompression und der inversen Vorhersagestufe der Dekompression:

1. sind p(x, y) die Pixelwerteingabe in die Kompressions- und die Ausgabe aus der Dekompressionsengine; und
2. sind d(x, y) die Differenzwerteingabe in den Codierer in der nächsten Stufe der Kompressions-Engine und Ausgabe aus der Umkehrung des Codierers in der Dekompressionsengine.

[0360] Die Vorhersage wird folgendermaßen berechnet:

d(x, y) = p(x, y) für x = 0, y = 0

d(x, y) = p(x, y) – p(x, y – 1) für x = 0, y > 0

d(x, y) = p(x, y) – p(x – 1, y) für x > 0

[0361] Die umgekehrte Vorhersage in der Dekompressionsengine wird folgendermaßen berechnet:

p(x, y) = d(x, y) für x = 0, y = 0

p(x, y) = p(x, y – 1) + d(x, y) für x = 0, y > 0

p(x, y) = p(x – 1, y) + d(x, y) für x > 0

[0362] Die Huffman/RLE-Codierung und -Decodierung ist in dieser Implementierung dieselbe wie für die verlustbehaftete Form der Kompression/Dekompression.

[0363] Die Kompressionsverfahren, die oben beschrieben werden, komprimieren Bilder in unabhängigen Blöcken von 8×8 Pixeln. Daher besteht bei der Chunking-Architektur, die oben beschrieben wird, jeder komprimierte 32×32-Pixelchunk aus 16 solcher Blöcke. Um die Kompression eines 32×32-Pixelchunks zu erleichtern, löst die Anti-Aliasing-Engine **412** Pixeldaten in 8×8-Pixelblöcke auf. Die 8×8-Pixelblöcke werden gepuffert, so dass ein erster Puffer gefüllt wird, während ein zweiter Puffer komprimiert wird.

Steuerungselemente und Parameter

[0364] Wie oben eingeführt, rendert der Tiler (Fig. 9A-Fig. 9C) Gsprites mit immer einem Chunk auf einmal. Diese Chunks bestehen aus Pixelblöcken (in diesem Fall 16 8×8-Pixelblöcke). Zur Texturabbildung, Schattierung und für einige Mehrfach-Renderingoperationen ruft der Tiler Gsprite- oder Texturblöcke aus dem Speicher ab. Um ein Teilbild zusammenzusetzen, ruft die Gsprite-Engine (Fig. 12A-B) Gsprite-Blöcke ab, transformiert Pixel in den Bildschirmraum und setzt Pixel in einem Zusammensetzungspuffer zusammen.

[0365] Es gibt eine Reihe von Steuerungsparametern, die die Verarbeitung von Gsprites, Chunks und Blöcken bestimmen. Eine Gsprite-Anzeigeliste speichert eine Liste von Gsprites, die ein Anzeigebild beinhalten. Diese Anzeigeliste umfasst Zeiger auf Gsprites, und spezieller, Gsprite-Headerblöcke. Wie weiter unten beschrieben wird, speichern Gsprite-Headerblöcke eine Reihe von Attributen eines Gsprites, einschließlich Gsprite-Breite, -Höhe und eine affine Transformation, die in Form eines Bildschirmraumparallelogramms definiert ist. Der Gsprite-Headerblock umfasst auch eine Liste seiner zugehörigen Chunks. In einer Implementierung ist diese Liste in der Form von Zeigern oder Handles zum Chunken der Steuerungsblöcke.

[0366] Chunksteuerungsblöcke umfassen Parameter pro Chunk und pro Block. Die Parameter pro Chunk umfassen eine YUV-Farbconverterumgehung, Vorgabe-Q-Faktoren, ein perzeptives Quantisierungs-Flag, Pixelformat und ob die Pixeldaten im Speicher liegen, der in den Speicherzuweisungseinheiten (MAU) in linearem Speicher gemanagt wird. Eine MAU ist ein Stück gemeinsamer Speicher, der zum Zuweisen von Chunkspeicher verwendet wird. MAU-gemanagter Speicher umfasst eine Liste von MAUs (124 Byte, zum Beispiel), wobei jede MAU einen Zeiger auf die nächste MAU hat. In einer speziellen Implementierung zum Beispiel werden die Chunksteuerungsblöcke in sequenziellen MAUs für jeden Gsprite gespeichert.

[0367] Die Parameter pro Block umfassen die Kompressionsart, Zahl der MAUs, die der Block überspannt, und einen Blockzeiger, der auf das erste Byte der Pixeldaten für den Block zeigt. Das spezielle Blockformat ist ein 8×8×4-Feld von Pixeln, die 32-Bit-Pixel codieren (8 Bit für RGB und Alpha).

[0368] Die Schritte zum Abrufen eines Pixels mit (X, Y)-Koordinaten in einem Gsprite unter Verwendung der obigen Steuerungsparameter umfassen:

1) Dividiere Y und X durch 32, um die Chunk-Zeile bzw. -Spalte abzuleiten.
2) Bilde die Chunknummer durch: (Chunkzeile)·(Spritebreite in Chunks) + Chunkspalte.
3) Bilde den Chunksteuerungsblockversatz durch: (Chunknummer)·(Größe des Chunkheaderblocks))
4) Bilde den Blockversatz innerhalb des Chunksteuerungsblocks durch: (Y<4:3>·4 + X<4:3>)·3.
5) Sende den Blockzeiger an die Logik des dekomprimierten Caches, empfange einen Block.
6) Bilde den Pixelversatz innerhalb des Blocks durch (Y<2:0>·8) + X<2:0>

[0369] Hier wird ein Chunkversatz zum Auswählen eines Chunks verwendet. Dann wird ein Blockversatz zum Auswählen eines Blockzeigers verwendet. Der Blockzeiger wählt einen Block, der das Pixel enthält, und der Pixelversatz wählt das Pixel.

[0370] Um auf den Block für ein gegebenes Pixel unter komprimierten Blöcken von Pixeldaten zuzugreifen, führen die Cache-Steuerelemente im Tiler und der Gsprite-Engine die folgenden Schritte aus:

1) Bilden der MAU-Adresse durch Aufsuchen des Blockzeigerwerts im Chunksteuerungsblock und durch Teilen durch die Größe des MAU.
2) Suchen der Zahl der MAUs, die im Chunksteuerungsblock für diesen Block zugewiesen ist.
3) Suchen der nächsten Blockzeigeradresse im Chunksteuerungsblock.
4) Bilden der Länge des komprimierten Blocks durch: MAUS, zugewiesen·MAU-Größe + Zweierkomplement von ((Blockzeiger) mod MAU-Größe) + (nächster Blockzeiger) mod (MAU-Größe)
5) Senden der Blockadresse und der Länge des komprimierten Blocks an die Logik des Komprimierten Caches.

[0371] Der komprimierte Cache liest die erste MAU, und wenn die Länge der Übertragung nicht erfüllt wurde, dann wird der Zeiger, der in der MAU enthalten ist, für den Zugriff auf den Beginn der nächsten MAU verwendet. Dieser Prozess setzt sich fort, bis die Übertragungslänge erreicht ist.

[0372] Zur Unterstützung von MIP-Abbildungstexturoperationen unterstützt der Tiler ein weiteres Indexierungsniveau. Ein Verfahren zum Indexieren eines MIP-Abbildungsniveaus umfasst die folgenden Schritte:

1) Bilde für einen gegebenen Sprite eine Tabelle von Versatzwerten auf MIP-Chunkniveau durch:
mipChunkOffset[0] = 0 \\\Versatz für Niveau von Detail 0
Für jedes Niveau der MIP-Abbildung:
mipChunkOffset[level + 1] = Breite von Sprite/(2^Niveau)·Höhe von Sprite/(2^Niveau) + mipChunkOffset[Niveau]
2) Verwenden Sie den LOD-Parameter, um den MIP-Chunkversatz zu erhalten.

[0373] An diesem Punkt kann mit MIP-Chunkversatz, Breite des Sprites/(2^Niveau) und Höhe des Sprites/(2^Niveau) jeder gewünschte Chunk innerhalb des gewählten Detailniveaus für den aktuellen Gsprite gefunden werden.

[0374] Oben haben wir das Konzept eines Gsprites vorgestellt. Um es kurz zu wiederholen, können ein oder mehrere Objekte im Betrachtungsvolumen einem Gsprite zugeordnet werden. Gsprites können unabhängig gerendert werden, was ermöglicht, sie bei unterschiedlichen Auflösungen zu rendern und mit variierender Häufigkeit zu aktualisieren. Um den Zusatzaufwand beim Rendern zu reduzieren, kann das System Bewegung eines Objektes durch Ausführen einer affinen Transformation am Gsprite angenähert werden, statt das Objekt neu zu rendern. Um die Gsprites, die eine Szene umfassen, anzuzeigen, setzt das System die Gsprites, die die Objekte in der Szene repräsentieren, zusammen. Wir werden diese und andere Merkmale detaillierter unten beschreiben.

[0375] Wie oben beschrieben, beginnt das System damit, geometrische Strukturen einem Gsprite zuzuordnen. Ein Gsprite ist eine zweidimensionale Region, die in den Koordinaten des physischen Ausgabegerätes gemessen wird. In der Implementierung, die unten im Detail dargestellt wird, ist die Gsprite-Form ein Rechteck, er kann aber auch andere Formen haben. Gsprites können affin durch die Gsprite-Engine transformiert werden (d.h. sie können skaliert, verschoben, gedreht, reflektiert und/oder geschert werden – es ist jede Transformation mit einer 2×2-Matrix plus Translation möglich). Eine Anwendung der 2D-Transformation besteht darin, eine 3D-Bewegung zu simulieren. Gsprites können derart instanziert werden, dass dasselbe Gsprite-Bild mehrmals auf dem Bildschirm mit unterschiedlichen Transformationen erscheinen kann. Das Instanzieren kann für rechteckige Teilmengen eines Gsprite-Bildes wie für das gesamte Bild gelten. Es kann auch auf eine Farbkomponentenbasis angewendet werden, z.B. könnte Alpha von einem Gsprite kommen, während die Farbe von einem anderen kommt.

[0376] Im allgemeinen ordnet der Bildpräprozessor ein Objekt einem Gsprite zu, jedoch kann auch mehr als ein Objekt einem Gsprite zugeordnet werden. Der Bildpräprozessor kombiniert sich durchdringende oder selbst verdeckende Objekte in einem einzelnen Gsprite. Er vereint auch Objekte auf der Basis von Speicher- und Verarbeitungsnebenbedingungen. Der Bildprozessor ist zum Beispiel möglicherweise nicht in der Lage, mehrere unabhängige, aber sich überlappende Gsprites in der Zeit zusammenzusetzen, die von der Auffrischungsrate des Ausgabegerätes benötigt wird. In diesem Fall kann das System diese überlappenden Objekte zu einem einzigen Gsprite vereinen.

[0377] Nach der Zuordnung von Objekten zu Gsprites rendert der Bildprozessor die Gsprites für das Teilbild (Frame). Das unabhängige Rendern der Objekte ermöglicht es dem System, den Rendering-Zusatzaufwand zu reduzieren, da es nicht jedes Objekt in einer Szene in jedem Teilbild zu rendern braucht. Wir werden dieses Merkmal ausführlich unten behandeln.

[0378] Um die Objekte in einer Szene anzuzeigen, setzt der Bildprozessor die Gsprites, die die Objekte in der Szene umfassen, zusammen. Das Zusammensetzen bezieht sich auf den Prozess des Kombinierens von Farbdaten aus Gsprite-Schichten. Zur Unterstützung der Lichtdurchlässigkeit berücksichtigt der Bildprozessor auch die Alphawerte von transformierten Gsprite-Pixeln, wenn er sie zum Anzeigen zusammensetzt.

[0379] Die Fig. 21a und Fig. 21B sind Flussdiagramme, die illustrieren, wie Gsprites in einer Ausführungsform verarbeitet werden. In der erläuterten Ausführungsform überspannt die Verarbeitung von Gsprites zwei Teilbildperioden. Objekte in einer Szene werden Gsprites zugeordnet und in der ersten Teilbildperiode gerendert; Gsprite in der Szene werden dann transformiert und in einer nächsten Teilbildperiode zusammengesetzt.

[0380] Zuerst bestimmt der Bildpräprozessor möglicherweise sichtbare Objekte. In Fig. 21A illustrieren wir diesen Prozess als Folge von Schritten. Für ein Teilbild bestimmt der Bildprozessor möglicherweise sichtbare Objekte durch das Durchlaufen einer Liste von Objekten (**696**, **698**) und Bestimmen, welche Objekte möglicherweise in einer Szene, d.h. in einem Betrachtungsraum, sichtbar sind.

[0381] Der Bildpräprozessor ordnet dann Gsprites zu, ordnet sie neu zu oder hebt die Zuordnung auf. Zuordnen eines Gsprites bezieht sich im allgemeinen auf das Erzeugen einer Datenstruktur, um den Gsprite im System zu repräsentieren. Wenn ein Objekt potentiell nicht sichtbar ist (**700**) und das System keinen Gsprite dafür zugewiesen hat (**702**), ist keine weitere Verarbeitung notwendig. Wenn ein Objekt möglicherweise nicht sichtbar ist (**702**) und das System bereits einen Gsprite dafür zugewiesen hat (**702**), dann hebt der Bildpräprozessor die Zuordnung des Gsprites für dieses Objekt auf (**704**).

[0382] Der Bildpräprozessor ordnet eine neue Gsprite-Datenstruktur möglicherweise sichtbaren Objekten zu, denen das System keinen Gsprite zugeordnet hat (**706**, **708**). In diesem Fall erzeugt der Bildpräprozessor eine

Gsprite-Datenstruktur und stellt Bilddaten, die dem Objekt entsprechen, zum Rendern in die Schlange (**710**). Dieses „Einreihen in die Schlange" zum Rendern wird als Hinzufügen zu einer Liste von Objekten zum 3D-Rendern dargestellt (**710**). Der Bildpräprozessor berechnet auch eine affine Transformation für den Gsprite (**714**). Die affine Transformation dient in dieser Ausführungsform zwei Zwecken. Erstens kann sie zum Annähern von Bewegung des Objektes verwendet werden, der sie in der Szene entspricht. Zweitens kann sie verwendet werden, um einen Gsprite aus dem Gsprite-Raum in die Ausgabegerätekoordinaten zu transformieren. Gsprite-Raum bezieht sich auf ein Koordinatensystem, das beim Unterteilen des Objektes in Chunks verwendet wird. Das Koordinatensystem, das zum Unterteilen des Objektes in Chunks verwendet wird, kann so optimiert werden, dass Chunkregionen das Objekt, das in den 2D-Raum transformiert wurde, sehr effizient abdecken.

[0383] Wenn ein Objekt möglicherweise sichtbar ist (**700**) und das System bereits einen Gsprite dafür zugewiesen hat (**706**), dann berechnet der erläuterte Bildpräprozessor eine affine Transformation (**714**). Wie wir unten detaillierter erklären, kann die affine Transformation zum Annähern der Bewegung des Objektes verwendet werden. Der Bildpräprozessor bewertet die Genauigkeit dieser Näherung, und wenn sie eine zu starke Verzerrung erzeugt (**716**), ordnet der Bildpräprozessor dem Objekt einen Gsprite neu zu (**708**). In diesem Fall bringt der Bildpräprozessor die geometrische Struktur, die in den Gsprite gerendert werden soll, zum Rendern in die Schlange (d.h. Orte in der 3D-Liste) (**710**) und fügt auch den Gsprite zur Anzeigeliste hinzu (**718**).

[0384] Wenn jedoch die affine Transformation zur genauen Näherung der Bewegung des Objektes verwendet werden kann (**716**, Verzerrung liegt innerhalb der vorgegebenen Toleranz), dann gibt es keine Notwendigkeit, das Objekt neu zu rendern, und der Bildpräprozessor stellt den Gsprite, der mit dem Objekt verknüpft ist, in die Anzeigeliste (**718**).

[0385] In der nächsten Teilbildperiode erzeugt der Bildprozessor das Anzeigebild. Die Teilbildperiode wird durch die gestrichelte Linie illustriert, die die Schritte (**718**) und (**720**) trennt. Der Bildprozessor durchläuft die Anzeigeliste und transformiert die Gsprites in der Liste auf die physischen Ausgabegerätekoordinaten (**720**). Die Transformation auf die Ausgabekoordinaten umfasst im allgemeinen das Scannen von Pixeldaten aus einem verdrehten, gedrehten oder skalierten Gsprite in die Pixelorte des Ausgabegerätes. Der Bildprozessor setzt dann diese transformierten oder „gescannten" Gsprite-Daten zusammen. Zum Schluss konvertiert der Bildprozessor die Pixeldaten in analoge Werte und zeigt das Bild an (**724**).

[0386] Die Fig. 5A und Fig. 5B sind Flussdiagramme, die den Prozess des Renderns von geometrischen Strukturen in einer Chunking-Architektur illustrieren. Es ist wichtig zu beachten, dass die Gspritekonzepte, die oben beschrieben werden, nicht auf eine Chunking-Architektur begrenzt sind. Die Fig. 5A und der begleitende Text oben liefern mehr Beschreibung bezüglich dessen, wie der Bildpräprozessor die Gsprite-Konfiguration aus der Geometrie in einer Szene bestimmt. Siehe Schritte (**240**-**244**) und den begleitenden Text. Speziell können Objekte auf Grund von Verarbeitungsbeschränkungen des Bildprozessors vereinigt und in einen einzigen Gsprite oder eine kleine Zahl von Gsprites gerendert werden, falls notwendig. Wenn zum Beispiel der Tiler, die Gsprite-Engine und der Zusammensetzungspuffer die aktuelle Zuordnung von Objekten zu Gsprites für ein Teilbild mit der erforderlichen Teilbildauffrischungsrate nicht verarbeiten können, dann können die Daten zurück zum DSP oder Wirtsprozessor geleitet werden, um Objekte zu vereinigen und mehrere Objekte in einen Gsprite zu rendern.

[0387] Fig. 6 liefert zusätzliche Informationen bezüglich der Verarbeitung von Gsprites in einer Ausführungsform. Wie in Fig. 6 gezeigt und oben beschrieben, bestimmt der Bildpräprozessor auch die Tiefenordnung von Gsprites (**280**).

[0388] Wenn der Bildpräprozessor einen Gsprite zuordnet, erzeugt er eine Datenstruktur, um den Gsprite zu repräsentieren. Die Gsprite-Datenstruktur umfasst einen Header zum Speichern verschiedener Attribute des Gsprites und um zu verfolgen, wo die zugehörigen Bilddaten im Speicher gespeichert werden. Die Datenstruktur umfasst Felder zum Speichern der Größe des Gsprites, zum Darstellen der Kantengleichungen für die Kanten des Gsprites, zum Führen der 2D-Transformationsdaten und anderer Bildattribute.

[0389] Nach dem Bestimmen der Gsprite-Konfiguration für den Betrachtungsraum bestimmt der Bildpräprozessor, welche Gsprites gerendert werden sollen. Statt alle Objekte in einer Szene zu rendern, kann das System Gsprites, die für ein anderes Teilbild gerendert wurden, wieder verwenden. Die Änderung in der Position eines Objektes von Teilbild zu Teilbild kann durch Ausführen einer affinen Transformation an einem gerenderten Gsprite genähert werden. Wie in Fig. 6 gezeigt, durchläuft der Bildpräprozessor zyklisch Gsprites (**282**-**286**) und berechnet Gsprite-Transformationen (**284**). Im folgenden Abschnitt behandeln wir weiter ausführlich die Gsprite-Aktualisierung und das Verzerren von Gsprites.

[0390] Das Bildverarbeitungssystem kann Bewegung eines 3D-Objektes durch Ausführen einer affinen Transformation an einem gerenderten 2D-Gsprite, der das Objekt repräsentiert, nähern. Wir bezeichnen den Prozess des Ausführens einer affinen Transformation an einem gerenderten Bild als „Verzerren" (warping) und einen Gsprite, der sich aus diesem Prozess ergibt, als „verzerrten Gsprite" (warped gsprite). In einer Implementierung umfasst der Prozess des Simulierens des 3D-Rendering eines Objektes die folgenden Schritte: 1) Berechnen einer affinen Transformationsmatrix, um die geometrische Bewegung von charakteristischen Punkten zu nähern; 2) Messen der Genauigkeit der Näherung in Schritt 1; und 3) Wenn die Genauigkeit ausreichend ist, dann Ausführen einer affinen Transformation am Gsprite zur Zeit t_0, um seine Position zu einer späteren Zeit t zu nähern.

[0391] Fig. 22 ist ein Flussdiagramm, das den Prozess der Ausführung einer affinen Transformation zur Simulierung der 3D-Bewegung illustriert. Zur Vervollständigung zeigt Fig. 22 „Wähle charakteristische Punkte" als ersten Schritt (**744**). Wie aus der Diskussion unten ersichtlich wird, werden charakteristische Punkte normalerweise nicht während der Bildverarbeitung gewählt, sondern werden vielmehr vom Autor der geometrischen Modells festgelegt.

[0392] Die affine Transformation, die zum Simulieren der Bewegung eines Objektes verwendet wird, wird unter Verwendung charakteristischer Punkte berechnet. Charakteristische Punkte sind Punkte, die für ein Objekt gewählt werden, damit sie seine Position oder andere wichtige Bildmerkmale repräsentieren, wie sie sich mit der Zeit ändern. Da wir uns auf die charakteristischen Punkte in Weltkoordinaten eines 3D-Modells und die Bildschirmkoordinaten des Models beziehen, die in den Bildschirmraum transformiert sind, ist es hilfreich, Begriffe zu klären, die wir verwenden, um diese Punkte zu beschreiben. Wir bezeichnen charakteristische Punkte im Bildschirmraum als „charakteristische Betrachtungspunkte", und wir bezeichnen charakteristische Punkte in Weltkoordinaten als „charakteristische Modellierungspunkte".

[0393] Durch Wählen eines repräsentativen Satzes von charakteristischen Punkten, statt die gesamte Menge der Objektpunkte zu berücksichtigen, vereinfachen wir die Berechnung der affinen Transformation beträchtlich. Die Zahl der charakteristischen Punkte, die benötigt werden, um eine genaue Näherung der 3D-Bewegung eines Objektes zu erhalten, variiert je nach dem Modell. Wenn das Objekt ein starrer Körper ist, können charakteristische Punkte aus einem Begrenzungskasten gewählt werden, der das ganze Objekt umschließt. Wenn die Punkte, die den Begrenzungskasten definieren, mit derselben Transformation transformiert werden, dann folgen die Punkte des Begrenzungskastens der Transformation der Objektgeometrie.

[0394] Für Objekte mit einer komplexeren Bewegung, können mehr charakteristische Punkte erforderlich sein, um eine zutreffende Näherung zu erhalten. Zum Beispiel kann ein Objekt in eine Reihe von starren Körpern unterteilt werden, jeder mit einem Begrenzungskasten, der seine Position nähert. Wenn das Objekt aus einer Hierarchie von starren Körpern mit individuellen Bewegungstransformationen besteht, dann können die charakteristischen Punkte aus der Vereinigung der Ecken der sich bewegenden Teilobjekt-Begrenzungskästen abgeleitet werden.

[0395] Als weitere Alternative kann der Autor des Modells charakteristische Punkte für das Modell festlegen. Dies ermöglicht es dem Autor des Modells, speziell charakteristische Punkte festzustellen, die zum Nähern der 3D-Bewegung des Objekts verwendet werden. Wie ferner unten beschrieben, kann die Genauigkeit der affinen Transformation gemäß einer Reihe von Metriken überprüft werden. Dadurch, dass der Autor die charakteristischen Punkte festlegen kann, kann er Punkte festlegen, die die größte Relevanz für die Metrik oder die Metriken besitzt, die zum Bewerten der Genauigkeit der affinen Transformation verwendet werden.

[0396] Wenn ein Satz von charakteristischen Punkten gegeben ist, kann eine affine Transformation berechnet werden, um die Änderung der Position eines Gsprites von der Zeit t_0 zur Zeit t zu nähern. Dieser Schritt wird als Schritt (**746**) in Fig. 22 illustriert.

[0397] Die affine Transformation wird aus den charakteristischen Betrachtungspunkten zur Zeit t_0 und t berechnet. Je nachdem, wie die charakteristischen Punkte gewählt werden, repräsentieren die charakteristischen Modellierungspunkte Punkte auf einem Objekt oder seinem Begrenzungskasten. Die Position der charakteristischen Modellierungspunkte ändert sich mit der Zeit gemäß der Modellierungstransformation. Um die charakteristischen Betrachtungspunkte zu finden, werden die charakteristischen Modellierungspunkte mit der Betrachtungstransformation multipliziert. Die folgende Diskussion wird bei der Klärung des Prozesses der Berechnung der affinen Transformationsmatrix helfen, die zum Transformieren eines 2D-Gsprites verwendet wird.

[0398] Das Format der affinen Transformationsmatrix ist folgendermaßen:

$$S = \begin{bmatrix} a & b & p_x \\ c & d & p_y \end{bmatrix}$$

[0399] Ein Maß zum Kontrollieren der Näherung ist die Positionsmetrik. Die Positionsmetrik bezieht sich auf die Differenz in der Position zwischen den charakteristischen Punkten zur Zeit t und der Position der charakteristischen Punkte zur Zeit t_0, multipliziert mit der affinen Transformationsmatrix. Die allgemeine Formel der Positionsmetrik ist folgendermaßen:

$\Sigma_i \| \overline{x}^i(t) - S(t)\overline{x}^i(t_0) \|^2$

[0400] Im Fall der Positionsmetrik ist die Position der charakteristischen Punkte im Bildschirmraum am relevantesten, weil der Unterschied in der Position auf dem Bildschirm angibt, wie genau der transformierte Gsprite die Bewegung seines entsprechenden 3D-Modells nähert. Für andere Metriken jedoch kann die Genauigkeit der Näherung in Form der charakteristischen Modellierungspunkte berechnet werden. Für das Beispiel der Positionsmetrik betrachten wir die Bildschirmraumpunkte direkt. Es seien

$\overline{x}^i(t) = V(t)T(t)x^i(t)$

die Bildschirmraumpunkte, wobei V(t) die Betrachtungstransformation und T(t) die Modellierungstransformation ist. Um die affine Transformationsmatrix zu berechnen, kann ein Standardverfahren der kleinsten Quadrate verwendet werden. Löst man das lineare System:

$[\overline{x}'(t_0)]S(t) = \overline{x}^i(t)$

erzeugen die Standardverfahren der kleinsten Quadrate ein Ergebnis, das die Positionsmetrik minimiert.

[0401] Für den Fall, daß es drei charakteristische Punkte gibt, kann die affine Transformationsmatrix direkt gelöst werden. Wenn zum Beispiel drei Punkte auf den Achsen eines Begrenzungskastens verwendet werden, ist das Ergebnis eine geschlossene Lösung für die zeitabhängige affine Transformationsmatrix, wie unten gezeigt.

$$\begin{bmatrix} x^0 & y^0 & 1 \\ x^1 & y^1 & 1 \\ x^2 & y^2 & 1 \end{bmatrix}_{t_0} \begin{bmatrix} a & c \\ b & d \\ p_x & p_y \end{bmatrix}_t = \begin{bmatrix} x^0 & y^0 \\ x^1 & y^1 \\ x^2 & y^2 \end{bmatrix}$$

$$S(t) = \left[X(t_0)\right]^{-1} X(t)$$

$$\left[X(t_0)\ 1\right]^{-1} = \frac{1}{D}\begin{bmatrix} y^1 - y^2 & y^2 - y^0 & y^0 - y^1 \\ x^2 - x^1 & x^0 - x^2 & x^1 - x^0 \\ x^1 y^2 - x^2 y^1 & x^2 y^0 - x^0 y^2 & x^0 y^1 - x^1 y^0 \end{bmatrix}$$

wo $D = x^1y^2 - x^2y^1 + x^2y^0 - x^0y^2 + x^0y^1 - x^1y^0$
ist.

[0402] Im allgemeinen Fall kann ein Verfahren der kleinsten Quadrate, wie zum Beispiel Normalgleichungen oder Einzelwertzerlegung, zum Lösen der affine Transformationsmatrix verwendet werden. Das verallgemeinerte Problem wird unten illustriert:

$$\begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ . & . \\ . & . \end{bmatrix} \begin{bmatrix} a & c \\ b & d \\ t_x & t_y \end{bmatrix} = \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ . & . \\ . & . \end{bmatrix}$$

[0403] Um eine Lösung für die affine Transformationsmatrix zu erhalten, muss die Pseudoinverse einer N × 3-Matrix berechnet werden. Für eine beliebige Zahl von charakteristischen Punkten verwenden wir ein Verfahren der kleinsten Quadrate zum Auflösen nach der Pseudoinversen. In einer Ausführungsform wird das Normalgleichungsverfahren verwendet.

[0404] $\tilde{X}$ sei die transponierte Matrix der charakteristischen Punkte zur Zeit t_0, und X sei die transponierte Matrix der charakteristischen Punkte zur Zeit t.

$$[\tilde{X}\ 1]S = X$$

$$\begin{bmatrix} \tilde{x}^0 & \tilde{y}^0 & 1 \\ \tilde{x}^1 & \tilde{y}^1 & 1 \\ \vdots & \vdots & \vdots \\ \tilde{x}^{n-1} & \tilde{y}^{n-1} & 1 \end{bmatrix} \begin{bmatrix} a & c \\ b & d \\ p_x & p_y \end{bmatrix} = \begin{bmatrix} x^0 & y^0 \\ x^1 & y^1 \\ \vdots & \vdots \\ x^{n-1} & y^{n-1} \end{bmatrix}$$

[0405] Für eine Lösung mit dem Verfahren der Normalgleichungen werden beide Seiten der Gleichung mit der Transponierten der geeigneten Matrix multipliziert, und dann wird die sich ergebende hermitesche invertiert. Die typische Schwäche der Normalgleichungen ist, dass die resultierende Matrix singulär ist oder zur Instabilität auf Grund des Rundungsfehlers neigt. Die Matrix ist singulär, wenn die charakteristischen Punkte entartet sind. In der speziellen Form der Matrix kann der Rundungsfehler durch Normalisieren der Terme beherrscht werden.

$[\tilde{X}1]^T[\tilde{X}1]S = [\tilde{X}1]^T X$

$$\begin{bmatrix} \sum_i \tilde{x}^i \tilde{x}^i & \sum_i \tilde{x}^i \tilde{y}^i & \sum_i \tilde{x}^i \\ \sum_i \tilde{x}^i \tilde{y}^i & \sum_i \tilde{y}^i \tilde{y}^i & \sum_i \tilde{y}^i \\ \sum_i \tilde{x}^i & \sum_i \tilde{y}^i & 1 \end{bmatrix} \begin{bmatrix} a & c \\ b & d \\ p_x & p_y \end{bmatrix} = \begin{bmatrix} \sum_i \tilde{x}^i x^i & \sum_i \tilde{x}^i y^i \\ \sum_i \tilde{y}^i x^i & \sum_i \tilde{y}^i y^i \\ \sum_i x^i & \sum_i y^i \end{bmatrix}$$

[0406] Es gibt nur 5 Terme in der resultierenden Matrix. Die 3×3-Matrix wird dann invertiert, um die affine Transformation zu erhalten. Da andererseits die Summe der Terme der x-Koordinaten und die Summe der y-Koordinaten dem Schwerpunkt der charakteristischen Punkte entspricht, können diese Terme durch eine Änderung des Koordinatensystems eliminiert werden, um den Schwerpunkt auf 0,0 zu verschieben. Die resultierende Matrix ist vom Typ 2 × 2 und leicht zu invertieren.

[0407] Nach dem Berechnen der affinen Transformationsmatrix wird die Genauigkeit der Näherung unter Verwendung von einer oder mehreren Metriken kontrolliert. Der Entscheidungsschritt (**748**) von **Fig.** 18 illustriert den Schritt der Kontrolle von einer oder mehreren Metriken und zeigt im allgemeinen, wie die Logik sich auf der Basis der Metrik/Metriken verzweigt. Wie oben beschrieben, ist die Positionsmetrik ein Beispiel dafür, wie die Genauigkeit der affinen Transformation kontrolliert werden kann. Um zu messen, ob die affine Transformation die Positionsmetrik erfüllt, werden die charakteristischen Betrachtungspunkte zur Zeit t_0, die unter Verwendung der berechneten affinen Transformation transformiert wurden, mit den charakteristischen Betrachtungspunkten zur Zeit t verglichen.

[0408] Ein weiterer Ansatz besteht darin, die innere Drehung des 3D-Modells als Metrik zu verwenden. In diesem Fall werden die charakteristischen Modellierungspunkte zur Zeit t_0, die unter Verwendung der berechneten affinen Transformation transformiert wurden, mit den charakteristischen Modellierungspunkten zur Zeit t verglichen.

[0409] Noch ein weiterer Ansatz besteht darin, eine Beleuchtungsmetrik zu verwenden. Wie bei der Metrik für die innere Drehung, werden die charakteristischen Modellierungspunkte dazu verwendet, die Genauigkeit der Näherung zu kontrollieren.

[0410] Neben den Metriken, die oben beschrieben werden, gibt es eine Vielzahl anderer Alternativen. Um diese Metriken zu berechnen, können relevante charakteristische Daten zusammen mit den charakteristischen Punkten geführt werden. Je nach der gewünschten Genauigkeit kann eine einzelne Metrik, oder eine Kombination von Metriken verwendet werden.

[0411] Wenn die charakteristischen Punkte, die den transformierten Gsprite repräsentieren, ausreichend genau sind, dann kann der transformierte Gsprite an Stelle eines neu gerenderten Gsprites verwendet werden. Um die 2D-Transformation zu berechnen, wird der Gsprite für die Zeit t_0 mit der affinen Transformationsmatrix multipliziert (**750**). Im Gegensatz zum Rendern des Gsprites braucht diese Berechnung beträchtlich weniger Verarbeitungszeit. Das Simulieren der 3D-Bewegung mit einer 2D-Transformation kann daher den Verarbeitungsaufwand, der zum Rendern eines Bildes erforderlich ist, beträchtlich reduzieren.

[0412] Auf der Basis der Näherungsgenauigkeit kann das System den Zusatzaufwand beim Rendern nach Bedarf reduzieren, um innerhalb der Grenzen seiner Renderingkapazität für ein Teilbild der Bilddaten zu bleiben. Um das Konzept allgemein zu illustrieren, zeigt **Fig. 22**, dass ein Gsprite neu gerendert wird, wenn die 2D-Transformation nicht ausreichend genau ist (**754**). Wie in weiteren Details unten beschrieben wird, ist es jedoch nicht notwendigerweise vorzuziehen, einen Gsprite auf der Basis einer Metrik zu akzeptieren oder abzulehnen. Es ist vielmehr oft nützlich zu bestimmen, wie genau die Näherung für eine Reihe von Gsprites in einer Szene ist und so viele Gsprites wie möglich neu zu rendern.

Farbverzerrung von Gsprites

[0413] Als weitere Optimierung kann das Renderingsystem die Beleuchtungsänderungen von Teilbild zu Teilbild abfragen und die Farbwerte des Gsprite modifizieren, um diese Änderungen zu nähern. Dieser Ansatz umfasst drei Grundschritte: 1) Abfragen der Beleuchtungsänderung zwischen den Teilbildern; 2) Bestimmen, wie die Farbwerte im Gsprite zu ändern sind, um die Beleuchtungsänderung zu nähern (d.h. eine Farbverzerrung zu berechnen); und 3) falls ausreichend genau, Ausführen einer Farbverzerrung am Gsprite, um die Beleuchtungsänderung zu nähern. Wenn nach der Bewertung der Beleuchtungsgleichung der Bildpräprozessor feststellt, dass sich die Beleuchtung um mehr als einen vorgegebenen Betrag verändert hat, dann weist er den Tiler an, das Objekt neu zu rendern.

[0414] Im ersten Schritt testet das Renderingsystem die Beleuchtungsänderung für ein Objekt, das mit dem Gsprite verknüpft ist. Es testet die Beleuchtungsänderung zwischen einem ersten Teilbild, in dem ein Objekt in einen Gsprite gerendert wird, und einem nachfolgenden Teilbild, in dem das Renderingsystem versucht, den Gsprite Farb-zu-verzerren, um die Beleuchtungsänderung zu nähern. Eine Möglichkeit zum Testen der Beleuchtungsänderung besteht darin, die Beleuchtungsgleichung an charakteristischen Punkten mit Normalen für das erste Teilbild und das nachfolgende Teilbild zu testen und die Ergebnisse des Tests bei jedem der Teilbilder zu vergleichen. Die charakteristischen Punkte sollten vorzugsweise auf dem Objekt verteilt sein, um einen genauen Test der Beleuchtungsänderung über den Gsprite zu liefern. Die spezielle Zahl und der Ort von charakteristischen Punkten kann variieren und ist im allgemeinen modellspezifisch.

[0415] Ein Beispiel für die Beleuchtungsgleichung ist:

$$I_\lambda = I_{a\lambda}k_\theta O_{d\lambda} + f_{att}I_{p\lambda}[k_d O_{d\lambda}(N\cdot L) + k_S O_{d\lambda}(R\cdot V)^n]$$

wo:
$I_{a\lambda}$ das Umgebungslicht ist,
k_θ der Umgebungsreflexionskoeffizient ist,
$O_{d\lambda}$ die diffuse Farbe des Objekts ist,
f_{att} der Lichtquellenabschwächungsfaktor ist, der beschreibt, wie sich die Lichtenergie verringert, wenn es sich von der Lichtquelle weiter wegbewegt.

$I_{p\lambda}$ das Licht von einer Punktlichtquelle ist,
k_d der diffuse Reflektionskoeffizient, eine Konstante zwischen 0 und 1, die von Material zu Material variiert, ist,
$O_{d\lambda}$ die Spiegelreflektionsfarbe des Objekts ist,
k_S der Spiegelreflektionskoeffizient des Materials ist, der im Bereich zwischen 0 und 1 liegt.

[0416] (N·L) ist das Skalarprodukt der Oberflächennormale N und der Richtung der Lichtquelle L.

[0417] (R·V) ist das Skalarprodukt zwischen der Richtung der Reflektion R und der Richtung zum Betrachtungspunkt V, der Exponent n ist der Spiegelreflektionsexponent des Materials, der normalerweise zwischen 1 und mehreren hundert variiert.

[0418] λ zeigt an, dass ein Term, der diesen Index aufweist, wellenlängenabhängig ist. Eine Annahme zum Vereinfachen der Beleuchtungsgleichung besteht darin anzunehmen, dass das RGB-Farbmodell die Wechselwirkung des Lichts mit Objekten ausreichend modellieren kann. Unter Verwendung dieser Annahme kann das Beleuchtungsmodell auf jede R-, G- und B-Farbkomponente angewendet werden.

[0419] Die Beleuchtungsgleichung oben ist nur ein Beispiel, das ein Verfahren zum Berechnen der Beleuchtung an Punkten auf der Oberfläche eines Objektes illustriert. Die Beleuchtungsgleichung kann zum Beispiel durch Nichtbeachten des Lichtabschwächungsfaktors oder der Spiegelreflektion vereinfacht werden. Auf dem Gebiet des 3D-Grafikrenderns gibt es eine Reihe von anderen konventionellen Beleuchtungsgleichungen, die zum Modellieren der Beleuchtung auf der Oberfläche eines grafischen Objektes verwendet werden. Daher kann jede beliebige Zahl von unterschiedlichen Beleuchtungsgleichungen verwendet werden, um die Beleuchtung an charakteristischen Punkten zu testen, die mit einem grafischen Objekt verknüpft sind. Im allgemeinen berechnet der Bildpräprozessor die Beleuchtungsgleichung und bestimmt, wie der resultierende Beleuchtungswert I (möglicherweise für jede RGB-Komponente) sich in der Größe von Teilbild zu Teilbild ändert.

[0420] Um die Änderung in der Beleuchtung von Teilbild zu Teilbild einzuschätzen, berechnet der Bildpräprozessor die Beleuchtungsgleichung für charakteristische Punkte in einem ersten Teilbild und einem nachfolgenden Teilbild unter Verwendung der Flächennormale am charakteristischen Punkt, der Richtung der Lichtquelle für jedes Teilbild und möglicherweise anderen Daten, die mit der speziellen Beleuchtungsgleichung verbunden sind.

[0421] Das System kann die Beleuchtungsänderung an charakteristischen Punkten auf einem Objekt, das durch den Gsprite repräsentiert wird, oder an charakteristischen Punkten auf einem Begrenzungsvolumen des Objektes testen. Ein Ansatz für das Testen der Beleuchtungsänderung besteht darin, die Beleuchtungsänderung auf der Oberfläche eines Begrenzungsvolumens des Objektes zu testen. Das System kann zum Beispiel Beleuchtungsänderungen an den Normalen auf der Oberfläche einer Begrenzungskugel des Objektes oder Teilen des Objektes testen. Eine Begrenzungskugel ermöglicht dem Bildpräprozessor, die wesentlichen Veränderungen zu verfolgen, die auf Grund dessen, dass eine lokale Lichtquelle innerhalb des „Raums" eines Objektes bewegt wird, auftreten können. Wenn der Bildpräprozessor einfach einen Satz von Vektoren verwendete, die sich am Schwerpunkt eines Objektes befinden, könnte die Bewegung einer lokalen Lichtquelle keine wesentlichen Beleuchtungsänderungen bewirken, sondern könnte einen Einfluss auf die Beleuchtung des Objektes als Ganzes haben. Unter diesen Umständen kann das Testen von Beleuchtungsänderungen auf der Oberfläche einer Begenzungskugel die Beleuchtungsänderungen für das Objekt genauer erfassen, die ansonsten durch selektives Betrachten von charakteristischen Punkten auf der Oberfläche des Objektes verlorengehen würden.

[0422] Als weitere Alternative kann eine Kombination von Normalen an den charakteristischen Punkten auf dem Objekt oder auf der Oberfläche einer Begrenzungskugel verwendet werden, um die Beleuchtungsänderungen abzufragen. Dieser Ansatz kann Beleuchtungsänderungen effektiver verfolgen, weil er die Beleuchtungsänderungen an charakteristischen Punkten auf dem Objekt und auf der Oberfläche eines Begrenzungsvolumens für das Objekt verfolgt.

[0423] Auf der Basis der Beleuchtungsänderungen kann das System bestimmen, wie die Gsprite-Farbwerte zu ändern sind, um diese Beleuchtungsänderungen zu nähern. Ähnlich wie bei der Geometrietransformation, die an einem Gsprite ausgeführt wird, berechnet das System, wie die Farbwerte des Gsprites zu ändern sind, um die Beleuchtungsänderung näherungsweise zu berücksichtigen. Eine Möglichkeit zur Berechnung der Farbverzerrung besteht darin, einen Anpassungsansatz nach der Fehlerquadratmethode zu verwenden, wie oben beschrieben. Das Ergebnis dieses Schritts ist eine konstante, lineare Änderung oder eine Änderung höherer Ordnung, mit der die Farbwerte an den Pixelorten über den Gsprite modifiziert (z.B. mit einem Skalie-

rungsfaktor multipliziert und/oder einem Versatz versehen) werden.

[0424] Die Farverzerrung umfasst einen Multiplikator oder ein Feld von Multiplikatoren, die über den Gsprite hinweg angewendet werden. Im einfachsten Fall kann die Farbverzerrung einfach ein konstanter Maßstabsfaktor sein, der auf alle Pixel im Gsprite angewendet wird. Ein genauerer Ansatz ist, eine Verzerrung mit linearer Änderung oder einer höheren Ordnung zu verwenden, um die Beleuchtungsänderungen anzunähern. Der Multiplikator ist vorzugsweise ein Vektorwert, so dass die Farbkomponenten unabhängig skaliert werden können. Um die Änderungen durch farbige Lichtquellen genau zu modellieren, sollte jede Farbkomponente unabhängig skaliert werden.

[0425] Zusätzlich zum Multiplikator kann auch ein Versatzwert, der zu einem Farbwert im Gsprite addiert wird, auf der Basis der Beleuchtungsänderungen an den charakteristischen Punkten berechnet werden.

[0426] Eine Möglichkeit, den Multiplikator und die Versatzwerte zu berechnen, ist nach einem Multiplikator und dem Versatzwert aufzulösen, der die Änderung in der Beleuchtungsgleichung an jedem charakteristischen Punkt repräsentiert, um festzustellen, ob die charakteristischen Punkte sich an der Oberfläche des Objektes, an der Oberfläche eines Begrenzungsvolumens oder von beiden befinden. Der Bildpräprozessor kann einen Multiplikator, einen Versatzwert oder beide durch Wählen eines Multiplikators oder Versatzwerts oder einer Kombination aus einem Multiplikator und einem Versatzwert berechnen, die dieselbe oder im wesentlichen dieselbe Änderung der Beleuchtungsgleichung an jedem charakteristischen Punkt bewirkt, wie während des Teststadiums beobachtet wurde. Sobald diese Multiplikatoren und/oder Versatzwerte berechnet sind, gibt es eine Reihe von Möglichkeiten, den Multiplikator und die Versatzwerte zu berechnen, die auf die Farbwerte im Gsprite angewendet werden. Eine Möglichkeit besteht darin, die Multiplikatoren zu mitteln, um einen einzigen Skalierungsfaktor für den Gsprite abzuleiten. Eine andere Möglichkeit besteht darin, die Versatzwerte zu mitteln, um einen einzigen Versatzwert für den Gsprite abzuleiten. Noch eine weitere Möglichkeit ist, eine Anpassung nach der Fehlerquadratmethode auf den Multiplikator und die Versatzwerte unabhängig anzuwenden, um Ausdrücke abzuleiten, die darstellen, wie sich die Multiplikatoren und die Versatzwerte mit dem Ort auf der Oberfläche des Objekts ändern. Dieser Ausdruck kann in Hardware unter Verwendung von Interpolatoren implementiert werden, um unabhängige Multiplikatoren und/oder Versatzwerte für Pixelorte im Gsprite zu berechnen. Die Gsprite-Engine kann zum Beispiel einen Rastergenerator mit Interpolatoren zum Interpolieren der Multiplikatoren und/oder Versatzwerte für jeden Pixelort umfassen, bevor ein Farbwert mit dem Multiplikator multipliziert wird oder bevor ein Versatzwert zu einem Farbwert oder einem skalierten (d.h. skaliert durch den entsprechenden Multiplikator, der für den Pixelort berechnet wurde) Farbwert addiert wird.

[0427] Gerade so wie das System die Genauigkeit der geometrischen Änderung bewertet, kann das System auch die Genauigkeit der Farbverzerrung durch Vergleichen der Farbwerte, die durch Farbänderung berechnet wurden, mit entsprechenden Farbwerten, die für das aktuelle Teilbild unter Verwendung des normalen Renderingprozesses berechnet wurden, bewerten. Wenn die Farbwerte sich um mehr als eine vorgegebene Toleranz unterscheiden, sollte der Gsprite neu gerendert werden.

[0428] Neben der Verringerung des zusätzlichen Renderingaufwandes kann das Verzerren der Gsprites die Transportverzögerung reduzieren. In Anwendungen, in denen sich die Betrachtungsperspektive schnell ändert, ist es auf Grund der Transportverzögerung schwierig, die sich schnell ändernde Perspektive anzuzeigen. Transportverzögerung bezieht sich auf die Verzögerung, die zwischen dem Erhalt der Eingabe, welche eine Änderung des Betrachtungspunktes bewirkt, und dem schließlichen Anzeigen des entsprechenden Bildes für diesen neuen Betrachtungspunkt auftritt. Fig. 23 illustriert ein Beispiel dafür, wie die Transportverzögerung reduziert werden kann. Die Abschnitte entlang der horizontalen Achse repräsentieren Zeitschritte entsprechend der Teilbildverzögerung.

[0429] In diesem Beispiel gibt es eine Verzögerung von drei Teilbildperioden zwischen der Abfrage der Eingabe und der Anzeige der Ausgabe auf einem Anzeigegerät. Zuerst wird die Eingabe in einem ersten Teilbild **774** abgefragt. Als Nächstes berechnet der Computer die affinen Transformationen und rendert Objekte in den Gsprites **776**. Zum Schluss werden die gerenderten Bilddaten für das Teilbild zusammengesetzt und auf das Anzeigegerät ausgegeben **778**. Obwohl die Zeit, die zum Ausführen jedes dieser Schritte nicht notwendigerweise eine ganze Teilbildverzögerung ist, die durch eine Teilbildperiode gemessen wird, verwenden wir den Zuwachs einer Teilbildperiode, um das Konzept zu illustrieren. Wie illustriert, liegen drei Teilbildperioden der Verzögerung zwischen Eingabe und Anzeige der entsprechenden Bilddaten.

[0430] Um die Transportverzögerung zu reduzieren, können Betrachtungspunktdaten aus einem nachfolgenden Bild auf die Renderingphase des aktuellen Bildes angewendet werden **782**. Dies wird durch den Pfeil von

der Eingabephase für ein nachfolgendes Bild **782** auf die Gsprite-Transformation und die Renderingphase **776** des aktuellen Bildes illustriert. Die Verarbeitungsschritte (**782**, **784**, **780**) für das nächste Teilbild der Bilddaten werden neben den Schritten (**776**, **778**) gezeigt, wie in Fig. 23 zu sehen ist. Wie illustriert, läuft die Verarbeitung in Pipelineart ab. Eingaben werden nach einem nachfolgenden Teilbild abgefragt, während die Gsprite-Transformationen berechnet und das Rendern für das aktuelle Teilbild ausgeführt wird.

[0431] Die Modellierungstransformation für das aktuelle Bild kann in Verbindung mit der Betrachtungstransformation für das nachfolgende Bild verwendet werden, um eine Gsprite-Transformation zu berechnen, die normalerweise in der Form einer affinen Transformationsmatrix ist. Ein gerenderter Gsprite kann dann verzerrt werden, um so seine Position relativ zum Betrachtungspunkt des nachfolgenden Bildes zu simulieren. Dieser Ansatz verringert den Effekt der Transportverzögerung auf den Nutzer, weil er dem System ermöglicht, sich schneller auf schnelle Änderungen in der Betrachtungspunktperspektive einzustellen.

[0432] Neben der Reduzierung der Transportverzögerung in diesem Kontext, können nachfolgende Bilddaten dazu verwendet werden, die Transportverzögerung auch in anderen Zusammenhängen zu reduzieren.

[0433] Wie oben angedeutet wurde, gibt es eine Reihe von Vorteilen für das unabhängige Rendern der Gsprites. Gsprites können unterschiedliche Aktualisierungsraten haben und daher variiert die Zahl der Gsprites, die in einem bestimmten Teilbild aktualisiert wird. Einige Gsprites müssen vielleicht bei jedem Teilbild aktualisiert werden, während andere Gsprites möglicherweise weniger häufig aktualisiert werden. Wenn eine Reihe von Gsprites in einem bestimmten Teilbild aktualisiert werden muss, kann sich der Rendering-Zusatzaufwand dramatisch erhöhen und das System belasten. Um dieses Problem anzugehen, führt das System ein Prioritätsrendering aus, was ihm ermöglicht, das Rendern über eine Reihe von Teilbildern zu verteilen und Gsprites effizienter zu verarbeiten.

[0434] Ohne Prioritätsrendering kann die Zahl der Gsprites, die für das Rendern in einem bestimmten Teilbild eingeteilt sind, variieren. Einige Gsprites können zum Beispiel vorgegebene Aktualisierungsraten haben. Die Aktualisierungsrate für einen Gsprite kann je nachdem variieren, ob er im Vordergrund oder Hintergrund einer Szene ist. Mit der Unterstützung für affine Verzerrungen, die oben beschrieben sind, kann das System mit einer affinen Transformation das erneute Rendern eines Gsprites durch Simulieren einer Positionsänderung vermeiden. Im Fall von affinen Verzerrungen kann die Notwendigkeit, einen Gsprite neu zu rendern, variieren, je nachdem, ob sich die Szene ändert.

[0435] Um das Prioritätsrendern zu implementieren, priorisiert das System das Rendern nach dem Grad der Verzerrung, die sich durch Wiederverwenden eines gerenderten Gsprites ergeben würde. Die Verzerrung wird auf der Basis von einer oder mehreren Fehlerschwellen berechnet. Um die Verzerrung eines Gsprites zu quantifizieren, misst das System, wie nahe, oder umgekehrt, wie weit ein Gsprite von seiner Fehlerschwelle entfernt ist. Die Fehlerschwelle kann für jeden Gsprite variieren und kann auf einem oder mehreren Faktoren beruhen. Ein nach Verzerrung geordnete Liste von Gsprites wird geführt, die die relative Qualität der Gsprites vor dem erneuten Rendern darstellt. Dann werden so viele Gsprites in einem Teilbild wie möglich unter Berücksichtigung der Systemressourcen neu gerendert. Gsprites werden neu gerendert, beginnend bei dem am stärksten verzerrten Gsprite, und dann weiter in absteigender Reihe zu den weniger verzerrten Gsprites. Das Verarbeiten in dieser Weise beseitigt die Möglichkeit einer Teilbildüberladung durch das Rendern der Gsprites, während statt dessen ein effizienter Mechanismus zum Ausbalancieren von Szenenkomplexität und Bewegung mit der Gsprite-Genauigkeit bereitgestellt wird.

[0436] Neben den Merkmalen, die oben beschrieben werden, ermöglicht das Rendern in Gsprites dem System, die Auflösung von Objekten in einer Szene zu variieren. Dies gibt dem System die Möglichkeit, Gsprites Verarbeitungs- und Speicherressourcen auf der Basis ihrer Bedeutung in einer Szene zuzuweisen.

[0437] Die Kosten eines Gsprites können in Form von Speicher, den er belegt, und der Verarbeitung, die zu seinem Rendern erforderlich ist, gemessen werden. Beide Kosten hängen stark von der Zahl der Pixel im Gsprite-Bild ab. Wenn Gsprite-Bilder bei einer festen Auflösung, der Bildschirmauflösung, gespeichert und gerendert werden, werden die Kosten, die für ein Gsprite anfallen, von seinen Bildschirmabmessungen bestimmt.

[0438] Es ist wichtig, Verarbeitungs- und Speicherressourcen auf der Basis der Art und des Ortes eines Objektes, statt lediglich seiner Größe, die es auf dem Bildschirm einnimmt, zuzuordnen. Aktive Objekte im Vordergrund einer Szene sind normalerweise wichtiger für die Szene als die im Hintergrund. Wenn jedoch dem Gsprite Ressourcen auf der Basis der Größe zugeordnet werden, dann sind die Verarbeitungs- und Speicherkosten für den Hintergrund auf Grund seiner größeren Bildschirmabmessungen viel größer.

[0439] Das System kann die Bildschirmauflösung von der Auflösung des Gsprites entkoppeln, so dass die Kosten eines Gsprites unabhängig von seiner endgültigen Bildschirmauflösung festgelegt werden können. Das System erreicht dies durch Wählen der geeigneten Auflösung des Gsprites und dann durch Skalieren des Gsprites auf die entsprechende Größe.

[0440] Der Vergrößerungs- oder Skalierungsfaktor kann aus den Bildschirmabmessungen des Bildes und der Gsprite-Auflösung abgeleitet werden. Normalerweise liefert die Grafikanwendung die Bildschirmabmessungen. Die Grafikanwendung kann auch die Auflösung angeben. Alternativ kann der Bildpräprozessor die Gsprite-Auflösung auf der Basis der verfügbaren Ressourcen und der relativen Bedeutung des Gsprites in der Szene bestimmen.

[0441] Beim Betrieb rendert der Bildprozessor den Gsprite in einen kleineren Bereich in Ausgabegerätekoordinaten, als er tatsächlich im Betrachtungsraum einnimmt. Die Größe des Bereichs, in den der Gsprite gerendert wird, leitet sich aus der Auflösung und den Bildschirmabmessungen ab. Der gerenderte Gsprite kann dann auf seine tatsächliche Größe, wie durch seine Bildschirmabmessungen definiert sind, skaliert werden. Da der Gsprite einen kleineren Bereich hat, verbraucht er weniger Speicher und weniger Verarbeitungsressourcen zum Rendern. Außerdem können in der erläuterten Ausführungsform Gsprites mit wechselnden Auflösungen immer noch in einer gewöhnlichen Grafikpipeline verarbeitet werden.

[0442] Eine Möglichkeit, diesen Ansatz zu unterstützen, besteht darin, den Vergrößerungs- oder Skalierungsfaktor in der Gsprite-Datenstruktur zu speichern. Der Skalierungsfaktor kann dann dazu verwendet werden, den Gsprite zu skalieren, bevor er mit anderen Gsprites zusammengesetzt wird, um das Anzeigebild zu erzeugen. Der Bildpräprozessor kann die Skalierung des Gsprites ausführen. Spezieller gesagt, skaliert in der Implementierung, die oben beschrieben wird, der DSP den Gsprite.

[0443] So wie ein Gsprite skaliert werden kann, um die Auflösung zu reduzieren, kann er auch auf eine Größe gerendert und dann auf eine kleinere Anzeigefläche skaliert werden. Dieses Verfahren kann auf Objekte in einer Szene angewendet werden, die in der Größe schwinden. Statt das Objekt für jedes Teilbild neu zu rendern, kann das System den Gsprite skalieren, der das Objekt repräsentiert. Dieser Ansatz kann auch durch Speichern des Skalierungsfaktors in der Gsprite-Datenstruktur implementiert werden.

[0444] Oben haben wir die Gsprite-Verarbeitung durch ein Bildverarbeitungssystem beschrieben, und wir haben auch beschrieben, wie eine Gsprite-Transformation berechnet und in einem Bildverarbeitungssystem angewendet werden kann. Wir beschreiben jetzt detaillierter, wie man Pixeldaten transformieren, zusammensetzen und anzeigen kann.

[0445] In dieser Ausführungsform richtet der DSP **176** die Gsprite-Datenstrukturen ein und speichert sie im gemeinsamen Speicher **216** auf der Bildverarbeitungsplatine **174**. Der DSP liest und schreibt in die Gsprite-Engineregister durch den Tiler über eine speicherabgebildete Schnittstelle. Die Register in der Gsprite-Engine umfassen einen Zeiger auf die aktuelle Anzeigeliste. Weitere Details zur Gsprite-Engine **436** werden oben unter Bezugnahme auf **Fig.** 12 bereitgestellt.

[0446] Die primäre Eingabe in die Gsprite-Engine **204** erfolgt über die Gsprite-Anzeigeliste. Fig. 24 illustriert ein Beispiel für eine Anzeigeliste **800** und Gsprite-Datenstrukturen. In dieser Implementierung umfasst die Anzeigeliste **800** ein Feld von Gsprite-Steuerblockadressen, die SCB (Sprite-Kontrollblock)-Handles **840** genannt werden, wobei auf jeden eine Bandmaske **802** folgt. Das erste Wort in der Liste **800** umfasst die Anzahl von Gsprites in der Liste. Ein gesetztes Bit in der Bandmaske zeigt an, dass der Gsprite im Band vorhanden ist. Obwohl wir hier ein spezielles Beispiel bereitstellen, kann die Anzeigeliste auf andere Weise implementiert werden. Die Liste kann zum Beispiel aus separaten Listen für jedes einzelne Band bestehen, wobei jede Bandliste die Gsprites aufzählt, die auf dieses Band fallen. Wie oben bemerkt, sind die Gsprites in der Anzeigeliste nach Tiefe geordnet und in diesem Fall sind sie von vorn nach hinten geordnet.

[0447] Der Gsprite-Kontrollblock (SCB) **806** umfasst Informationen zum Scannen des Gsprites in Ausgabegerätekoordinaten. Rechteckige Gsprites bilden sich auf ein Parallelogramm im Bildschirmraum unter einer affinen Transformation ab. Die Kantengleichungen des Gsprites haben die Form: $A_0x + B_0y + C_0 = F_0$; $A_1x + B_1y + C_1 = F_1$; $-A_0x - B_0y + C_2 = F_2$; $-A_1x - B_1y + C_3 = F_3$. Die rechte Seite dieser Gleichungen ist an den entsprechenden Kanten gleich null. Der DSP **176** bestimmt den Wert der Koeffizienten aus der affinen Transformation für den Gsprite. Nach der affinen Transformation ist die Form des Gsprites ein Parallelogramm, und daher brauchen nur 2 Sätze von A- und B-Koeffizienten gespeichert zu werden. Die C-Terme werden überhaupt nicht gebraucht, da die Gsprite-Engine nur die F-Werte am Startpunkt benötigt, und sie braucht auch eine Beschrei-

bung, wie sich die F-Werte mit Schritten im Bildschirmraum X und Y ändern, was durch die A- und B-Koeffizienten angegeben wird. Zur Unterstützung der Abbildung von gespeicherten Gsprite-Daten in Ausgabegerätekoordinaten wird das Vorzeichen des Koeffizienten so gesetzt, dass das Ergebnis eine positive Zahl ist, wenn die Koordinaten eines Punktes im Parallelogramm in der Kantengleichung ermittelt werden.

[0448] Speziell umfasst der SCB A_0, B_0; A_1, B_1; F_0, F_1, F_2, F_3; den am weitesten links liegenden Punkt xs, ys; den am weitesten rechts liegenden Punkt xf, yf; die Neigung des am weitesten links liegenden Punktes gegenüber der Oberseite des Gsprites und die Neigung des am weitesten links liegenden Punktes gegenüber dem Boden und die Breite und Höhe des Parallelogramms.

[0449] Der Startpunkt für den Scan ist der am weitesten links liegende Punkt des Parallelogramms, und der Scan bewegt sich Spalte für Spalte von links nach rechts im Bildschirmraum. Um den Gsprite auf jedes 32-Scanzeilenband zu beschneiden, umfasst der SCB auch die Neigungen dx/dy vom Startpunkt (dem am weitesten links liegenden Punkt) zu den oberseitigen und bodenseitigen Punkten des Gsprites, so dass der am weitesten links liegende Punkt in einem bestimmten Bildschirmband bestimmt werden kann.

[0450] Die Kantengleichungen des Parallelogramms sind im DSP **176** derart normalisiert worden, dass F = 0 an einer Kante des Parallelogramms und F = Gsprite-Breite oder -Höhe an der gegenüberliegenden Kante gilt. Die F-Werte für die Kanten 0 und 1 des Parallelogramms können daher direkt verwendet werden, um ein Sample S, T eines bestimmten Gsprite-Bildes an einem bestimmten Bildschirmort X, Y nachzusehen. Da die Abbildung von einem Bildschirm X, Y auf einen Gsprite S, T selten direkt auf einer Gsprite-Bildabfrage landen wird, interpoliert die Gsprite-Engine die nächstgelegenen 4-(oder 16-) Gsprite-Bildabfragen, um das Ausgabesample zu finden.

[0451] Der SCB **806** umfasst die Größe des ursprünglichen Gsprites (horizontale und vertikale Schrittweite) und die Größe und den Ort des Teil-Gsprites, der gescannt werden soll (Breite, Höhe, Start S und T). Er kann auch Kennzeichen (Flags) umfassen, die beschreiben, wie die Bildchunks komprimiert wurden und welches Pixelformat in den Chunks verwendet wird.

[0452] Bei dieser Chunking-Architektur wird der Gsprite in 32×32-Pixelchunks unterteilt. Es ist nicht notwendig, Gsprites zum Rendern in Chunks zu teilen. Eine Chunking-Architektur besitzt aber eine Reihe von Vorteilen, wie oben dargelegt. Zur Unterstützung der Chunking-Architektur umfasst der SCB ein zweidimensionales Feld von Zeigern (Chunk-Handles), das die Adresse für das erste Wort des komprimierten Chunks im gemeinsamen Speicher repräsentiert. Der Chunkspeicher wird in 512-Bit-Blöcken gehandhabt. Jeder Zeiger oder Chunk-Handle hat 18 Bit, was einen Gesamtwert von 16 MB addressierbarer Speicher ermöglicht. Da die Menge an Speicher, die zum Komprimieren jedes Chunks benötigt wird, variabel ist, enthält jeder 512-Bit-Block einen 18-Bit-Zeiger auf den nächsten Block. Blöcke, die nicht mehr benötigt werden, werden einer verlinkten Liste von freien Blöcken hinzugefügt, so dass sie für andere Chunks verwendet werden können.

[0453] Wenn Objekte, die einem Gsprite zugeordnet sind, in Chunks geteilt werden, wird die Gsprite-Datenstruktur aktualisiert, so dass sie einen Verweis auf die Chunks enthält, die Bilddaten für den Gsprite umfassen.

[0454] Gsprite-Daten können aus einem anderen Gsprite instanziert werden. In dem Beispiel, das in Fig. 20 gezeigt wird, instanziert ein Gsprite Bilddaten von einem anderen. Hier zeigt der erste Chunk-Handle (**808**) für den SCB auf den SCB **810** eines anderen Gsprites. In einer alternativen Implementierung zeigen Chunk-Handles nur auf Orte im Speicher, an denen Chunks gespeichert sind.

[0455] Fig. 25 ist ein Beispiel, das illustriert, wie ein Gsprite von 6 Chunks mal 2 Chunks sich auf horizontale Bänder in der Anzeige abbilden könnte. Fig. 25 zeigt den Startpunkt **836** und den Endpunkt **834**, die beim Scannen von Bilddaten aus dem Gsprite-Raum auf den physischen Ausgabegerätraum verwendet werden. Wir erklären unten mit mehr Details, wie Gsprite-Bilddaten auf den Ausgabegeräteraum abgebildet werden.

[0456] Nach dem Rendern und der Berechnung der affinen Transformationen für Gsprite in einem Teilbild führt der Bildprozessor dann die Erzeugung der Anzeige aus. Wie in Fig. 21B gezeigt, transformiert der Bildprozessor Gsprites in physische Ausgabekoordinaten und setzt die Gsprites zusammen. Nach dem Zusammensetzen der Pixeldaten transferiert der Bildprozessor diese zur Anzeige.

[0457] In dieser Ausführungsform liest die Gsprite-Engine in der Anzeigeliste und bildet das Gsprite-Bild auf die Ausgabegerätekoordinaten ab. Während die Gsprite-Engine die Gsprite-Daten transformiert, sendet sie Pixeldaten an einen Zusammensetzungspuffer zur Anzeige. Der Zusammensetzungspuffer ist vorzugsweise

doppelt gepuffert, so dass zusammengesetzte Pixeldaten von einem Puffer übertragen werden können, während Pixeldaten im anderen Puffer zusammengesetzt werden.

[0458] Spezieller gesagt, liest die Gsprite-Engine Gsprite-Bilddaten im AYUV-Format aus dem gemeinsamen Speicher, dekomprimiert, transformiert und filtert sie, konvertiert sie ins ARGB-Format und sendet sie an den Zusammensetzungspuffer mit Videoraten (z.B. 75 Hz). Der Zusammensetzungspuffer setzt die dekomprimierten ARGB-Pixel in einem 1344 × 32-Puffer zur Anzeige zusammen.

[0459] Fig. 26 ist ein Flussdiagramm, das illustriert, wie die Gsprite-Engine Bilddaten verarbeitet. Beim Erhalt eines Bildsynchronisierungssignals (**858**) durchläuft die Gsprite-Engine jedes Band (**860**) für ein Teilbild und scannt jeden Gsprite in einem Band (**862**). Nach dem Scannen der Gsprites für ein Band fährt sie dann mit dem nächsten Band (**860**) fort. Die Gsprite-Engine wiederholt den Scanprozess für jedes Band im Betrachtungsraum.

[0460] Da in einer Echtzeitanwendung die Gsprite-Engine den Scan innerhalb einer Zeit abschließen muss, die von der Bildwiederholrate vorgegeben wird, ist es möglich, dass die Gsprite-Engine nicht in der Lage ist, jeden Gsprite in jedem Band zu verarbeiten. Um diesen Fall verhüten zu helfen, meldet die Gsprite-Engine für jedes Teilbild die freie Verarbeitungszeit für jedes Band an den Host-Rechner. Unter Verwendung dieser Informationen kann der Bildpräprozessor Objekte nach Notwendigkeit vereinigen, um eine Überladung eines bestimmten Bandes zu verhindern.

[0461] Beim Scannen der Pixel aus dem Gsprite-Raum konvertiert die Gsprite-Engine die Pixeldaten in die Ausgabegerätekoordinaten (**866**). Es kann eine beliebige Zahl von herkömmlichen Scanverfahren verwendet werden, um den Gsprite in Ausgabegerätekoordinaten zu scannen. Es kann entweder ein Backward Mapping oder ein Forward Mapping verwendet werden. Die Gsprite-Engine verwendet in dieser Ausführungsform einen Backward Mapping-Ansatz.

[0462] Die Gsprite-Engine bestimmt unter Verwendung der Kantengleichungsdaten im SCB den Ort für den Start des Scans in jedem Band durch Zuschneiden des Gsprites auf das Band. Fig. 25 zeigt zum Beispiel, wie die Kanten des Gsprites in das dritte Band hineinragen (**830**, **832**). Die Schnittpunkte sind der Start- und Stoppunkt für den Scan des Gsprites in diesem speziellen Band. Ein Ansatz für das Scannen besteht darin, das Scannen zickzackartig vom Startpunkt aus durchzuführen. Der Startpunkt in einem Band kann gefunden werden, indem die zum Schnittpunkt nächstgelegenen Pixel in Ausgabegerätekoordinaten herangezogen werden. Sobald der Startpunkt berechnet ist, schreitet die Gsprite-Engine schrittweise in Inkrementen weiter, bis sie aus dem Gsprite oder aus dem Band heraustritt. Sie schreitet dann eine Spalte nach rechts und dann nach unten, bis sie entweder aus dem Gsprite oder aus dem Band heraustritt. Bei jedem Schritt interpoliert sie aus den Pixeldaten im Gsprite-Raum, um einen Pixelwert für einen Pixelort zu finden. Wenn sie diesen Pixelwert an jedem Ort berechnet, sendet sie die Pixeldaten zum Zusammensetzen an den Zusammensetzungspuffer.

[0463] Fig. 27 ist ein Schema, das illustriert, wie die Gsprite-Engine und die Zusammensetzungspuffer Bänder von Bilddaten verarbeiten. In diesem Schema bezieht sich der Begriff „Band" auf die Zeitdauer (Bandperiode), die zur Verarbeitung eines Bandes von Pixeldaten zugewiesen ist. Diese Zeit kann zum Teil aus der Bildwiederholrate und der Zahl der Bänder im Anzeigegerät abgeleitet werden. Wie in Fig. 27 gezeigt, füllt die Gsprite-Engine **204** die Zusammensetzungspuffer **210** für ein Band **888**, und diese zusammengesetzten Bilddaten werden zur Anzeige **892** herausgescannt. Diese Schritte können unter Verwendung der Doppelpufferung für aufeinander folgende Bänder überlappt werden. Während die Gsprite-Engine **204** einen Zusammensetzungspuffer für ein Band füllt **890**, überträgt der Zusammensetzungspuffer die zusammengesetzten Bilddaten für ein weiteres Band an den DAC **212**, **892**. In der nächsten Bandperiode wird das Band, das gerade zusammengesetzt wurde, dann angezeigt **894**. Dieser Prozess wiederholt sich für Bänder in der Anzeige. Wegen dieser Doppelpufferung kann der Prozess des Transformierens und des Zusammensetzens von Pixeln gleichzeitig mit dem Prozess des Anzeigens eines Bandes ablaufen.

[0464] Gsprites können in Echtzeit zusammengesetzt werden, um das Bild zu erzeugen, das auf dem Ausgabegerät angezeigt wird. Die Gsprite-Pixeldaten, die aus der Gsprite-Adressierungs- und Bildverarbeitungsengine erzeugt wurden, werden an einen Zusammensetzungspuffer weitergeleitet. Der Zusammensetzungspuffer hat zwei 32-Scanline-Puffer, einen, der zum Zusammensetzen zu Videodaten zur Anzeige verwendet wird, und einen, der zum Erzeugen der Videodaten zur Anzeige verwendet wird. Die zwei Puffer schalten vor und zurück, so dass während des Anzeigens einer Scanline-Region die nächste zusammengesetzt wird.

[0465] Die Gsprite-Engine leitet die primären Farbdaten und Alphadaten an den Zusammensetzungspuffer

für jedes Pixel, das zusammengesetzt werden soll, weiter. Ein 32-Scanline-Alphapuffer ist mit dem Scanlinepuffer verbunden, der gerade zum Zusammensetzen verwendet wird. Da die Gsprites in der Reihenfolge vorn nach hinten verarbeitet werden, kann der Alphapuffer zum Ansammeln von Opazität für jedes Pixel verwendet werden, was ordnungsgemäßes Anti-Aliasing und Transparenz ermöglicht.

[0466] Der Scanline-Farbpuffer wird mit 0,0 initialisiert. (alle Bits zurückgesetzt), während der Alphapuffer auf 1,0 (alle Bits gesetzt) initialisiert wird. Für jedes Pixel berechnet sich die Farbe, die in den Scanlinepuffer geladen wird, zu Farbe(neu) = Farbe(dst) + Farbe(src)·Alpha(src)·Alpha(dst). Der Alphawert, der im Alphapuffer gespeichert wird, berechnet sich zu Alpha neu) = Alpha(dst)·(1 minus Alpha(sic)). Die Farbverweistabelle (LUT) ist vorzugsweise 256 × 10 Bit: die Extrabits (10 gegenüber 8) können dazu verwendet werden, für eine genauere Gammakorrektur zu sorgen.

Tiling

[0467] Wie oben angeführt, führt der Bildprozessor (Fig. 1) eine Scan-Konvertierung aus, entfernt verborgene Flächen, führt Anti-Aliasing, Lichtdurchlässigkeitsberechnungen, Texturierung und Schattierung aus. In diesem Abschnitt beschreiben wir detailliert Scan-Konvertierung, Anti-Aliasing und Lichtdurchlässigkeitsberechnung.

[0468] Fig. 4B ist ein Schema, das Teile des Bildprozessors **462** zum Erzeugen von gerenderten Bilddaten aus geometrischen Grundelementen illustriert. Der Bildprozessor umfasst einen Rastergenerator **464**, eine Pixelengine **466**, eine Anti-Aliasing-Engine **468** und einen Rastererzeugungspuffer, der die Pixelpuffer **470** und einen Fragmentpuffer **472** in dieser Ausführungsform umfasst. Der „Rastergenerator" bezieht sich auf den Teil des Bildprozessors, der Pixelwerte aus den geometrischen Grundelementen, d.h. Polygonen, bestimmt. Der Rastergenerator **464** liest die Daten für jedes Grundelement und erzeugt Pixeldaten, die mit dem Pixelort verknüpft sind. Diese Pixeldaten umfassen Farbe, Alpha und Tiefe (Abstand vom Betrachtungspunkt). Wenn ein Pixel nicht vollständig von einem Polygon bedeckt ist, erzeugt der Rastergenerator Pixelfragmentdaten.

[0469] Während er ein Polygon scannt, leitet der Rastergenerator Pixeldaten an die Pixelengine zur Verarbeitung weiter. Die Pixelengine **468** liest Pixeldaten aus dem Rastergenerator und bestimmt, welche Pixeldaten in Pixel- und Fragmentpuffern zu speichern sind. Die Pixelpuffer **472** sind zweidimensionale Felder, wobei die Elemente in den Feldern Pixelorten entsprechen, und umfassen Speicher zum Speichern von Farb-, Alpha- und Tiefendaten. Der Fragmentpuffer **470** speichert Fragmentdaten, die das teilweise Abdecken eines Pixels repräsentieren.

[0470] Die Pixelengine **466** führt eine Entfernung verborgener Flächen unter Verwendung der Tiefenwerte aus, die durch den Rastergenerator erzeugt wurden, und hält auch die Pixelfragmente und lichtdurchlässigen Pixel für das Anti-Aliasing und die Lichtdurchlässigkeitsverarbeitung aufrecht. Für einen gegebenen Pixelort bewahrt die Pixelengine das nächstgelegene voll abgedeckte lichtundurchlässige Pixel, falls vorhanden. In diesem Kontext bedeutet „voll abgedeckt", dass das Pixel vollständig von einem Polygon bedeckt wird, das gerade im Rastergenerator scan-konvertiert wird. Die Pixelengine bewahrt auch Pixel mit Lichtdurchlässigkeit (Alpha kleiner als 1) und Pixelfragmente vor dem nächstgelegenen lichtundurchlässigen Pixel. Die Pixelengine speichert das nächstgelegene lichtundurchlässige Pixel für einen Pixelort im Pixelpuffer und speichert im Fragmentpuffer alle Fragmente oder lichtdurchlässigen Pixel an diesem Pixelort, die sich vor dem nächstgelegenen lichtundurchlässigen Pixel befinden.

[0471] Nachdem die Pixelengine Pixeldaten erzeugt hat, löst die Anti-Aliasing-Engine **468** die Pixeldaten in Pixel- und Fragmentpuffer auf. Die Konstruktion des Bildprozessors, der in Fig. 4B illustriert wird, unterstützt die Doppelpufferung von Pixeldaten und die einfache Pufferung von Fragmentdaten. Die Pixelengine erzeugt Pixeldaten in einem der Pixelpuffer und fügt Fragmentinformationen zum Fragmentpuffer hinzu, während die Anti-Aliasing-Engine die Pixeldaten aus dem anderen Pixelpuffer und Fragmentdaten aus dem Fragmentpuffer auflöst. Wenn jedes Fragment aufgelöst ist, wird der Fragmenteintrag zur Liste der freien Fragmente zur Verwendung durch neue Pixeldaten hinzugefügt.

[0472] Nachdem wir einen Überblick über den Prozess der Erzeugung und Auflösung von Pixeldaten gegeben haben, beschreiben wir nun detaillierter eine Ausführungsform.

[0473] Die Komponenten von Fig. 4B können im Tiler implementiert werden. Der Tiler liest Grundelementedaten und Renderinganweisungen aus dem gemeinsamen Speichersystem **216** (Fig. 4A), erzeugt gerenderte Bilddaten und speichert komprimierte Bilddaten im gemeinsamen Speicher. Wie oben beschrieben, sind die grundlegenden 3D-Grafikgrundelemente im System Dreiecke. Das Rendern von Dreiecken sorgt für zahlreiche

Vereinfachungen in der Hardware, die zur Grafikerzeugung verwendet wird, da das Dreieck immer planar und konvex ist. Alternativ können jedoch auch n-seitige Polygone verwendet werden.

[0474] Oben haben wird die Komponenten des Tilers **200** erklärt. Hier beschreiben wir detaillierter den Datenfluss durch den Tiler.

[0475] Da der Tiler Eingaben aus dem DSP erhält, beginnen wir mit einer kurzen Wiederholung von Funktionen des DSP **176** (**Fig.** 4). Wie oben beschrieben, kann der DSP **176** die Geometrievorverarbeitung und die Beleuchtungsberechnungen ausführen, die für die 3D-Grafik benötigt werden. Der DSP **176** berechnet Modell- und Betrachtungstransformationen, Beschneiden, und Beleuchtung usw. Renderbefehle werden in Hauptspeicherpuffern gespeichert und zur Bildverarbeitungsplatine über einen PCI-Bus geDMAt (durch direkten Speicherzugriff erfasst). Die Renderbefehle werden dann im gemeinsamen Speicher **216** (Fig. 4A) gepuffert, bis sie vom DSP benötigt werden. Die Renderingbefehle werden vom Tiler **200** (Fig. 4A) gelesen, wenn er zur Ausführung von Bildverarbeitungsoperationen bereit ist.

[0476] Wie im Flussdiagramm in den Fig. 28A und Fig. 28B gezeigt, verarbeitet der Setup-Block Anweisungen zum Rendern von Grundelementen, die aus dem gemeinsamen Speicher gelesen wurden. Der Eckpunkteingabeprozessor parst den Eingabestrom (**914**) (Fig. 28A) und speichert die Informationen, die für die Grundelemente-Dreiecksverarbeitung in den Eckpunktsteuerregistern (**916**) benötigt werden.

[0477] Die zwei Eckpunktsteuerregister speichern sechs Eckpunkte, drei für jedes Dreieck, in jedem Register. Die zwei Eckpunktsteuerregister ermöglichen das Doppelpuffern der Dreiecksinformationen, um sicherzustellen, dass die Setup-Engine immer über Dreiecksinformationen zum Verarbeiten verfügt.

[0478] Die Setup-Engine berechnet dann die linearen Gleichungen (**918**), die die Kanten-, Farb- und Texturkoordinateninterpolation über die Fläche des Dreiecks bestimmen. Diese linearen Gleichungen werden dazu verwendet festzustellen, welche Texturblöcke benötigt werden, um das Dreieck zu rendern. Die Kantengleichungen werden auch in den Scanumwandlungsblock (**920**) geschoben und in den Grundelementeregistern innerhalb des Scanumwandlungsblocks gespeichert, bis sie von der Scanumwandlungs-Engine benötigt werden. Die Grundelementeregister sind in der Lage, mehrere Sätze von Kantengleichungen zu speichern.

[0479] Die Setup-Engine sendet Texturadressen an die Texturlese-Schlange (**922**), die Anforderungen für Texturchunks puffert. Der Texturadressgenerator bestimmt dann die Adresse im Speicher der angeforderten Texturchunks (**924**) und sendet Texturleseanforderungen an den Befehls- und Speichersteuerungsblock (**926**) (Fig. 28B), der die Texturdaten (**928**), die vom Scanumwandlungsblock verwendet werden, abruft.

[0480] Texturdaten werden im gemeinsamen Speicher (**216**) (Fig. 4A) in einem komprimierten Bildformat gespeichert, das mit dem Format der Bilddaten übereinstimmen kann. Das Kompressionsformat wird auf individuelle 8×8-Pixelblöcke angewendet. Die 8×8-Blöcke werden für Zwecke des Speichermanagements zu 32×32-Blöcken zusammengestellt, um den Zusatzaufwand für das Speichermanagement zu reduzieren.

[0481] Wenn Texturblöcke benötigt werden, werden sie in den Tiler abgerufen, von der Dekompressionsengine (**930**) dekomprimiert und in einem chipintegrierten Texturcache gecacht (**932**). Es können insgesamt 32×32-Pixelblöcke gecacht werden, obwohl jeder einzelne Block nur eine Farbkomponente speichert. Die Texturdaten werden im RGB- und Alpha-Format gecacht.

[0482] Die Scanumwandlungs-Engine liest dann die Kantengleichungen aus den Grundelementeregistern (**934**), um die Dreieckskanteninformationen zu scan-konvertieren. Die Scanumwandlungs-Engine umfasst die Interpolatoren für das Bewegen der Kanten der Dreiecke, das Interpolieren von Farben, Tiefen, Lichtdurchlässigkeit usw.

[0483] Die Scanumwandlungs-Engine sendet Texturadressen an die Texturfilter-Engine (**936**). Die Texturfilter-Engine berechnet Texturdaten für die Polygone, die gerade gerendert werden. Die Texturfilterengine berechnet einen Filterkern, der auf der Z-Neigung und Orientierung des Dreiecks und auf den s- und t-Koordinaten beruht. Der Texturcache, der an der Texturfilterengine hängt, speichert Texturdaten für sechzehn 8×8-Pixelblöcke. Der Texturcache steht auch mit der Dekompressionsengine in Verbindung, die die Texturdaten (welche in einem komprimierten Format gespeichert sind) zur Verwendung durch die Texturfilterengine dekomprimiert.

[0484] Wenn die Texturfilterung abgeschlossen ist, leitet die Texturfilterengine die Informationen zur Scanum-

wandlungs-Engine (**938**) zurück, so daß sie von der Scanumwandlungs-Engine zur weiteren Verarbeitung verwendet werden können. Neben der Texturverarbeitung scankonvertiert die Scanumwandlungs-Engine die Dreieckskantendaten (**940**), und die individuellen Pixeladressen werden zusammen mit Farb- und Tiefeninformationen an die Pixelengine zur Verarbeitung geleitet (**942**).

[0485] Das Verfahren, das in den Fig. 28A und Fig. 28B illustriert wird, variiert für die alternativen Verfahren, die in Verbindung mit den Fig. 10 und Fig. 11 beschrieben werden. Die Fig. 28C und Fig. 28D illustrieren ein Verfahren für den Zugriff auf Bilddaten entsprechend Fig. 10 und Fig. 9B. In ähnlicher Weise illustrieren die Fig. 28E und Fig. 28F ein Verfahren für den Zugriff auf Bilddaten entsprechend Fig. 11 und Fig. 9C.

[0486] Bezieht man sich zuerst auf die Fig. 28C und Fig. 28D, so beginnt diese Implementierung des Verfahrens im Setup-Block **381** von Fig. 9B. Der Eckeneingabeprozessor **384** verarbeitet den Eingabedatenstrom (**947**). Als Nächstes puffern die Eckpunktsteuerregister **386** Dreiecksdaten aus dem Eingabedatenstrom (**948**). Die Setup-Engine **388** berechnet dann die Kantengleichungen (**949**) und leitet sie an den Scanumwandlungsblock **395** weiter (**950**).

[0487] Der Scanumwandlungsblock **395** liest die Kantengleichungen, die in den Grundelementeregistern (**951**) gespeichert sind, und scan-konvertiert die Dreiecksdaten (**952**). Die Scanumwandlungs-Engine **398** schreibt dann Pixeldaten, einschließlich Pixeladresse, Farb- und Alphadaten und Bedeckungsdaten, in einen Eintrag in die Texturreferenzdatenschlange **399** (**953**) (Fig. 28D). Im Fall der Texturabbildungsoperationen umfasst dieser Eintrag auch Texturreferenzdaten, nämlich die Koordinaten des Texturmittelpunkts. Der Eintrag kann auch Texturfilterdaten umfassen, wie zum Beispiel Ebenendetails oder anisotrope Filtersteuerungsdaten.

[0488] Aus den Texturreferenzdaten bestimmt die Texturcachesteuerung **391**, welche Texturblöcke abzurufen sind, und bewirkt, dass der/die entsprechende(n) Textblock/-blöcke aus dem Speicher (**954**) abgerufen werden.

[0489] Die Texturadresscachesteuerung **391** sendet Texturleseanforderungen an den Befehls- und Speichersteuerungsblock **380** (**955**). Die Texturleseschlange **393** puffert Leseanforderungen für Texturblöcke an das gemeinsame Speichersystem. Die Speichersteuerung **380** ruft die Texturdaten aus dem gemeinsamen Speicher ab, und wenn sie komprimiert sind, stellt sie den komprimierten Block oder Blöcke in den komprimierten Cache **416** (**956**). Die Dekompressionsengine **404** dekomprimiert komprimierte Bilddaten und setzt sie in den Texturcache **402** (**957**, **958**). Wie oben in Verbindung mit Fig. 10 beschrieben, setzt sich das Ersetzen von Blöcken im Texturcache gemäß einem Cacheersetzungsalgorithmus fort.

[0490] Um eine Texturabbildung oder andere Pixeloperationen auszuführen, die Bilddaten im Texturcache erfordern, liest die Texturfilterengine **401** Texturadressen aus der Texturreferenzdatenschlange **399** (**959**). Die Texturfilter-Engine **401** greift auf die Bilddaten im Texturcache **402** zu, berechnet den Beitrag aus der Textur und kombiniert diesen Beitrag mit den Farb- und möglicherweise Alphadaten aus der Texturreferenzdatenschlange **399**.

[0491] Die Texturfilterengine **401** leitet Pixeldaten an die Pixelengine **406**, die das Entfernen verborgener Flächen ausführt und die Speicherung der Pixeldaten in einem Rastererzeugungspuffer steuert.

[0492] Die Fig. 28E und Fig. 28F illustrieren ein Verfahren für den Zugriff auf Bilddatenblöcke aus dem Speicher entsprechend dem Ansatz in Fig. 11. In dieser alternativen Implementierung beginnt das Verfahren durch Einreihen von Grundelementen in die Schlange im Setup-Block **383**. Der Eckpunkteingabeprozessor **384** parst den Eingabedatenstrom und reiht Dreiecksdaten in die Schlange in Eckpunktsteuerregistern **387** ein (**961**, **962**). Wenn auf Bilddatenblöcke aus dem Speicher zugegriffen werden muss, wie dies im Fall einer Texturabbildungsoperation ist, scankonvertiert der Vorrastergenerator **389** die Grundelemente, die sich in der Schlange in den Eckpunktsteuerregistern **386** befinden, um Leseanforderungen für Texturdatenblöcke im gemeinsamen Speicher zu erzeugen (**963**).

[0493] Wenn der Vorrastergenerator ein Grundelement scannt, das sich in der Schlange im Setup-Block befindet, leitet er Texturleseanforderungen an die Texturcachesteuerung **391** (**964**). Die Texturcachesteuerung **391** bestimmt die geeigneten Texturblöcke (**965**) und überträgt Leseanforderungen über die Texturleseschlange **393** an den Befehls- und Speichersteuerungsblock **380** (**989**) (Fig. 28F). Der Speichersteuerungsblock ruft die angeforderten Texturdaten ab, und wenn sie komprimiert sind, stellt er sie in den komprimierten Block **416** (**990**). Die Dekompressionsengine dekomprimiert Texturblöcke im komprimierten Cache **416** und schreibt die dekomprimierten Bilddaten in den Texturcache **402** (**991**, **992**). Die Texturcachesteuerung leitet den Strom von

Texturblöcken aus dem komprimierten Cache **416** durch die Dekompressions-Engine **404** in den Texturcache **402**.

[0494] Der Scanumwandlungsblock **397** liest die geometrischen Grundelemente, die sich in der Schlange im Setup-Block befinden. Der Scanumwandlungsblock **397** führt Pixelerzeugungsoperationen aus, sobald angeforderte Texturdaten im Texturcache **402** verfügbar sind. Beim Prozess der Ausführung dieser Pixeloperationen liest die Scanumwandlungs-Engine **398** Kantengleichungen aus dem Grundelementeregistern (**393**) und leitet Texturadressen an die Texturfilterengine **403** (**994**). Die Texturfilterengine greift auf die entsprechenden Bilddaten zu, die im Texturcache **402** gespeichert sind, und gibt dann gefilterte Daten an den Scanumwandlungsblock **397** zurück (**995**). Der Scanumwandlungsblock **397** konvertiert die Dreiecksdaten und berechnet aus den konvertierten Dreiecksdaten und den gefilterten Daten Ausgabepixeldaten (**996**). Er leitet dann diese Ausgabepixeldaten an die Pixelengine **406** weiter.

[0495] Die Pixelengine **406** führt Berechnungen auf Pixelniveau aus, einschließlich Entfernen von verborgenen Flächen und Mischoperationen. Um verborgene Flächen zu entfernen, vergleicht die Pixelengine **406** die Tiefenwerte für einlaufende Pixel (vollständig bedeckte Pixel oder Pixelfragmente) mit Pixeln an entsprechenden Orten in den Pixel- oder Fragmentpuffern. Bei Schattierungsoperationen führt die Pixelengine **406** Tiefenvergleichsoperationen aus, um das erste und zweite nächstgelegene Grundelement zur Lichtquelle an Orten in einer Schattenabbildung festzustellen, und aktualisiert den ersten und zweiten nächstgelegenen Tiefenwert, wo notwendig. Nach der Ausführung der Berechnungen auf Pixelebene speichert die Pixelengine die entsprechenden Daten im Pixel- oder Fragmentpuffer.

[0496] Der Tiler implementiert einen hochwertigen Anti-Aliasing-Algorithmus zur Behandlung nicht lichtundurchlässiger Pixel. Der Pixelpuffer speichert die Pixeldaten für das am weitesten vorn gelegene nichttransparente Pixel für Pixelorte in einem Chunk. Der Fragmentpuffer speichert Pixelfragmente für lichtdurchlässige Pixel und für teilweise bedeckte Pixel, die näher am Betrachtungspunkt liegen als die Pixel im Pixelpuffer für entsprechende Pixelorte. Unter Verwendung einer Fragmentlistenstruktur kann mehr als ein Fragment für einen Pixelort gespeichert werden. In einem Prozess, der als Auflösen bezeichnet wird, verarbeitet die Anti-Aliasing-Engine die Fragmentlisten, um Farb- und Alphawerte für Pixelorte zu berechnen.

[0497] Um die Zahl der Fragmente zu reduzieren, die erzeugt werden, implementiert die Pixelengine ein Verfahren zum Vereinigen von Pixelfragmenten, das das Fragment, das gerade erzeugt wird, mit Fragment(en), die aktuell im Fragmentpuffer gespeichert sind, vergleicht. Wenn die neuen und vorherigen Attribute eines Fragments (Farbe und Tiefe) sich innerhalb einer vorgegebenen Toleranz ähnlich sind, werden die Fragmente im Vorbeigehen kombiniert, und es wird kein zusätzliche Fragment erzeugt.

[0498] Wenn bei einem kombinierten Fragment festgestellt wird, dass es vollständig bedeckt ist (mit einer Bedeckungsmaske und lichtundurchlässigem Alpha), dann wird das Fragment in den Farbpuffer geschrieben und der Fragmentort wird freigegeben, so dass er für nachfolgende Polygone innerhalb des aktuellen Chunks verwendet werden kann.

[0499] Sobald alle Polygone für den Chunk gerendert sind, werden die Pixelpuffer getauscht. Während die Anti-Aliasing-Engine die Pixeldaten im Fragmentpuffer und einem der Pixelpuffer auflöst, schreibt die Pixelengine Pixeldaten für den nächsten Chunk in den anderen Pixelpuffer und die restlichen freien Stellen im Fragmentpuffer. Das Pixelauflösen umfasst im allgemeinen das Berechnen eines einzelnen Farbwerts (und möglicherweise Alphawerts) für einen Pixelort auf der Grundlage der Pixeldaten im Pixel- und Fragmentpuffer entsprechend dem Ort. Wir stellen zusätzliche Details unten bereit, die diese Fragen betreffen.

[0500] In den Implementierungen des Tilers, die in den Fig. 9A-Fig. 9C gezeigt werden, haben die Pixelengine und die Anti-Aliasing-Engine Zugriff auf einen einzelnen Fragmentpuffer und ein Paar Pixelpuffer. Die zwei 32×32-Pixelpuffer werden zur Doppelpufferung zwischen der Pixelengine und der Anti-Aliasing-Engine bereitgestellt. Der Pixelpuffereintrag umfasst die folgenden Daten:

R	G	B	α	Z	P

wo R, G, B die roten, grünen bzw. blauen Farbkomponenten sind, α die Alphakomponente ist, die die Lichtdurchlässigkeit des Pixels repräsentiert, und Z die Tiefenkomponente ist, die die Tiefe des Pixels vom Betrachtungspunkt repräsentiert. Die x, y-Adresse ist fest und implizit in der Pixelpufferadressierung. Es werden acht

Bit pro Farbkomponente verwendet (d.h. Rot, Grün und Blau), acht Bit werden für die α-Komponente verwendet, und sechsundzwanzig Bit werden verwendet, um den Z-Wert, Schablonenwert und einen Prioritätswert zu speichern. Von diesen 26 Bit können bis zu 24 Bit als Z-Werte verwendet werden, bis zu 3 können als Schablonenebenen verwendet werden, und bis zu drei können als Prioritätswerte verwendet werden. Wie oben mit Bezug auf **Fig.** 9 beschrieben, umfasst der Puffer auch einen 9-Bit-Fragmentpufferzeiger.

[0501] Der Prioritätswert ist pro Grundelement fixiert und wird verwendet, um bei der Auflösung von Objekten zu helfen, die coplanar sind, wie zum Beispiel Straßen auf einem Gelände, indem während der Z-Vergleichsoperation Prioritätsbeziehungen verwendet werden, die von der Tiler-Engine genutzt werden, um die eingehenden Pixel-Z-Werte im Vergleich zu dem gespeicherten Z-Wert abzugrenzen.

[0502] Der Fragmentpuffer wird zum Speichern von Informationen über Pixelfragmente für Polygone, deren Kanten ein gegebenes Pixel überschneiden, oder für Polygone mit Lichtdurchlässigkeit verwendet. Jeder Eintrag im Fragmentpuffer liefert Farb-, α-, Z- und Bedeckungsdaten, die mit der Oberfläche verbunden sind.

[0503] Mehrere Fragmentpuffereinträge können mit einem einzigen Pixel (über einen verlinkten Listenmechanismus) für Fälle verknüpft sein, in denen mehrere Polygone eine Teilbedeckung für denselben Pixelort aufweisen. Der Fragmentpuffer hat zwei Ports, so dass er parallel von der Anti-Aliasing-Engine und der Pixelengine betrieben werden kann. In einer Implementierung ist der Fragmentpuffer ein eindimensionales Feld von Fragmentdatensätzen und umfasst insgesamt 512 Fragmentdatensatzeinträge. Das Speichermanagement des Fragmentpuffers erfolgt unter Verwendung einer verlinkten Listenstruktur. Jeder Fragmentpuffereintrag umfasst die folgenden Daten:

R	G	B	α	Z	M	P	S

wo R, G, B die roten, grünen bzw. blauen Farbkomponenten sind, α die Alphakomponente ist, die die Lichtdurchlässigkeit des Pixels repräsentiert, und Z die Tiefenkomponente ist, die die Tiefe des Pixels vom Betrachtungspunkt repräsentiert. M ist eine 4×4-Pixelbedeckungsmaske für jedes Pixel, das teilweise bedeckt ist. P ist ein Zeiger auf den nächsten Fragmentpuffereintrag, und S wird zur Darstellung einer Fragmentschablone verwendet. Es werden acht Bit pro Farbkomponente (d.h. Rot, Grün und Blau) und acht Bit für die α-Komponente verwendet. Sechsundzwanzig Bit werden verwendet, um den Z-Wert plus Schablone und Priorität zu speichern, und neun Bit werden für den Fragmentzeiger P verwendet.

[0504] Die Pixelbedeckungsmaske wird durch Festlegen eines Bedeckungsmaskenwertes für jede Kante und ihre bitweise AND-Verknüpfung berechnet. Die Berechnung der Bedeckungsmaske ist ein zweistufiger Prozess. Die erste Stufe besteht darin zu bestimmen, wie viele von den Subpixelbits in der Bedeckungsmaske auszuschalten sind, und die zweite Stufe besteht darin zu bestimmen, welche speziellen Bits aktiviert werden sollen.

[0505] Die erste Stufe verwendet die Fläche des Pixels, die von der Kante verdeckt wird, um festzustellen, wie viele von den Bedeckungsmaskenbits eingeschaltet werden sollen. Diese Fläche wird durch eine Datenentnahme aus der Tabelle berechnet, die durch die Kantenneigung und den Abstand von der Pixelmitte indexiert ist. Die zweite Stufe verwendet die Kantenneigung, um die Reihenfolge festzustellen, in der die Abfragebits eingeschaltet werden sollen. Der Satz der Bitreihenfolgen wird in einer vorherberechneten Tabelle gespeichert, die ‚Bedeckungsordnungstabelle' genannt wird. Jeder Eintrag in der Bedeckungsordnungstabelle besteht aus einer speziellen Ordnung der Abtastbits, die für einen Bereich von Neigungswerten korrekt ist. Die Kantenneigung wird gegenüber einem Satz von Neigungsbereichen geprüft, und der Index, der mit dem Bereich verknüpft ist, welcher diesen Neigungswert umfasst, wird als Index in die Bedeckungsordnungstabelle verwendet.

[0506] Ein Verfahren zum Berechnen der Bedeckungsmaske wird in Schilling, A, "A New Simple and Efficient Anti Aliasing with Subpixel Masks [Ein neues und effizientes Anti-Aliasingverfahren mit Subpixelmasken]", Computer Graphics, Bd. 25, Nr. 4, Juli 1991, S. 133-141, beschrieben.

Entfernen verborgener Flächen und Vereinen von Fragmenten

[0507] Oben haben wir angegeben, dass die Pixelengine das Entfernen verborgener Flächen vornimmt, indem sie Tiefenvergleichsoperationen an einlaufenden Pixeldaten vornimmt. Wir haben auch bemerkt, dass Pixelfragmente vereinigt werden können, um Fragmentspeicher freizusetzen. Verschmelzen von Fragmenten re-

duziert die Speicheranforderungen, um eine gegebene Szene dem Anti-Aliasing zu unterwerfen, und beschleunigt das Auflösen der Fragmente, um ein Endbild zu erzeugen. Wir beschreiben nun eine Implementierung für das Entfernen verborgener Flächen, die das Verschmelzen eines einlaufenden Pixelfragments mit einem gespeicherten Pixelfragment umfasst, wenn das einlaufende Fragment innerhalb vorgegebener Farb- und Tiefentoleranzen des gespeicherten Fragments liegt.

[0508] Fig. 4B ist ein Schema, das die Komponenten im Tiler **462**, einschließlich Rastergenerator **464**, Pixelengine **466**, Pixel- und Fragmentpuffern **470** und **472**, illustriert. Die Pixel- und Fragmentpuffer dienen als Rastergenerierungspuffer zum Speichern gewählter Pixeldaten. Wenn der Rastergenerator ein geometrisches Grundelement scannt, erzeugt er Beispiele von Pixeldaten. Die Pixelengine steuert die Z-Pufferung und stellt auch fest, ob ein eingehendes Pixelfragment mit einem Pixelfragment verschmolzen werden kann, das im Fragmentpuffer an einem entsprechenden Pixelort gespeichert ist. Die Illustrationen von Tilern, die in den Fig. 9A-Fig. 9C gezeigt werden, und der begleitende Text oben liefern weitere Details bezüglich spezieller Implementierungen des Tilers. Das Verfahren und die Hardware zum Verschmelzen von Pixelfragmenten, die unten beschrieben werden, können in diesen Tiler-Konstruktionen und auch in alternativen Konstruktionen implementiert werden.

[0509] Wie oben beschrieben, erzeugt der Scanumwandlungsblock (Rastergenerator) im Tiler Beispiele von Pixeldaten, die repräsentieren: 1) voll abgedeckte, lichtundurchlässige Pixel; 2) voll abgedeckte lichtdurchlässige Pixel; 3) teilweise abgedeckte, lichtundurchlässige Pixel; oder 4) teilweise abgedeckte, lichtdurchlässige Pixel.

[0510] Der Pixelpuffer speichert Farbe und Tiefe (Z) der am weitesten vorn gelegenen, voll abgedeckten lichtundurchlässigen Pixel. Der Pixelpuffer speichert auch einen Zeiger auf eine Fragmentliste, einschließlich der Fragmente, die eine Bedeckungsmaske haben, die nicht voll abgedeckt ist oder die ein Alpha haben, das nicht vollständig lichtundurchlässig ist. Der Kopf oder das erste Fragment in der Fragmentliste ist das neueste Pixelfragment, das verarbeitet wird. In dieser speziellen Implementierung versucht die Pixelengine, einlaufende Pixelfragmente mit dem neuesten Pixelfragment zu verschmelzen. Da es einen bestimmten Betrag an räumlicher Kohärenz beim Rendern von Polygonen gibt, erhöht der Versuch, mit dem neuesten Fragment zu verschmelzen, das für einen gegebenen Pixelort erzeugt wurde, die Wahrscheinlichkeit, dass die Verschmelzung erfolgreich sein wird.

[0511] Die Fragmentlisten für jeden Pixelort werden in unsortierter Form geführt, wobei das oberste Fragment das neueste Fragment ist, das für einen bestimmten Pixelort erzeugt wurde. Die Pixelfragmente hinter dem obersten Fragment bleiben unsortiert, können aber auch sortiert werden, wenn zusätzliche Rechenzeit verfügbar ist, um bei der Optimierung der Fragmentauflösungsphase zu helfen.

[0512] In einer alternativen Implementierung umfasst die Pixelengine eine zusätzliche Logik, um Fragmentlisten nach einem Pixelfragment abzusuchen, das die Fragmentverschmelzungskriterien erfüllt. Dieser Ansatz wird nicht bevorzugt, weil der zusätzliche Aufwand für die Suchlogik nicht die kleine Verbesserung bei der Identifizierung von mehr Verschmelzungskandidaten rechtfertigt. Dies gilt besonders in einem Echtzeitsystem, wo zusätzliche Taktzyklen, die im Verschmelzungsprozess verbraucht werden, die Zeit erhöhen, die zum Rendern eines Animationsteilbildes benötigt wird.

[0513] In einer anderen Implementierung führt die Pixelengine eine nach Tiefe sortierte Liste von Pixelfragmenten und versucht, mit dem Fragment zu verschmelzen, das dem Betrachtungspunkt für einen gegebenen Pixelort am nächsten liegt. Dieser letztere Ansatz wird jedoch nicht bevorzugt, weil er weniger wahrscheinlich erfolgreiche Verschmelzungskandidaten findet, d.h. Fragmente mit Z- und Farbwerten innerhalb der Toleranz zum einlaufenden Fragment. Er hat jedoch den potenziellen Vorteil, das Freisetzen von zusätzlichem Fragmentspeicher zu vereinfachen. Wenn ein verschmolzenes Pixel vollständig abgedeckt wird und lichtundurchlässig ist, können alle Pixelfragmente an diesem Pixelort freigesetzt werden, da das verschmolzene Pixel näher am Betrachtungspunkt als die anderen Pixelfragmente ist, die für diesen Pixelort gespeichert werden.

[0514] Fig. 29 ist ein Schema, das eine Implementierung des Entfernens von verborgenen Flächen und der Fragmentverschmelzung im Tiler illustriert. Die Verarbeitung beginnt mit der Erzeugung einer neuen Gruppe von Pixeldaten, die Farbe, Z und Bedeckungsmaske für einen Pixelort haben (**968**). Wenn das Pixelpuffer-Z für diesen Pixelort näher als das Z einer neuen Gruppe der Pixeldaten ist (ein vollständig oder teilweise abgedecktes Pixel) (**970**), dann wird die neue Gruppe von Pixeldaten vollkommen verdunkelt und wird verworfen (**972**). Die Verarbeitung setzt sich dann mit der nächsten Gruppe von Pixeldaten fort, solange der Rastergenerator nicht alle Pixel für den aktuellen Satz von Grundelementen, die gerendert werden, erzeugt hat.

[0515] Wenn das Pixelpuffer-Z nicht näher als das Z der neuen Gruppe von Pixeldaten ist (d.h. das Z der neuen Gruppe von Pixeldaten ist näher am Betrachtungspunkt), dann kontrolliert die Pixelengine die Bedeckungsmaske der einlaufenden Pixel (**974**). In Fällen, wo die Bedeckungsmaske für die einlaufenden Pixel voll ist, ersetzt die Pixelengine die Pixelpuffer Farbe und Z durch die neue Farbe und das neue Z (**976**). Es werden in diesem Fall keine neuen Daten zur Fragmentliste hinzugefügt, und mit dem Speicher wird sparsam umgegangen.

[0516] Wenn die Bedeckungsmaske der neuen Gruppe von Pixeldaten nicht voll ist, dann führt die Pixelengine einen Verschmelzungstest aus, um festzustellen, ob die neue Farbe und das neue Z innerhalb der vorgegebenen Toleranzen der Farbe und des Z des obersten Fragmentes ist (**978**). Dieser Verschmelzungstest kann auch die Feststellung umfassen, ob das Alpha (Lichtdurchlässigkeit) der einlaufenden Pixel innerhalb einer vorgegebenen Toleranz des Alpha im obersten Fragment liegt. Wenn die neue Fragmentfarbe und Z nicht innerhalb der vorgegebenen Toleranzen liegen, dann wird ein neues Pixelfragment zum Fragmentpuffer am Kopf der Fragmentliste hinzugefügt (**980**).

[0517] Wenn die neue Fragmentfarbe und Z innerhalb der vorgegebenen Toleranzen liegen und die neue Bedeckungsmaske nicht voll ist, dann wird das einlaufende Pixelfragment dem oberen Fragment in der Fragmentliste hinzugefügt (**982**). Die Pixelengine verschmilzt die Pixelfragmente durch Ausführen einer OR-Operation auf die obere Bedeckungsmaske und die neue Bedeckungsmaske, wobei das Ergebnis in der oberen Bedeckungsmaske gelassen wird.

[0518] Nach dem Kombinieren der Bedeckungsmasken wird die verschmolzene obere Bedeckungsmaske kontrolliert, um festzustellen, ob sie ein vollständig bedecktes Pixel repräsentiert (**984**). Wenn die verschmolzene obere Bedeckungsmaske nicht voll ist, dann setzt sich die Verarbeitung mit der nächsten Gruppe der Pixeldaten fort (**966**). Wenn die verschmolzene obere Bedeckungsmaske voll ist, hat die verschmolzene obere Bedeckungsmaske zur vollständigen Pixelbedeckung geführt. Daher wird der Speicher, der für das oberste Fragment verwendet wird, freigesetzt und die oberste Fragment-Farbe, -Z und -Bedeckungsmaske werden durch die neue Fragment-Farbe, -Z und -Bedeckungsmaske ersetzt.

[0519] In Fällen, in denen die Pixelengine einen Eintrag im Pixelpuffer durch ein neues, vollständig bedecktes Pixel ersetzt, setzt die Pixelengine auch alle Pixelfragmente in der entsprechenden Fragmentliste mit Tiefenwerten frei, die größer als dieses vollständig bedeckte Pixel sind (**988**).

[0520] Dies tritt auf, wenn ein einlaufendes vollständig bedecktes, lichtundurchlässiges Pixel einen niedrigeren Z-Wert als der Pixelpuffereintrag am selben Pixelort hat. Dies tritt auch auf, wenn ein verschmolzenes Fragment vollständig bedeckt, lichtundurchlässig ist und einen niedrigeren Z-Wert als der Pixelpuffereintrag am selben Pixelort hat. Unter diesen Umständen läuft die Pixelengine durch die Fragmentliste, vergleicht den Z-Wert des neuen vollständig bedeckten Pixels mit dem Z-Wert der Fragmente in der Liste und setzt jedes Fragment mit einem größeren Z als das Z des neuen, vollständig bedeckten Pixels frei. Alternativ könnte der Z-Puffer für den Packungsprozess gespeichert werden, was die Notwendigkeit beseitigt, die Fragmentliste zu durchsuchen, und die Echtzeitleistung verbessert.

[0521] Der Ansatz, der in Fig. 29 gezeigt ist, reduziert die Speicheranforderungen, um eine gegebene Szene dem Anti-Aliasing zu unterwerfen, und beschleunigt das Auflösen der Fragmente, um ein Endbild durch Verwerfen der Pixelfragmente, die nicht verwendet werden, zu erzeugen. Das Einstellen von Farb- und Z-Toleranz ermöglicht es, die Zahl der erzeugten Fragmente, die verworfen werden, mit der Anti-Aliasing-Genauigkeit zu verrechnen, je nach den Bedürfnissen des Nutzers. Wenn Farbe und Z an der Kante des Polygons bewertet werden, die zur Pixelmitte am nächsten gelegen ist, können engere Farbtoleranzen und Z-Toleranzen verwendet und trotzdem Speicher gespart werden.

[0522] Fig. 30 ist ein Schema, das eine Implementierung der Fragmentverschmelzungsschaltkreise illustriert, die zur Ausführung eines Verschmelzungstests an einlaufenden Pixelfragmenten verwendet werden. In dieser Implementierung vergleicht die Pixelengine die einlaufenden Farb-(RGB), Alpha- und Tiefenwerte mit den Farb-, Alpha- und Tiefenwerten des neuesten Pixelfragments für den Pixelort der einlaufenden Pixel. Die Farb-, Tiefen- und Alphakomponenten, die als „neu" dargestellt werden, beziehen sich auf eine einlaufende oder „neu" erzeugte Gruppe von Pixeldaten, während die Komponenten, die als „prev. [vorherig]" dargestellt werden, sich auf das neueste Pixelfragment für einen Pixelort beziehen.

[0523] In einer alternativen Ausführungsform, bei der die Pixelengine die Fragmentliste durchläuft, um ein Pixelfragment innerhalb der Farb- und Tiefentoleranzen zu finden, beziehen sich die Komponenten, die als „prev.

[vorherig]" dargestellt werden, auf jedes der Pixelfragmente in der Fragmentliste für den Pixelort, die unter Verwendung des Verschmelzungstests analysiert werden.

[0524] Die Verschmelzungstestblöcke **1000**-**1008** vergleichen die Tiefen-, Farb- und Alphakomponenten für neue und vorherige Pixelfragmente, und wenn die neuen und vorherigen Werte innerhalb einer vorgegebenen Toleranz liegen, geben sie ein Bit aus, das anzeigt, dass das neue Pixelfragment ein Verschmelzungskandidat ist. Die Pixelengine führt dann eine bitweise AND-Operation aus (**1010**), um festzustellen, ob jeder der Verschmelzungstests bestanden wurde. Wenn ja, verschmilzt die Pixelengine die neuen und vorherigen Pixelfragmente. Die Pixelengine berechnet durch OR-Verknüpfung der neuen und vorherigen Bedeckungsmaske eine neue Bedeckungsmaske für das vorherige Fragment. Wenn einer der Verschmelzungstests versagt, fügt die Pixelengine das neue Pixelfragment an oberster Stelle in die Fragmentliste ein. Dieses neue Pixelfragment wird Teil der verlinkten Liste und weist auf den vorherigen Kopf der Fragmentliste.

[0525] Fig. 31 ist ein Schema, das ein Verschmelzungstestmodul **1012** detailliert illustriert. Das Verschmelzungstestmodul berechnet den absoluten Wert des Unterschieds zwischen einem neuen und einem vorherigen Wert **1014**. Ein Komparator **1016** im Verschmelzungstestmodul vergleicht den Unterschied mit einem Referenzwert, der in einem Tiler-Register **1018** gespeichert ist, und liefert einen booleschen Wert, der anzeigt, ob der neue und der vorherige Wert innerhalb der vorgegebenen Toleranz liegen. Die Ausgabe des booleschen Werts aus den Verschmelzungstestmodulen sind Eingabe in den bitweisen AND-Block **1010**, wie in Fig. 30 gezeigt. Die Ausgabe des bitweisen AND zeigt an, ob jede der Farben, das Alpha und der Tiefenwert innerhalb der vorgegebenen Toleranzen liegen. Wenn ja, verschmilzt die Pixelengine die einlaufenden und die obersten Pixelfragmente, wie oben beschrieben.

[0526] Wie oben bemerkt, gibt es eine Reihe von möglichen Variationen zu dem Verfahren zum Verschmelzen von Pixelfragmenten. In einer alternativen Implementierung kann die Pixelengine eine Fragmentliste durchsuchen und einen Verschmelzungstest an jedem Pixelfragment ausführen, bis:1) sie das Ende der Liste erreicht, oder 2) ein gespeichertes Pixelfragment findet, das den Verschmelzungstest besteht. In einer anderen Implementierung kann die Pixelengine die Pixelfragmente in sortierter Form bewahren, indem sie zum Beispiel eine Einfügesortierung mit jedem einlaufenden Fragment ausführt. Die Pixelengine kann versuchen, ein einlaufendes Pixelfragment nur mit dem Pixelfragment zu verschmelzen, das dem Betrachtungspunkt (mit dem niedrigsten Z-Wert) am nächsten ist, oder kann versuchen, es mit mehreren Pixelfragmenten zu verschmelzen, die für einen Pixelort gespeichert sind.

Unterteilen von Bildregionen, um Pixelspeicherüberlauf zu verhindern

[0527] Wenn der Tiler geometrische Grundelemente rastert, speichert er Pixeldaten in Pixel- und Fragmentpuffern. Der Tiler löst dann diese Pixeldaten in einem Nachverarbeitungsschritt auf. Weil der Tiler einen Speicher fester Größe zum Speichern dieser Pixeldaten verwendet, ist es möglich, dass er die Speicherkapazität überschreitet. Um dieses Problem anzugehen, überwacht der Tiler die Speicherkapazität und, falls notwendig, reduziert er die Größe des Bildbereichs, der aktuell gerade gerendert wird, um ein Überlaufen des Fragmentspeichers zu verhindern.

[0528] In einer Ausführungsform baut der Tiler das Grafikausgabebild durch Verarbeiten einer Reihe von 32×32-Pixelchunks auf. Fig. 32 ist ein Diagramm, das einen Teil des Pixel- und Fragmentpuffers illustriert. Wie in diesem Beispiel gezeigt, löst der Tiler den 32×32-Pixelpuffer (**1118**) unter Verwendung eines zugehörigen 512-Eintrags-Fragmentpuffers auf (**1120**). In dieser Implementierung kann der Fragmentpuffer bis zu 512 Pixelfragmenten speichern, die in einem späteren Verarbeitungsschritt kombiniert werden, um den 32×32-Pixelausgabepuffer zu bilden. Bei der Verwendung eines Fragmentpuffers mit 512 Einträgen, um einen 32×32-Ausgabepixelpuffer zu erzeugen, gibt es eine eindeutige Möglichkeit, dass der Fragmentspeicher knapp wird, wenn fein mosaikartig gemusterte grafische Objekte oder Objekte, die eine beträchtliche Lichtdurchlässigkeit aufweisen, gerastert werden. In diesen Fällen ist mehr Fragmentspeicher erforderlich, um Pixelfragmentdaten für teilweise bedeckte oder lichtdurchlässige Pixel zu speichern. Ein Fragmentpuffer mit 512 Pixeleinträgen speichert nur halb so viele Pixel wie der 32×32-Ausgabepuffer, der 1024 (32 × 32 = 1024) Pixel speichert.

[0529] Zur Abmilderung der Auswirkungen dieser Speicherbegrenzung ist das Pixelspeicherformat im Tiler so strukturiert, dass es 2 Ebenen von hierarchischer Zerlegung unterstützt. Fig. 33 ist ein Diagramm, das diese hierarchische Zerlegung abbildet. Wenn der Fragmentspeicher bei der Verarbeitung eines 32×32-Pixelpuffers erschöpft ist, löscht der Tiler die Pixel- und Fragmentpuffer und verarbeitet den Eingabestrom von geometrischen Grundelementen für einen Satz von vier 16×16-Pixelteilpuffern neu (**1122**). Das Verarbeiten eines 16×16-Pixelpuffers mit dem 512-Eintrag-Speichersystem ergibt doppelt so viele Fragment-Einträge wie ge-

wünschte Ausgabepixel, womit eine sehr große Mehrheit von Fällen mit zahlreichen teilweise bedeckten oder lichtdurchlässigen Pixeln erfasst wird.

[0530] Wenn der Fragmentspeicher bei der Verarbeitung eines der 16×16-Pixelteilpuffers erschöpft ist, löscht der Tiler die Pixel- und Fragmentpuffer und verarbeitet den Eingabestrom von Grundelementen für einen Satz von vier 8×8-Pixelteilpuffern neu (**1124**). Jeder 16×16-Pixelteilpuffer kann in vier 8×8-Pixelteilpuffer für insgesamt sechzehn 8×8-Teilpuffer aufgeteilt werden. Das Verarbeiten eines 8×8-Pixelpuffers mit dem 512-Eintrag-Speichersystem ergibt acht mal so viele Einträge wie gewünschte Ausgabepixel, womit die meisten denkbaren komplexen Grafikobjekte erfasst werden. Ein zusätzlicher Vorteil der 8×8-Teilpuffer ist, dass sie im Format sind, das von der Kompressions-Engine gefordert wird, die zum Komprimieren der Pixeldaten verwendet wird. Also ist keine weitere Pixelpufferzerlegung vor der Kompression erforderlich.

[0531] Wenn jeder Pixelteilpuffer (d.h. entweder der 16×16 oder der 8×8) erfolgreich verarbeitet ist, werden die Pixel aufgelöst und an die Kompressions-Engine gesandt. Da der Tiler die 16×16- und 8×8-Teilpuffer in der Reihenfolge der Auflösung und Kompression eines vollständigen 32×32-Pixelpuffers verarbeitet, führt der Abschluss der gesamten Verarbeitung aller Teilpuffer zu einem vollständigen 32×32-Pixelpuffer, der im Systemspeicher in einem komprimierten Format, ohne zusätzliche Verarbeitungsanforderungen, gespeichert wird.

[0532] Der Pufferzerlegungsprozess wird rekursiv im Vorübergehen angewendet, um mit anspruchsvollen Fällen (z.B. überlappende fein mosaikartig strukturierte Objekte mit beträchtlicher Lichtdurchlässigkeit, Schatten und Beleuchtung mit mehr als einer Lichtquelle) umzugehen. Die folgende Beschreibung illustriert das Verfahren.

[0533] Die **Fig. 34A**-B ist ein Flussdiagramm, das ein Verfahren zur Pufferzerlegung im Tiler illustriert. In einer Vorverarbeitungsphase erzeugt der DSP den Eingabedatenstrom einschließlich der Renderbefehle und der Polygone, die unter Bildregionen, Chunks genannt, sortiert sind. Der DSP leitet dann einen Eingabedatenstrom an den Tiler zur Verarbeitung. Als Reaktion auf die Renderbefehle im Eingabedatenstrom rastert ein Rastergenerator im Tiler Polygone im Eingabedatenstrom, um Pixeldaten zu erzeugen (**1130**, **1132**, **1136**).

[0534] In diesem speziellen Beispiel illustriert das Flussdiagramm, dass Polygone seriell verarbeitet werden. Es gibt jedoch eine Reihe von Möglichkeiten, Grundelemente zu rendern. Die Art und Weise, wie die Grundelemente gerastert werden, ist für den Zerlegungsprozess nicht kritisch.

[0535] Wenn der Rastergenerator Pixeldaten erzeugt, überwacht er die Kapazität des Fragmentpuffers. In dieser Implementierung erhöht der Rastergenerator schrittweise einen Pufferzähler für jeden Eintrag, der dem Fragmentspeicher hinzugefügt wird, und kontrolliert den Wert des Zählers, während er Pixeldaten erzeugt (**1138**, **1142**). Wenn der Wert des Pufferzählers **512** erreicht, dann ist der Fragmentspeicher voll. An diesem Punkt kontrolliert der Tiler die aktuelle Chunkgröße, um festzulegen, wie man ihn unterteilt (**1144**, **1150**).

[0536] In der speziellen Implementierung, die hier beschrieben und illustriert wird, wird die Speicherzerlegung ausgelöst, wenn der Fragmentspeicher seine Kapazität, 512 Pixelfragmente, erreicht. Es ist jedoch möglich, die Zerlegung in Gang zu setzen, bevor der Fragmentspeicher seine volle Kapazität erreicht.

[0537] Wenn die Chunkgröße 32×32 Pixel ist (**1144**), dann teilt der Tiler die Chunkgröße in vier 16×16-Pixelchunks (**1146**). Der Tiler leert dann die Pixel- und Fragmentpuffer (**1146**) und beginnt mit dem Rastern des Eingabestroms für den aktuellen Chunk in die vier 16×16-Teilchunks (**1158**). Bei dieser Implementierung sendet der DSP den Eingabedatenstrom für den Chunk erneut. Statt die Polygone unter den Teilchunks neu zu sortieren, verarbeitet der Tiler den Eingabestrom von Polygonen wiederholt für jeden Teilchunk und weist Polygone ab, die außerhalb der jeweiligen Teilchunks fallen. Als Alternative kann der DSP den Eingabedatenstrom erneut verarbeiten, wobei die Polygone im Strom unter den jeweiligen Teilchunkregionen sortiert werden. Diese Alternative reduziert die Zahl der Polygone für jeden Teilchunk, erhöht aber den Zusatzaufwand für die Verarbeitung im DSP.

[0538] Der Tiler verarbeitet 16×16-Teilchunks in ähnlicher Weise (**1150**, **1152**). Wenn die aktuelle Chunkgröße 16×16 Pixel ist, dann teilt der Tiler den Chunk in vier 8×8 Pixel und leert die Pixel- und Fragmentpuffer (**1152**). Bei dieser Implementierung unterteilt der Tiler die Chunks nicht in kleinere als 8×8-Blöcke. Die Kapazität des Fragmentspeichers, in diesem Fall **512** Elemente, sollte ausreichend sein, selbst fein mosaikartig strukturierte und/oder lichtdurchlässige Objekte durch Unterteilen von Bildchunks in 8×8-Blöcke zu handhaben. Jedoch ist der Tiler, der hier beschrieben wird, nur eine mögliche Implementierung: Die Notwendigkeit, die Größe des Bildes aufzuteilen, kann je nach solchen Faktoren, wie Komplexität der Szene, Form des Anti-Aliasing und der

Lichtdurchlässigkeit, die unterstützt wird, und Speicherkapazität des Fragmentpuffers, variieren.

[0539] Wenn der Pufferzähler **512** für einen 8×8-Pixelblock erreicht, löst der Tiler die Pixelfragmente auf, die mit dem 8×8-Pixelchunk verbunden sind, und führt einen Puffertausch aus (**1154**). Nachdem der 8×8-Chunk aufgelöst ist, kontrolliert der Tiler, ob es mehr 8×8-Pixelchunks gibt (**1156**). Wenn es weitere 8×8-Pixelchunks gibt, dann setzt sich die Verarbeitung durch Neustart der Polygonverarbeitung für den nächsten 8×8-Teilchunk fort (**1158**).

[0540] Wenn es keine 8×8-Chunks mehr gibt, dann kontrolliert der Tiler, ob es weitere 16×16-Pixelchunks gibt (**1148**). Wenn zusätzliche 16×16-Pixelchunks bleiben, dann startet der Tiler die Polygonverarbeitung für alle restlichen 16×16-Teilchunks neu (**1158**). Wenn es keine zusätzlichen 16×16-Pixelchunks mehr gibt, dann erhält der Tiler den Eingabedatenstrom für den nächsten Chunk (**1160**) und fährt mit der Verarbeitung der Polygone darin fort (**1158**).

[0541] Wenn die Kapazität des Fragmentpuffers während der Verarbeitung des Eingabedatenstroms für einen Chunk oder Teilchunk nicht überschritten wird, dann fährt der Tiler mit der Auflösung der Pixeldaten in den Pixel- und Fragmentpuffern fort (**1132**, **1134**). Wenn der Tiler die Verarbeitung des Eingabedatenstroms für den aktuellen Chunk abschließt, dann setzt er die Auflösephase für den Chunk oder Teilchunk in Gang. Wenn zum Beispiel die Chunkgröße 32×32 Pixel ist (**1162**), dann wird der 32×32-Pixelchunk aufgelöst, und die Puffer werden getauscht (**1164**). Die Verarbeitung setzt sich dann mit dem Erhalt des nächsten Chunks fort (**1160**) (**Fig. 34A**).

[0542] Wenn die Chunkgröße 16×16 Pixel ist (**1166**), dann wird der 16×16-Pixelchunk aufgelöst, und die Puffer werden getauscht (**1168**). Der Tiler fährt dann mit der Kontrolle fort, ob weitere 16×16-Chunks übrig sind (**1148**). Wenn ja, startet die Polygonverarbeitung neu durch erneutes Senden der Polygone für den nächsten Teilchunk (**1158**). Wenn nicht, dann ruft er den Eingabestrom für den nächsten Chunk ab und startet die Verarbeitung der Polygone für diesen Chunk (**1160**).

[0543] Wenn die Chunkgröße nicht 16×16 Pixel ist, dann beträgt sie als Vorgabe 8×8 Pixel. Der Tiler fährt mit dem Auflösen des 8×8-Pixelchunks und dem Tauschen der Puffer fort (**1154**). Der Tiler verarbeitet dann alle restlichen 8×8-Teilchunks und dann alle restlichen 16×16-Teilchunks. Nach dem Abschluss der Verarbeitung aller restlichen Teilchunks fährt der Tiler mit dem nächsten Chunk fort. Die Verarbeitung endet schließlich, wenn es keine weiteren Chunks im Eingabedatenstrom mehr gibt.

[0544] Während der Chunkverarbeitung werden Daten gesammelt, um die Maximalzahl der Pixelfragmente festzustellen, die jeder Chunk erzeugt. Die Zahl der Einträge, die im 512-Fragmentpuffer nach der Verarbeitung jedes Chunks frei ist, wird ebenfalls gesammelt. Diese Daten werden als Hilfe bei der Feststellung verwendet, wann die Pufferzerlegung automatisch beim Wiederverarbeiten eines Objektes ausgeführt werden sollte. Wenn zum Beispiel ein komplexes Objekt eine Reihe von Malen im Verlauf eines Spiels neu gezeichnet wird, würde die Verarbeitung des komplexen Objektes automatisch die Pufferzerlegung auf der Basis der Pixelpufferdaten einschalten, die gesammelt wurden, um das kontinuierliche Wiederverarbeiten des Eingabestroms von Pixelinformationen zu verhindern.

[0545] Die Pufferzerlegung in 16×16- oder 8×8-Teilpuffer kann auch angefordert werden, wenn ein bekannter komplexer (d.h. fein mosaikartig strukturierter) Pixelchunk an den Tiler gesendet wird. Dadurch wird die Bestimmung einer Notwendigkeit der Pufferzerlegung, Leeren der Pixel- und Fragmentpuffer und Wiederverarbeitung des Eingabestroms beseitigt, wenn ein Pixelchunks bekanntermaßen komplex ist und eine intensive Verarbeitung erfordert.

[0546] Es gibt mindestens zwei alternative Verfahren zum Neustarten des Scanumwandlungsprozesses, wenn ein Uberlauf entdeckt wird. Bei einem Verfahren kann die Pixelengine den Scanumwandlungsblock anweisen, anzuhalten, wenn ein Überlauf festgestellt wird, und dann alle Fragmentlisten im Pixelspeicher für Pixelorte außerhalb des Teilchunks, der verarbeitet werden soll, zu leeren. Um dies zu erreichen, findet die Pixelengine Fragmentlisten außerhalb des Teilchunks durch Lesen der Fragmentlistenzeiger im Pixelpuffer an den Pixelorten außerhalb des Teilchunks, und durch Freisetzen der Fragmente im Fragmentpuffer, der mit diesen Pixelorten verbunden ist. Der Scanumwandlungsblock fährt dann mit dem Rastern des aktuellen Satzes von geometrischen Grundelementen für den Chunk fort, wo er aufgehört hat.

[0547] In einem zweiten Verfahren startet der Scanumwandlungsblock nach dem Löschen des gesamten Fragmentspeichers neu. In diesem Fall startet der Scanumwandlungsblock neu und beginnt mit dem Rastern

der geometrischen Grundelemente am Beginn des Satzes von Grundelementen für einen Chunk.

[0548] Pufferzerlegung im Vorübergehen (On-the-fly) sorgt für eine Möglichkeit, einen kleinen Pixelausgabepuffer, einen kleinen Umfang an Fragmentpufferspeicher zu verwenden und den Fragmentdatenspeicherüberlauf während der Verarbeitung von grafischen Objekten zu reduzieren, selbst beim Verarbeiten von grafischen Objekten, die sehr komplexe Kennwerte besitzen (z.B. mehrere Lichtquellen, fein mosaikartige Struktur, Lichtdurchlässigkeit usw.).

[0549] Obwohl wir die Zerlegung in Form von speziellen Ausführungsformen beschrieben haben, versteht es sich, dass die Erfindung in einer Reihe von anderen Arten und Weisen implementiert werden kann. Es ist nicht notwendig, Bildregionen in der speziellen Weise, die beschrieben wurde, aufzuteilen. Vielmehr können Bildregionen in Teilregionen unterschiedlicher Größe aufgeteilt werden. Obwohl die Chunking-Architektur für die Bildaufteilung besonders geeignet ist, kann auch ein vollständiger Bildpuffer in kleinere Regionen zerlegt werden, um die Fragmentspeicheranforderungen zu reduzieren. Die speziellen Arten von Logik oder Software, die zum Verfolgen des Speicherverbrauchs verwendet werden, können ebenfalls variieren. Kurz gesagt, es gibt eine Reihe von möglichen alternativen Implementierungen innerhalb des Geltungsbereichs der Erfindung.

Pixelnachverarbeitung

[0550] Nachdem der Bildprozessor Fragmentdaten für einen Pixelort erzeugt hat, sortiert er dann diese Fragmentdaten und löst sie auf, um die Farbe an diesem Ort zu berechnen. Wie oben beschrieben, erzeugt der Bildprozessor Fragmente für teilweise bedeckte Pixel und bewahrt sie auf. Ein Pixel wird von einem Polygon teilweise bedeckt, wenn eine oder mehrere Kanten des Polygons das Pixel schneiden oder wenn das Polygon Lichtdurchlässigkeit besitzt. Das Aufbewahren der Fragmentdaten zum Ausführen von Anti-Aliasing- und Lichtdurchlässigkeitsberechnungen kann eine beträchtliche Speicherplatzmenge erfordern. Wenn sich die Zahl der gerenderten Polygone erhöht, erhöht sich auch die Speicherplatzmenge zum Speichern der Pixeldaten und Fragmente.

[0551] Neben den erhöhten Speicheranforderungen kann auch der Verarbeitungsumfang, der zum Auflösen der Fragmente erforderlich ist, beträchtlich sein. Bei einem Z-Puffer-Ansatz werden Fragmentdaten nach Tiefe sortiert. Die Grundelementedaten werden im allgemeinen nicht nach der Tiefenreihenfolge sortiert, wenn sie zum Rendern ankommen. Da die Grundelementedaten in beliebiger Tiefenreihenfolge ankommen, muss der Bildprozessor die Fragmentdaten nach ihrer Erzeugung sortieren. Die sortierten Daten werden dann verarbeitet, um die Farbe und möglicherweise das Alpha an einem Pixelort zu bestimmen. An jedem Pixelort können mehrere Fragmente zur Farbe beitragen. Wenn Alpha ebenfalls berechnet wird, steigt auch die Zahl der Fragmente und die Komplexität der Verarbeitung.

[0552] Aus den Gründen, die oben hervorgehoben wurden, können die Speicher- und Verarbeitungsanforderungen zur Unterstützung des fortgeschrittenen Anti-Aliasing und der Lichtdurchlässigkeit beträchtlich sein. Es gibt einen Konflikt zwischen der Unterstützung der anspruchsvollen Anti-Aliasing- und Lichtdurchlässigkeitsberechnungen einerseits und Verringerung der Speicheranforderungen andererseits. Um die Kosten des Systems zu reduzieren, sollte die Verwendung von Speicher minimiert werden, jedoch erfordern fortgeschrittene Anti-Aliasing- und Lichtdurchlässigkeitsmerkmale mehr Speicher. Es ist sogar noch schwieriger, diese fortgeschrittenen Merkmale in einem Echtzeitsystem bei gleichzeitiger Minimierung der Speicheranforderungen zu unterstützen.

[0553] In einer Ausführungsform rendert unser System Grundelemente, jeweils einen Chunk auf einmal, was den Speicher reduziert und mehr Fragmentauflösung in einem Nachverarbeitungsschritt ermöglicht. Während Pixeldaten für einen Chunk erzeugt werden, können Pixeldaten eines weiteren Chunks aufgelöst werden. Eine Reihe von Vorteilen, die sich auf die Fragmentsortierung und die Pixelauflösung auswirken, folgen aus dem Chunkingkonzept. Speicheranforderungen werden wesentlich reduziert, weil ein großer Teil der Daten, die während des Rasterungsprozesses erzeugt wurden, nicht behalten werden muss, nachdem der Bildprozessor die Pixel in einem Chunk aufgelöst hat. Der Bildprozessor braucht nur den aufgelösten Farbanteil nach dem Auflösen eines Chunks zu behalten.

[0554] Ein weiterer Vorteil des Renderns von Chunks in Serie ist, dass der Pixel- und Fragmentspeicher implementiert werden kann, um den Zusatzaufwand für Speicherzugriffe zu reduzieren. Typische Grafiksysteme verwenden externe Speicher zum Implementieren von Farb-, Tiefen- und Fragmentpuffern. Es ist sehr schwierig, diesen externen Speicher zu organisieren, so dass die rigorosen Bandbreitenanforderungen der Echtzeit-Bildverarbeitung erfüllt werden. Der Pixel- und Fragmentspeicher, der zur Unterstützung des Renderns ei-

nes Chunks, wie zum Beispiel einer 32×32-Pixelregion, benötigt wird, muss sich nicht im externen Speicher befinden. Er kann statt dessen auf derselben Hardware implementiert werden, die die Raster- und Anti-Aliasing-Funktionen ausführt. In der Implementierung, die oben beschrieben wird, kann zum Beispiel der Fragment- und Pixelpuffer auf einem einzelnen integrierten Schaltkreischip implementiert werden.

[0555] Die Verwendung von chipintegrierten Speichern vereinfacht die Bandbreitenprobleme, die mit externem Speicher verbunden sind. Chipintegrierte Speicher ermöglichen eine effiziente Nutzung mehrerer Speicherbänke. Eine Bank kann zum Beispiel für den Pixelpuffer verwendet werden, und eine weitere Bank kann für Fragmentdatensätze verwendet werden.

[0556] Ein weiterer Vorteil des chipintegrierten Speichers ist, dass er weniger teuer und leichter zu implementieren ist als Mehr-Port-Speicher. Die Leistungsfähigkeit der Pixel- und Fragmentpuffer kann durch die Verwendung von Mehr-Port-Speichern verstärkt werden, die gleichzeitiges Lesen und/oder Schreiben ermöglichen, um eine Verarbeitungsrate von einem Takt pro Pixel zu erreichen. Da der Fragmentpuffer viel kleiner ist, wenn Chunks separat gerendert werden, kann er auf dem Chip implementiert werden. Sowohl die geringere Größe des Speichers als auch sein Vorhandensein auf dem Chip machen die Verwendung von Mehr-Port-Speicher technisch anwendbar und kostengünstig. Externe Mehr-Port-Speicher sind andererseits auf Grund der höheren Kosten pro Bit und Verbindungen zwischen den Chips teuer.

[0557] Ein weiterer wichtiger Vorteil in Bezug auf das Chunking ist, dass Pixel für einen Teil eines Teilbildes (Frame) erzeugt werden können, während Pixel für einen anderen Teil aufgelöst werden. Statt Pixel für ein ganzes Teilbild zu erzeugen und dann diese Pixel aufzulösen, kann unser Ansatz die Prozesse der Erzeugung des Auflösens von Pixeln überlappen lassen, was die Systemtransportverzögerung reduziert.

[0558] In einer Ausführungsform unseres Systems löst der Bildprozessor Fragmente in einem Nachverarbeitungsschritt auf. Während die Pixelengine Pixeldaten für einen Teil eines Bildes erzeugt, löst die Anti-Aliasing-Engine Fragmente für einen anderen Teil eines Bildes auf. Wie oben bemerkt, sind die Pixeldaten doppelgepuffert: Die Pixelengine kann auf einen Puffer zugreifen, während die Anti-Aliasing-Engine auf den anderen zugreift. Nachdem die Pixelengine Pixel für einen Chunk erzeugt hat, führt der Tiler einen Puffertausch aus. Die Pixelengine erzeugt dann Pixel für den nächsten Chunk, und die Anti-Aliasing-Engine löst die Pixel für den vorherigen Chunk auf.

[0559] Obwohl er in der bevorzugten Ausführungsform auch doppelgepuffert sein könnte, hat der Fragmentpuffer zwei Ports, so dass die Pixelengine und die Anti-Aliasing-Engine darauf gleichzeitig zugreifen können. Die Pixelengine kann dann Fragmentdaten in den Fragmentpuffer durch einen Port schreiben, während die Anti-Aliasing-Engine auf Fragmentdaten durch einen anderen Port zugreift.

[0560] In dieser Ausführungsform ermöglichen es die doppelgepufferten und zwei Ports umfassende Speichersysteme dem Bildprozessor, Pixeldatenerzeugung und Pixelauflösung zu überlappen. Es gibt eine Reihe von alternativen Möglichkeiten, um eine Doppelpufferungsmethode zu implementieren.

[0561] Der Bildprozessor sortiert die Fragmentdaten nach der Tiefe, bevor er den Auflösungsprozess abschließt. Der Bildprozessor kann im allgemeinen Pixeldaten sortieren, während er Pixel erzeugt, und nachdem er Pixel für einen Teil eines Bildes, das gerendert werden soll, erzeugt hat. Die Pixelengine kann zum Beispiel eine Einfügesortierung ausführen, während sie Fragmentdaten in den Fragmentpuffer schreibt. Außerdem kann die Pixelengine Fragmentdaten sortieren, nachdem sie das Erzeugen von Pixeldaten für das ganze Bild oder einen Teil eines Bildes abgeschlossen hat. Die Pixelengine kann auch Fragmente in Fällen sortieren, in denen sie einlaufende Pixeldaten zurückweist. Da die Pixelengine nicht in den Fragmentpuffer schreiben muss, wenn die einlaufenden Pixeldaten zurückgewiesen werden, kann sie eine Sortierung von Fragmenten vornehmen, bevor das nächste hereinkommende Pixel ankommt. Wir bezeichnen diesen letzteren Ansatz als „Hintergrundsortieren" von Fragmenten.

[0562] Eine Einfügesortierung bezieht sich auf die Tiefensortierung eines ankommenden Fragmentes mit anderen Fragmenten im Fragmentpuffer. In einem Echtzeitsystem wird möglicherweise eine Einfügesortierung nicht bevorzugt, weil sie potenziell den Prozess der Erzeugung von Pixeldaten verlangsamen kann. Das Durchsuchen des Fragmentpuffers, um den richtigen Einfügungspunkt für ein ankommendes Fragment zu finden, kann einen unerwünschten Zusatzaufwand verursachen. Zusätzlich erfordert sie bei Hardware-Implementierungen zusätzliche Hardware und verkompliziert die Konstruktion der Pixelengine.

[0563] Als Alternative zur Einfügesortierung können Fragmente sortiert werden, nachdem der Bildprozessor

die Pixelerzeugung für einen Teil eines Bildes abgeschlossen hat. Einige Systeme rendern einen ganzen Rahmen von Bilddaten auf einmal. In solchen Systemen kann das Sortieren von Fragmenten für jeden Pixelort im Betrachtungsraum zusätzliche Verarbeitungszeit erfordern und eine unerwünschte Verzögerung hinzufügen, besonders für ein Echtzeitsystem. Die Zeitdauer, die zur Ausführung der Sortierung benötigt wird, kann je nach der Zahl der Fragmente pro Pixel und je nach dem Grad, bis zu dem die Einfügesortierung bereits ausgeführt ist, variieren. Die Sortieroperation kann daher andere Pixeloperationen am Ablaufen hindern, was die Leistungsfähigkeit verringert.

[0564] Durch Rendern jeweils eines Teils des Betrachtungsraums auf einmal kann die Fragmentsortierung für einen Teil eines Bildes ablaufen, während ein nächster Teil gerade gerastert wird. Die Anti-Aliasing-Engine kann im wesentlichen eine Fragmentsortierung in einem Nachverarbeitungsschritt ausführen. In einer Ausführungsform sortiert die Anti-Aliasing-Engine Fragmente für einen Chunk, während Fragmente für den nächsten Chunk gerade erzeugt werden.

[0565] Selbst in Fällen, in denen die Pixelerzeugung und -auflösung auf diese Weise überlappt sind, kann es trotzdem vorteilhaft sein, eine Sortierung von Fragmenten für einen Teil eines Bildes vorzunehmen, während die Pixelengine Pixel für diesen Teil des Bildes erzeugt. Die Hintergrundsortierung von Pixelfragmenten reduziert den Zusatzaufwand des Sortierens von Fragmenten, nachdem die Pixelengine die Erzeugung von Pixeln für einen Satz von Grundelementen beendet hat.

[0566] In einer Ausführungsform wird die Hintergrundsortierung gleichzeitig mit Pixeloperationen ausgeführt, die an Pixeln vorgenommen werden, um die Verzögerung, die für das Sortieren von Fragmenten erforderlich ist, zu reduzieren und in einigen Fällen zu beseitigen. Die Konstruktion nutzt die Tatsache aus, dass viele der Pixel teilweise nicht bedeckt sind, und macht daher keinen Gebrauch vom Fragmentpuffer. Die Hintergrundsortierung nutzt diese Reservebandbreite, um eine Sortierung eines Satzes von Fragmenten im Fragmentpuffer auszuführen.

[0567] Nach dem Sortieren löst der Bildprozessor die Fragmente für einen Pixelort auf, um die Farbe für diesen Pixelort festzustellen. Wenn Alpha nicht berücksichtigt wird, berechnet der Bildprozessor die Farbakkumulation auf der Grundlage der Farb- und Bedeckungsdaten für Fragmente in einer nach der Tiefe sortierten Liste für einen Pixelort. Wenn Alpha zusätzlich zu den Bedeckungsdaten berücksichtigt wird, berechnet der Bildprozessor die Farbakkumulation auf der Grundlage von Farbe, Bedeckung und Alpha der Fragmente in einer nach der Tiefe sortierten Liste für einen Pixelort.

[0568] Der Bildprozessor kann im allgemeinen jeweils Fragmente für Pixelorte auflösen, die dem gesamten Betrachtungsraum entsprechen, oder nur für einen Teil des Betrachtungsraums. In der Ausführungsform, die oben beschrieben wird, löst der Bildprozessor Pixelorte in einem Teil des Betrachtungsraums auf, der Chunk genannt wird. Die Fragmentauflösung läuft ab, nachdem die Fragmente erzeugt und sortiert wurden.

[0569] Die Fragmentauflösung ist der Prozess, während dem alle Fragmente für ein Pixel kombiniert werden, um einen einzelnen Farb- und Alphawert zu berechnen. Dieser einzelne Farb- und Alphawert wird in den Farbpuffer geschrieben (und dann komprimiert und in einen Gsprite abgelegt).

[0570] Das Berechnen der aufgelösten Farbe umfasst das Akkumulieren eines korrekt skalierten Farbbeitrags von jeder Schicht, während gleichzeitig die Bedeckungsinformationen berechnet und bewahrt werden, mit denen die nachfolgenden Schichten skaliert werden sollen. Diese Akkumulation kann in der Tiefenreihenfolge von vorn nach hinten oder von hinten nach vorn ausgeführt werden. Bei einem Ansatz von vorn nach hinten, als Gegensatz zu von hinten nach vorn, können räumliche Bedeckungsdaten zur Bestimmung der Bedeckung für aufeinander folgende Schichten verwendet werden. Anders als die Bedeckung, gelten Alphadaten gleichermaßen für den gesamten Pixelbereich.

[0571] Für die Reihenfolge vorn nach hinten sind die Gleichungen zum Berechnen von Farbe und Alpha für sortierte Fragmentdatensätze:
Alpha initialisiert auf Maximalwert (inverses Alpha).
Farbe initialisiert auf 0.

Anew = Aold – (Aold·Ain);

Cnew = Cold + (Cin·(Aold·Ain));

[0572] Für die Reihenfolge hinten nach vorn sind die Gleichungen zum Berechnen von Farbe und Alpha für sortierte Fragmentdatensätze:
Alpha und Farbe initialisiert auf 0.

Anew = Ain + ((1 – Ain)·Aold);

Cnew = (Cin·Ain) + ((1 – Ain)·Cold);

[0573] Für eine Hardware-Implementierung ist vorn nach hinten vorzuziehen, weil der Auflösungsprozess weniger hardwareintensiv ist.

[0574] Ein Pseudocode-Beispiel für die Akkumulation von Fragmenten nur mit Tiefe, Farbe und Bedeckung (kein Alpha) wird unten dargelegt:

```
NUM_CVG BITS ist die Zahl der Bits in der Bedeckungsmaske
MAX_ALPHA ist der maximale Alphawert
for (jeden fragmentierten Pixelort) {
        ColorAccum = 0;
        CoverageAccum = 0;
        while (Fragmentliste ist nicht leer) {
                scanne Fragmentliste und extrahiere nächstgelegenes Fragment(Bedeckung,
Farbe);
                ColorScale = CountSetBits(coverage & ~(CoveragcAccum))/NUM_CVG_BITS;
                ColorAccum += ColorScale * color;
                CoverageAccum (= coverage
        }
        ColorAccum ist die Pixelfarbe
}
```

[0575] Das Akkumulieren von Fragmenten mit Tiefe, Farbe, Bedeckung und Alpha erfordert, dass ein Alphawert für jedes Subsample berechnet und aufbewahrt wird. Dies ist auf die Kombination von Bedeckungsmasken und Alphawerten für jedes Fragment zurückzuführen. Es trifft im allgemeinen zu, dass das akkumulierte Alpha in jeder Schicht während der Akkumulation eine Funktion aller Alphawerte der vorherigen Schichten ist. Mit Bedeckungsmasken kann jedes Subsample potenziell einen unterschiedlichen Satz von ‚vorherigen' Alphawerten haben, da eine Schicht, für die das Bedeckungsbit nicht gesetzt ist, nicht zu diesem Subsample beiträgt.

[0576] Ein Ansatz zur Auflösung von Fragmenten sowohl mit Alpha als auch Bedeckung besteht darin, Farbe für jedes Teilpixel in einer Schicht separat zu berechnen und dann den Beitrag von jedem Teilpixelort zu addieren, um dem gesamten Farbbeitrag zu bestimmen. Der Alphamaßstabsfaktor für jedes Teilpixel wird aus dem Alpha in dieser Schicht zusätzlich zu dem Alpha bestimmt, das sich aus anderen Schichten angesammelt hat. Dieser Alphamaßstabsfaktor wird dann mit der Farbe für das Teilpixel multipliziert, um den Farbbeitrag des Teilpixels zu bestimmen. Die Farbe für eine Schicht wird dann durch Summieren der Farbbeiträge aus den Teilpixeln bestimmt.

[0577] Ein Beispiel für die getrennte Akkumulation von Farbe und Alpha für Teilpixel ist:

```
for (jeden fragmentierten Pixelort) {
        ColorAccum = 0;
        AlphaAccum[NUM CVG BITS] = {MAX_ALPHA, MAX_ALPHA,..., MAX.ALPHA };
        while (Fragmentliste ist nicht leer) {
                scanne Fragmentliste und extrahiere nächstgelegenes Fragment(Bedeckung,
Farbe);
        for (i=0; i<NUM_CVG_BITS; i++) {
//      wenn dieses Bit in der Bedeckungsmaske gesetzt ist
                if (Bedeckung >> 1) & 0x1 {
//      berechne Alphamaßstabswert - Beitrag für diese Farbe
                        AlphaScale = (alpha * AlphaAccum[i]);
//      füge Farbe hinzu, die die Alpha skaliert ist
                        ColorAccum += (color*AlphaScale) * (1/NUM_CVG_BITS));
//      berechne akkumuliertes Alpha für den Teilabfragewert
//      AlphaAccum = AlphaAccum*(MAX_ALPHA-alpha)=
//      AlphaAccum - AlphaAccum*alpha
AlphaAccum[i] = AlphaScale;
}
}
}
ColorAccum ist die Pixelfarbe
}
```

[0578] Ein Beispiel, das 4 Teilpixelorte verwendet, hilft, die Fragmentauflösung zu illustrieren. Bei diesem Beispiel betrachten wir drei Fragmente, wobei jedes eine Bedeckungsmaske, Alpha- und Farbwert hat. Der Anfangszustand wird unten in der Tabelle illustriert. In diesem Beispiel akkumulieren wir Farbe und Alpha unter Verwendung des Ansatzes vorn nach hinten. Das anfängliche Alpha wird auf 1 gesetzt, was volle Transparenz bedeutet. Die Daten für jede Schicht sind folgende: Fragment 0, Alpha = 0,5, Bedeckungsmaske (BM) = 0011, und Farbe = C_0; Fragment 1, Alpha = 0,3, BM = 1000, Farbe = C_1; Fragment 2, Alpha = 0,8, BM = 0101, Farbe = C_2. Die Daten für jedes Fragment werden in Tabellen unten bereitgestellt.

[0579] Mit den Alphawerten auf eins initialisiert, wird das Alpha-Bedeckungsfeld unten gezeigt.

1	1
1	1

[0580] Um die Farbe zu berechnen, werden die Farbwerte für jeden Teilpixelort mit dem neuen Alpha und dem Alpha aus dem Bedeckungsfeld multipliziert. Das Ergebnis für die Teilpixelorte wird dann durch vier dividiert (eins dividiert durch die Zahl der Teilpixelorte). Zum Schluss wird der Beitrag von allen Teilpixelorten summiert, um die akkumulierte Farbe festzustellen.

Bedeckungs-maske	Farbe	Alpha für neues Frag.	Alpha aus Bedeckungsfeld Alpha	Teilpixelbeitrag
1	C_0	0,5	1	¼
1	C_0	0,5	1	¼
0	C_0	0,5	1	¼
0	C_0	0,5	1	¼

[0581] Unter Verwendung der Formel, Alpha' = Alpha·(Max_alpha – new_alpha), berechnet der Bildprozessor das neue Alpha separat für jeden Pixelort und speichert ihn im Alpha-Bedeckungsfeld in der Tabelle unten.

0,5	0,5
1	1

[0582] Der Beitrag von Fragment 1 wird in der Tabelle unten dargestellt.

Bedeckungs-maske	Farbe	Alpha für neues Frag.	Alpha aus Bedeckungsfeld Alpha	Teilpixelbeitrag
0	C_1	0,3	0,5	¼
0	C_1	0,3	0,5	¼
0	C_1	0,3	1	¼
1	C_1	0,3	1	¼

[0583] Das neue Alphabedeckungsfeld ist folgendes:

0,5	0,5
0,7	1

[0584] Der Beitrag von Fragment 2 wird in der Tabelle unten dargestellt.

Bedeckungs-maske	Farbe	Alpha für neues Frag.	Alpha aus Bedeckungsfeld Alpha	Teilpixelbeitrag
1	C_2	0,8	0,5	¼
0	C_2	0,8	0,5	¼
1	C_2	0,8	1	¼
0	C_2	0,8	0,7	¼

[0585] Das Alpha-Bedeckungsfeld für die Fragmente nach Fragment 2 ist folgendes:

0,5	0,1
0,7	0,2

[0586] Dieses Verfahren erfordert 2·NUM_CVG_BITS Multiplikationen (2·16 = 48 im 4×4-Fall) pro Fragment für die Berechnung von Alpha und des Farbbeitrags. Man beachte, dass die (1/NUM_CVG_BITS)-Skalierung mit einer Verschiebung erfolgen kann, wenn die Zahl der Bits in der Bedeckungsmaske von der Größe 2**n ist (was normalerweise der Fall ist).

[0587] Fig. 35 ist ein Schema, das eine Hardware-Implementierung des Ansatzes illustriert, der oben für ein Pixel beschrieben wird, das in 4 × 4 Teilpixelregionen aufteilt ist (**1224**). Die Auflösungshardware umfasst einen Satz von 16 identischen Verarbeitungs- und Speichereinheiten, die Alpha- und Farbakkumulatoren genannt werden (ACA) (**1226**), wobei jeder ACA einer Teilpixelregion des Pixels zugewiesen ist. Während der Verarbeitung der Fragmentliste für jeden Pixelort werden die Bedeckungsmasken für jedes Fragment als Verarbeitungsmaske für die Auflösungshardware verwendet. Der ACA führt eine Multiplikation für den Alpha-Maßstab, Farbakkumulation und Alpha-Akkumulation aus. Die (1/NUM_CVG_BITS)-Skalierung wird mit einer Verschiebung ausgeführt, wie oben dargelegt. Nachdem alle Fragmente für einen gegebenen Pixelort verarbeitet wurden, kombiniert der Ausgabeabschnitt die Farb- und Alphawerte für alle 16 Teilpixel in einer hierarchischen Weise (**1228**). Die Prozessoren in der Ausgabe kombinieren die zwei eingehenden Werte und teilen durch 2. Bei Hardware-Pipelineverarbeitung verwendet der Pixelauflösungsprozess nur einen einzigen Hardware-Zeittakt pro Fragmenteintrag.

[0588] Ein alternatives Verfahren reduziert die Hardware-Anforderungen, indem Teilpixel, die denselben akkumulierten Alphawert haben, in jeder Schicht ähnlich behandelt werden. Dieses Verfahren beruht auf der Beobachtung, dass der Zustand, in dem Subsamples eindeutige akkumulierte Alphawerte haben, allmählich eintritt. Zu Anfang werden alle Subsample-Alphawerte auf null gesetzt (transparent). Die erste Fragmentakkumulation kann höchstens einen eindeutigen Alphawert hinzufügen, was zu einer Gruppe von Subsamples führt, die den anfänglichen Alphawert behalten, und der anderen Gruppe führt, die denselben neuen Alphawert hat. Die zweite Fragmentakkumulation kann zu nicht mehr als vier eindeutigen Alphawerten führen. Insgesamt gesehen, beträgt die Zahl der eindeutigen Teilabfrage-Alphawerte, die nach ‚n' Fragmentakkumulationen möglich ist, 2**n (oder genauer MIN(2**n, NUM_CVG_BITS).

[0589] Dieses alternative Verfahren verwendet dieses Merkmal, um die Zahl der Akkumulationen zu verringern, die zur Ausführung nur der Farbskalierung und der Akkumulation für jeden eindeutigen Alphawert innerhalb der Subsamples erforderlich sind, statt für jedes Subsample. Bei diesem Verfahren braucht höchstens nur ein Akkumulat für das erste Fragment aufzutreten, zwei für des zweite Fragment, vier für das dritte Fragment und so weiter, bis zu der Zahl der Subsamples im Pixel (z.B. bei einem 4×4-Subsample-Feld ist der schlimmste Fall 16 Akkumulationen pro Fragment).

[0590] Die Grundlage für dieses Verfahren besteht darin, den Satz von eindeutigen Alphawerten und ihren zugehörigen Bedeckungsmasken während der Fragmentakkumulation beizubehalten, wobei die Absicht ist, eine minimale Zahl von Farbakkumulationen auszuführen.

[0591] Das Alpha und die Bedeckungsmasken werden in NUM_CVG_BITS-Elementfeldern gespeichert, von

denen eine gewisse Teilmenge dieser Einträge zu einer beliebigen Zeit tatsächlich gültig (oder ‚in Gebrauch') ist. Die ‚in Gebrauch befindlichen' Einträge sind diejenigen, die den aktuellen Satz von eindeutigen Alphawerten enthalten. Die in Gebrauch befindlichen Einträge werden durch eine NUM_CVG_BITS-Bitmaske identifiziert, wobei ein gesetztes Bit anzeigt, dass das Feldelement an diesem Bitindex in Gebrauch ist. Es wird eine Konvention verwendet, bei der das erste gesetzte Bit in der Bedeckungsmaske eines Paares (eindeutiges Alpha, Bedeckungsmaske) definiert, in welchem Feldelement dieses Paar gespeichert ist. Man betrachte das folgende Beispiel dafür, wie das Feld initialisiert und bei der Akkumulation von drei Fragmenten (unter Verwendung von 4 Teilabfragewerten) aktualisiert wird:
Anfangszustand (X weist auf einen Wert hin, der nicht von Interesse ist):

0b0001 // in Gebrauch befindliche Maske

{ 1., 0b1 111 } l/ Paar Alpha, Bedeckung

{ X, 0bXXXX }

{ X, 0bXXXX }

{ X , 0bXXXX }

Akkumuliere Fragment { 0,5 /* Alpha */. 0b0011 /* Bedeckungsmaske */}

0b0101 // in Gebrauch befindliche Maske

{ .5, 0b001 I } // Alpha, Bedeckungspaare

{ X, 0bXXXX }

{ 1., 0b1100 }

{ X, 0bxxXX }

Fragment akkumulieren {0.3, 0b1000 }

0b1101 // in Gebrauch befindliche Maske

{0.5. 0b0011 } // Paare Alpha, Bedeckung

{ X, 0bXXXX }

{ 1.0, 0b0100 }

{0.7, 0b 1000 }

Fragment akkumulieren {0.8, 0b0101 }

0b1lll // in Gebrauch befindliche Maske

{0.1, 0b0001 } // Paare Alpha, Bedeckung

{0.5, 0b0010 }

{0.2, 0b0100 }

{0.7, 0b1000 }

[0592] Das anfängliche Alphabedeckungsfeld wird unten dargelegt:

x	1
x	x

[0593] Die in Gebrauch befindliche Maske ist 0001, was den Ort festlegt, wo die Feldmaske gespeichert ist. Die entsprechende Feldmaske ist folgende:

xxxx	1111
xxxx	xxxx

[0594] Nach Fragment 0 erscheint die Alpha-Bedeckungsmaske wie folgt.

x	0,5
x	1

[0595] Die in Gebrauch befindliche Maske ist 0101, die Feldmaske ist folgende:

xxxx	0011
xxxx	1100

[0596] Für Elemente in der in-Gebrauch-Maske, die gesetzt sind, wird die Feldmaske durch AND mit der Bedeckungsmaske für das neue Fragment verknüpft, um festzustellen, ob es eine Änderung im Alphawert gibt. Wenn es ein neues Alpha gibt, wird der neue Wert für die Feldmaske folgendermaßen berechnet: Feldmaske AND NOT Bedeckungsmaske. Wenn es einen neuen Wert für die Feldmaske gibt, wird er am entsprechenden Ort gespeichert.

[0597] Nach Fragment 1 erscheint die Alpha-Bedeckungsmaske wie folgt.

x	0,5
0,7	1

[0598] Die in-Gebrauch-Maske ist 0101, und die Feldmaske ist folgende:

xxxx	0011
1000	0100

[0599] Nach Fragment 2 erscheint die Alpha-Bedeckungsmaske wie folgt.

05	0,1
0,7	0,2

[0600] Die in-Gebrauch-Maske ist 1111, und die Feldmaske ist folgende:

0010	0001
1000	0100

[0601] Die Zahl von eindeutigen Alphawerten zu einem beliebigen Zeitpunkt ist gleich der Zahl der gesetzten Bits in der in-Gebrauch-Maske. Die vollständige Lösung umfasst zwei Schritte. Der erste Schritt ist die Ausführung der notwendigen Farbakkumulationen, wobei eine Akkumulation pro ‚in-Gebrauch'-Eintrag im Bedeckungs-/Alpha-Feld erforderlich ist. Der zweite Schritt besteht darin, das Bedeckungs-/Alpha-Feld mit den Werten des neuen Fragments zu aktualisieren.

[0602] Eine vollständige Implementierung dieses Verfahrens (für 4×4-Teilabfragewerte) folgt.

```
for (jeden fragmentierten Pixelort) {

//      Anfangszustand (pro Pixel)
InUseMask = 0x0001;
CoverageArrayMask[16]= { 0xfff, 0, ..., 0 );
CoverageArrayAlpha[16] = { MAX_ALPHA. MAX_ALPHA, ..., MAX-ALPHA
ColorAccum = 0;

while (Fragmentliste ist nicht leer) {
scanne Fragmentliste und extrahiere nächstgelegenes Fragment(Bedeckung, Farbe, Alpha);

// akkumuliere die Farbe dieses Fragments in ColorAccum für jedes In-Gebrauch-Element
InUseMaskScratch = InUseMask;
while (InUseMaskScratch != 0x0000) {
//      finde erstes gesetztes Bit in scratch in-use Mask
Index  FindFirstSetBit(InUseMaskScratch);
InUseMaskScratch &=~(0x1<<Index);

//      lies altes (oder aktuelles) Alpha für diesen Eintrag - dies wird verwendet
//      bei Aktualisieren der nicht bedeckten Fläche (die neu ‚in Gebrauch' sein kann)
AlphaOld = CoverageArrayAlpha[Index];

//      Alphaskalierungsfaktor - verwendet zum Skalieren von Farbe für Akkumulation und
//      zum  Berechnen von Alpha für nachfolgende Schichten
AlphaScale = AlphaOld * alpha;

//      berechne Alpha für nächste Schicht - verwende dies zum Aktualisieren des Alphafeldes
//      AlphaNext = AlphaOld*(MAX_ALPHA-alpha) = AlphaOld-AlphaOld*alpha
AlphaNext = AlphaOld - AlphaScale;
```

```
//      berechne Maske für überlappte Bedeckung - dies ist der Anteil von diesem
//      Feldeintrag, der von einem neuen Fragment bedeckt wird, also akkumulieren Sie die
//      Farbe und aktualisieren Sie das Feld mit dem neuen Alphawert
AccumCvgMask = Coverage & CoverageArrayMask[Index];
if (AccumCvgMask != 0x0000) (

//      Farbe akkumulieren

nCoverageBits = CountSetBits(AccumCvgMask);
ColorAccum += color*(AlphaScale * nCoverageBits/NUM CVG_BITS));
// Alpha für abgedeckten Teil aktualisieren (dies kann zu einem 'neuen'
//      in-Gebrauch-Element führen oder einfach den alten überschreiben)
Index2 = FindFirstSetBit(AccumCvgMask);
InUseMask |= (0x1 << Index2);
CoverageArrayMask[Index2] = AccumCvgMask;
CoverageArrayAlpha[Index2] = AlphaNext;
}
//      Berechne die Maske für den nicht abgedeckten Bereich - das ist der Abschnitt
//      dieses Feldeintrags, der vom neuen Fragment verdeckt wird, also
//      aktualisieren Sie nur die Bedeckung (das Alpha bleibt dasselbe )
UpdateCvgMask = ~Coverage & CoverageArrayMask[Index];
if (UpdateCvgMask!= 0X0000) {
        Index2 = FindFirstSetBit(UpdateCvgMask);
        InUseMask = (0x l << Index2);
//      Aktualisieren für den nicht abgedeckten Bereich - dies kann zu einem 'neuen'
//      in-Gebrauch-Element führen oder nur das alte überschreiben (daher den
//      Alpha-Wert kopieren, falls er neu ist ...)
CoverageArrayMask[Index2] = UpdateCvgMask;
CoverageArrayAlpha[Index2] = AlphaOld;
}
}
}
```

ColorAccum ist die Pixelfarbe

[0603] Die Kernarithmetikoperation ist die Farbakkumulation, die insgesamt drei Multiplikationen pro eindeutigem Alphawert erfordert.

[0604] ColorAccum += color·(alpha·AlphaOld·(nCoverageBits/NUM_CVG_BITS)); Man beachte, dass die dritte Multiplikation durch die Zahl der Subsamples etwas vereinfacht werden kann. Für 16 Subsamples beinhaltet die dritte Multiplikation den Festpunktwert 0,4, der Multiplikator kann also 8 × 4 sein (wo die anderen Multiplikatoren wahrscheinlich 8 × 8 sind). Man beachte auch, dass für 2**n große Bedeckungsmasken die Division, die oben gezeigt wird, nur eine Verschiebung ist.

[0605] Dieses Verfahren erfordert im schlimmsten Fall insgesamt:

$$\sum_{n=1}^{NumFrags} MIN(2^n, 16)$$

Akkumulationen. Der Normalfall kann kleiner als dies sein, weil der schlimmste Fall nur auftritt, wenn die Bedeckung eines neuen Fragments sowohl gesetzte als auch ungesetzte Werte in jedem ‚in-Gebrauch'-Feldelement hat.

[0606] Eine nützliche Optimierung besteht darin, die Bedeckungsmaskenorte zu verfolgen, die einen vollständig lichtundurchlässigen Alphawert haben. Dies ist nützlich in dem Fall, dass Fragmente gerade auf Grund der teilweisen geometrischen Bedeckung und nicht auf Grund von nicht lichtundurchlässigen Transparenzwerten erzeugt werden. Diese Fragmente haben normalerweise einen vollständig lichtundurchlässigen Transparenzwert. Die Implementierung dieser Optimierung erfolgt durch Bewahren eines zusätzlichen Maskenwertes, der OpaqueAlphaMask. Die OpaqueAlphaMask wird durch einen O-Ring in Bedeckungsmasken von Fragmenten gesetzt, für die der Alpha vollständig lichtundurchlässig ist (dies wird nach dem Akkumulieren des Beitrags des Fragments ausgeführt). Diese Maske wird dann dazu verwendet, Bits in den Masken nachfolgender Fragmente zu ignorieren, da es keinen weiteren Farbbeitrag zu den entsprechenden Subsamples gibt.

[0607] Eine weitere mögliche Optimierung besteht darin, Orte mit identischen Alphawerten zu konsolidieren, dies ist aber beträchtlich aufwändiger zu implementieren, und das Vorhandensein von identischen Alphawerten, die weder 0 noch MAX_ALPHA sind, ist nicht wahrscheinlich.

[0608] Das Beispiel und der Pseudocode, der oben angegeben wird, verwenden eine Tiefensortierung vorn nach hinten. Es ist gleichermaßen möglich, dieselben Berechnungen mit einer Tiefensortierung von hinten nach vorn auszuführen. Die Berechnungen, die oben angeführt werden, verwenden auch Farbkomponenten, die nicht vorher mit der Alphakomponente multipliziert wurden. Dasselbe Verfahren gilt für vormultiplizierte Farbkomponenten, mit leicht unterschiedlichen arithmetischen Berechnungen (und identischem Kontrollfluss).

[0609] Fig. 36 ist ein Schema, das eine Implementierung des hardwareoptimierten Fragmentauflösungsteilsystems in der Anti-Aliasing-Engine illustriert. Die Eingabe in das Teilsystem ist ein Strom von tiefensortierten Fragmentdatensätzen. Wie gezeigt, umfasst ein Fragmentdatensatz RGB-Farbwerte, einen Alphawert A und eine Bedeckungsmaske (Cov Mask). Dieses besondere Fragmentauflösungsteilsystem verarbeitet Fragmentdatensätze in der Reihenfolge vorn nach hinten und akkumuliert Farbwerte für den Pixelort, während es jede Fragmentschicht verarbeitet. Dieses Teilsystem minimiert die Hardware, die zum Akkumulieren der Farbwerte notwendig ist, weil es eindeutige Pixelregionen, die ein gemeinsames Alpha besitzen, verfolgt. Dies ermöglicht dem Fragmentauflösungsteilsystem, Farbe einmal für jede eindeutige Pixelregion zu skalieren und zu akkumulieren, anstatt separat für jede Teilpixelregion.

[0610] Wie im Pseudocode oben dargelegt, initialisiert das Fragmentauflösungssystem eine in-Gebrauch-Maske **1236**, ein Feld von Bedeckungsmasken **1230** und ein Feld von akkumulierten Alphawerten **1230** vor dem Auflösen einer Liste von Fragmentdatensätzen. Die Elemente in der in-Gebrauch-Maske **1236** repräsentieren Pixelregionen, wobei jedes ein oder mehrere Teilpixelregionen umfasst, die ein gemeinsames akkumuliertes Alpha haben. Die Bedeckungsmasken ergeben die Teilpixelorte, die durch eine Pixelregion bedeckt werden. Das Feld von akkumulierten Alphawerten speichert die eindeutigen akkumulierten Alphawerte für entsprechende Pixelregionen, die ein gemeinsames Alpha haben. Dieses besondere Bedeckungsfeld **1236** speichert die akkumulierten Alphawerte und Bedeckungsmasken.

[0611] Nach dem Initialisieren der in-Gebrauch-Maske, Bedeckungsfeldmaske und Bedeckungsfeldalpha beginnt das Teilsystem mit der Verarbeitung eines Fragmentdatensatzes, mit dem Fragmentdatensatz, der dem Betrachtungspunkt am nächsten liegt, beginnend. In einer Implementierung der Anti-Aliasing-Engine **412** im Tiler sortiert die Anti-Aliasing-Engine die Fragmentliste in einer Nachverarbeitungsstufe, nachdem der Scan-

umwandlungsblock **395** und die Texturfilterengine **401** das Rastern eines Chunks beendet haben. Die Anti-Aliasing-Engine liest jedes Fragment in einer Fragmentliste, mit dem Kopf beginnend, und legt dabei Einträge in ein sortiertes Feld von Indizes und Tiefen. Jeder Index in diesem Feld zeigt auf einen Fragmentpufferort, der die RGB-, Alpha- und Bedeckungsdaten für ein Pixelfragment in der Liste speichert. Während die Anti-Aliasing-Engine Pixelfragmente liest, führt sie eine Einfügesortierung derart aus, dass die Feldeinträge ein tiefensortiertes Feld von Indizes auf Pixelfragmente und entsprechende Tiefenwerte umfassen. Sobald die Liste sortiert ist, ruft das Fragmentauflösungsteilsystem tiefensortierte Fragmente durch Lesen jedes Eintrags im sortierten Feld in der Reihenfolge ab, in der diese Einträge im Feld gespeichert sind. Dies ermöglicht dem Fragmentauflösungssystem, die RGB-Farbwerte, Alpha und Bedeckungsmasken für die Pixelfragmente in einer Liste nach der Tiefe sortiert abzurufen.

[0612] Während es jeden Fragmentdatensatz in der Liste verarbeitet, verfolgt das Teilsystem die Pixelregionen, die ein gemeinsames Alpha haben. Das Teilsystem bestimmt, ob jeder Fragmentdatensatz in der Liste jede Pixelregion überlappt, die ein gemeinsames Alpha hat. Wenn ja, berechnet das Teilsystem die akkumulierte Farbe für den Abschnitt der aktuellen Pixelregion, die sich mit dem aktuellen Fragment überlappt. Wenn es eine Überlappung mit der aktuellen Pixelregion gibt, bestimmt das Teilsystem auch die neue Pixelregion oder -regionen, die durch diese Überlappung hervorgerufen werden, und verfolgt diese.

[0613] Für ein aktuelles Fragment (**1232**) durchläuft das Teilsystem jedes Element in der In-Gebrauch-Maske. Die Bedeckungsfeldschleifenkontrolle **1234** führt die In-Gebrauch-Maske (**1236**) und aktualisiert sie nach Notwendigkeit, während sie jeden Fragmentdatensatz verarbeitet. Während sie durch die Einträge in der In-Gebrauch-Maske schleift, kommuniziert die Bedeckungsfeldschleifenkontrolle mit der neuen Bedeckungskontrolle **1238** und steuert ihren Betrieb. Die neue Bedeckungskontrolle **1238** aktualisiert die Bedeckungsfeldmaske und Alpha **1230** nach Notwendigkeit, wenn das aktuelle Fragmente die aktuelle Pixelregion überlappt.

[0614] Die neue Bedeckungskontrolle **1238** liest das gespeicherte akkumulierte Alpha (Aold) aus dem Bedeckungsfeld-Alpha, das mit dem aktuellen Eintrag in der In-Gebrauch-Maske verknüpft ist, und berechnet den Alpha-Skalenfaktor (A·Aold), der zum Skalieren der Farbe und zum Berechnen von Alpha für nachfolgende Fragmentschichten verwendet wird Anext (1 – A·Aold). Die neue Bedeckungskontrolle **1238** überträgt den Alpha-Skalenfaktor (A·Aold) auf die Skalen- und Akkumulationskontrolle **1246** zur Verwendung beim Skalieren der Farbdaten des aktuellen Fragments. Die neue Bedeckungskontrolle **1238** berechnet auch das Alpha für nachfolgende Schichten, Anext (1 – A·Aold), und berechnet es zusammen mit seiner entsprechenden Bedeckungsfeldmaske im Bedeckungsfeld **1230**.

[0615] Für jede Pixelregion mit einem gemeinsamen akkumulierten Alpha bestimmt das Fragmentauflösungsteilsystem, ob das aktuelle Fragment die aktuelle Pixelregion überlappt, indem der Schnitt der Bedeckungsmasken des Fragments und der Pixelregion gesucht wird.

[0616] Wenn das aktuelle Fragment die aktuelle Pixelregion überlappt, 1) berechnet das Teilsystem die akkumulierte Farbe für den überlappten Abschnitt der Pixelregion, 2) aktualisiert das Teilsystem das In-Gebrauch-Element und die entsprechende Bedeckungsfeldmaske und Alpha (Bededeckungsfeld-Alpha) für dieses In-Gebrauch-Element.

[0617] Die Skalen- und Akkumulationskontrolle **1246** berechnet die akkumulierte Farbe für jede eindeutige Pixelregion, die vom aktuellen Fragment bedeckt wird. Die Skalen- und Akkumulationskontrolle umfasst einen Bedeckungsskalierer **1240**, einen Farbskalierer **1242** und einen Farbakkumulator **1244**. Der Bedeckungsskalierer **1240** berechnet einen Bedeckungsskalenfaktor (Zahl der Teilpixelorte in der aktuellen Pixelregion, die vom aktuellen Fragment bedeckt wird/Gesamtzahl der Teilpixelorte·A·Aold). Der Farbskalierer **1242** liest dann die Farbwerte (RGB) für das aktuelle Fragment (**1232**) und multipliziert sie mit dem Bedeckungsskalenfaktor aus dem Bedeckungsskalierer **1240**. Zum Schluss addiert der Farbakkumulator **1244** die skalierten Farben zu den akkumulierten Farben, um die aktualisierten akkumulierten Farbwerte zu berechnen.

[0618] Wenn das aktuelle Fragment die aktuelle Pixelregion überlappt, aktualisiert die Bedeckungsfeldschleifenkontrolle **1234** die In-Gebrauch-Maske **1236**, so dass sie einen Eintrag umfasst, der der neuen Pixelregion entspricht. Damit kann nur das existierende In-Gebrauch-Element überschrieben oder ein neues erzeugt werden. Die Bedeckungsfeldschleifenkontrolle weist auch die neue Bedeckungskontrolle **1238** an, die Bedeckungsfeldmaske **1230** auf die Bedeckung der neuen Pixelregion zu aktualisieren und das akkumulierte Alpha für diese neue Pixelregion einzustellen. Die neue Bedeckungskontrolle **1238** stellt einen neuen Alpha-Bedeckungsfeldeintrag, der der neuen Pixelregion entspricht, auf Anext ein.

[0619] Wenn das aktuelle Fragment nur einen Abschnitt einer Pixelregion bedeckt (statt sie ganz zu überlappen), dann erzeugt die neue Bedeckungskontrolle **1238** zwei neue Pixelregionen: 1) einen Abschnitt der Pixelregion, die das aktuelle Fragment überlappt, und 2) einen Abschnitt der Pixelregion, die vom aktuellen Fragment nicht bedeckt wird. In diesem Fall berechnet das Teilsystem die Bedeckung für den nicht bedeckten Abschnitt und stellt das Alpha dafür ein, welches dasselbe wie die ursprüngliche Pixelregion bleibt. Um dies zu erreichen, aktualisiert die Bedeckungsfeldschleifenkontrolle **1234** die In-Gebrauch-Maske **1236** und weist die neue Bedeckungskontrolle **1238** an, die Bedeckungsfeldmaske **1230** zu aktualisieren. Der Bedeckungsfeld-Alphaeintrag, der dieser zweiten Pixelregion entspricht, bleibt derselbe wie bei der aktuellen Pixelregion (Aold), weil sie durch das aktuelle Fragment nicht verändert wird.

[0620] Wenn der Ansatz, der oben beschrieben wird, wiederholt wird, durchschleift das Teilsystem jeden In-Gebrauch-Eintrag für das aktuelle Fragment und berechnet den Effekt, falls vorhanden, des aktuellen Fragments auf jede Pixelregion. Es wiederholt dann den Prozess für nachfolgende Fragmente in der Liste, bis die Liste leer ist.

[0621] Der Klemm- und Einstellblock **1248** führt das Festhalten der akkumulierten Farbe auf dem richtigen Bereich (dies wird auf Grund des Rundens, das zu Farben oder Alphawerten führen kann, die den 8-Bitbereich verlassen, im Bedeckungsskalierblock benötigt) und eine Anpassung für Fehler aus, die durch das Skalieren eines Wertes durch eine 8-Bit-Binärzahl, die 1 repräsentiert, eingeführt werden. Eine Anpassung für diese Art von Fehler kann unter einigen Umständen notwendig sein, weil ein Wert von 1 tatsächlich durch den Hexwert „FF" repräsentiert wird. Mit anderen Worten, ein Alphabereich von 0 bis 1 wird durch einen Bereich von 8 Bitzahlen von 00 bis FF repräsentiert. Wenn man eine Zahl x mit FF multipliziert, muss das Ergebnis also x sein. Die Anpassung stellt sicher, dass das Ergebnis der Multiplikation mit FF ordentlich auf x gerundet wird.

[0622] Der Rückkopplungsweg **1250** zu den Pixelpuffern besteht, um einen Modus zu unterstützen, bei dem aufgelöste Pixelwerte in die Pixelpuffer zurück gespeichert werden. Das ermöglicht das Mehrfachrendern an aufgelösten Pixeldaten, ohne einen Chunk von aufgelösten Daten zum gemeinsamen Speicher außerhalb des Tilers zu übertragen.

[0623] Wenn das Fragmentauflösungsteilsystem nicht im Rückkopplungsmodus ist, dann überträgt der Klemm- und Einstellblock **1248** die aufgelösten Pixeldaten in die Blockeinspeicherungspuffer über den Datenweg **1252**, der in **Fig. 36** gezeigt wird. Diese Blockeinspeicherungspuffer werden dazu verwendet, aufgelöste Pixeldaten zu puffern, bevor sie in 8×8-Pixelblöcke komprimiert werden.

Texturabbildung

[0624] Das Bildverarbeitungssystem umfasst eine Reihe von fortgeschrittenen Texturabbildungsmerkmalen. Seine Unterstützung für die Texturabbildung umfasst die anisotrope Filterung von Texturdaten. Das System kann die anisotrope Filterung von Texturdaten in Echtzeit ausführen.

[0625] Wir beginnen mit der Beschreibung einiger Begriffe, die die Grundlage für unseren Ansatz zur anisotropen Filterung bilden, und beschreiben dann eine Implementierung mit mehr Details.

[0626] Texturabbildung bezeichnet die Abbildung eines Bildes auf eine Fläche. Schwierige Details auf der Oberfläche eines Objektes sind unter Verwendung von Polygonen oder anderen geometrischen Grundelementen sehr schwierig zu modellieren, und dies vergrößert sehr den Rechenaufwand für das Objekt. Texturabbildung ermöglicht einem Bildverarbeitungssystem, feine Details effizient auf der Oberfläche eines Objektes darzustellen. Eine Texturabbildung ist eine digitale Abbildung, die wir auch als das „Quellbild" bezeichnen. Die Textabbildung ist normalerweise rechteckig geformt und hat ihren eigenen (u, v)-Koordinatenraum. Einzelne Elemente der Texturabbildung werden als „Texel" bezeichnet. Bei der Texturabbildung wird eine Textur oder ein "Quellbild" auf ein Zielbild abgebildet.

[0627] Bei digitalen Bildern werden die Quell- und Zielbilder an diskreten Punkten abgetastet, normalerweise auf einem Punktgitter mit ganzzahligen Koordinaten. Im Quellbild befinden sich die Texel an ganzzahligen Koordinaten im (u, v)-Koordinatensystem. Im Zielbild befinden sich die Pixel in ähnlicher Weise an ganzen Koordinaten im (x, y)-Koordinatensystem.

[0628] Eine geometrische Transformation beschreibt, wie ein Punkt aus dem Quellbild auf das Zielbild abgebildet wird. Die Inverse dieser Transformation beschreibt, wie ein Punkt im Zielbild zurück auf das Quellbild abgebildet wird. Der Bildprozessor kann diese inverse Transformation verwenden, um festzustellen, wo im

Quellbildfeld der Texel eine Pixelintensität herkommen sollte. Die Intensität an diesem Punkt im Quellbild kann dann auf der Basis der benachbarten Texeldaten bestimmt werden. Ein Punkt im Ziel, der zurück in das Quellbild abgebildet wird, fällt nicht notwendigerweise exakt auf ganzzahlige Koordinaten eines Texels. Um die Intensität an diesem Punkt festzustellen, werden die Bilddaten aus benachbarten Texeln berechnet.

[0629] Da die Quellbildintensitäten nur an diskreten Werten bekannt sind, werden Werte aus benachbarten Texeln interpoliert und die resultierenden Daten werden dann durch ein Tiefpassfilter geschickt. Im allgemeinen ist die Vorgehensweise folgendermaßen. Zuerst wird ein Punkt aus dem Zielbild in das Quellbild abgebildet. Dann werden Texeldaten interpoliert, um die Intensität an dem Punkt, der in das Quellbild abgebildet wird, zu rekonstruieren. Zum Schluss wird ein Tiefpassfilter angewendet, um Raumfrequenzen im Quellbild zu entfernen, die in einen zu hohen Bereich transformiert werden, um ordentlich im diskreten Zielbild neu abgetastet zu werden. Dieses Tiefpassfilter wird manchmal als Anti-Aliasing-Filter bezeichnet, weil es hohe Frequenzen entfernt, die sich wegen der erneuten Abtastung als Wellen von niedrigerer Frequenz maskieren oder ausgeben. Dieses Konzept wird detaillierter unten beschrieben.

[0630] Fig. 37 ist ein Beispiel, das illustriert, wie ein Pixel **1300** auf der Fläche **1302** des Zielbildes auf die Fläche der Texturabbildung **1304** abgebildet wird. In diesem Beispiel wird das Pixel aus dem Zielbild als Quadrat **1306** dargestellt. Die Rückwärtsabbildung dieses Pixels **1300** auf die Texturabbildung **1304** ist ein Viereck **1308**, das die komplexere Form annähert, in die das Pixel sich auf Grund der Krümmung der Zielfläche **1302** abbildet. Nach dem Abbilden des Pixels **1300** auf die Textur wird ein Intensitätswert aus Texelsamples innerhalb des Vierecks berechnet. In einer Ausführungsform zum Beispiel wird der Intensitätswert eines Pixels durch Bilden einer gewichteten Summe von Texeln im Viereck berechnet.

[0631] Sowohl die Interpolation als auch die Tiefpassfilterfunktionen können zu einem einzigen Filter kombiniert werden, das durch Bilden eines gewichteten Durchschnitts der Punkte implementiert wird, die jeden inversen Transformationspunkt in der Quelle umgeben, der auf einen diskreten Punkt im Ziel abgebildet wird. Wir bezeichnen die Region von Punkten, die zu diesem gewichteten Mittel beitragen, als Fußspur des Filters. Die Fußspur hat im allgemeinen eine andere Form in der Quelle für jeden Zielpunkt. Da die Fußspur für jeden einzelnen Punkt variieren kann, ist es schwierig, die richtige Form der Fußspur und die Gewichtungsfaktoren zu finden, um sie auf die Punkte im Innern der Fußspur anzuwenden. Einige konventionelle Systeme nehmen die Näherung der Verwendung derselben Form für das Filter in jedem Punkt vor, obwohl die Größe des Filters variiert werden darf. Diese Vorgehensweise kann jedoch zu Verzerrungen im Endbild führen.

[0632] Wir bezeichnen Filter, die entweder quadratische oder kreisförmige Fußspuren von variabler Größe erzeugen, als isotrope Filter. Ein Kreis ist wirklich isotrop, da er in allen Richtungen dieselbe Länge hat. Wir betrachten auch das Quadrat als im wesentlichen isotrop, da es horizontal und vertikal dieselben Abmessungen hat.

[0633] Die isotrope Filterung kann Verzerrungen bewirken, weil sie ziemlich grobe Näherungen verwendet. In Bereichen des Quellbildes, wo die tatsächliche Fußspur sehr länglich ist, ist eine im wesentlichen isotrope Form, wie zum Beispiel ein Quadrat oder ein Kreis, ein armseliger Ersatz für die Fußspur, selbst wenn die Größe einstellbar ist. Da ein isotropes Filter nur eine Form hat, kann es Texel in einer länglichen Fußspur nicht genau erfassen. Ein quadratisches Filter zum Beispiel kann Texelwerte aus einer vierseitigen Fußspur, die in einer Richtung gestreckt ist, nicht genau abtasten. Das Abtasten von Texeln außerhalb der tatsächlichen Fußspur kann zu Unschärfe führen. Werden Texel in der Fußspur andererseits nicht abgetastet, so kann dies bewirken, dass das Endbild auf Grund des Aliasing funkelt.

[0634] In einem Ansatz, der MIP-(multum in parvo – vieles an einer kleinen Stelle) Abbildung genannt wird, wird eine Reihe von Texturabbildungen bei verschiedenen Auflösungen gespeichert. Wenn zum Beispiel die eine Textur bei 512 × 512 Texeln liegt, kann das System auch Texturen bei 256 × 256, 128 × 128, 64 × 64 usw. speichern. Ein Bildverarbeitungssystem kann diese Texturabbildungen bei variierender Auflösung verwenden, um die beste Anpassung für ein isotropes Filter an die Fußspur des Pixels zu finden, das in die Textur abgebildet wird. Der Bildprozessor findet zuerst die zwei Texturen, bei denen die Fußspur in der Größe der Größe des Filters am nächsten kommt. Er führt dann eine Interpolation für die zwei Texturen aus, die am besten zur Fußspur passen, um zwei Zwischenwerte zu berechnen. Zum Schluss interpoliert er zwischen den zwei Zwischenwerten, um einen Wert für das Pixel zu finden.

[0635] Während die MIP-Abbildung einige verbesserte Ergebnisse für isotrope Filter liefern kann, verursacht sie immer noch Verzerrungen, besonders wo die Fußspur in einer Richtung gestreckt ist. Ein genaueres Filter für die tatsächliche Fußspur in jedem einzelnen Punkt kann durch die Kaskade eines im wesentlichen isotropen

Rekonstruktionsfilters erzeugt werden, das mit einem im wesentlichen isotropen Resamplingfilter gefaltet ist, dessen Form durch die Inverse der geometrischen Transformation verzerrt wurde. Diese Verzerrung kann einen hohen Grad von Anisotropie eines Filters erzeugen. Wenn die Transformation das Bild in einer Richtung viel stärker als in einer anderen Richtung kontrahiert, dann dehnt oder streckt die inverse Transformation die Fußspur in der Quelle in Richtung der maximalen Kontraktion im Ziel. Dies kann auftreten, wenn man eine ebene Fläche aus einer Perspektive in der Nähe des Randes betrachtet. Beim isotropen Filtern würde das Endbild in diesem Beispiel verzerrt erscheinen, weil das Filter die Texelwerte in der gestreckten Fußspur nicht richtig abtasten kann.

[0636] Eine Ausführungsform unseres anisotropen Filterverfahrens umfasst die folgenden zwei Schritte: 1) Finden einer ungefähren Richtung der maximalen Ausdehnung der Filterfußspur, und 2) Anwenden eines Resamplingfilters in dieser Richtung auf die Ausgabe eines Rekonstruktionsfilters, um ein zusammengesetztes Filter zu erzeugen, das der tatsächlichen Fußspur stärker angepaßt ist.

[0637] Die Richtung der maximalen Ausdehnung kann aus der Rückwärtsabbildung eines Filters vom Zielbild auf die Texturabbildung abgeleitet werden. In der perspektivischen Abbildung zum Beispiel (wo ein Objekt in Richtung des Fluchtpunktes abklingt), ist die Abbildung einer n × n-Pixel-Fußspur vom Zielbild auf die Textur ein Viereck. Die Linie der Anisotropie wird als Linie definiert, die die Richtung der maximalen Ausdehnung hat und durch einen Punkt vom Ziel geht, der in das Quellbild zurück abgebildet wird.

[0638] In dieser Ausführungsform bildet der Bildprozessor die Filterfußspur zurück auf die Textur ab, um die Richtung der maximalen Ausdehnung zu finden. Er lässt dann ein interpolierendes Filter (das Rekonstruktionsfilter, das oben angeführt wird) entlang der Richtung der maximalen Ausdehnung durchlaufen. Um einen Pixelwert für das Zielbild zu berechnen, wendet er ein Resamplingfilter auf die Ausgabe des interpolierenden Filters an.

[0639] In einer Implementierung ist das Resamplingfilter ein eindimensionales digitales Filter, das entlang der Linie der Anisotropie angewendet wird. Für dieses Filter kann eine Reihe von eindimensionalen Filtern verwandt werden. Daher beabsichtigen wir nicht, den Geltungsbereich unserer Erfindung auf ein spezielles eindimensionales Filter zu beschränken.

[0640] In dieser Implementierung ist das interpolierende Filter ein zweidimensionales isotropes Filter. Wie beim Resamplingfilter beabsichtigen wir nicht, den Geltungsbereich unserer Erfindung auf eine spezielle Art von interpolierendem Filter zu beschränken. Das zweidimensionale isotrope Filter ist nur eine mögliche Implementierung. Das interpolierende Filter liefert Werte an Positionen entlang der Linie der Anisotropie durch Interpolieren dieser Werte aus benachbarten Texeldaten. Die einzelnen Positionen, an denen das interpolierende Filter auf das Quellbild angewendet wird, können durch vertikales oder horizontales schrittweises Fortschreiten in Inkrementen und Interpolieren eines Wertes auf der Linie der Anisotropie an jeder Position bestimmt werden. Wenn zum Beispiel die Linie der Anisotropie mehr vertikal als horizontal ist, wäre eine Vorgehensweise, schrittweise in vertikaler oder V-Richtung im (u, v)-Koordinatensystem der Textur fortzuschreiten. Wenn analog zum Beispiel die Linie der Anisotropie mehr horizontal als vertikal ist, wäre eine andere Vorgehensweise, schrittweise in horizontaler oder U-Richtung im (u, v)-Koordinatensystem der Textur fortzuschreiten.

[0641] Ein mögliches Verfahren zum Fortschreiten entlang der Linie der Anisotropie ist, das interpolierende Filter an diskreten Orten entlang dieser Linie, mit konstanten, ungefähr der Länge der minimalen Ausdehnung entsprechenden Abständen, anzuwenden. Speziell können die Abtastwerte entlang der Linie der Anisotropie mit konstanten Intervallen bei einem Abstand angeordnet sein, der etwa gleich der Länge der minimalen Ausdehnung ist, wobei die Mittelabtastung sich an dem Punkt befindet, wo die Pixelmitte in die Texturabbildung abbildet wird. Sobald diese Abtastorte berechnet sind, kann ein isotropes Filter wiederholt an jeder Stelle angewendet werden. Ein isotropes Filter kann zum Beispiel an den Abtastorten angewendet werden, um die Interpolation mit benachbarten Textursamples für jedes Sample auszuführen, wobei die Größe des Filters von der Länge der minimalen Ausdehnung abhängt. Eine spezielle Möglichkeit, dieses Verfahren zu implementieren, besteht darin, eine dreilinige Interpolation an jedem diskreten Ort entlang der Linie der Anisotropie vorzunehmen.

[0642] Nach der Anwendung der Ausgabewerte des interpolierenden Filters auf das digitale Filter ist der resultierende Pixelwert ein gewichtetes Mittel der Ausgabewerte des interpolierenden Filters entlang der Linie der Anisotropie. Obwohl wir spezielle Arten hier beschreiben, können die Arten von Filtern, die zur Annäherung der Rekonstruktions- und Resamplingfunktionen verwendet werden, variieren.

[0643] Die **Fig.** 38A-D illustrieren ein Beispiel für den Prozess der anisotropen Filterung. Die **Fig.** 38A-D illustrieren die Texel in einer Texturabbildung (**1400A**-D) und zeigen, wie ein anisotropes Filter erzeugt werden kann. Der erste Schritt besteht darin, eine Filterfußspur im Quellbild zu berechnen, indem eine inverse Transformation an der Filterfußspur für einen Pixelort im Zielbild ausgeführt wird. In diesem Beispiel wird die Filterfußspur in der Textur **1400A** als Viereck **1402** illustriert.

[0644] Im nächsten Schritt wird die inverse Transformationsmatrix an einem Punkt angenähert, der in das Quellbild abgebildet wird. In diesem Beispiel, das in **Fig.** 38B gezeigt wird, wird diese Annäherung durch ein Parallelogramm **1404** dargestellt. Dieses Parallelogramm nähert die viereckige Fußspur an. Diese Annäherung kann im allgemeinen durch Berechnen der Jacobi-Matrix für die inverse geometriche Transformation gefunden werden. Obwohl wir das Beispiel für Zwecke der Erläuterung vereinfacht haben, kann dasselbe Konzept auf Fälle erweitert werden, wo die inverse Transformation komplexer ist. Dies wird aus zusätzlichen Details ersichtlich, die unten bereitgestellt werden.

[0645] Wieder mit Bezug auf das Beispiel in Fig. 38, kann die Größe der Rekonstruktions- und Resamplingfilter aus der Jacobi-Matrix abgeleitet werden. In **Fig.** 38B repräsentieren wir die Jacobi-Matrix als Parallelogramm **1404**.

[0646] Die Länge des Parallelogramms kann dazu verwendet werden, die Größe des Resamplingfilters zu bestimmen. In diesem Beispiel wird die Länge entlang der Richtung der maximalen Ausdehnung **1406** gemessen, die wir als Richtung der Anisotropie bezeichnen. Die Höhe des Parallelogramms kann analog dazu verwendet werden, um die Größe des Rekonstruktionsfilters zu bestimmen. Die Höhe ist die Richtung der minimalen Ausdehnung **1408**.

[0647] **Fig.** 38C zeigt ein Rechteck **1406**, das das Parallelogramm annähert. Die Abmessungen dieses Rechtecks entsprechen der Höhe und Länge des Parallelogramms. Das Rechteck repräsentiert das „balkenförmige" Filter, das zum Annähern der Anisotropie der tatsächlichen Filterfußspur verwendet wird.

[0648] **Fig.** 38D illustriert, wie dieses „balkenförmige" Filter berechnet werden kann. Die Fußspur des Rekonstruktionsfilters wird durch das Quadrat **1408** repräsentiert. In diesem Beispiel hat das Rekonstruktionsfilter eine quadratische Fußspur und ist daher ein im wesentlichen isotropes Filter. Um Werte entlang der Linie der Anisotropie zu berechnen, die durch die Linie **1410** in **Fig.** 38D repräsentiert wird, werden Werte aus Texeln (**1400D**) interpoliert, die die Linie der Anisotropie **1410** umgeben. Das Rekonstruktionsfilter ist daher ein interpolierendes Filter, wie oben bemerkt. Die Ausgabe dieses Filters wird dann auf ein eindimensionales Filter angewendet, das das Resamplingfilter darstellt. Die Linie der Anisotropie **1410** repräsentiert die Orientierung des Resamplingsfilters. Die Werte, die berechnet werden, während das Rekonstruktionsfilter entlang der Linie der Anisotropie bewegt wird, werden summiert, um den Pixelwert für das Zielbild zu berechnen.

[0649] Die oben beschriebene Vorgehensweise kann in einer Reihe von Weisen implementiert werden. Sie kann in Hardware und Software implementiert werden. Um die anisotrope Echtzeitfilterung zu unterstützen, wird das Verfahren vorzugsweise in Hardware implementiert. Eine Ausführungsform dieses Ansatzes wird auf dem Tiler-Chip implementiert.

[0650] Im Tiler, der in den Fig. 9A-C illustriert wird, wird die anisotrope Filterung im Scanumwandlungsblock und in der Texturfilterengine unterstützt. Der Scanumwandlungsblock berechnet Kontrollparameter für die Resampling- und Rekonstruktionsfilter, indem die Jacobi-Matrix der partiellen Ableitungen der inversen geometrischen Transformation an einem Punkt im Quellbild gebildet wird. Die Jacobi-Matrix repräsentiert den linearen Teil der besten, lokal affinen Approximation an die inverse Transformation. Spezieller gesagt, ist sie der Abschnitt erster Ordnung der Taylor-Reihe in zwei Dimensionen der inversen Transformation, die um den gewünschten Quellpunkt zentriert ist.

[0651] Der lineare Teil der affinen Transformation von Texturkoordinaten auf Bildschirmkoordinaten hat eine 2×2-Jacobi-Matrix J; die inverse Transformation von Bildschirmkoordinaten auf Texturkoordinaten hat eine Jacobi-Matrix J^{-1}. Die Längen der zwei Spaltenvektoren der Matrix J^{-1} sind die Längen der zwei Seiten des Parallelogramms für ein Pixel der Einheitsgröße. Die Komponenten der zwei Spaltenvektoren in der inversen Jacobi-Matrix bestimmen die Längen der zwei Seiten des Parallelogramms.

[0652] Die Transformationen nehmen die Form von Attributkantengleichungen an, die der Scanumwandlungsblock bewertet, während er jedes Grundelement scannt. Die folgende Gleichung ist typisch:

$F_{s/w} = A_{s/w}x + B_{s/w}y$

wobei am Pixelort (x, y) gilt:

1) $F_{s/w}$ ist der Wert der Texturkoordinate (s), dividiert durch die homogene Koordinate (w).
2) $A_{s/w}$ ist der Wert des Gradienten der Texturkoordinate (s), dividiert durch die homogene Koordinate (w) in Bezug auf die x-Koordinate.
3) $B_{s/w}$ ist der Wert des Gradienten der Texturkoordinate (s), dividiert durch die homogene Koordinate (w) in Bezug auf die y-Koordinate. F, A und B sind alle relativ zum Scanstartpunkt des Grundelementes normalisiert. Der Scanumwandlungsblock bewertet die Kantengleichungen für 1/w, s/w und t/w.

[0653] Die Elemente der inversen Jacobi-Matrix ergeben die Längen der Seite und die Fläche des Parallelogramms. Die Fläche des annähernden Rechtecks und die lange Seite des Rechtecks sind dasselbe: die kurze Seite des Rechtecks ist die kurze Seite des Parallelogramms, multipliziert mit dem Sinus des Winkels zwischen der x- und der y-Achse im (s, t)-Koordinatensystem.

[0654] Die Ableitungen für die inverse Jacobi-Matrix leiten sich direkt aus dem Fs, As und Bs der Kantengleichungen an jeder Texturkoordinate (s, t) ab.

$$w = \frac{1}{1/w} = \frac{1}{F_{1/w}}$$

$$s = w(s/w) = wF_{s/w} \qquad t = w(t/w) = wF_{t/w}$$

$$\frac{\partial s}{\partial x} = w\left(\frac{\partial(s/w)}{\partial x} - s\left(\frac{\partial(1/w)}{\partial x}\right)\right) \qquad \frac{\partial s}{\partial y} = w\left(\frac{\partial(s/w)}{\partial y} - s\left(\frac{\partial(1/w)}{\partial y}\right)\right)$$

$$= w\left(A_{t/w} - tA_{1/w}\right) \qquad = w\left(B_{t/w} - tB_{1/w}\right)$$

$$LängeXQuadriert = \left(\frac{\partial s}{\partial x}\right)^2 + \left(\frac{\partial t}{\partial x}\right)^2$$

$$LängeYQuadriert = \left(\frac{\partial s}{\partial y}\right)^2 + \left(\frac{\partial t}{\partial y}\right)^2$$

$$Fläche = Abs\left(\left(\frac{\partial s}{\partial x}\right)\left(\frac{\partial t}{\partial y}\right) - \left(\frac{\partial s}{\partial y}\right)\left(\frac{\partial t}{\partial x}\right)\right) = L \times S \sin\theta$$

[0655] Nachdem die Jacobi-Matrix gefunden ist, sucht der Scanumwandlungsblock den längeren der zwei Spaltenvektoren. Die Richtung dieses Vektors stellt die Richtung der Linie der maximalen Ausdehnung oder Linie der Anisotropie dar. Das Verhältnis der Länge dieses Spaltenvektors zur Länge des anderen wird als Anisotropieverhältnis bezeichnet. Die Länge des eindimensionalen anisotropen Filters wird aus diesem Verhältnis bestimmt. Die Länge des längeren Vektors dividiert durch das Anisotropieverhältnis steuert die Breite des Rekonstruktionsfilters.

[0656] Die längere Seite wird zur Hauptachse und kann dazu verwendet werden, die zu inkrementierende Bildschirmkoordinate zu bestimmen, wenn man schrittweise (Takt) in Texturkoordinaten fortschreitet. Sie kann auch dazu verwendet werden, die Größe der Inkremente zu bestimmen.

```
// DsDx ist die partielle Ableitung von s nach x usw.
// (DsDc, DtDc) sind Schritte in (s, t) entlang der Anisotropieachse.

if (LengthXSquared >= LengthYSquared) {
MajorSquared = LengthXSquared
InverseMajor = 1./sgrt(MajorSquared)

DsDc = DsDx * InverseMajor
DtDc = DtDx * InverseMajor I

else {

MajorSquared = LengthYSquared
InverseMajor = 1./sqrt(Majorsquared)
DsDc = DsDy * InverseMajor
DtDc = DtDy * InverseMajor
}
```

[0657] Die Schrittgößen DsDc und DtDc sind Grundeingaben in die Texturfilterengine, die das Abtasten und Filtern ausführt. Diese Schritte erzeugen eine Orientierung, die um (höchstens) sieben Grad ungenau ist, was im Fall eines gleichseitigen Parallelogramms auftritt.

[0658] In dieser Implementierung nähert die Länge des kürzeren Vektors normalerweise die Breite des Rekonstruktionsfilters an, wenn nicht das Anisotropieverhältnis eine vorgegebene Schranke übersteigt. Wenn die Schranke überstiegen wird, dann wird das Anisotropieverhältnis in der Rechnung durch diese vorgegebene Schranke ersetzt. Durch das Begenzen des Verhältnisses in dieser Weise wird verhindert, dass das Filter mehr als eine vorgegebene Zahl von Texelpunkten zur Berechnung eines Wertes verwendet. Damit setzt die Schranke für das Verhältnis eine Grenze dafür, wie lange das Rekonstruktionsfilter benötigt, um einen Ausgabewert zu berechnen.

[0659] Ein weiterer einschränkender Fall tritt auf, wenn die Länge eines der Vektoren kleiner als eins ist. In diesem Fall wird die tatsächliche Länge des Vektors durch die Länge von eins ersetzt. Dies stellt sicher, dass die Filterlängen niemals für die Ausführung der Interpolation zu kurz sind.

[0660] Nachdem der Scanumwandlungsblock die Kontrollparameter für die Filter berechnet hat, berechnet er dann einen Pixelwert. Das eindimensionale digitale Filter berechnet ein gewichtetes Mittel der Ausgabe aus dem interpolierenden Filter. Das interpolierende Filter berechnet diese Ausgabe durch Interpolieren der Texeldaten aus dem Quellbild, das neben der Linie der Anisotropie liegt.

[0661] Die Größe des interpolierenden Filters kann so eingestellt werden, dass sie die wahre Fußspurbreite annähert, die in einer Richtung senkrecht zu der der maximalen Ausdehnung gemessen wird. Wenn die Fußspur groß ist, was in Bereichen des Bildes auftritt, die die Transformation schrumpft, müssen viele Punkte des Quellbildes mit den Gewichtungskoeffizienten des Filters multipliziert werden, um einen einzelnen Ausgabepunkt zu erzeugen, was zu einer sehr langsamen oder kostspieligen Implementierung führt.

[0662] Wie oben vorgestellt, reduzieren existierende isotrope Filtersysteme die Berechnungszeit unter Verwendung der MIP-Abbildung. MIP-Abbildung bezeichnet das Bilden einer Bildpyramide, die auf dem Quellbild beruht, und die anschließende Verwendung der Bilder in dieser Pyramide, um die beste Anpassung für ein isotropes Filter in einem Quellbild zu finden. Jede Ebene der Pyramide ist in der Abtastdichte im Vergleich zu der darunterliegenden um einen Faktor von zwei in jeder Dimension reduziert. Der Boden der Pyramide ist das originale Quellbild. Durch das Interpolieren eines Bildes mit reduzierter Abtastdichte wird ein ähnlicher Effekt wie das Filtern des Originalbildes mit einem isotropen Filter erzeugt, dessen Fußspur relativ zu der des Interpolators um das Verhältnis der ursprünglichen Abtastdichte zur reduzierten Dichte vergrößert ist. Daher kann die Potenz von zwei Vergrößerungen der Fußspur durch Auswählen der richtigen Ebene der Pyramide zum Interpolieren erreicht werden. Jedes Vergrößerungsverhältnis kann durch Mischen der Ergebnisse von Interpolationen der zwei Pyramidenebenen erhalten werden, die das gewünschte Verhältnis einschließen.

[0663] In einer Ausführungsform kann die Größe des isotropen Filters durch Verwendung eines MIP-Abbildungsansatzes modifiziert werden, um sie besser an die Länge der minimalen Ausdehnung anzupassen. Die Größe des isotropen Filters, die aus der Analyse der Jacobi-Matrix bestimmt wird, kann zum Auswählen der begrenzenden Pyramidenebenen und des Mischungsfaktors verwendet werden. In einer Implementierung ist die Basispyramidenebene der ganzzahlige Teil von log Basis 2 der Filtergröße, und der Mischungsfaktor ist der Bruchteil.

[0664] Ein spezifisches Beispiel hilft bei der Erläuterung der Operation der speziellen Implementierung, die oben beschrieben wird. Wenn die gewünschte isotrope Größe 3 ist, dann ist $\log_2 3$ gleich 1,585. Der ganzzahlige Teil des Ergebnisses ist 1, was die Ebene 1 und 2 mit Dichte Reduzierungen von 2 bzw. 4 auswählt. Ebene 0 ist das originale Quellbild ohne Reduzierung. Der Mischungsfaktor ist 0,585.

[0665] In einer Implementierung verzögert die Texturfilterengine das Mischen. Zuerst wendet die Texturfilterengine 1D-Filter mit einer Länge an, die proportional zum Anisotropieverhältnis, auf den gewünschten Punkt in jeder Ebene zentriert, ist. Sie mischt dann die Ausgabe von jeder Ebene.

[0666] In einer alternativen Implementierung schreitet die Texturfilterengine entlang der Linie der Anisotropie fort und führt eine dreilineare Interpolation an diskreten Samples entlang dieser Linie aus. Die Texturfilterengine wendet dann das eindimensionale Filter auf das Ergebnis der dreilinearen Interpolation bei jedem Sample an.

[0667] Zusätzlich zur Steuerung der Größe des interpolierenden Filters kann auch die Größe des Resamplingfilters gesteuert werden. In einer Implementierung verwendet die Texturfilterengine Tabellen von Koeffizienten für 1D-Resamplingfilter verschiedener Größen und Mischungen zwischen ihnen, um ein Filter einer Größe zwischen den in der Tabelle angegebenen zu erzeugen. Eine besonders nützliche Implementierung für Hochgeschwindigkeits-Hardware besteht darin, die Filterlängen als Potenzen von zwei zu wählen und den Filterimpulsprofilen eine dreieckige oder trapezförmige Form zu geben. Die individuellen Filter haben dann sehr einfache Koeffizienten und der Aufwand für die Multiplikation wird auf ein paar Additionen und Verschiebungen in der Hardware reduziert.

[0668] Das Folgende ist eine Tabelle von Koeffizienten für diese Filter für die ersten vier Potenzen von 2:

0								1								/1
1							1	2	1							/4
2					1	2	3	4	3	2	1					/16
3	1	2	3	4	5	6	7	8	7	6	5	4	3	2	1	/64
4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	/15

[0669] In diesem Beispiel wird der log Basis 2 des Anisotropieverhältnisses verwendet, um eine Ebene und einen Mischungsfaktor auszuwählen. Wenn die Ebene über 4 hinausgeht, dann verwendet die Texturfilterengine das letzte Filter und nimmt keine Mischung vor. In diesem Beispiel haben alle Filter den Einheitsgain, was bedeutet, dass sich alle ihre Koeffizienten zu eins addieren. Multiplikationen mit 1, 2, 4 und 8 können durch Verschiebungsoperationen ausgeführt werden.

[0670] Multiplikationen mit 3, 5 und 6 können durch eine einfache Addition plus einer Verschiebungsoperation

ausgeführt werden. Und schließlich können Multiplikationen mit 7 durch eine einzelne Subtraktion und Verschiebungsoperationen ausgeführt werden. Die Divisionen durch die Potenzen von zwei sind nur Verschiebungen. Die Division durch 15 kann sehr gut durch die Multiplikation mit 1,00010001 Basis 2, gefolgt von einer Verschiebung um 4 Stellen (Division durch 16) angenähert werden. Die Multiplikation sind nur zwei Additionen.

[0671] Der Ansatz, der oben beschrieben wird, ermöglich zwei Freiheitsgrade in der Steuerung des zusammengesetzten Filters. In der Ausführungsform, die oben beschrieben wird, sind die Freiheitsgrade die Größen der Filter in der Richtung minimaler und maximaler Ausdehnung. Dieser Ansatz erzeugt ein Bild mit viel weniger Aliasing und Unschärfe ohne den Aufwand der Berechnung der wahren Fußspur an jedem Punkt, die das Ergebnis einer stark nichtlinearen Abbildung sein kann. Dieser Ansatz nähert ein kontinuierliches Filter an, das das tatsächliche Fußspurfilter entlang der Linie der Anisotropie abtastet. Es erreicht eine viel bessere Anpassung an die tatsächliche Fußspur als ein Kreis oder Quadrat, weil es einen „balkenförmigen" Filter entlang der Linie der Anisotropie ergibt. Wir haben dieses Verfahren in einem Echtzeit-Grafikrenderingsystem implementiert. Dieses Verfahren unterstützt also die qualitativ hochwertige Texturabbildung mit anisotroper Filterung, während trotzdem gleichzeitig Echtzeitraten erreicht werden, d.h. Berechnung eines neuen Teilbildes von Bilddaten bei einer Rate größer als 10 Hz und speziell bei Auffrischungsraten des Anzeigegerätes (z.B. 75 Hz).

[0672] Fig. 39 ist ein Schema, das eine Implementierung der Texturfilterengine (**401**, Fig. 9B) illustriert. Die Texturfilterengine liest Gruppen von Texturreferenzdaten aus einer Pixelschlange (Texturreferenzdatenschlange **399** in Fig. 9B) und berechnet Alpha- und Farbwerte (Alpha und RGB-Farbfaktoren) oder Schattenkoeffizienten für diese Gruppen. Diese Implementierung unterstützt sowohl Textur- als auch Schattenfilterung. Für Texturabbildungsoperationen berechnet die Texturfilterengine Texturfarben und Alpha und filtert die Texturfarben, um Alpha- und Farbfaktoren zu berechnen. Für Schattierungsoperationen führt die Texturfilterengine Tiefenvergleiche aus und filtert die resultierenden Werte, um Schattendämpfungskoeffizienten (s) zu berechnen.

[0673] Die Pixelschlange empfängt Texturreferenzdaten von einem Rastergenerator (wie zum Beispiel dem Scanumwandlungsblock **395** in Fig. 9B) und fungiert als FIFO-Puffer für die Texturfilterengine **401**. Die „Abtastung gültig" Daten geben an, welche Samples oder Abtastungen in einem Satz von Textur- oder Schattenabbildungselementen, die aus dem Texturcache abgerufen wurden, für die aktuelle Filteroperation gültig sind.

[0674] Für eine Texturabbildungsoperation umfassen die Texturreferenzdaten die Koordinaten eines Pixelortes, der in die Textur (s, t) abgebildet wird. Zur Unterstützung der dreilinearen MIP-Abbildung umfassen die Eingaben die (s, t)-Koordinaten für die nächstliegenden MIP-Abbildungsebenen (hi, lo) und die Detailebene (LOD). Die „Akkumulationskala"-Daten werden verwendet, um die Gewichtungsfaktoren zu steuern, die auf die Ausgabe der Farbkomponenteninterpolatoren angewendet werden. Die „Erweiterungskontroll"-Daten sind Datenbits, die die Texturerweiterungsmodi steuern. Die Texturerweiterungsmodi weisen die Texturfilterengine an, entweder eine Klemm-, Wickel- oder Reflektionsoperation auszuführen, wenn eine Texturanforderung sich außerhalb des Texturabbildungsbereichs befindet.

[0675] Für Schattierungsoperationen umfassen die Eingaben einen Abtastindex, (s, t)-Koordinaten eines Pixelortes, der in die Schattenabbildung abgebildet wird, und ein Beta, das die Tiefe des geometrischen Grundelementes von der Lichtquelle für einen gegebenen Pixelort repräsentiert. Der Abtastindex betrifft die spezielle Weise, in der das Schattenfilter auf Schattenabbildungselemente oder „Abtastungen" einwirkt. In dieser speziellen Implementierung arbeitet die Texturfilterengine mit 8 Abtastungen pro Taktzyklus. Im Fall der Schattenfilterung entsprechen diese Abtastungen einem 4×2-Gitter. Das Schattenfilter arbeitet zum Beispiel mit insgesamt 2 Sätzen von Abtastungen für den 4×4-Modus (4×2 + 4×2 = 4×4) und 8 Sätzen für den 8×8-Modus. Im Fall des 4×4-Modus wendet das Schattenfilter ein 3×3-Filter vier Mal an, jeweils auf die oberen linken, oberen rechten, unteren linken und unteren rechten 3×3-Blöcke in der insgesamt 4×4-Fußspur. Im ersten Taktzyklus verarbeitet es das obere 4×2-Gitter, und im zweiten Takt verarbeitet es das untere 4×2-Gitter im 4×4-Block. Der Abtastindex ist ein Index, der zum Identifizieren des Satzes von 8 Abtastungen verwendet wird, die gerade verarbeitet werden. Der Abtastindex durchläuft 2 Taktzyklen für den 4×4-Fall und 8 Taktzyklen für den 8×8-Fall und stellt fest, welche 4×2-Teilmenge gerade verarbeitet wird.

[0676] Wie in Fig. 41 gezeigt, umfasst die Texturfilterengine einen Keygenerator **1310**, Bruchteilkontrolle **1312**, Farbkomponenteninterpolator **1314**, Schattenfilterakkumulator **1316** und Akkumulator und Postprozessor **1318**.

[0677] Bei einer Texturabbildungsoperation liest der Keygenerator **1310** die (s, t)-Koordinaten und LOD und erzeugt die Cache-Keys, um die entsprechenden Texturdaten aus dem Texturcache abzurufen. Der Texturcache gibt Alpha und die RGB-Komponenten als Reaktion auf die Texturanforderungen zurück. Die Bruchteilkon-

trolle **1312** empfängt die (s, t)-Koordinaten als Eingabe und steuert die Operation der bilinearen und trilinearen Interpolatoren im Farbkomponenteninterpolator **1314**. Der Farbkomponenteninterpolator **1314** interpoliert die Texelabtastwerte, um interpoliertes Alpha und RGB-Komponenten zu berechnen. Der Akkumulator und Postprozessor **1318** skaliert dann die Alpha- und RGB-Komponenten, akkumuliert die skalierten Komponenten und gibt Alpha- und Farbfaktoren aus, die einem Pixelort entsprechen, der gerade verarbeitet wird. Diese Alpha- und Farbfaktoren sind Farb- und Alphawerte, die in die Pixelengine als Eingabe für den Texturmodulationsprozess eingegeben werden.

[0678] Bei der anisotropen Texturabbildunglaufen die Farbkomponenteninterpolatoren **1314** entlang der Linie der Anisotropie und führen eine dreilineare Interpolation bei jedem Schritt aus. Der Akkumulator **1318** fungiert als eindimensionales Filter, das Alpha und die Farbkomponenten skaliert und dann die skalierten Komponenten akkumuliert. In einer speziellen Implementierung skaliert der Akkumulator **1318** Alpha und die Farbkomponenten unter Verwendung der trapezförmigen oder Dreiecksfilterung auf der Basis des Anisotropieverhältnisses. In jedem Fall skaliert der Akkumulator Komponenten an den entfernten Rändern des Resamplingfilters, um eine Dämpfung (roll off) an den Filterkanten anzunähern. Um eine trapezförmige Filterung zu erreichen, entspricht der Skalenfaktor einer linearen Dämpfung an den Filterkanten und ist bei Schritten zwischen den Filterkanten konstant.

[0679] In einer speziellen Implementierung werden die Skalenfaktoren für Schritte entlang der Linie der Anisotropie folgendermaßen berechnet. Für ein Anisotropieverhältnis von 1:1 bis 2:1 wendet der Akkumulator einen Gewichtungsfaktor von 0,5 bei jedem Schritt das anisotropen Walkers an. Für ein Anisotropieverhältnis von 2:1 und darüber gewichtet der Akkumulator Komponenten mit 1,0/Anisotropie für Schritte n < (Anisotropie – 1)/2, und gewichtet Komponenten mit 0,5(Anisotropie – 2n)/Anisotropie für n größer als oder gleich (Anisotropie – 1)/2. Das Anisotropieverhältnis in diesem speziellen Beispiel ist das Verhältnis der langen zur kurzen Seite des bestmöglich angepassten Rechtecks für eine inverse Jacobi-Matrix. Die inverse Jacobi-Matrix ist eine Matrix von partiellen Ableitungen der geometrischen Transformation aus Betrachtungsraumkoordinaten in Texturkoordinaten (d.h. von (x, y)- in (s, t)-Koordinaten). Die Linie der Anisotropie ist eine Linie durch die (s, t)-Koordinaten in Richtung des längeren Spaltenvektors der inversen Jacobi-Matrix.

[0680] Für Schattierungsoperationen liest der Keygenerator **1310** die (s, t)-Koordinaten des Pixelorts, der in die Schattenabbildung abgebildet wird, und erzeugt Cache-Keys. Der Texturcache gibt die Schattenabbildungselemente (Shadels) an den Schattenfilterakkumulator **1316** zurück. Das Schattenfilter empfängt den Schattenindex und Beta als Eingabe und vergleicht die Tiefe der aktuellen Gruppe der Pixeldaten im hellen Raum mit den Tiefenwerten in der Filterfußspur, um eine Schattenmaske zu erzeugen. Der Schattenfilterakkumulator summiert Elemente in der Schattenmaske und teilt die Summe durch die Zahl der Abtastwerte. Bei dieser Implementierung erreicht die Texturfilterengine das glatte Abfallen an den Kanten der Filterfußspur durch Anwenden eines trapezförmigen Filters auf das Ergebnis des Tiefenvergheichsschritts. Um das trapezförmige Filter zu implementieren, berechnet das Schattenakkumulationsfilter vier vorläufige Schattenkoeffizienten durch viermaliges Anwenden eines 3×3- oder 7×7-Kastenfilters auf eine 4×4- bzw. 8×8-Filterfußspur und leitet die vier vorläufigen Koeffzienten an einen der Farbinterpolatoren **1314** weiter. Dieser Farbinterpolator **1314** führt eine bilineare Interpolation an den vorläufigen Koeffizienten aus, um einen endgültigen Schattenkoeffizienten zu berechnen.

[0681] Wie oben vorgestellt, liest der Keygenerator **1310** (s, t)-Koordinaten aus der Pixelschlange und erzeugt Cache-Keys, um Texturdaten aus dem Texturcache abzurufen. Fig. 40 ist ein Schema, das den Keygenerator detaillierter illustriert. Auf der Grundlage der (s, t)-Koordinaten in den Hi- und Lo-MIP-Abbildungen (den zwei nächsten MIP-Traps) berechnet der Keygenerator die Texturabtastorte in den Hi- und Lo-MIP-Abbildungen (**1340**). Der Keygenerator berechnet dann die Cache-Keys aus diesen Abtastwerten (**1342**). Der Keygenerator überträgt die Cache-Keys, (s, t)-Koordinaten und LOD für die Hi- und Lo-MIP-Abbildungsebenen auf den Texturcache, der die angeforderten Texturabtastwerte zurückgibt. Wenn natürlich nur eine Texturabbildungsebene der Details verwendet wird, erzeugt der Keygenerator Keys nur für eine Texturabbildung.

[0682] Die Bruchteilkontrolle **1312** in Fig. 39 steuert die Interpolation zwischen den Abtastungen in einer Textur- oder Schattenabbildung und zwischen MIP-Abbildungsebenen für die trilineare Interpolation. Zur Unterstützung der bilinearen Interpolation steuert die Bruchteilkontrolle das Gewichten zwischen den Abtastungen in einer Textur- oder Schattenabbildung. Zur Unterstützung der trilinearen Interpolation weist die Bruchteilkontrolle die Interpolatoren an, zwischen den vier nächsten Abtastwerten auf einen Punkt zu interpolieren, der in die zwei nächstgelegenen MIP-Abbildungsebenen (bilineare Interpolation) abgebildet wird, und weist dann einen linearen Interpolator an, das Ergebnis aus den zwei MIP-Abbildungsebenen zu mischen. Die Bruchteilkontrolle empfängt die LOD und (s, t)-Koordinaten für die Hi- und Lo-MIP-Abbildungsebenen als Eingabe und steu-

ert die Interpolation zwischen Abtastwerten auf jeder MIP-Ebene und zwischen MIP-Abbildungsebenen.

[0683] Der Farbkomponenteninterpolator **1314** umfasst Interpolatoren für Alpha und RGB-Farbkomponenten. Fig. 41 ist ein Schema, das einen der vier Interpolatoren detaillierter illustriert. Dieser Interpolator handhabt die Farbkomponenteninterpolation für eine Komponente und führt eine bilineare Interpolation an den Schattenkoeffizienten aus. Die anderen Farbkomponenteninterpolatoren handhaben nur eine Farbkomponente.

[0684] Der Farbkomponenteninterpolator empfängt Texel oder Schattenabbildungselemente aus dem Texturcache und wendet sie auf eine Bank von Multiplexern **1350** an. Wenn sie in die Bank der Multiplexer **1350** eingegeben werden, geben die Abtastwert-gültigen Daten an, welche der Abtastwerte gültig sind, d.h. diejenigen, die für die aktuelle Textur- oder Schattierungsoperation verwendet werden sollten. Auf der Basis der Abtastwert-gültigen Steuersignale wählen die Multiplexer entweder den eingehenden Abtastwert oder eine Texturhintergrundfarbe **1352** aus. Für Schattierungsarbeiten leitet der Farbkomponenteninterpolator **1314** Schattenelemente an den Schattenfilterakkumulator **1316** weiter. Die drei Farbkanäle werden dazu verwendet, ein einzelnes 24-Bit breites Schattenabbildungselement zu bilden, und der Alphakanal wird bei Schattierungsoperationen ignoriert. Für Texturabbildungsoperationen überträgt der Farbkomponenteninterpolator Textabtastwerte in die Stufen der linearen Interpolatoren **1354**, **1365** und **1358**.

[0685] Bei der trilinearen Interpolation verwendet der Farbkomponenteninterpolator drei Stufen von linearen Interpolatoren, zwei zum Interpolieren zwischen Abtastwerten auf jeder MIP-Abbildungsebene (**1354** und **1356**) und eine weitere zum Mischen der Ergebnisse aus jeder MIP-Ebene (**1358**). Der Farbkomponenteninterpolator führt eine bilineare Interpolation aus, um Schattenkoeffizienten zu kombinieren, die aus 4 Filterfußspuren berechnet wurden. Wie in Fig. 43 gezeigt, verwendet er die letzten zwei Stufen (**1356** und **1358**), um diese bilineare Interpolation auszuführen. Eine zweite Bank von Multiplexern **1360** wählt zwischen vier Schattenkoeffizienten und der Ausgabe der ersten Stufe von linearen Interpolatoren **1354** aus. Sowohl bei der Texturabbildung als auch bei Schattierungsoperationen überträgt der Farbkomponenteninterpolator die Ausgabe der Interpolatorstufen auf den Akkumulator und Postprozessor **1318**.

[0686] Der Schattenfilterakkumulator **1316** empfängt einen Abtastwerteindex und Lichttiefenwert (Beta) aus der Pixelschlange, vergleicht den Lichttiefenwert mit den Schattenabbildungselementen, die vom Texturcache zurückgegeben werden, um Schattenmasken zu erzeugen, und filtert die Schattenmasken, um vorläufige Schattenkoeffizienten zu berechnen. **Fig.** 44 ist ein Schema, das den Schattenfilterakkumulator detaillierter illustriert. Tiefenkomparatoren im Schattenfilterakkumulator vergleichen die Tiefe der Schattenelemente in der Filterfußspur und erzeugen eine Schattenmaske. In diesem speziellen Fall beträgt die Schattenmaske 8 Bit mit booleschen Werten, was einem 4×2-Abschnitt der Filterfußspur entspricht.

[0687] Die Fußspurkontrolle **1372** wählt den aktuellen 4×2-Abschnitt der gesamten Fußspur auf der Basis des Abtastindexwertes aus der Pixelschlange aus. Die Fußspurkontrolle überträgt eine Fußspurmaske auf jeden der vier Schattenbeitragsblöcke auf der Basis des Taktzyklus und des Filtermodus (2×2, 4×4 oder 8×8). Die Fußspurmaske zeigt an, welche der 8 Schattenmaskenelemente bei dem aktuellen Taktzyklus für jedes der vier Kastenfilter in den Modi 4×4 und 8×8 gültig sind. Im Modus 2×2 gibt der Schattenfilterakkumulator vier boolesche Werte aus, die anzeigen, ob jeder der vier nächstgelegenen Abtastwerte im Schatten ist oder nicht.

[0688] Der Schattenfilterakkumulator wendet vier Kastenfilter (z.B. 3×3 oder 7×7) auf die Abtastwerte in der Filterfußspur an. Jeder der Schattenbeitragsblöcke kombiniert die Fußspurmaske und die Schattenmaske, um festzustellen, welche Elemente der Schattenmaske für den aktuellen Taktzyklus gültig sind, und summiert dann die gültigen Elemente. Nach dem Akkumulieren der gültigen Elemente in der Schattenmaske für die ganze Filterfußspur teilen die Schattenbeitragsblöcke die Summe durch die Zahl der Abtastwerte, um vorläufige Schattenkoeffizienten zu berechnen, die in eine bilineare Interpolationsstufe im Farbinterpolator übertragen werden. Dieser Farbinterpolator interpoliert dann zwischen den vorläufigen Koeffizienten, um einen endgültigen Schattenkoeffizienten zu berechnen.

[0689] Der Akkumulator und Postprozessor **1318** empfängt Alpha- und Farbkomponenten aus dem Farbkomponenteninterpolator **1314** und berechnet Farb- und Alphafaktoren für jede Gruppe der Texturreferenzdaten. Für Schattierungsoperationen verwendet die Texturfilterengine einen Kanal (Alpha oder RGB), um einen Schattendämpfungskoeffizienten zu berechnen. Die Schattenfilterungslogik kann auch separat implementiert werden. Fig. 43 ist ein Schema, das den Akkumulator und Postprozessor detaillierter illustriert. Wie gezeigt, hat jede Farbkomponente (Alpha und RGB) einen Skalierer und Akkumulator. Der Skalierer und Akkumulator **1380** für jede Komponente empfängt die Akkumulationsskala und eine Farbkomponente als Eingabe und skaliert als Reaktion die Farbkomponente und addiert sie zu einem akkumulierten Komponentenwert im Kompo-

nentensummenblock **1382**. Bei der Anisotropiefilterung zum Beispiel wichten die Skalier- und Akkumulationsblöcke **1380** die Ausgabe des Rekonstruktionsfilters (trilinearer Interpolator), während die Texturfilterengine die Linie der Anisotropie abläuft. Nach dem letzten Schritt geben die Skalierer und Akkumulatoren für Alpha und RGB-Komponenten die endgültigen Farbkomponentenfaktoren aus.

[0690] Für Schattierungsarbeiten umgeht der Skalier- und Akkumulationsblock die Multiplikationsoperation, addiert aber einen Umgebungsversatz hinzu. Der Umgebungsversatz stellt sicher, dass selbst Objekte, die vollkommen im Schatten liegen, sichtbar sind. Ein Schattenkoeffizient von zum Beispiel 1 bedeutet vollständig beleuchtet, ein Schattenkoeffizient von 0 bedeutet vollständig im Schatten. Wenn Farben mit einem Koeffizienten von null multipliziert werden würden, wäre das Objekt an diesem Pixelort nicht sichtbar. Daher wird ein Versatz addiert und die Schattenkoeffizienten werden auf 1 festgehalten, so dass die versetzten Schattenkoeffizienten vom Versatzwert bis 1 reichen.

[0691] Der Schattenpostprozessor **1384** erledigt die Vervielfältigung der skalaren Schattendämpfung ‚s' für alle drei Farbkanäle und (bedingungsweise) für den Alphakanal. Es gibt auch eine bedingte Ergänzung (s = 1 – s) der Schattendämpfung, um ein Schattenbild zu berechnen. Ein Schattenbild ist ein Feld von Schattenkoeffizienten oder ein Feld der Komplemente von Schattenkoeffizienten, die zum Schattieren eines Gsprites verwendet werden können.

[0692] Zum Schluss wählt die Multiplexerstufe **1386** entweder die Schattenkoeffizienten für Schattierungsoperationen oder RGB- und Alphakomponenten für Texturabbildungsoperationen aus. Zusammengefasst, die Texturfilterengine **401** führt sowohl Schattierungs- als auch Texturierungsoperationen aus. Sie leitet das Ergebnis der Texturabbildungsoperationen an eine Texturmodulationsstufe. Texturmodulation umfasst normalerweise das Multiplizieren der RGB-Farbwerte aus dem Texturfilter mit einer interpolierten Farbe oder Farbwerten, die im Scanumwandlungsblock berechnet wurden. Neben grafischen Objekten mit Lichtdurchlässigkeit kann die Texturmodulation auch das Multiplizieren des Alphawerts aus dem Texturfilter mit einem interpolierten Alphawert aus dem Scanumwandlungsblock umfassen. Je nach der Implementierung kann die Texturmodulation in der Texturfilterengine (Element **401** in **Fig. 9B**) oder der Pixelengine (Element **406** in **Fig. 9b**) implementiert werden. Sie kann auch im Scanumwandlungsblock (Element **394** in **Fig. 9A** oder Element **397** in **Fig. 9C**) implementiert werden. In einer Implementierung kombiniert die Texturfilterengine **401** einen interpolierten Wert mit dem gefilterten Wert, um einen zusammengesetzten Wert zu berechnen. Die Pixelengine **406** stellt dann fest, ob ein zusammengesetzter Wert gespeichert oder mit einer entsprechenden RGB-Komponente oder einer Alphakomponente, die in den Pixel- oder Fragmentpuffern gespeichert ist, für einen entsprechenden Pixelort kombiniert werden soll.

[0693] Im Fall von Schattierungsoperationen können die Schattenkoeffizienten auf die RGB- und Alphawerte an entsprechenden Pixelorten in den Pixel- oder Fragmentpuffern angewendet oder auf interpolierte RGB- oder Alphawerte, die während des aktuellen Durchlaufs erzeugt und in einer Schlange gepuffert werden, angewendet werden. Wenn zum Beispiel ein Objekt keine Textur hat, die mit ihm verknüpft ist, kann der Texturmodulator in der Texturfilterengine **401** interpolierte und unaufgelöste RGB- und Alphawerte, die das beleuchtete Bild repräsentieren und in der Texturreferenzdatenschlange **391** (**Fig. 9B**) gespeichert sind, mit den Schattendämpfungskoeffizienten aus dem Akkumulator und Postprozessor multiplizieren.

[0694] Wir haben verschiedene Aspekte eines Bildverarbeitungssystems, seine Architektur und damit verbundene Verfahren mit Bezug auf mehrere Ausführungsformen beschrieben. Obwohl wir mehrere Ausführungsformen detailliert beschrieben haben, ist es nicht unsere Absicht, unsere Erfindung auf diese speziellen Ausführungsformen zu beschränken. Unsere neuartige Architektur kann zum Beispiel auf eine Reihe von Hardware-Implementierungen angewendet werden, einschließlich von und ohne Einschränkung auf Computersysteme, die von tragbaren Geräten bis zu Arbeitsplatzrechnern reichen, Spielestationen, Settopboxen, Grafikverarbeitungshardware, Grafikverarbeitungssoftware und Videobearbeitungsgeräte. Abwandlungen unserer Systeme und Verfahren können in Hardware oder Software oder einer Kombination beider implementiert werden.

[0695] In Anbetracht der vielen möglichen Ausführungsformen, in die die Prinzipien unserer Erfindung umgesetzt werden können, betonen wir, dass die detailliert dargestellten Ausführungsformen, die oben beschrieben werden, nur einen erläuternden Charakter besitzen und nicht als Einschränkung für den Geltungsbereich unserer Erfindung angesehen werden dürfen. Vielmehr beanspruchen wir, dass alle solche Ausführungsformen, die kommen mögen, innerhalb des Geltungsbereichs der folgenden Ansprüche unserer Erfindung liegen.

Patentansprüche

1. Verfahren zum Erzeugen von Bildern zur Anzeige in einem Vollbild oder einem anderen Betrachtungsraum einer physikalischen Ausgabevorrichtung, wobei das Verfahren die Schritte umfaßt:
Identifizieren eines oder mehrerer potentiell sichtbarer Objekte in einer Szene, wobei jedes von dem einen oder den mehreren potentiell sichtbaren Objekten mehrere geometrische Grundelemente aufweist und entsprechende zweidimensionale Bereiche des Betrachtungsraums umfaßt, um das eine oder die mehreren potentiell sichtbaren Bilder darin zu rendern;
Aufteilen jedes zweidimensionalen Bereichs unter mehreren Bildabschnitten oder Stücken;
Aufteilen der geometrischen Grundelemente des einen oder der mehreren potentiell sichtbaren Objekte unter den mehreren Bildabschnitten oder Stücken durch Zuweisen geometrischer Grundelemente des Objektes zu Stücken des entsprechenden zweidimensionalen Bereichs;
Serielles Rendern des einen oder der mehreren Objekte durch serielles Rendern der Stücke des entsprechenden zweidimensionalen Bereichs, wobei jedes entsprechende Stück durch Rendern der dem Stück zugeordneten geometrischen Grundelemente gerendert wird;
Speichern der gerenderten Stücke für Objekte in der Szene in einem wahlfrei zugänglichen Stück-Format;
Abrufen der gerenderten Stücke für die Objekte im Stück-Format;
Zusammensetzen von Pixeln aus den gerenderten Stücken der Objekte, um ein Anzeigebild zu berechnen, das die Objekte in der Szene wiedergibt; und
Wiederholen der obigen Schritte zur Verarbeitung und Anzeige nachfolgender Anzeigebilder.

2. Verfahren nach Anspruch 1, wobei die Objektgeometrie für jedes Stück der Szene seriell gerendert wird, um das gerenderte Bild zu produzieren.

3. Verfahren nach Anspruch 1, wobei die Objektgeometrie für die zumindest zwei Stücke in einen gemeinsamen Tiefenpuffer gerendert wird.

4. Verfahren nach einem der Ansprüche 1 bis 3, wobei die Szene mehrere Objekte umfaßt und der Schritt des Sortierens die Objektgeometrie unter mehreren Bildabschnitten oder Stücken den Schritt eines Zuweisens von geometrischen Grundelementen jedes der mehreren Objekte zu Stücken entsprechender zweidimensionaler Bildbereiche umfaßt.

5. Verfahren nach Anspruch 4, wobei der Schritt des seriellen Renderns der zumindest zwei Stücke umfaßt: serielles Rendern der mehreren Objekte, wobei das serielle Rendern jedes der mehreren Objekte ein serielles Rendern der geometrischen Grundelemente der mehreren Objekte in die Stücke der entsprechenden zweidimensionalen Bildbereiche umfaßt, um getrennte Bildschichten für jeden der zweidimensionalen Bildbereiche zu erzeugen.

6. Verfahren nach Anspruch 1, wobei die Stücke sich an variablen und adressierbaren Abschnitten des Betrachtungsraums befinden.

7. Verfahren nach Anspruch 1, wobei der Aufteilungsschritt den Schritt eines Aufteilens des zweidimensionalen Bildbereichs unter Stücken an nicht festgelegten Orten des Betrachtungsraums umfaßt.

8. Verfahren nach Anspruch 1, wobei die Stücke rechteckige Bereiche des Betrachtungsraums sind.

9. Verfahren nach Anspruch 1, wobei der Schritt des seriellen Renderns der Objektgeometrie für die zumindest zwei Stücke die Schritte umfaßt:
Rastern geometrischer Grundelemente für ein Stück, um Pixeldaten zu erzeugen, und dann Auflösen der Pixeldaten für das eine Stück; und
Wiederholen der Rasterungs- und Auflösungsschritte für nachfolgende Stücke.

10. Verfahren nach Anspruch 9, wobei der Rasterungsschritt für ein erstes Stück ausgeführt wird, während der Auflösungsschritt für ein zweites Stück ausgeführt wird.

11. Graphik-Rendersystem zum seriellen Rendern einer Objektgeometrie in einer Szene in einen Betrachtungsraum, wobei das System umfaßt:
einen Speicher zum Speichern gerendeter Bilddaten;
einen Bildvorprozessor, der zur Transformation der Objektgeometrie zum Betrachtungsraum betrieben werden kann und zur Aufteilung der transformierten Objektgeometrie unter mehreren Abschnitten oder Stücken des

Betrachtungsraums betrieben werden kann, wobei das Objekt mehrere geometrische Grundelemente umfaßt, die unter den mehreren Bildabschnitten oder Stücken aufgeteilt sind durch Zuweisen der geometrischen Grundelemente zu den mehreren Bildabschnitten oder Stücken; und
einen Bildprozessor, der mit dem Bildvorprozessor zum Empfang der transformierten Objektgeometrie für die mehreren Stücke kommuniziert, der zum seriellen Rendern der transformierten Objektgeometrie für die mehreren Stücke betreibbar ist, um Pixelwerte für Pixelorte im Betrachtungsraum durch serielles Rendern der Stücke des Betrachtungsraums zu berechnen, wobei jedes entsprechende Stück durch Rendern der diesem Stück zugeordneten geometrischen Grundelemente gerendert wird; und mit dem Speicher kommuniziert, um die Pixelwerte für die mehreren Stücke im Speicher zu speichern; und
wobei gerenderte Stücke für Objekte in der Szene in einem wahlfrei zugänglichen Stück-Format gespeichert werden, im Stück-Format abgerufen und zusammengesetzt werden, um ein Anzeigebild zu berechnen, das die Objekte in der Szene wiedergibt.

12. System nach Anspruch 11, wobei der Bildprozessor eine Rasterungsvorrichtung und einen Rasterungspuffer umfaßt, wobei die Rasterungsvorrichtung zum Rastern der transformierten Objektgeometrie für die mehreren Stücke und zur Erzeugung von Pixeldaten für die mehreren Stücke, mit jeweils einem Stück zu einem Zeitpunkt, betrieben werden kann, und zur Speicherung der Pixeldaten für das eine Stück im Rasterungspuffer betrieben werden kann.

13. System nach Anspruch 12, wobei der Bildprozessor eine Pixelmaschine umfaßt, die mit der Rasterungsvorrichtung zum Empfangen der Pixeldaten kommuniziert und mit dem Rasterungspuffer zur Speicherung ausgewählter Pixeldaten im Rasterungspuffer kommuniziert und zur Durchführung von Tiefenvergleichsoperationen zwischen den von der Rasterungsvorrichtung erzeugten Pixeldaten und den gewählten, im Rasterungspuffer gespeicherten Pixeldaten betrieben werden kann.

14. System nach Anspruch 13, wobei die gewählten Pixeldaten Pixelfragmentaufzeichnungen für Pixelorte für die Stücke umfassen, die verarbeitet wurden, wobei die Pixelfragmentaufzeichnungen Farb-, Tiefen- und Bedeckungsdaten umfassen und das System des weiteren eine Antialiasing-Engine in Kommunikation mit dem Rasterungspuffer zur Auflösung von Pixelfragmenten für Pixelorte im Stück, das bearbeitet wurde, und zum Berechnen der Pixelwerte umfaßt.

15. System nach Anspruch 14, wobei der Rasterungspuffer doppelt gepuffert ist, und in dem die Antialiasing-Engine zur Auflösung von Pixelfragmentaufzeichnungen für ein erstes Stück betreibbar ist, während die Rasterungsvorrichtung Pixeldaten für ein zweites Stück erzeugt.

16. System nach Anspruch 11, wobei der Bildvorprozessor ein programmierter Datenprozessor ist, der zur Sortierung der Geometrie von Objekten in einer Szene unter mehreren Stücken betrieben werden kann.

17. System nach Anspruch 16, wobei der programmierte Datenprozessor zur Transformation von Umgrenzungsfeldern der Objekte in Betrachtungsraumkoordinaten betreibbar ist, zur Unterteilung der transformierten Umgrenzungsfelder in zwei oder mehrere Stücke betreibbar ist und zur Zuordnung von geometrischen Grundelementen der Objekte zu den zwei oder mehreren, den Objekten entsprechenden Stücken betreibbar ist.

Es folgen 54 Blatt Zeichnungen

Anhängende Zeichnungen

FIG. 1

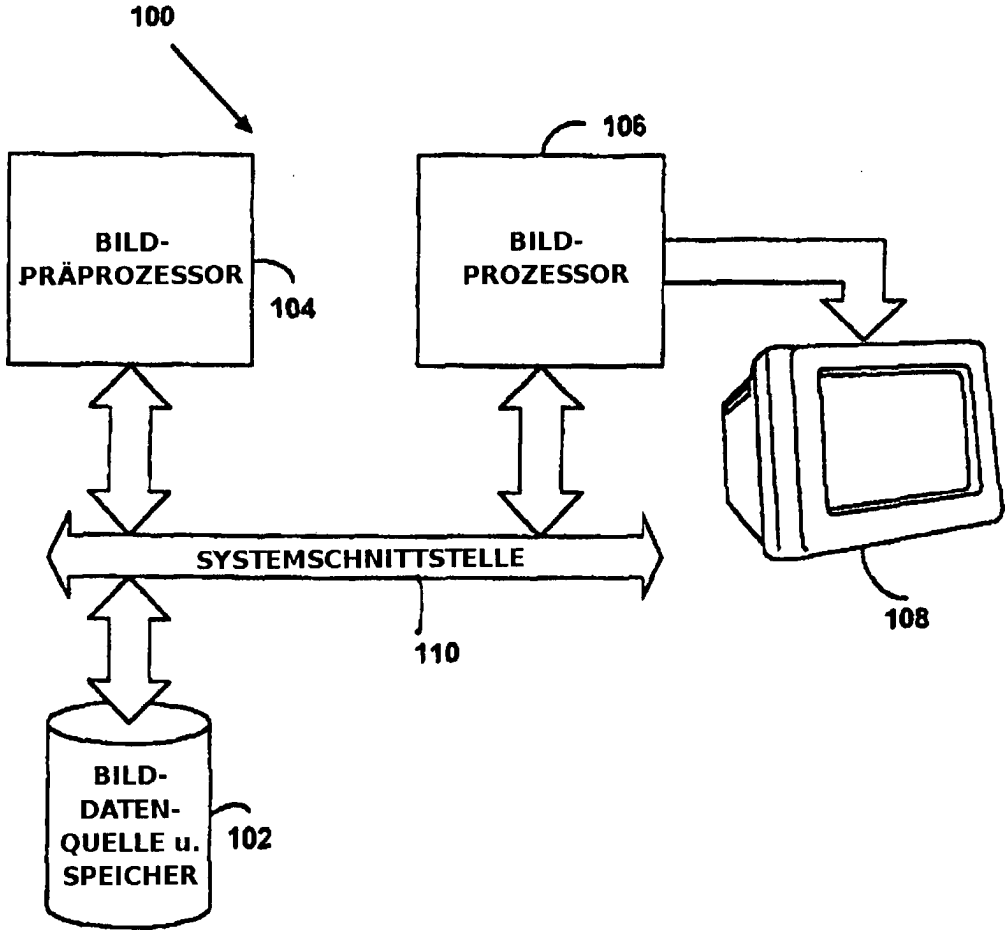


FIG. 2

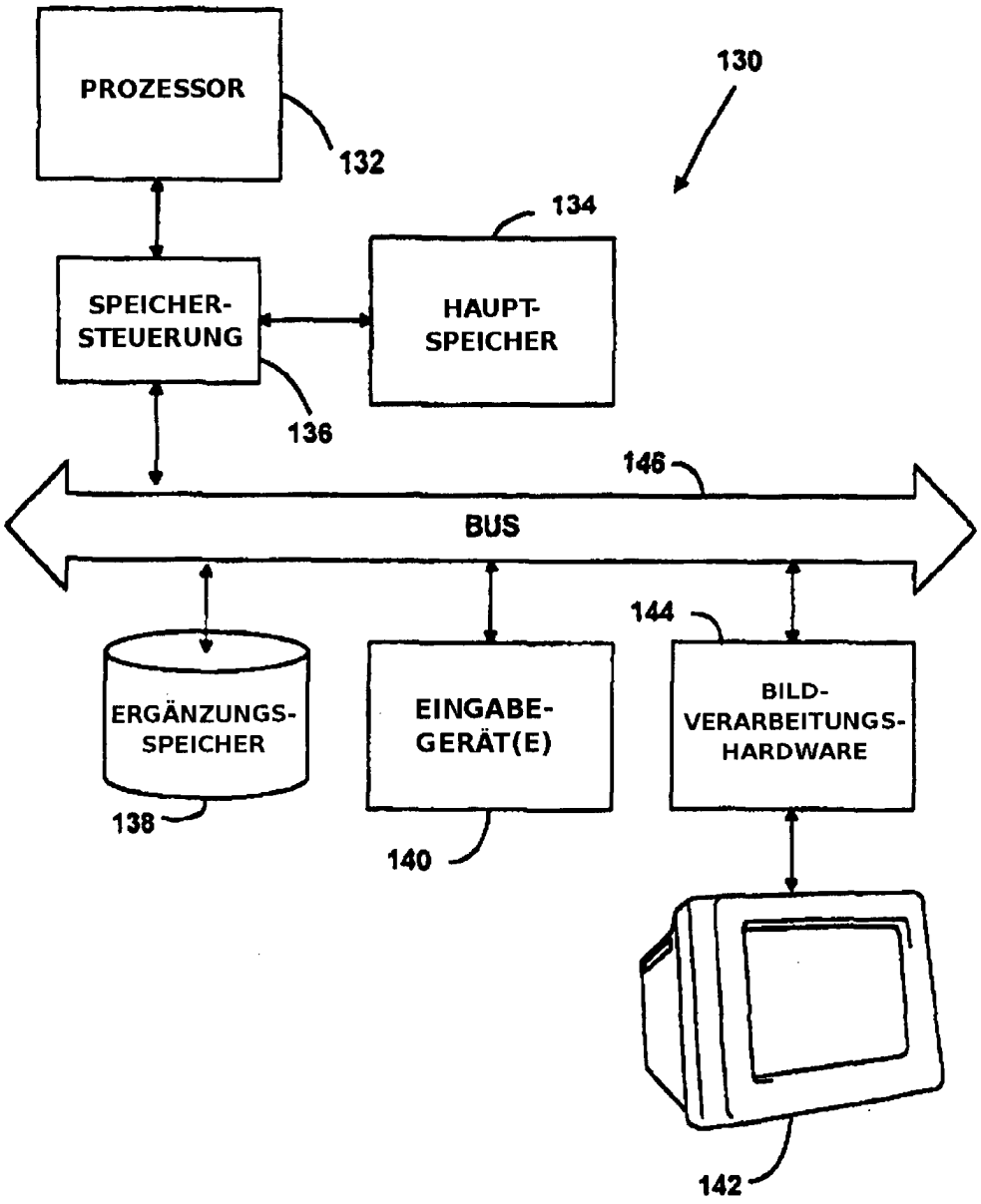

PROZESSOR
132
130
134
SPEICHER-
STEUERUNG
HAUPT-
SPEICHER
136
146
BUS
144
ERGÄNZUNGS-
SPEICHER
EINGABE-
GERÄT(E)
BILD-
VERARBEITUNGS-
HARDWARE
138
140
142

FIG. 3

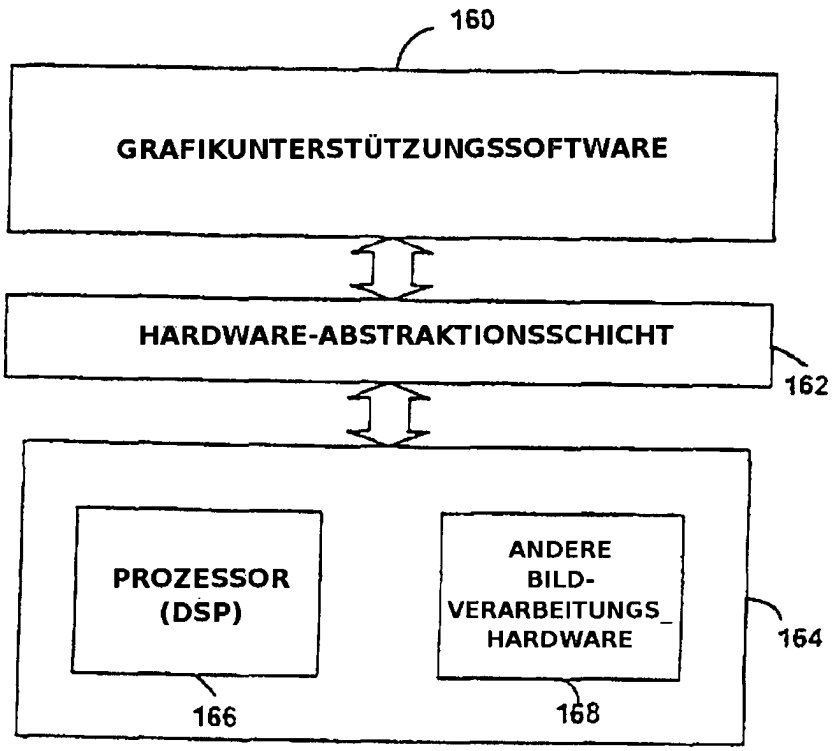

160
GRAFIKUNTERSTÜTZUNGSSOFTWARE
HARDWARE-ABSTRAKTIONSSCHICHT
162
PROZESSOR (DSP)
ANDERE BILD-VERARBEITUNGS_ HARDWARE
164
166
168

FIG. 4A

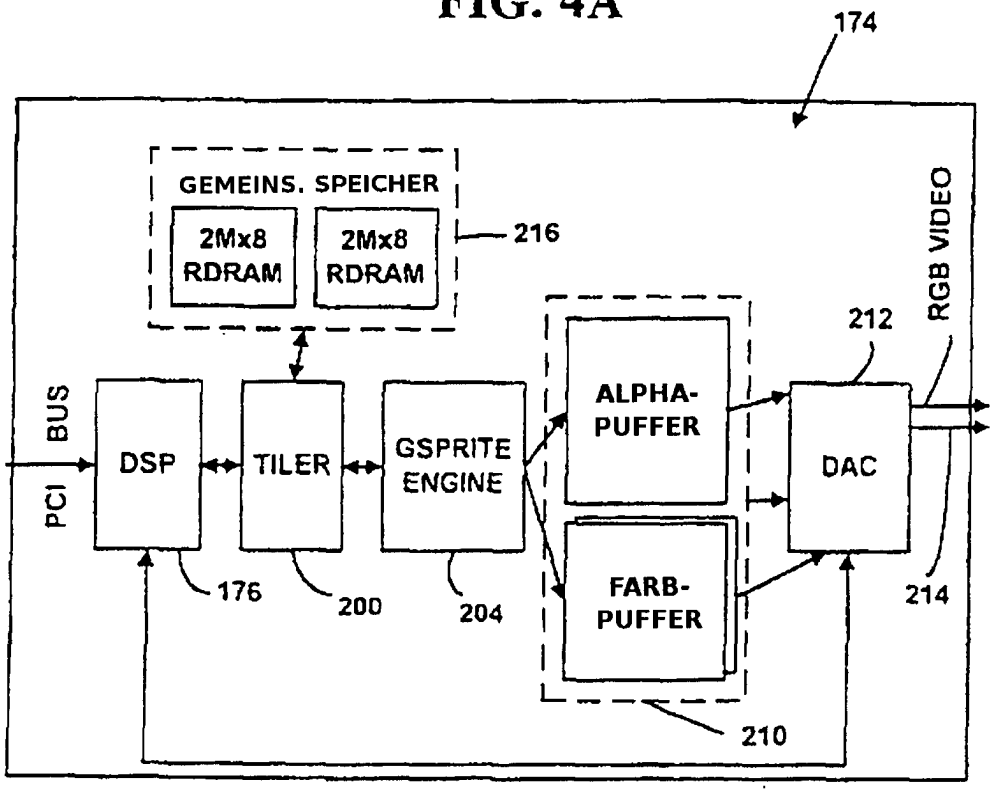

174
GEMEINS. SPEICHER
2Mx8 RDRAM
2Mx8 RDRAM
216
RGB VIDEO
212
PCI BUS
DSP
TILER
GSPRITE ENGINE
ALPHA-PUFFER
FARB-PUFFER
DAC
176
200
204
214
210

FIG. 4B

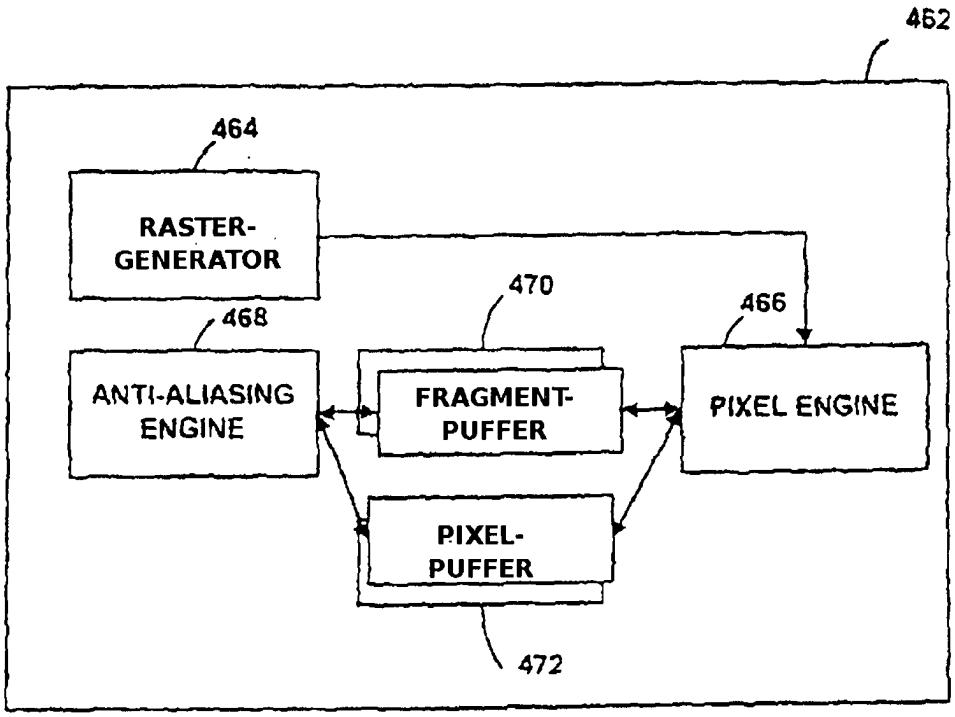

462
464
RASTER-GENERATOR
470
466
468
ANTI-ALIASING ENGINE
FRAGMENT-PUFFER
PIXEL ENGINE
PIXEL-PUFFER
472

FIG. 5A

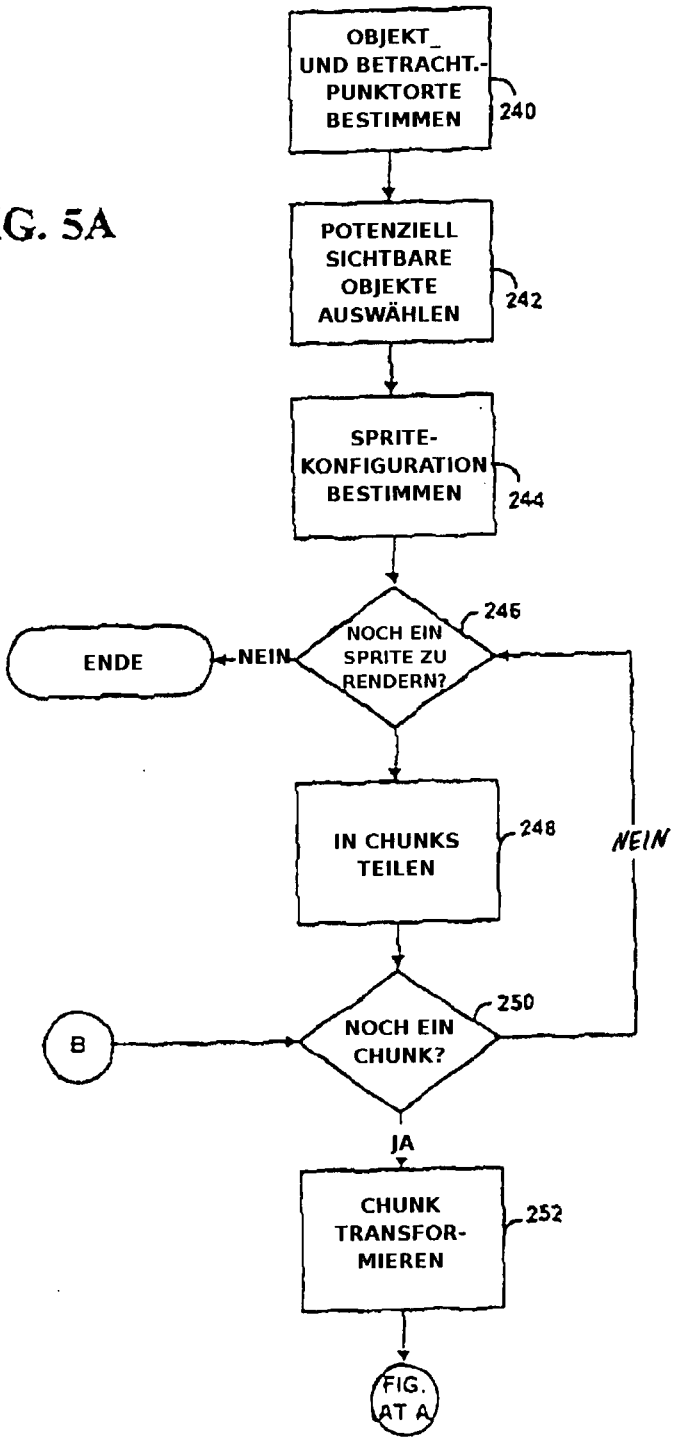

OBJEKT_ UND BETRACHT.-PUNKTORTE BESTIMMEN
240
POTENZIELL SICHTBARE OBJEKTE AUSWÄHLEN
242
SPRITE-KONFIGURATION BESTIMMEN
244
NOCH EIN SPRITE ZU RENDERN?
246
NEIN
ENDE
IN CHUNKS TEILEN
248
NEIN
NOCH EIN CHUNK?
250
B
JA
CHUNK TRANSFOR-MIEREN
252
FIG. AT A

FIG. 5B

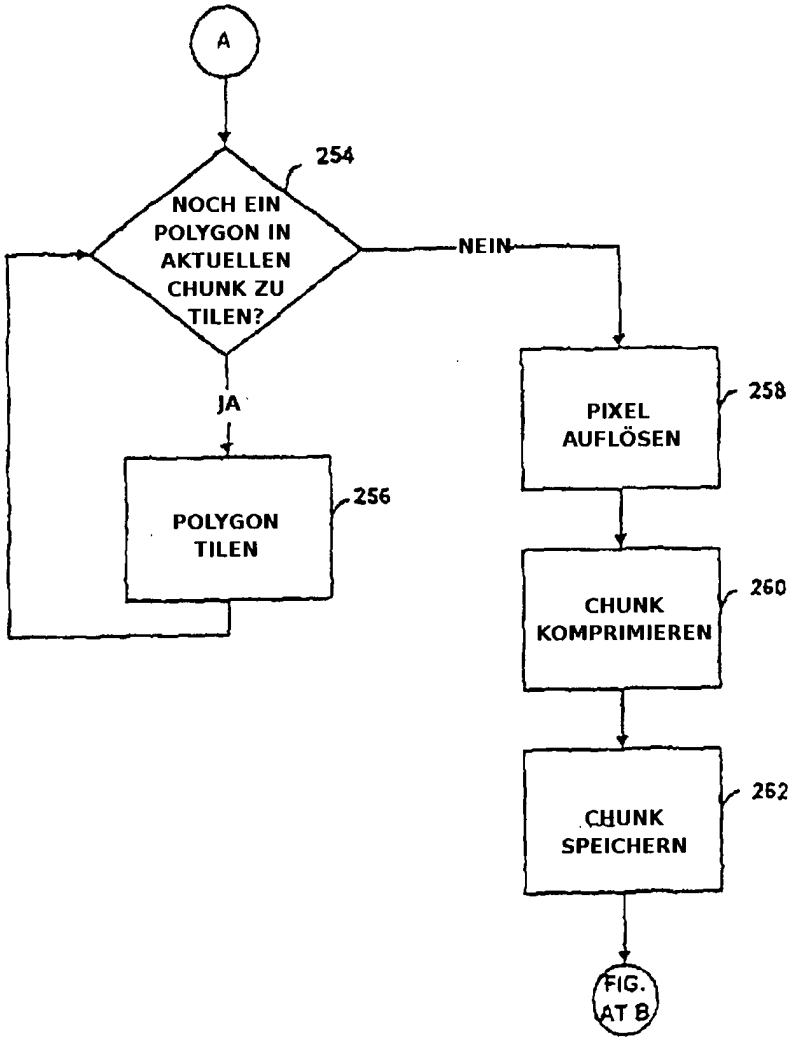

A
254
NOCH EIN POLYGON IN AKTUELLEN CHUNK ZU TILEN?
NEIN
JA
POLYGON TILEN
256
PIXEL AUFLÖSEN
258
CHUNK KOMPRIMIEREN
260
CHUNK SPEICHERN
262
FIG. AT B

FIG. 6

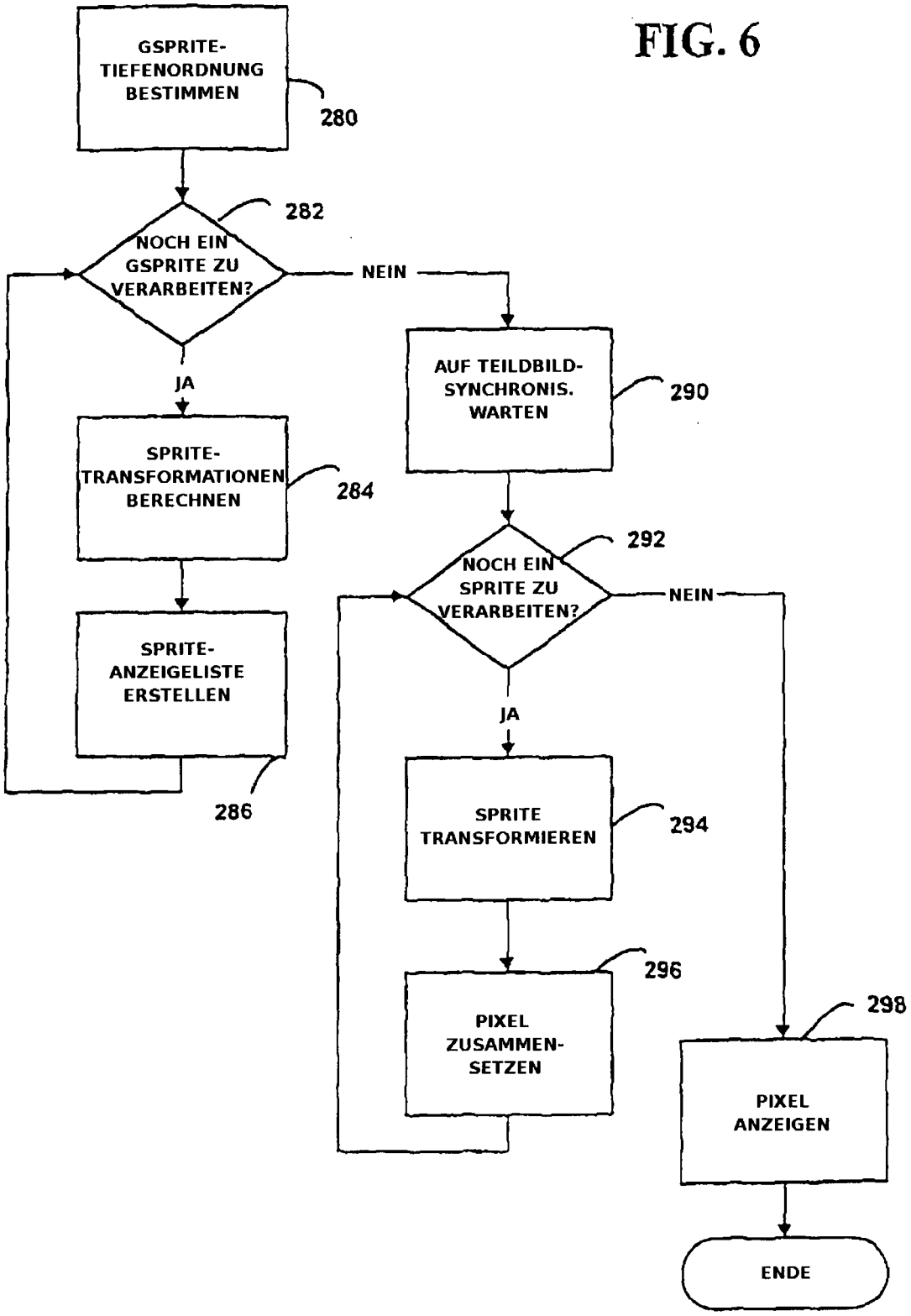

GSPRITE-TIEFENORDNUNG BESTIMMEN
280
282
NOCH EIN GSPRITE ZU VERARBEITEN?
NEIN
JA
SPRITE-TRANSFORMATIONEN BERECHNEN
284
SPRITE-ANZEIGELISTE ERSTELLEN
286
AUF TEILDBILD-SYNCHRONIS. WARTEN
290
292
NOCH EIN SPRITE ZU VERARBEITEN?
NEIN
JA
SPRITE TRANSFORMIEREN
294
296
PIXEL ZUSAMMEN-SETZEN
298
PIXEL ANZEIGEN
ENDE

FIG. 7

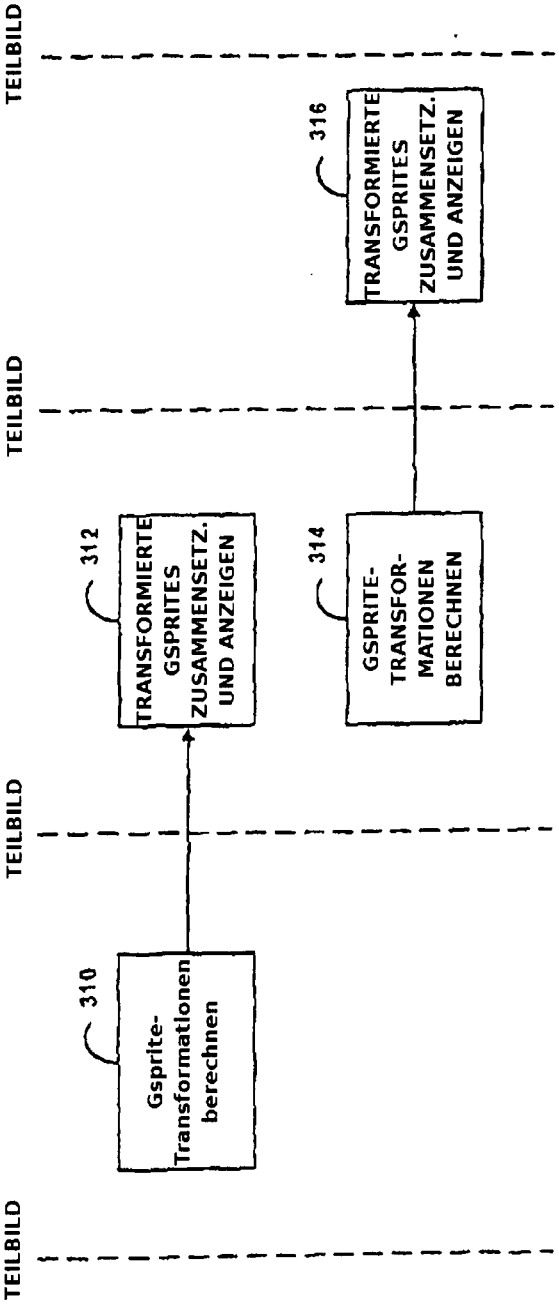

TEILBILD
TEILBILD
TEILBILD
TEILBILD
310
Gsprite-Transformationen berechnen
312
TRANSFORMIERTE GSPRITES ZUSAMMENSETZ. UND ANZEIGEN
314
GSPRITE-TRANSFOR-MATIONEN BERECHNEN
316
TRANSFORMIERTE GSPRITES ZUSAMMENSETZ. UND ANZEIGEN

FIG. 8

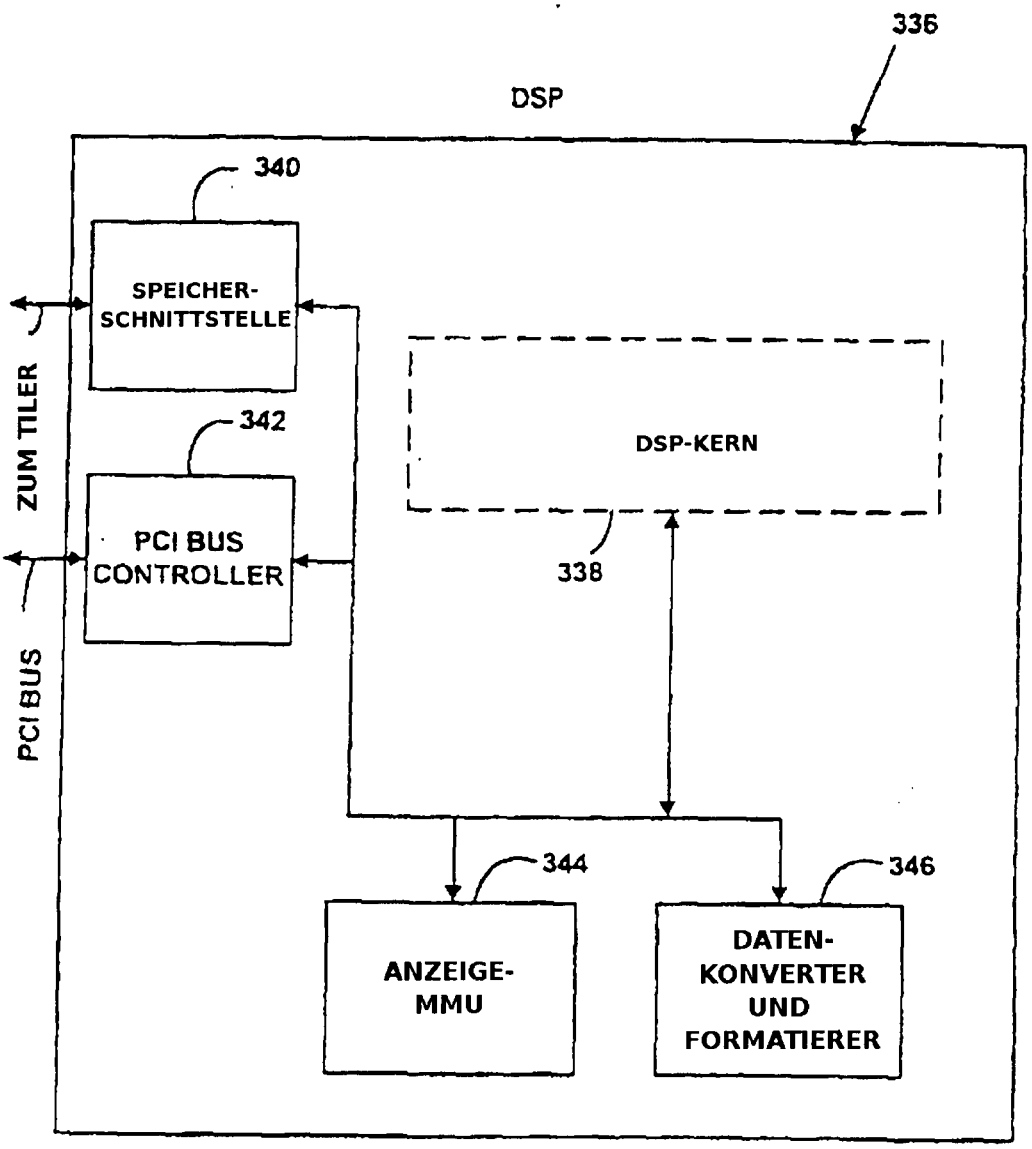

336
DSP
340
SPEICHER-SCHNITTSTELLE
342
PCI BUS CONTROLLER
ZUM TILER
PCI BUS
DSP-KERN
338
344
ANZEIGE-MMU
346
DATEN-KONVERTER UND FORMATIERER

FIG. 9A

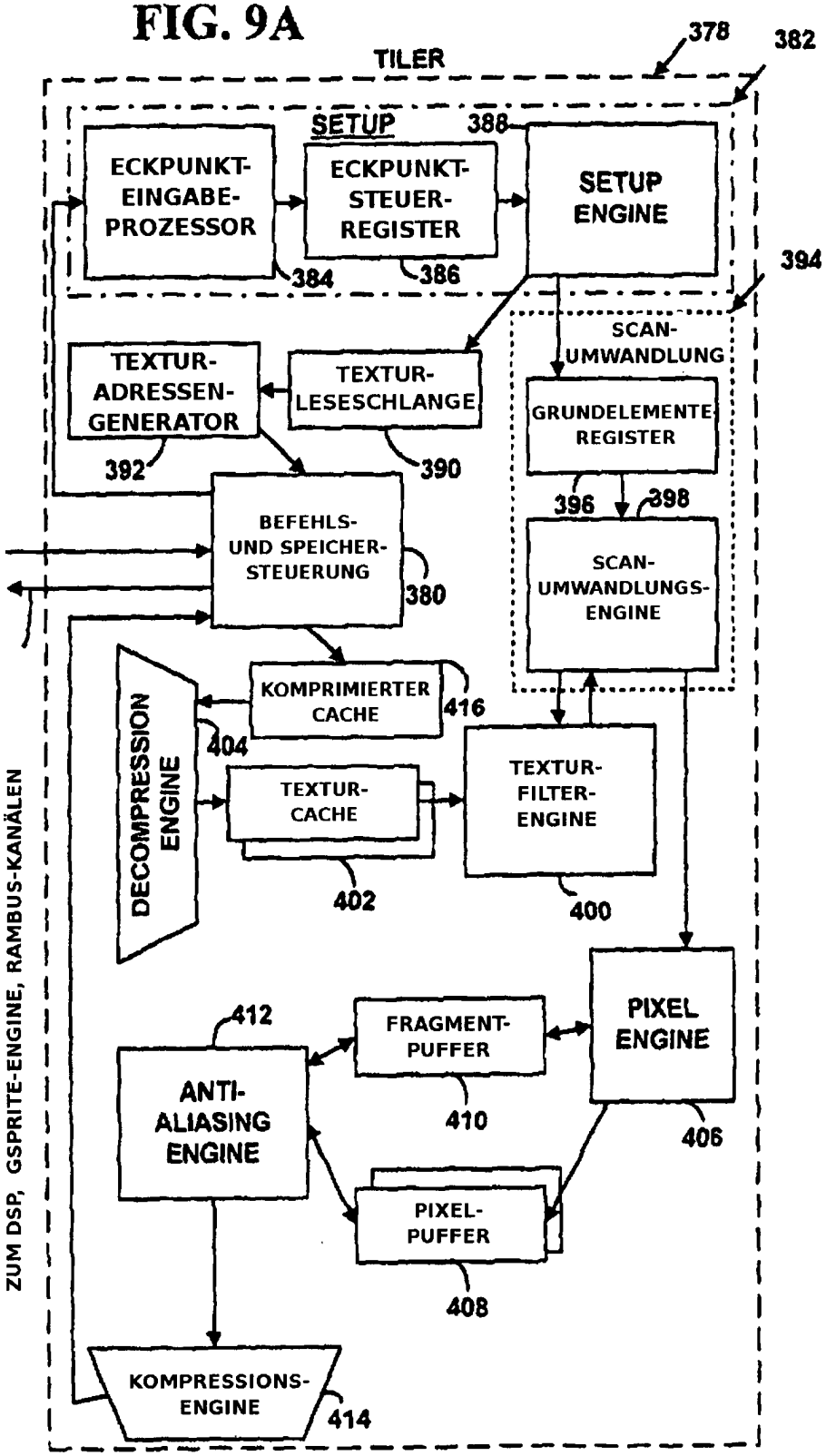

TILER
378
382
SETUP
388
ECKPUNKT-EINGABE-PROZESSOR
ECKPUNKT-STEUER-REGISTER
SETUP ENGINE
384
386
394
SCAN-UMWANDLUNG
TEXTUR-ADRESSEN-GENERATOR
TEXTUR-LESESCHLANGE
GRUNDELEMENTE-REGISTER
392
390
396
398
BEFEHLS- UND SPEICHER-STEUERUNG
SCAN-UMWANDLUNGS-ENGINE
380
KOMPRIMIERTER CACHE
416
DECOMPRESSION ENGINE
404
TEXTUR-CACHE
TEXTUR-FILTER-ENGINE
402
400
412
FRAGMENT-PUFFER
PIXEL ENGINE
ANTI-ALIASING ENGINE
410
406
PIXEL-PUFFER
408
KOMPRESSIONS-ENGINE
414
ZUM DSP, GSPRITE-ENGINE, RAMBUS-KANÄLEN

FIG. 9B

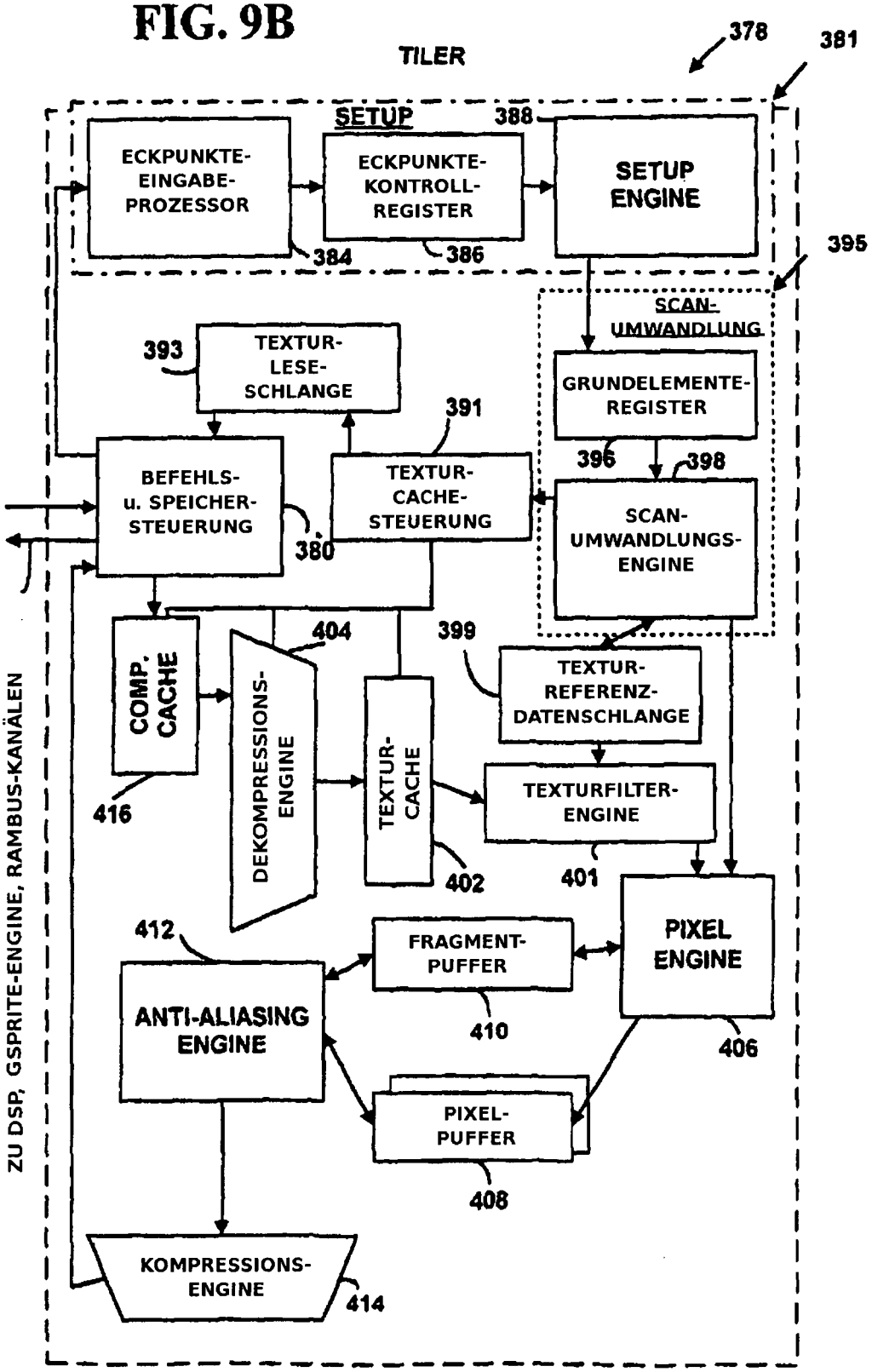

TILER
378
381
SETUP
388
ECKPUNKTE-EINGABE-PROZESSOR
ECKPUNKTE-KONTROLL-REGISTER
SETUP ENGINE
384
386
395
SCAN-UMWANDLUNG
393
TEXTUR-LESE-SCHLANGE
GRUNDELEMENTE-REGISTER
391
396
398
BEFEHLS- u. SPEICHER-STEUERUNG
TEXTUR-CACHE-STEUERUNG
SCAN-UMWANDLUNGS-ENGINE
380
404
399
COMP. CACHE
DEKOMPRESSIONS-ENGINE
TEXTUR-REFERENZ-DATENSCHLANGE
TEXTUR-CACHE
TEXTURFILTER-ENGINE
416
402
401
412
FRAGMENT-PUFFER
PIXEL ENGINE
ANTI-ALIASING ENGINE
410
406
PIXEL-PUFFER
408
KOMPRESSIONS-ENGINE
414
ZU DSP, GSPRITE-ENGINE, RAMBUS-KANÄLEN

FIG. 9C

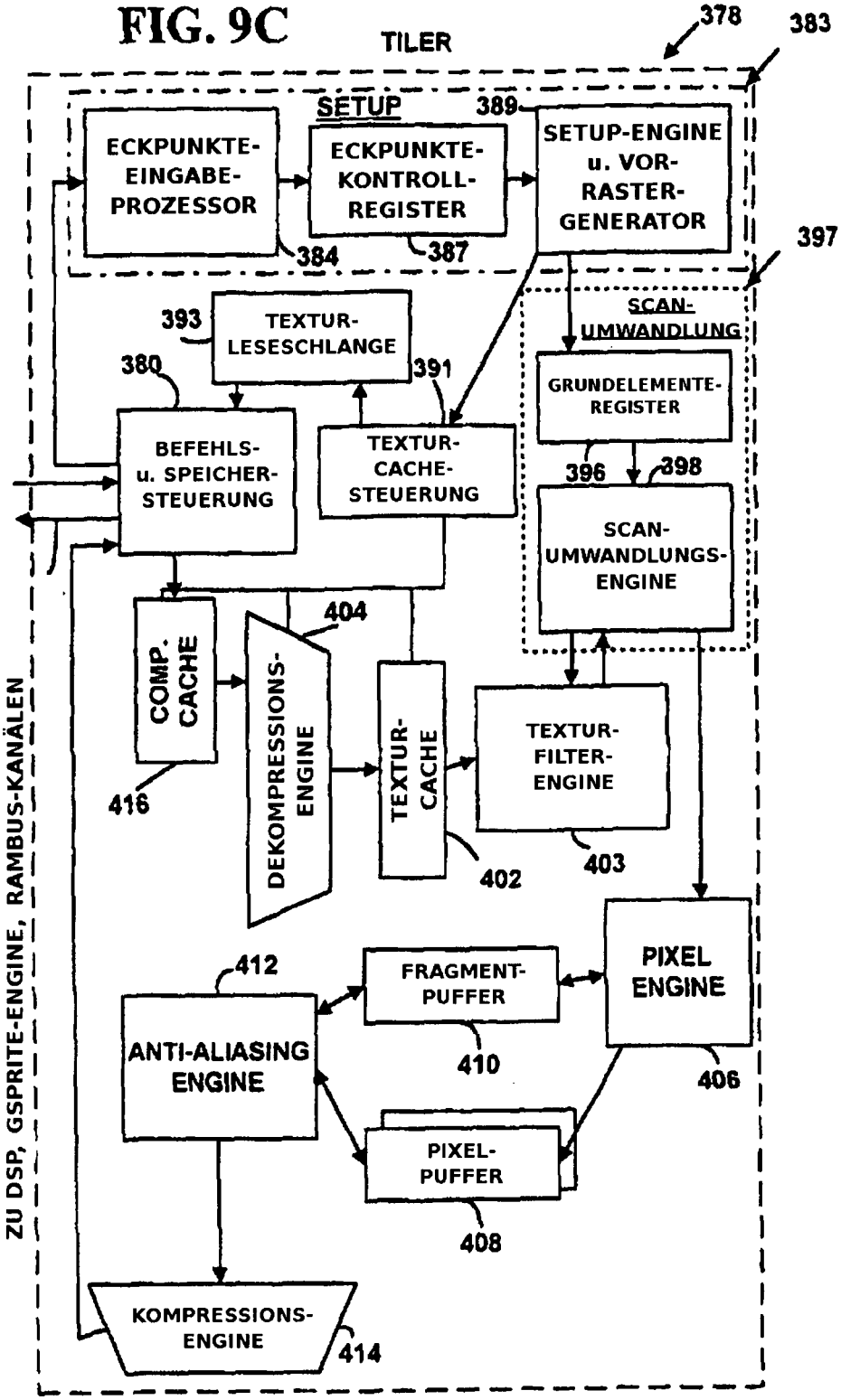

TILER
378
383
SETUP
389
ECKPUNKTE-EINGABE-PROZESSOR
ECKPUNKTE-KONTROLL-REGISTER
SETUP-ENGINE u. VOR-RASTER-GENERATOR
384
387
397
SCAN-UMWANDLUNG
393
TEXTUR-LESESCHLANGE
380
391
GRUNDELEMENTE-REGISTER
BEFEHLS- u. SPEICHER-STEUERUNG
TEXTUR-CACHE-STEUERUNG
396
398
SCAN-UMWANDLUNGS-ENGINE
404
COMP. CACHE
DEKOMPRESSIONS-ENGINE
TEXTUR-CACHE
TEXTUR-FILTER-ENGINE
416
402
403
412
FRAGMENT-PUFFER
PIXEL ENGINE
ANTI-ALIASING ENGINE
410
406
PIXEL-PUFFER
408
KOMPRESSIONS-ENGINE
414
ZU DSP, GSPRITE-ENGINE, RAMBUS-KANÄLEN

FIG. 10

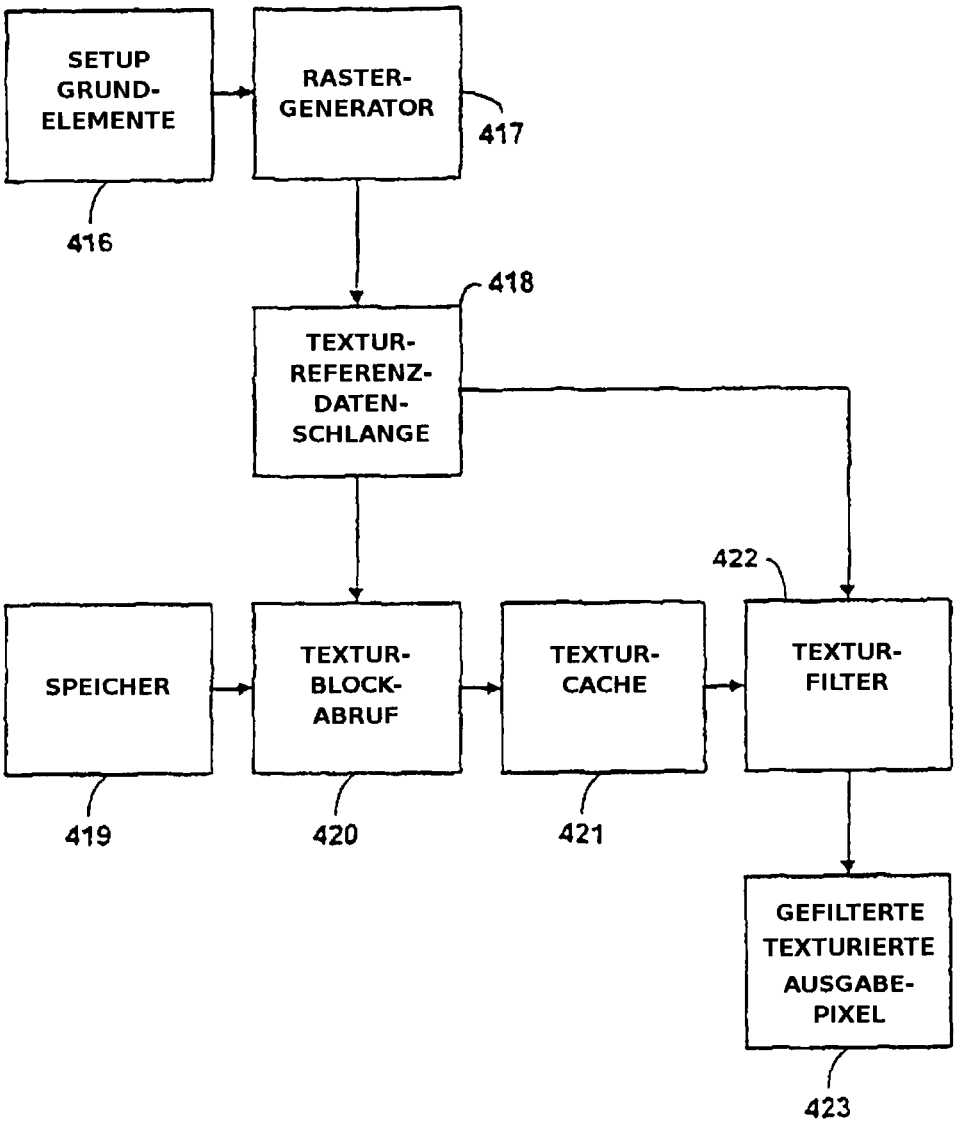

SETUP GRUND-ELEMENTE
RASTER-GENERATOR
417
416
418
TEXTUR-REFERENZ-DATEN-SCHLANGE
422
SPEICHER
TEXTUR-BLOCK-ABRUF
TEXTUR-CACHE
TEXTUR-FILTER
419
420
421
GEFILTERTE TEXTURIERTE AUSGABE-PIXEL
423

FIG. 11

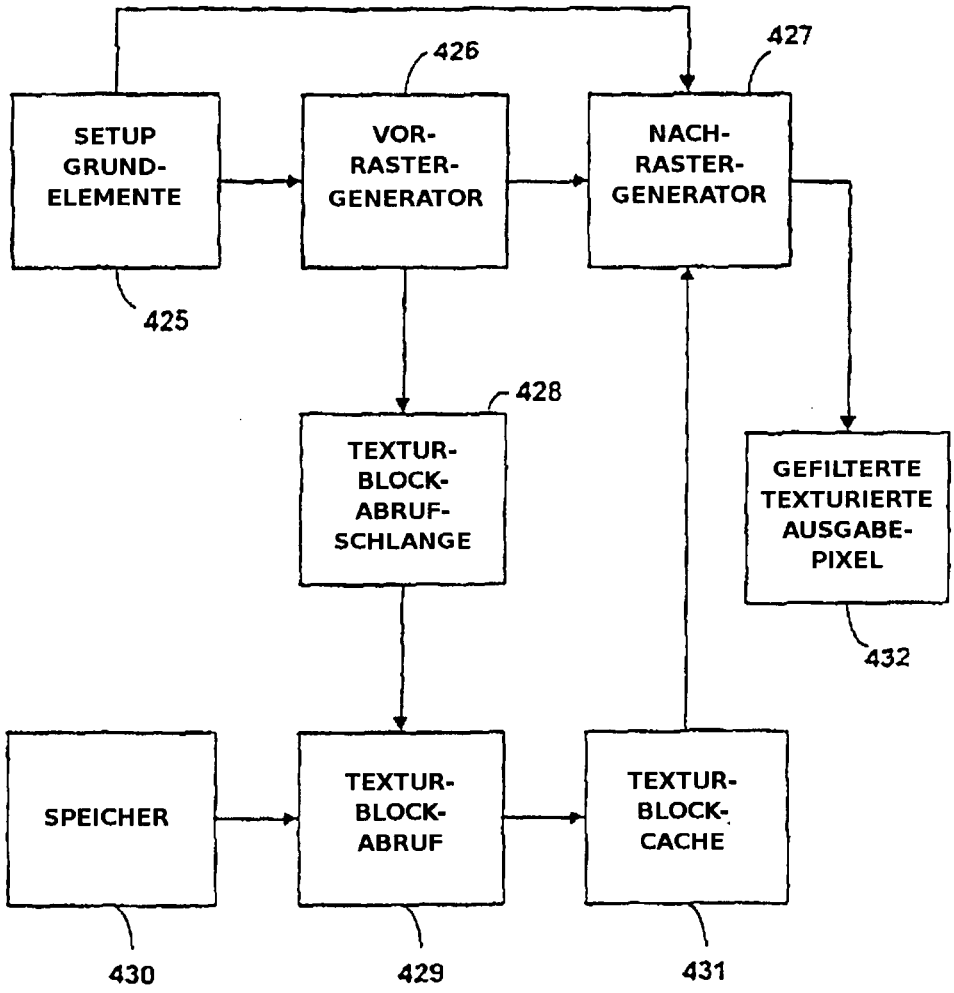

SETUP GRUND-ELEMENTE
425
VOR-RASTER-GENERATOR
426
NACH-RASTER-GENERATOR
427
TEXTUR-BLOCK-ABRUF-SCHLANGE
428
GEFILTERTE TEXTURIERTE AUSGABE-PIXEL
432
SPEICHER
430
TEXTUR-BLOCK-ABRUF
429
TEXTUR-BLOCK-CACHE
431

FIG. 12A

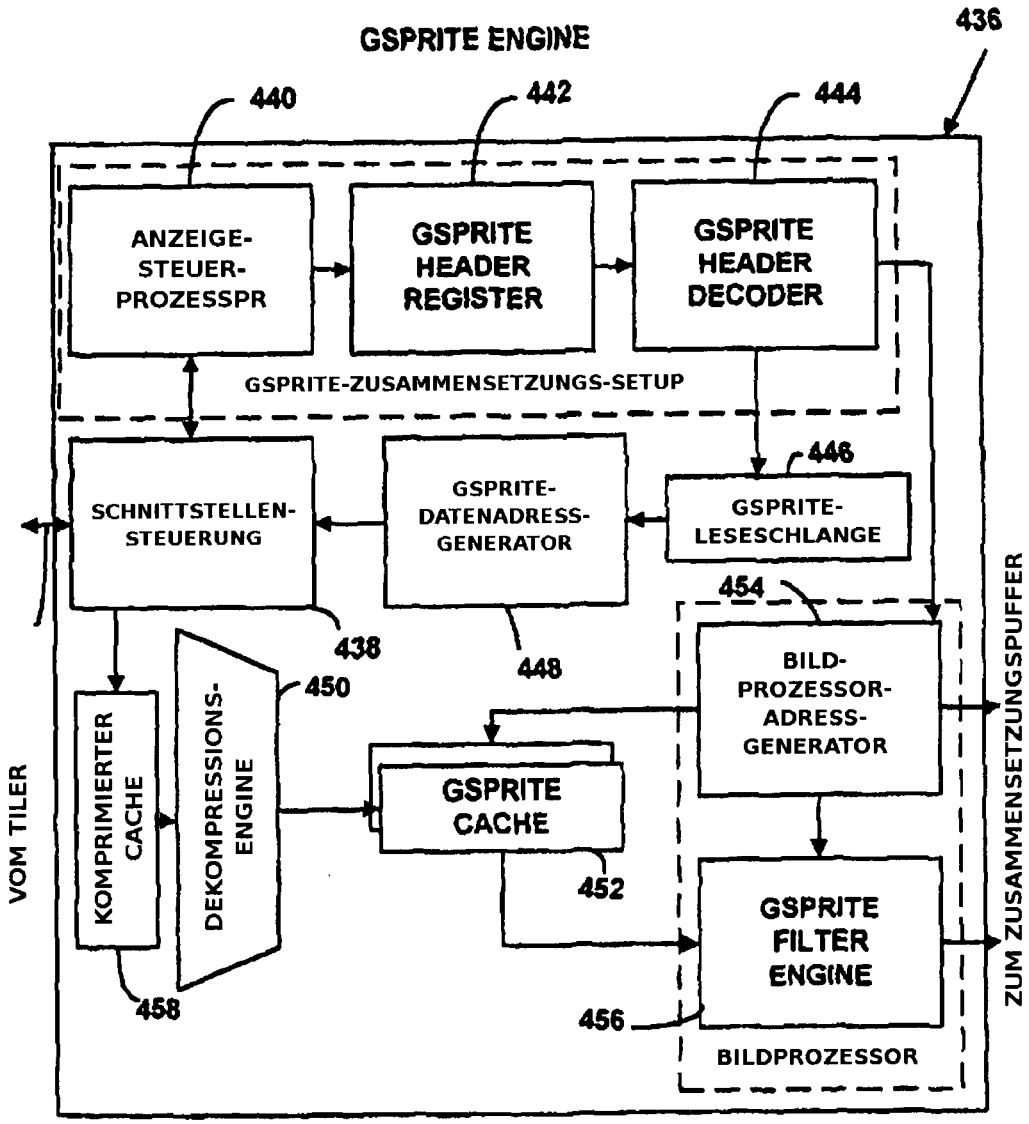

GSPRITE ENGINE
436
440
442
444
ANZEIGE-STEUER-PROZESSPR
GSPRITE HEADER REGISTER
GSPRITE HEADER DECODER
GSPRITE-ZUSAMMENSETZUNGS-SETUP
446
SCHNITTSTELLEN-STEUERUNG
GSPRITE-DATENADRESS-GENERATOR
GSPRITE-LESESCHLANGE
454
438
448
BILD-PROZESSOR-ADRESS-GENERATOR
450
KOMPRIMIERTER CACHE
DEKOMPRESSIONS-ENGINE
GSPRITE CACHE
452
GSPRITE FILTER ENGINE
458
456
BILDPROZESSOR
VOM TILER
ZUM ZUSAMMENSETZUNGSPUFFER

FIG. 12B

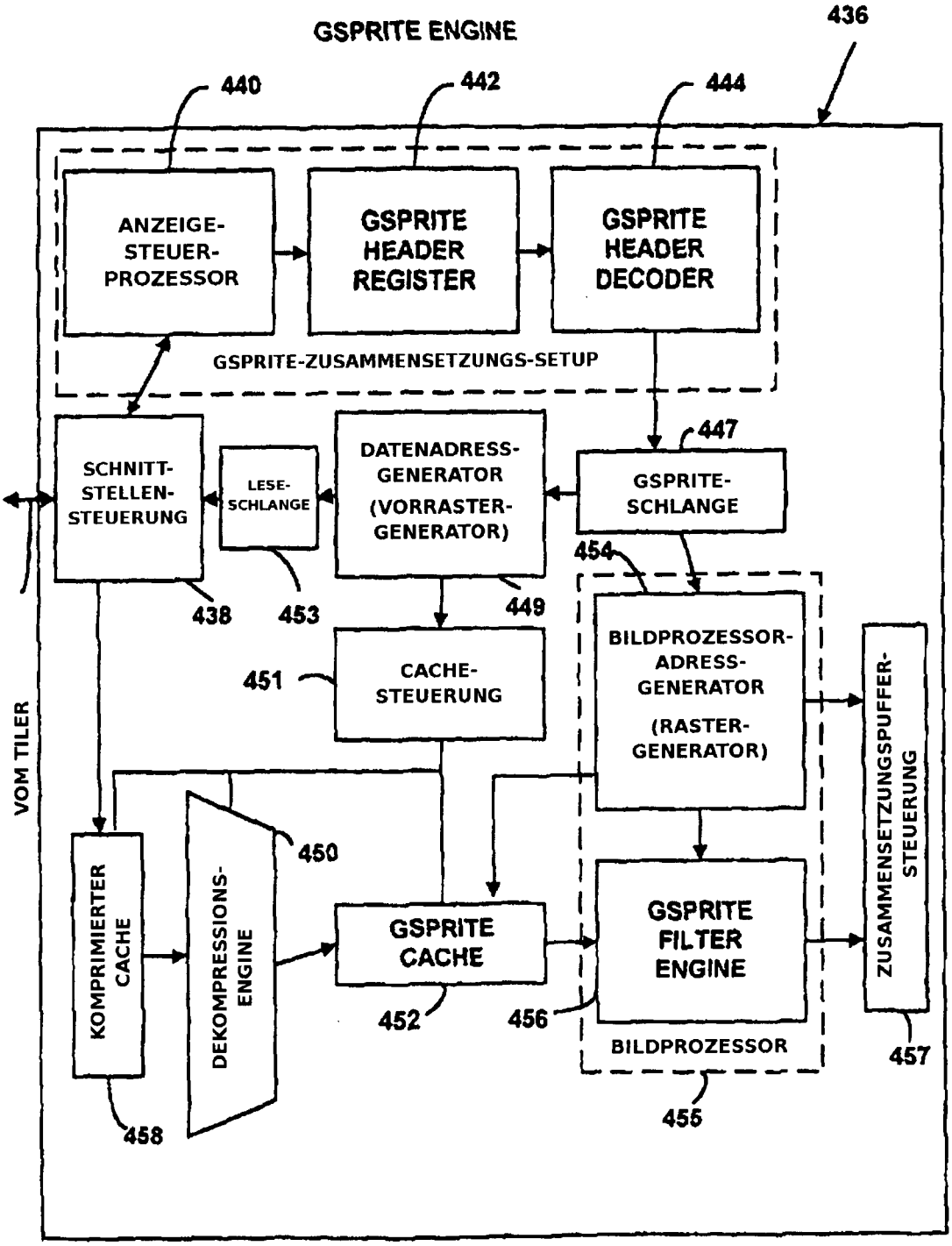

GSPRITE ENGINE
436
440
442
444
ANZEIGE-STEUER-PROZESSOR
GSPRITE HEADER REGISTER
GSPRITE HEADER DECODER
GSPRITE-ZUSAMMENSETZUNGS-SETUP
447
SCHNITT-STELLEN-STEUERUNG
LESE-SCHLANGE
DATENADRESS-GENERATOR (VORRASTER-GENERATOR)
GSPRITE-SCHLANGE
454
438
453
449
451
CACHE-STEUERUNG
BILDPROZESSOR-ADRESS-GENERATOR (RASTER-GENERATOR)
ZUSAMMENSETZUNGSPUFFER-STEUERUNG
VOM TILER
450
KOMPRIMIERTER CACHE
DEKOMPRESSIONS-ENGINE
GSPRITE CACHE
GSPRITE FILTER ENGINE
452
456
457
BILDPROZESSOR
458
455

FIG. 13

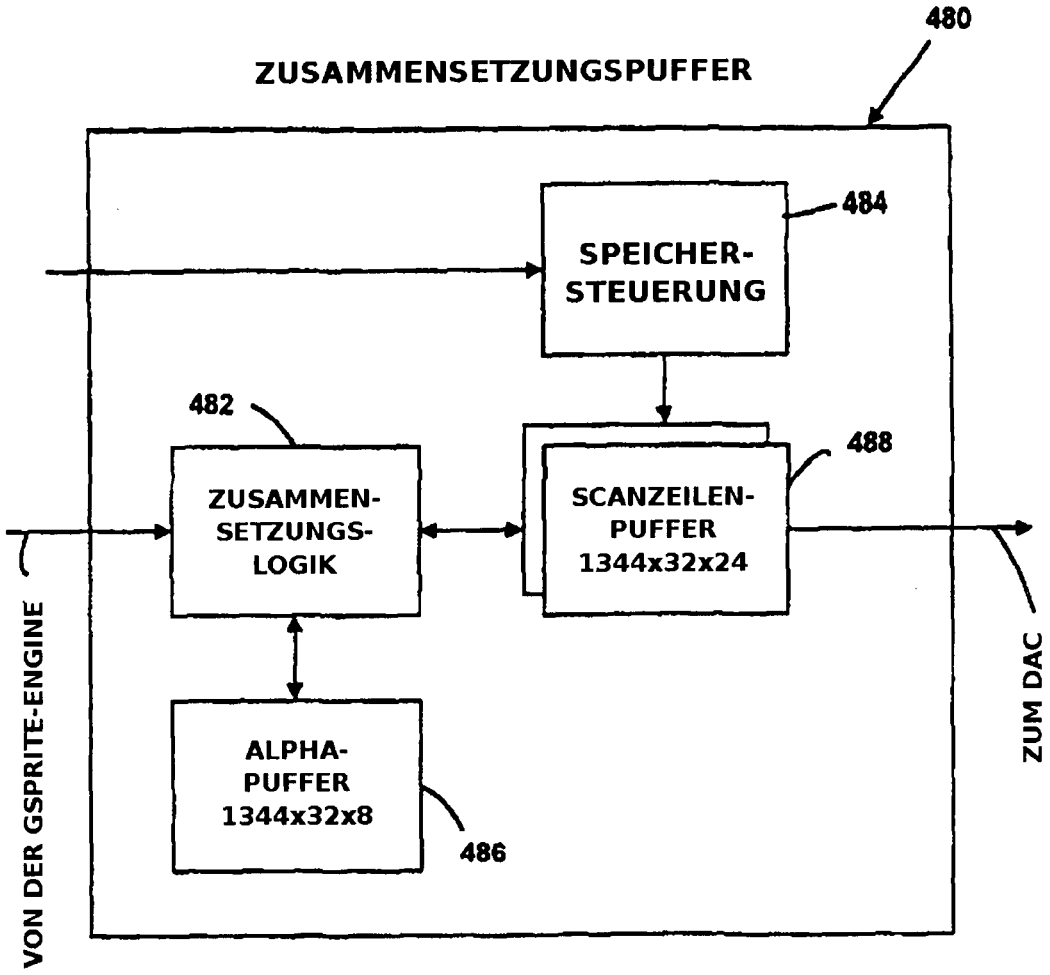

480
ZUSAMMENSETZUNGSPUFFER
484
SPEICHER-
STEUERUNG
482
ZUSAMMEN-
SETZUNGS-
LOGIK
488
SCANZEILEN-
PUFFER
1344x32x24
ALPHA-
PUFFER
1344x32x8
486
VON DER GSPRITE-ENGINE
ZUM DAC

FIG. 14

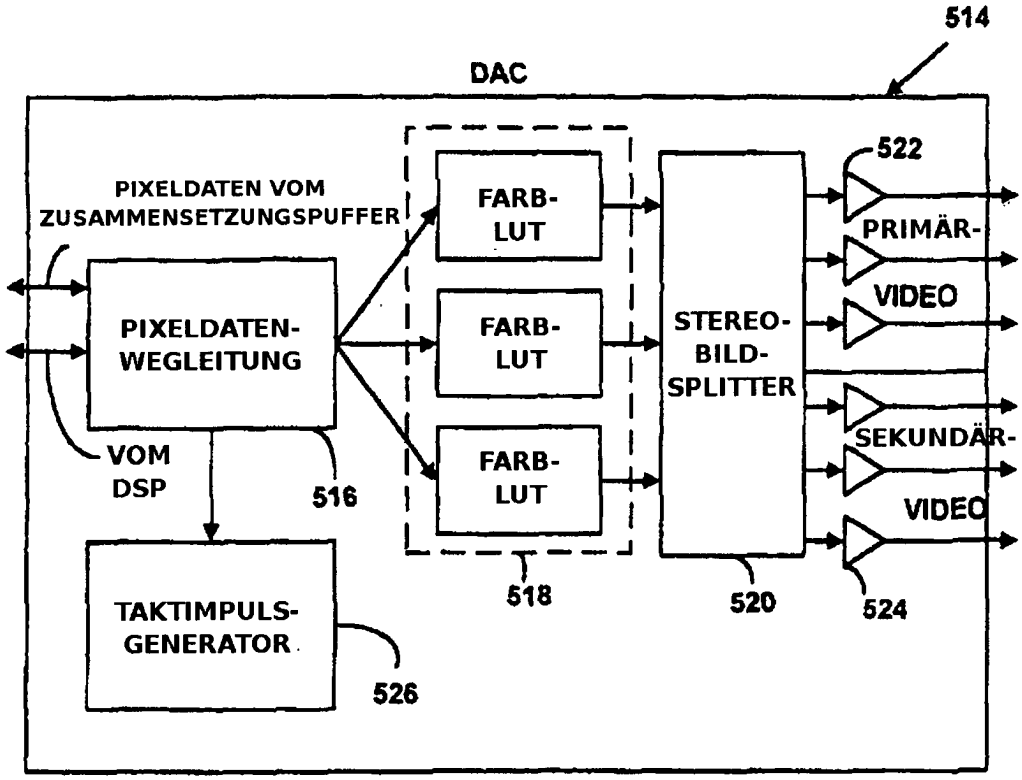

514
DAC
PIXELDATEN VOM ZUSAMMENSETZUNGSPUFFER
PIXELDATEN-WEGLEITUNG
VOM DSP
516
TAKTIMPULS-GENERATOR
526
FARB-LUT
FARB-LUT
FARB-LUT
518
STEREO-BILD-SPLITTER
520
522
PRIMÄR-VIDEO
SEKUNDÄR-VIDEO
524

FIG. 15A

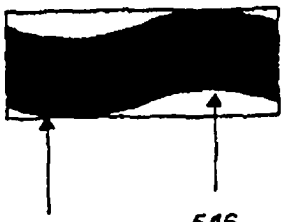


FIG. 15B

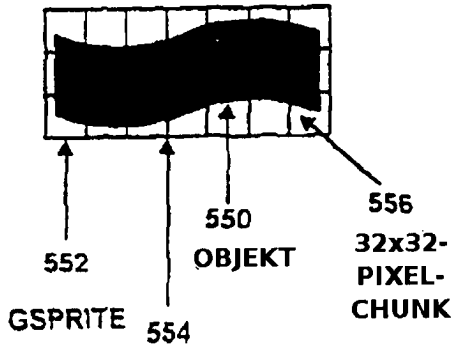


FIG. 15C

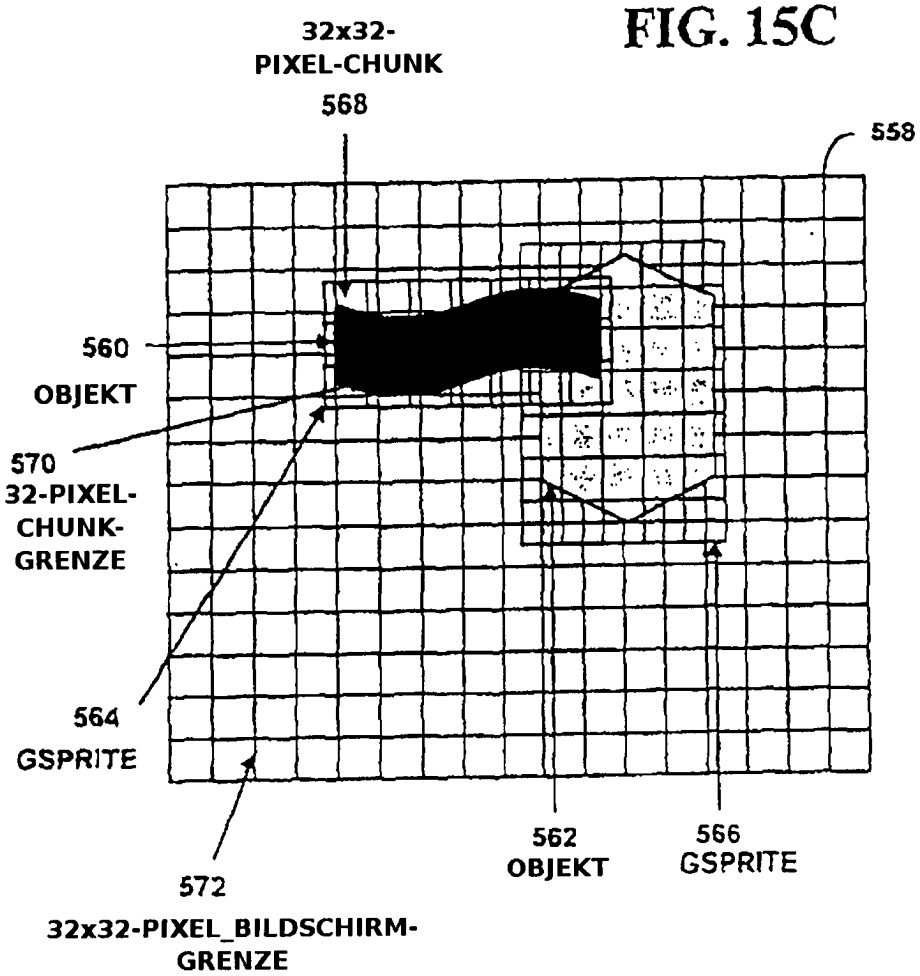


FIG. 16A

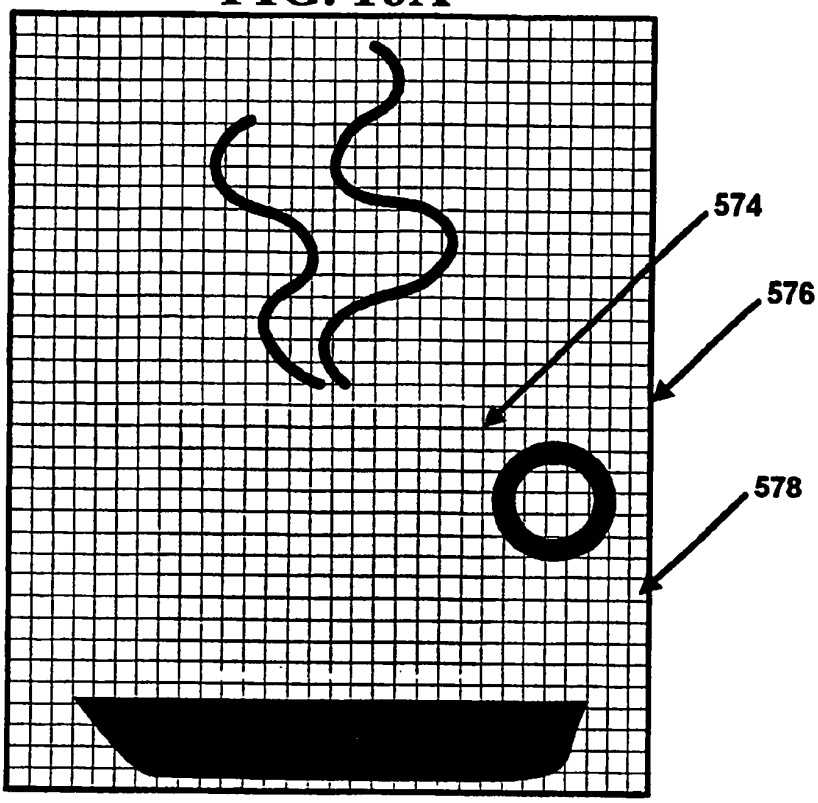
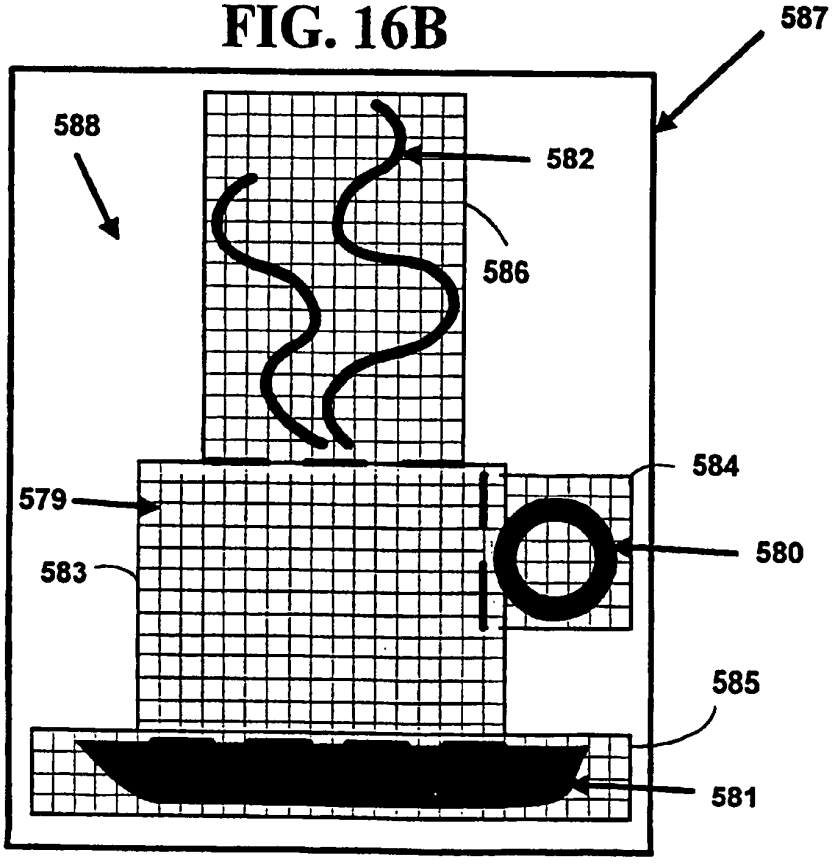


FIG. 16B



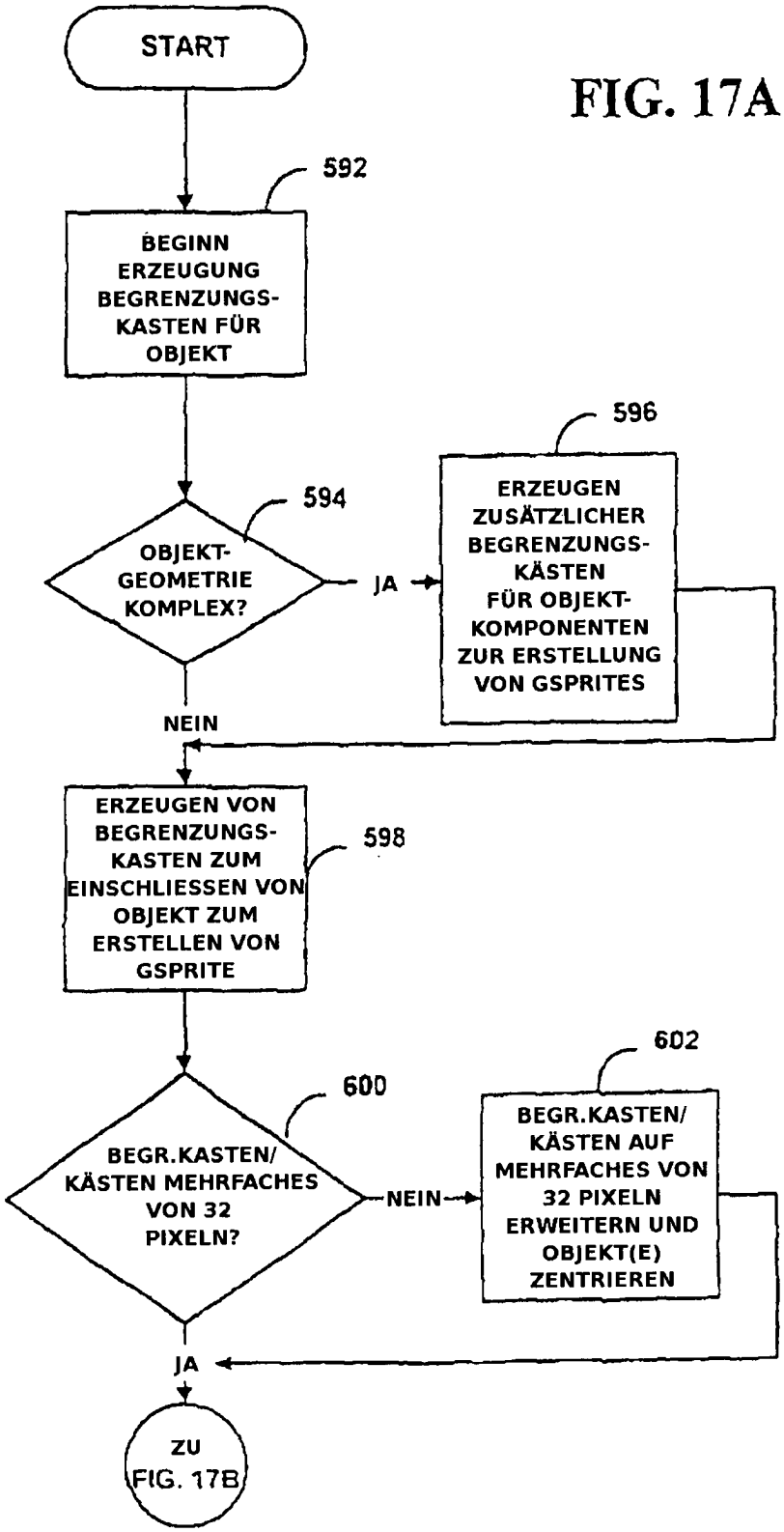

START
592
BEGINN ERZEUGUNG BEGRENZUNGS-KASTEN FÜR OBJEKT
596
ERZEUGEN ZUSÄTZLICHER BEGRENZUNGS-KÄSTEN FÜR OBJEKT-KOMPONENTEN ZUR ERSTELLUNG VON GSPRITES
594
OBJEKT-GEOMETRIE KOMPLEX?
JA
NEIN
ERZEUGEN VON BEGRENZUNGS-KASTEN ZUM EINSCHLIESSEN VON OBJEKT ZUM ERSTELLEN VON GSPRITE
598
602
600
BEGR.KASTEN/ KÄSTEN MEHRFACHES VON 32 PIXELN?
NEIN
BEGR.KASTEN/ KÄSTEN AUF MEHRFACHES VON 32 PIXELN ERWEITERN UND OBJEKT(E) ZENTRIEREN
JA
ZU FIG. 17B

FIG. 17A

FIG. 17B

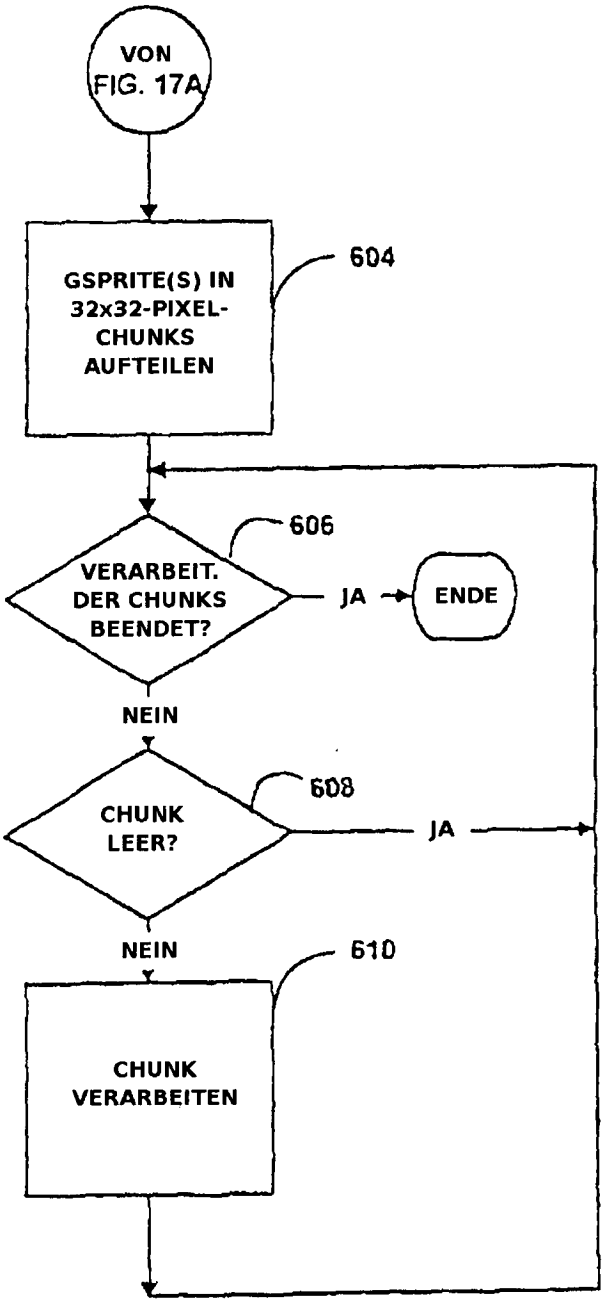

VON FIG. 17A
GSPRITE(S) IN 32x32-PIXEL-CHUNKS AUFTEILEN
604
606
VERARBEIT. DER CHUNKS BEENDET?
JA
ENDE
NEIN
608
CHUNK LEER?
JA
NEIN
610
CHUNK VERARBEITEN

FIG. 18A

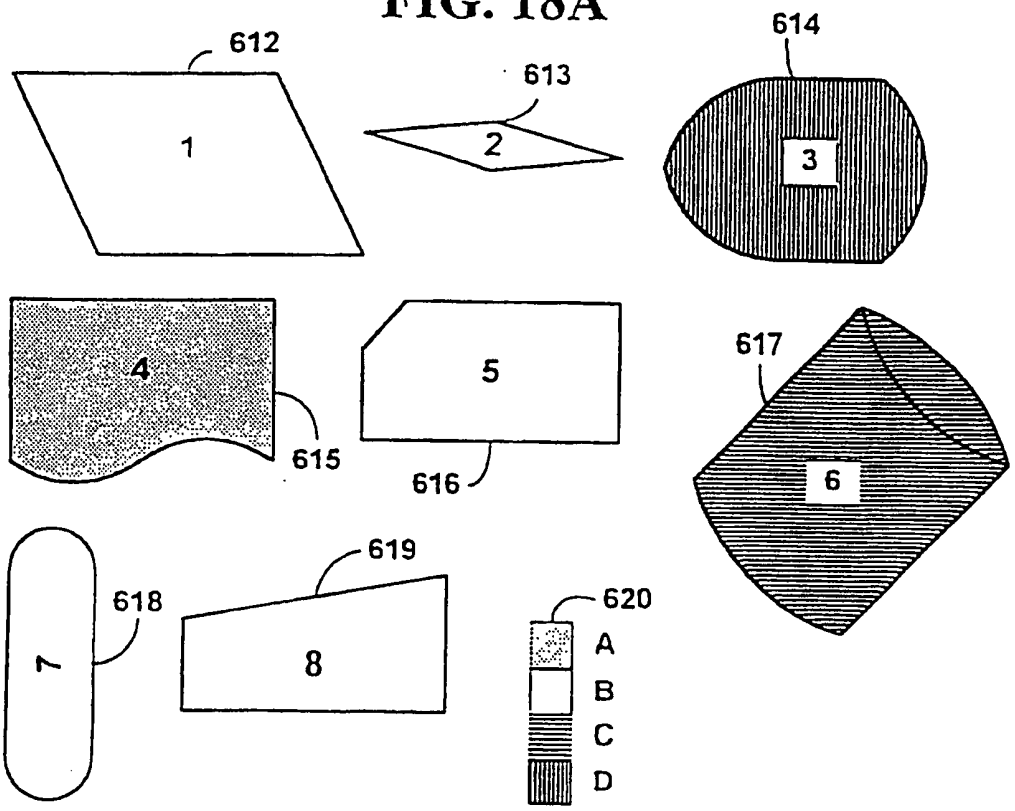


FIG. 18B

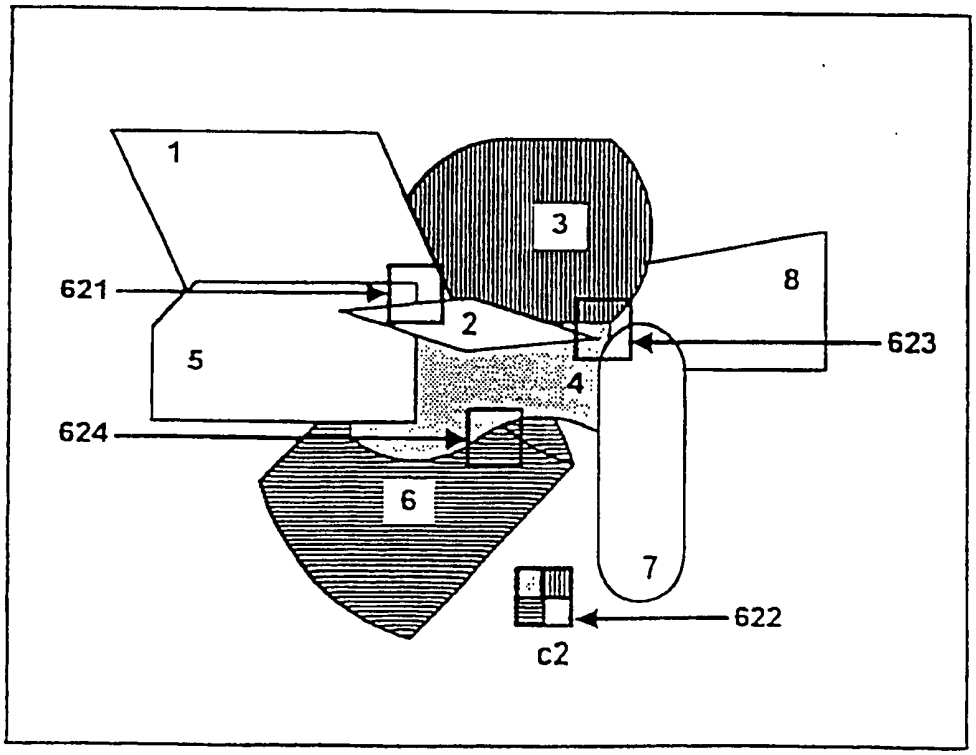


FIG. 19A

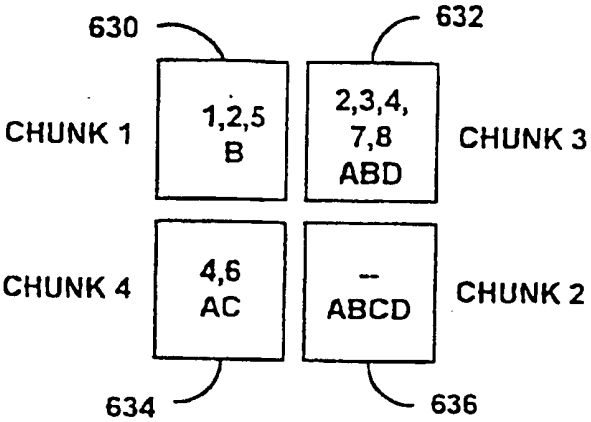


FIG. 19B

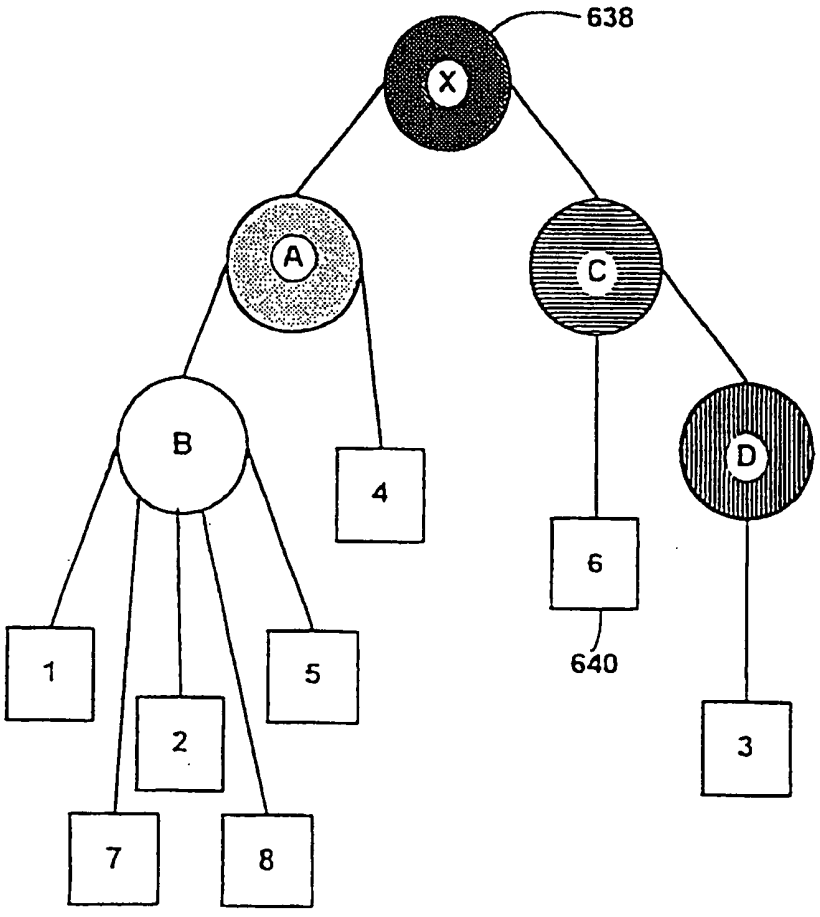


FIG. 20

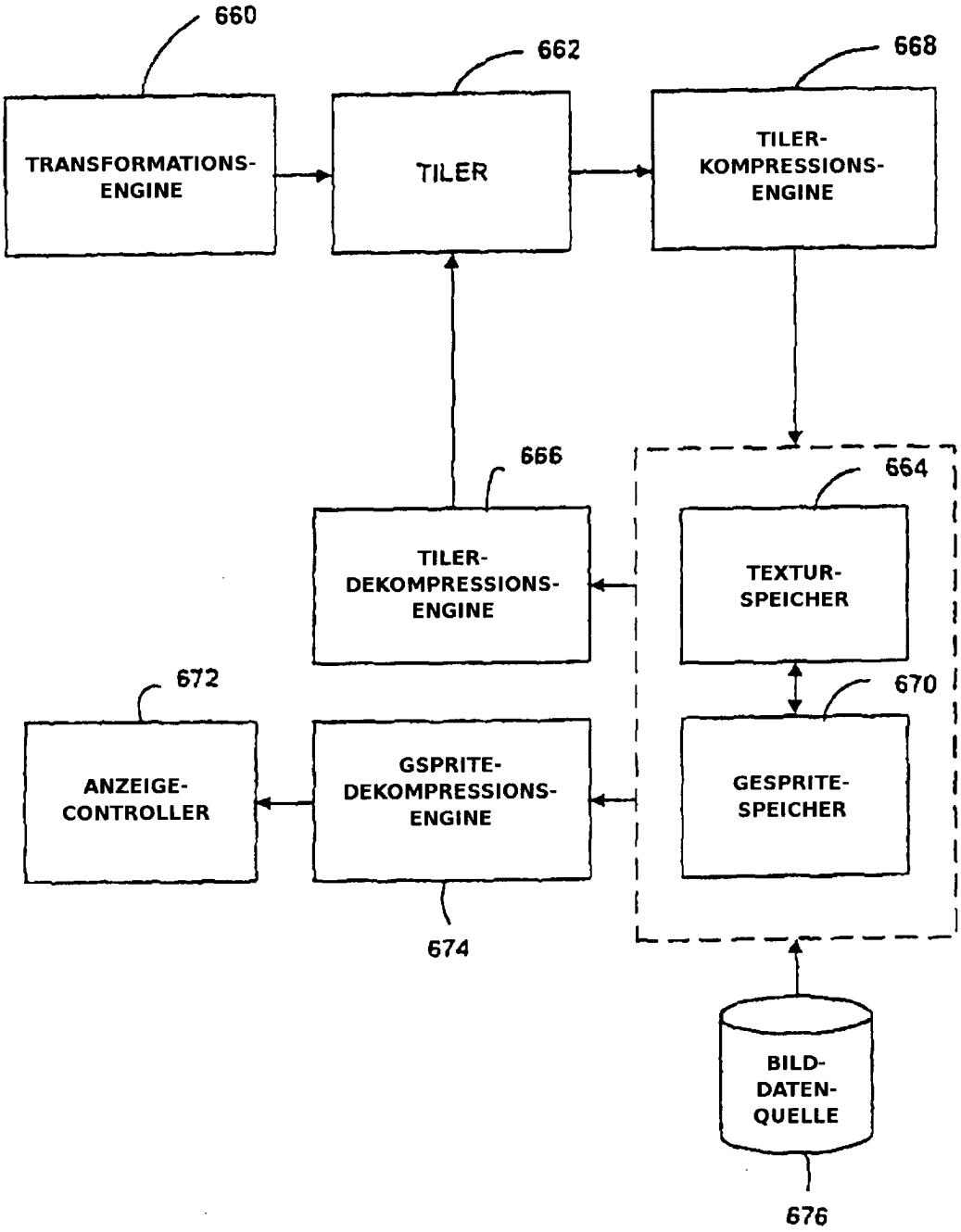

660
662
668
TRANSFORMATIONS-ENGINE
TILER
TILER-KOMPRESSIONS-ENGINE
666
664
TILER-DEKOMPRESSIONS-ENGINE
TEXTUR-SPEICHER
672
670
ANZEIGE-CONTROLLER
GSPRITE-DEKOMPRESSIONS-ENGINE
GESPRITE-SPEICHER
674
BILD-DATEN-QUELLE
676

FIG. 21A

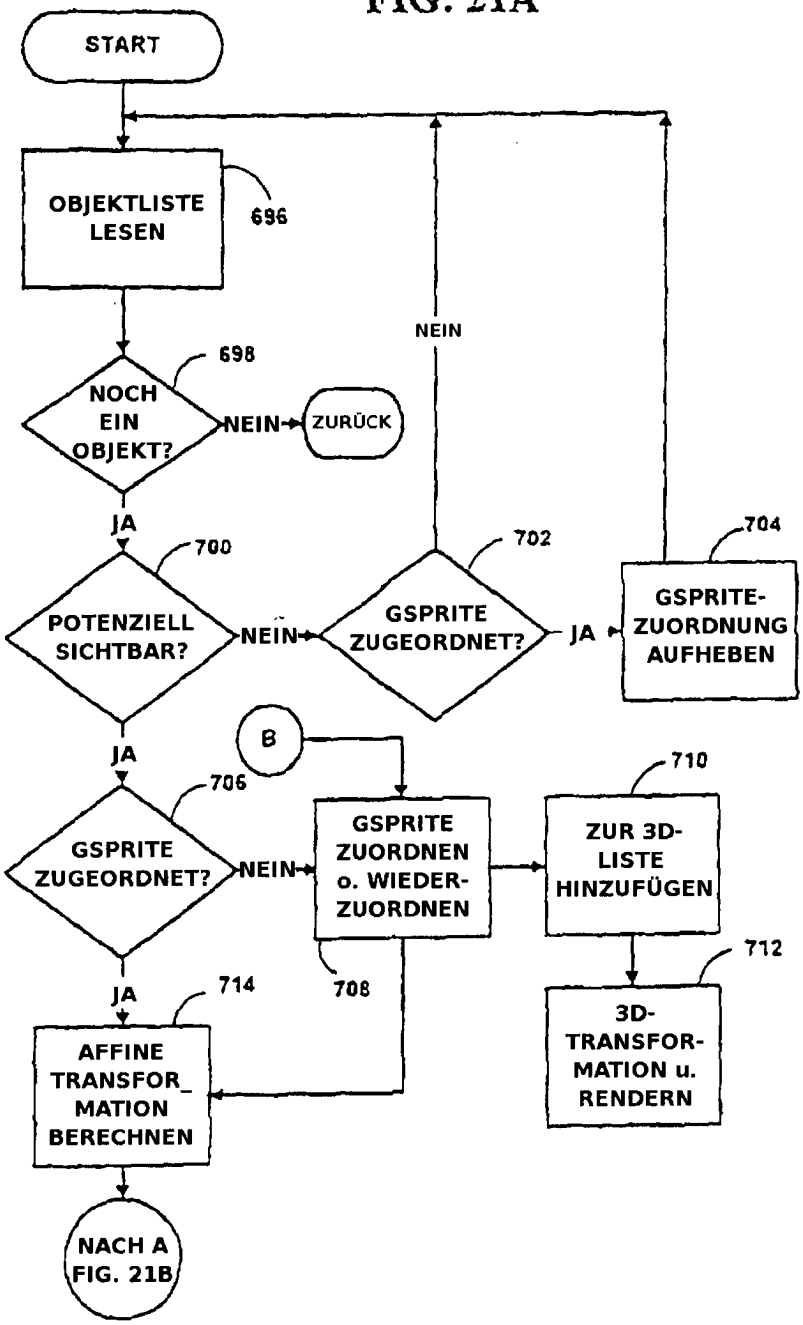

START
OBJEKTLISTE LESEN
696
NOCH EIN OBJEKT?
698
NEIN
ZURÜCK
JA
POTENZIELL SICHTBAR?
700
NEIN
GSPRITE ZUGEORDNET?
702
JA
GSPRITE-ZUORDNUNG AUFHEBEN
704
NEIN
JA
B
GSPRITE ZUGEORDNET?
706
NEIN
GSPRITE ZUORDNEN o. WIEDER-ZUORDNEN
708
ZUR 3D-LISTE HINZUFÜGEN
710
3D-TRANSFOR-MATION u. RENDERN
712
JA
AFFINE TRANSFOR_ MATION BERECHNEN
714
NACH A FIG. 21B

FIG. 21B

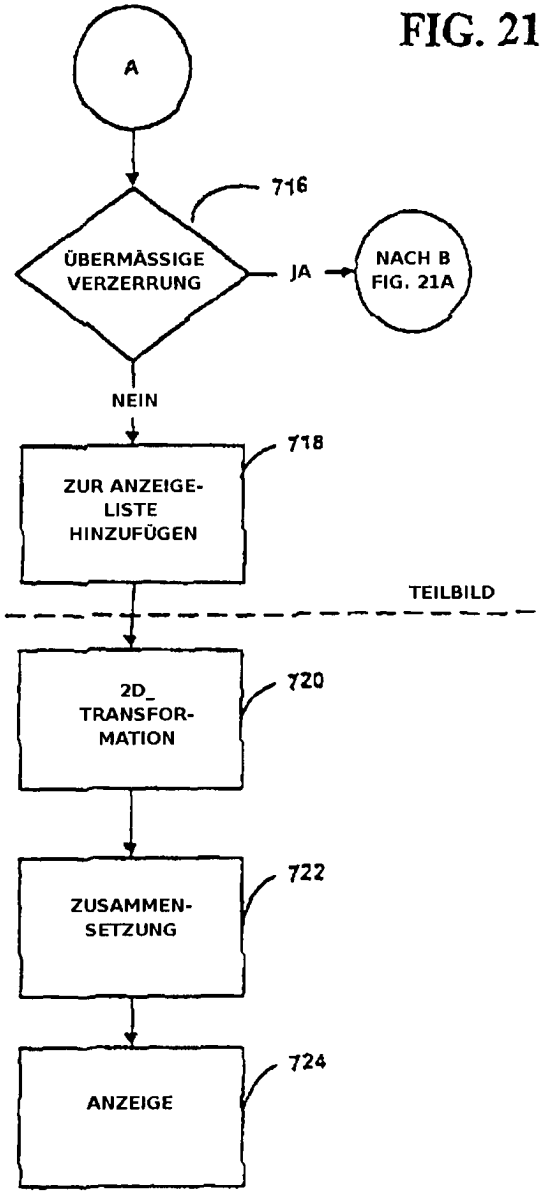

A
716
ÜBERMÄSSIGE VERZERRUNG
JA
NACH B FIG. 21A
NEIN
718
ZUR ANZEIGE-LISTE HINZUFÜGEN
TEILBILD
720
2D_ TRANSFOR-MATION
722
ZUSAMMEN-SETZUNG
724
ANZEIGE

FIG. 22

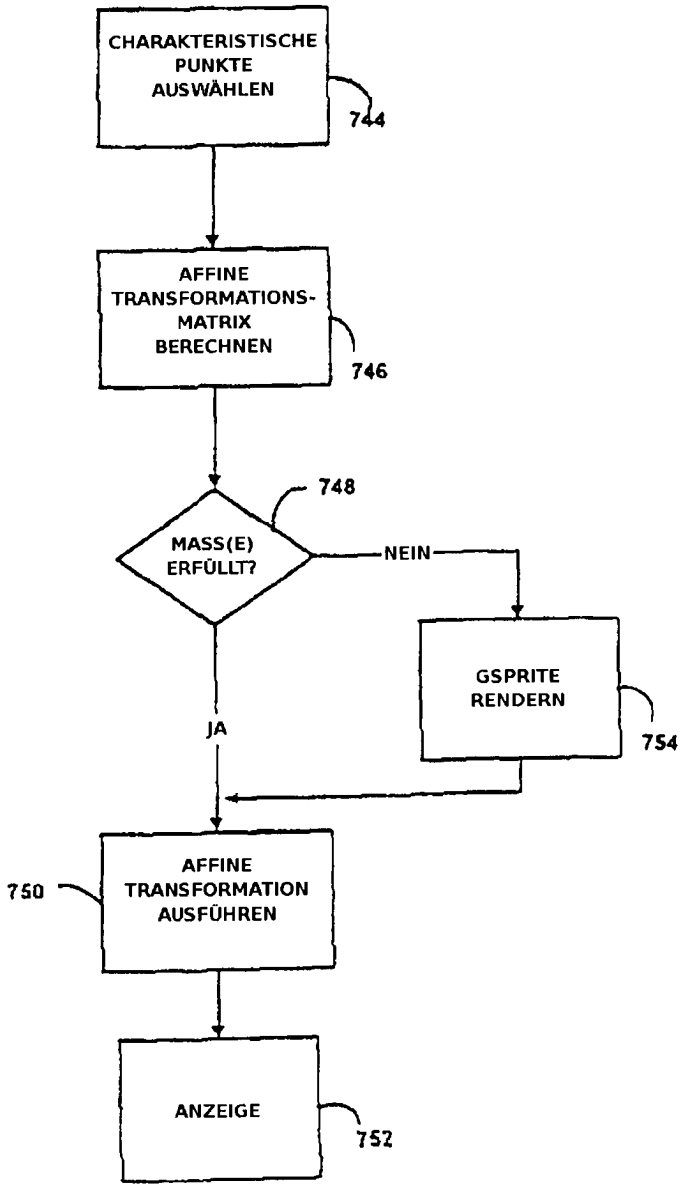

CHARAKTERISTISCHE PUNKTE AUSWÄHLEN
744
AFFINE TRANSFORMATIONS-MATRIX BERECHNEN
746
748
MASS(E) ERFÜLLT?
NEIN
GSPRITE RENDERN
754
JA
750
AFFINE TRANSFORMATION AUSFÜHREN
ANZEIGE
752

FIG. 23

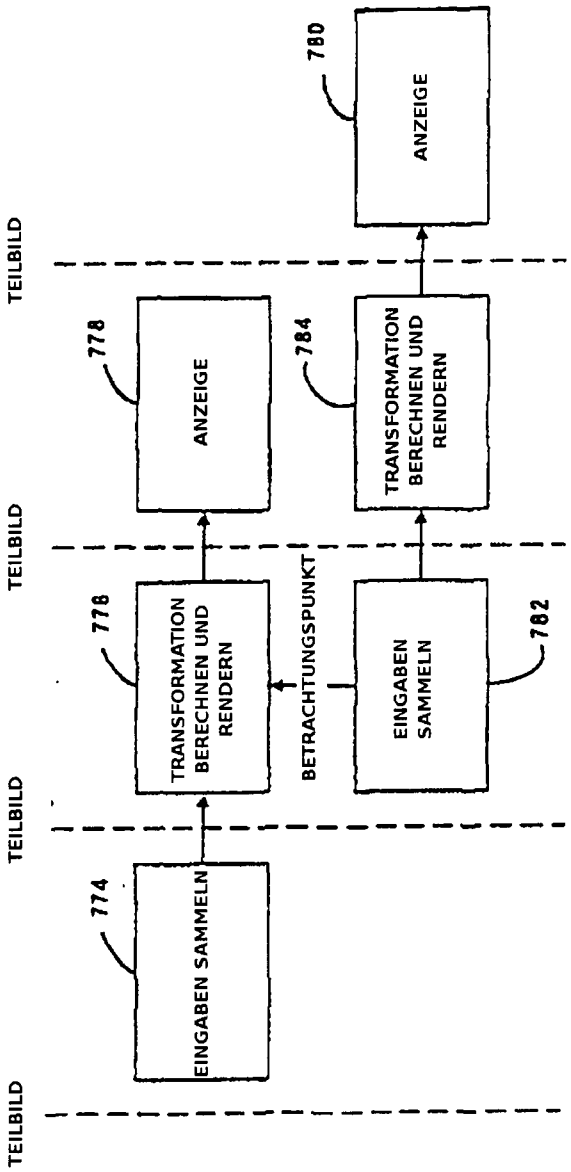

780
ANZEIGE
TEILBILD
778
ANZEIGE
784
TRANSFORMATION BERECHNEN UND RENDERN
TEILBILD
776
TRANSFORMATION BERECHNEN UND RENDERN
BETRACHTUNGSPUNKT
EINGABEN SAMMELN
782
TEILBILD
774
EINGABEN SAMMELN
TEILBILD

FIG. 24

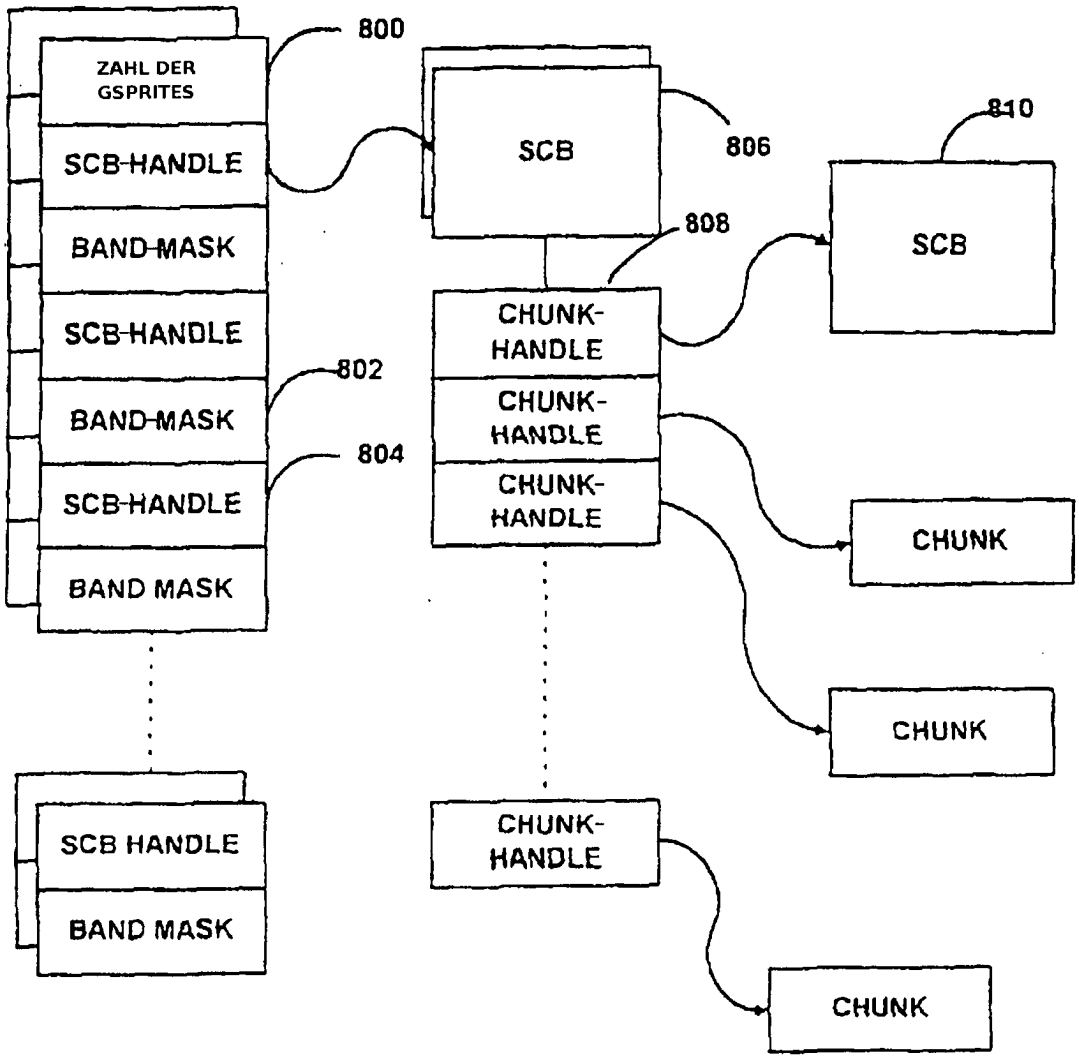

800
ZAHL DER GSPRITES
SCB-HANDLE
BAND-MASK
SCB-HANDLE
BAND-MASK
SCB-HANDLE
BAND MASK
802
804
SCB HANDLE
BAND MASK
SCB
806
808
CHUNK-HANDLE
CHUNK-HANDLE
CHUNK-HANDLE
CHUNK-HANDLE
810
SCB
CHUNK
CHUNK
CHUNK
CHUNK

FIG. 25

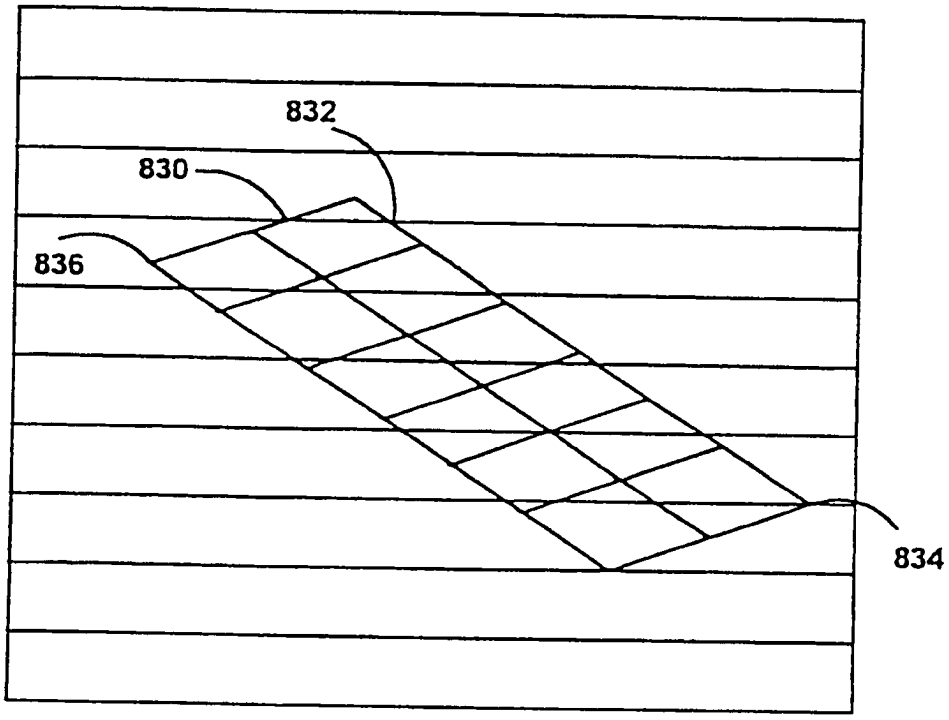


FIG. 26

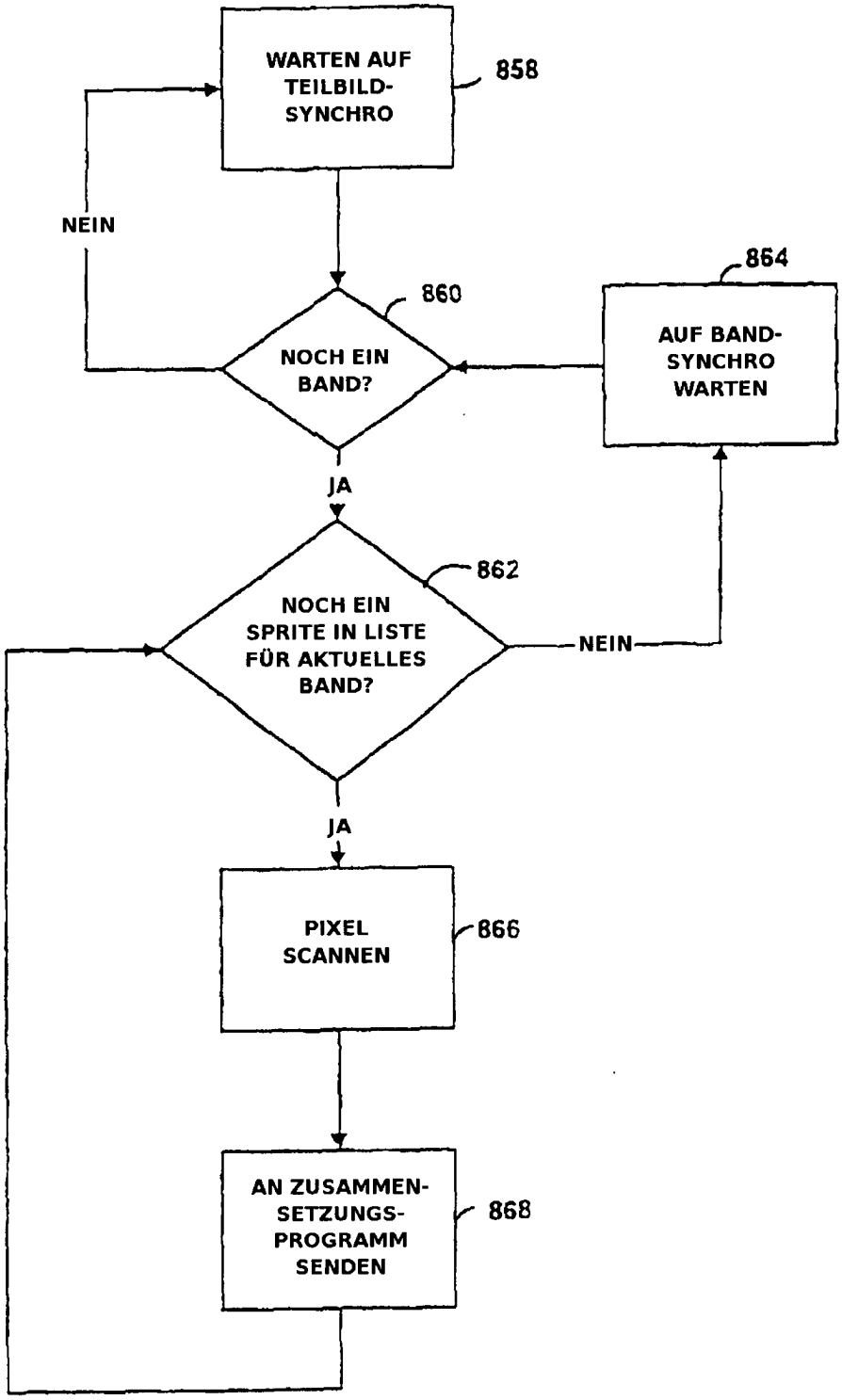

WARTEN AUF TEILBILD-SYNCHRO
858
NEIN
860
NOCH EIN BAND?
864
AUF BAND-SYNCHRO WARTEN
JA
862
NOCH EIN SPRITE IN LISTE FÜR AKTUELLES BAND?
NEIN
JA
PIXEL SCANNEN
866
AN ZUSAMMEN-SETZUNGS-PROGRAMM SENDEN
868

FIG. 27

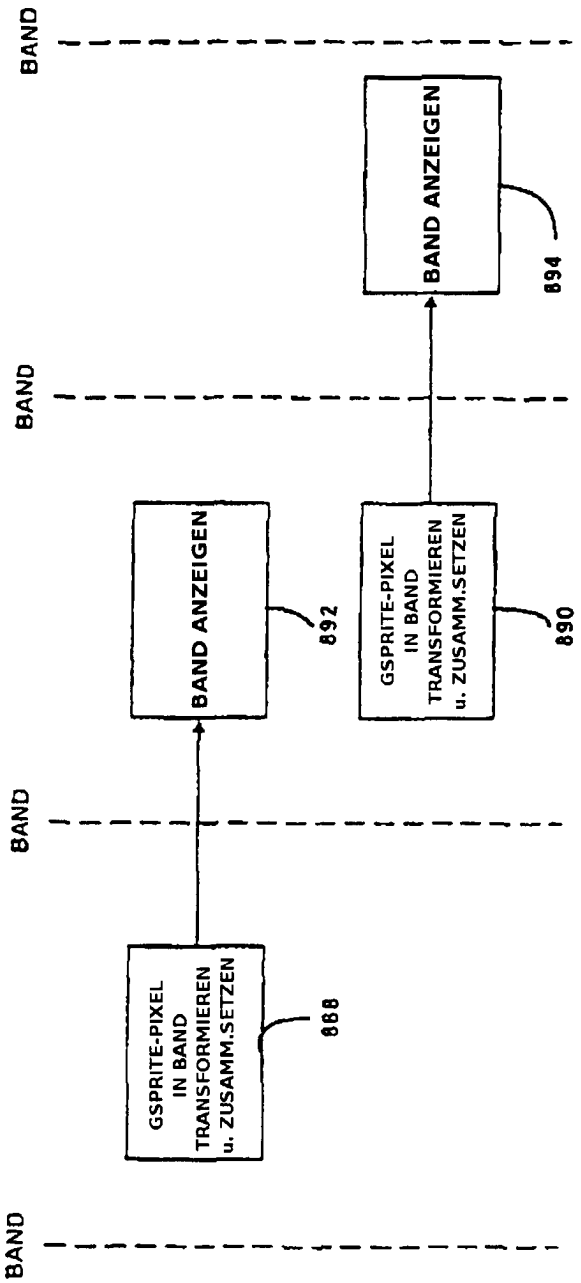

BAND
BAND
BAND
BAND
GSPRITE-PIXEL IN BAND TRANSFORMIEREN u. ZUSAMM.SETZEN
888
BAND ANZEIGEN
892
GSPRITE-PIXEL IN BAND TRANSFORMIEREN u. ZUSAMM.SETZEN
890
BAND ANZEIGEN
894

FIG. 28A

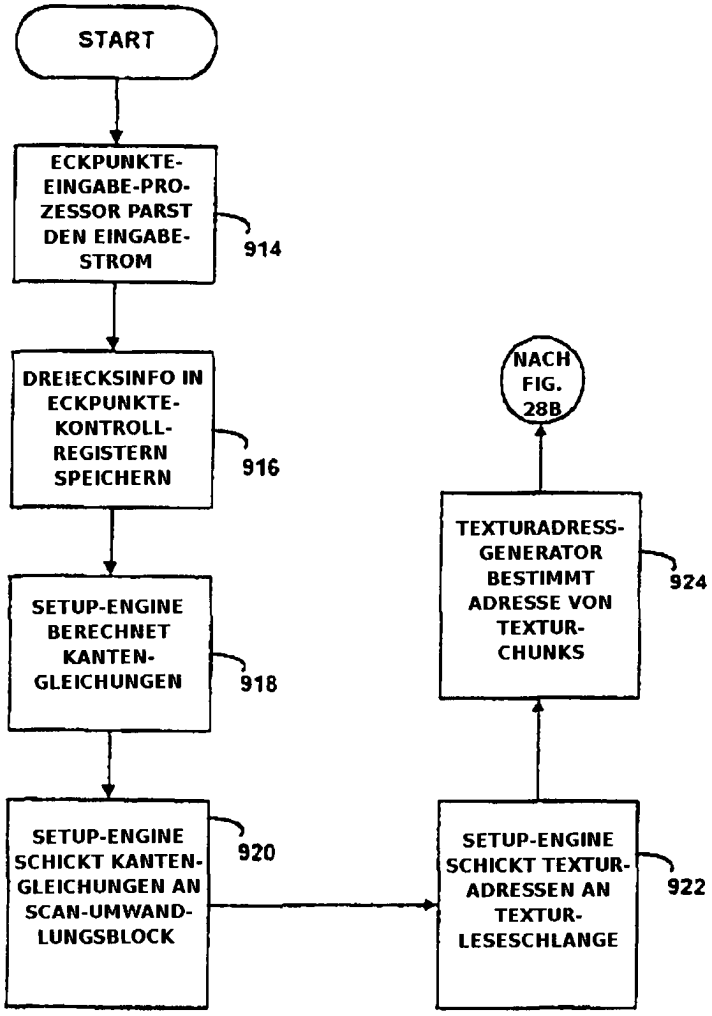

START
ECKPUNKTE-EINGABE-PROZESSOR PARST DEN EINGABE-STROM
914
DREIECKSINFO IN ECKPUNKTE-KONTROLL-REGISTERN SPEICHERN
916
SETUP-ENGINE BERECHNET KANTEN-GLEICHUNGEN
918
SETUP-ENGINE SCHICKT KANTEN-GLEICHUNGEN AN SCAN-UMWAND-LUNGSBLOCK
920
SETUP-ENGINE SCHICKT TEXTUR-ADRESSEN AN TEXTUR-LESESCHLANGE
922
TEXTURADRESS-GENERATOR BESTIMMT ADRESSE VON TEXTUR-CHUNKS
924
NACH FIG. 28B

FIG. 28B

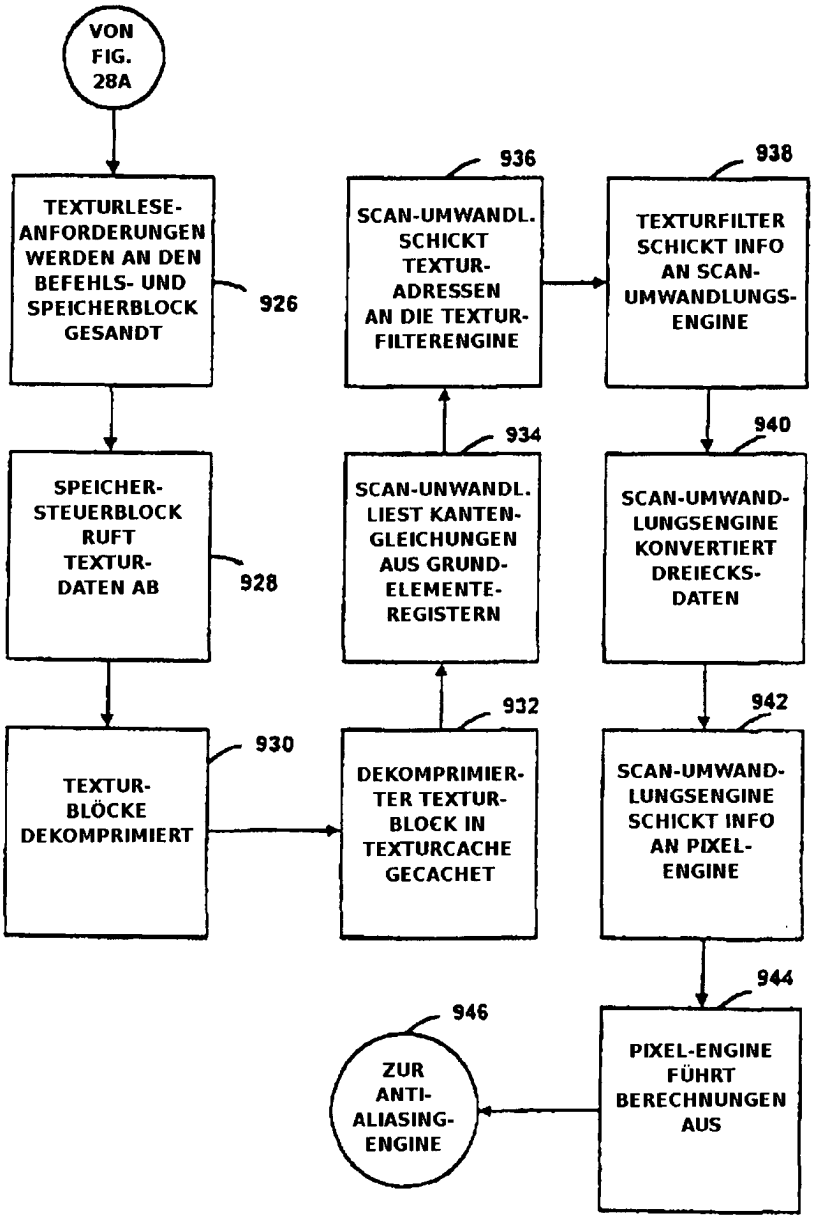

VON FIG. 28A
TEXTURLESE-ANFORDERUNGEN WERDEN AN DEN BEFEHLS- UND SPEICHERBLOCK GESANDT
926
SPEICHER-STEUERBLOCK RUFT TEXTUR-DATEN AB
928
TEXTUR-BLÖCKE DEKOMPRIMIERT
930
DEKOMPRIMIER-TER TEXTUR-BLOCK IN TEXTURCACHE GECACHET
932
SCAN-UNWANDL. LIEST KANTEN-GLEICHUNGEN AUS GRUND-ELEMENTE-REGISTERN
934
SCAN-UMWANDL. SCHICKT TEXTUR-ADRESSEN AN DIE TEXTUR-FILTERENGINE
936
TEXTURFILTER SCHICKT INFO AN SCAN-UMWANDLUNGS-ENGINE
938
SCAN-UMWAND-LUNGSENGINE KONVERTIERT DREIECKS-DATEN
940
SCAN-UMWAND-LUNGSENGINE SCHICKT INFO AN PIXEL-ENGINE
942
PIXEL-ENGINE FÜHRT BERECHNUNGEN AUS
944
ZUR ANTI-ALIASING-ENGINE
946

FIG. 28C

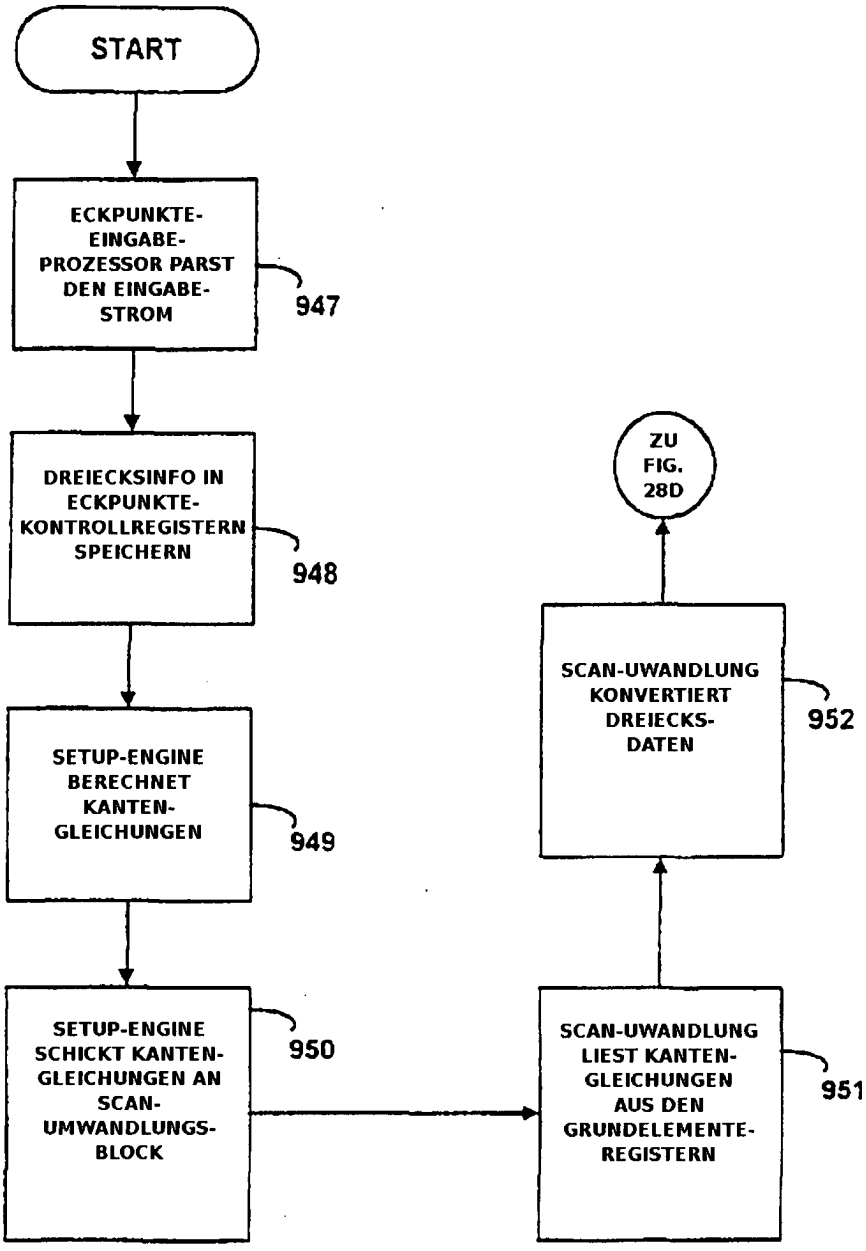

START
ECKPUNKTE-EINGABE-PROZESSOR PARST DEN EINGABE-STROM
947
DREIECKSINFO IN ECKPUNKTE-KONTROLLREGISTERN SPEICHERN
948
SETUP-ENGINE BERECHNET KANTEN-GLEICHUNGEN
949
SETUP-ENGINE SCHICKT KANTEN-GLEICHUNGEN AN SCAN-UMWANDLUNGS-BLOCK
950
SCAN-UWANDLUNG LIEST KANTEN-GLEICHUNGEN AUS DEN GRUNDELEMENTE-REGISTERN
951
SCAN-UWANDLUNG KONVERTIERT DREIECKS-DATEN
952
ZU FIG. 28D

FIG. 28D

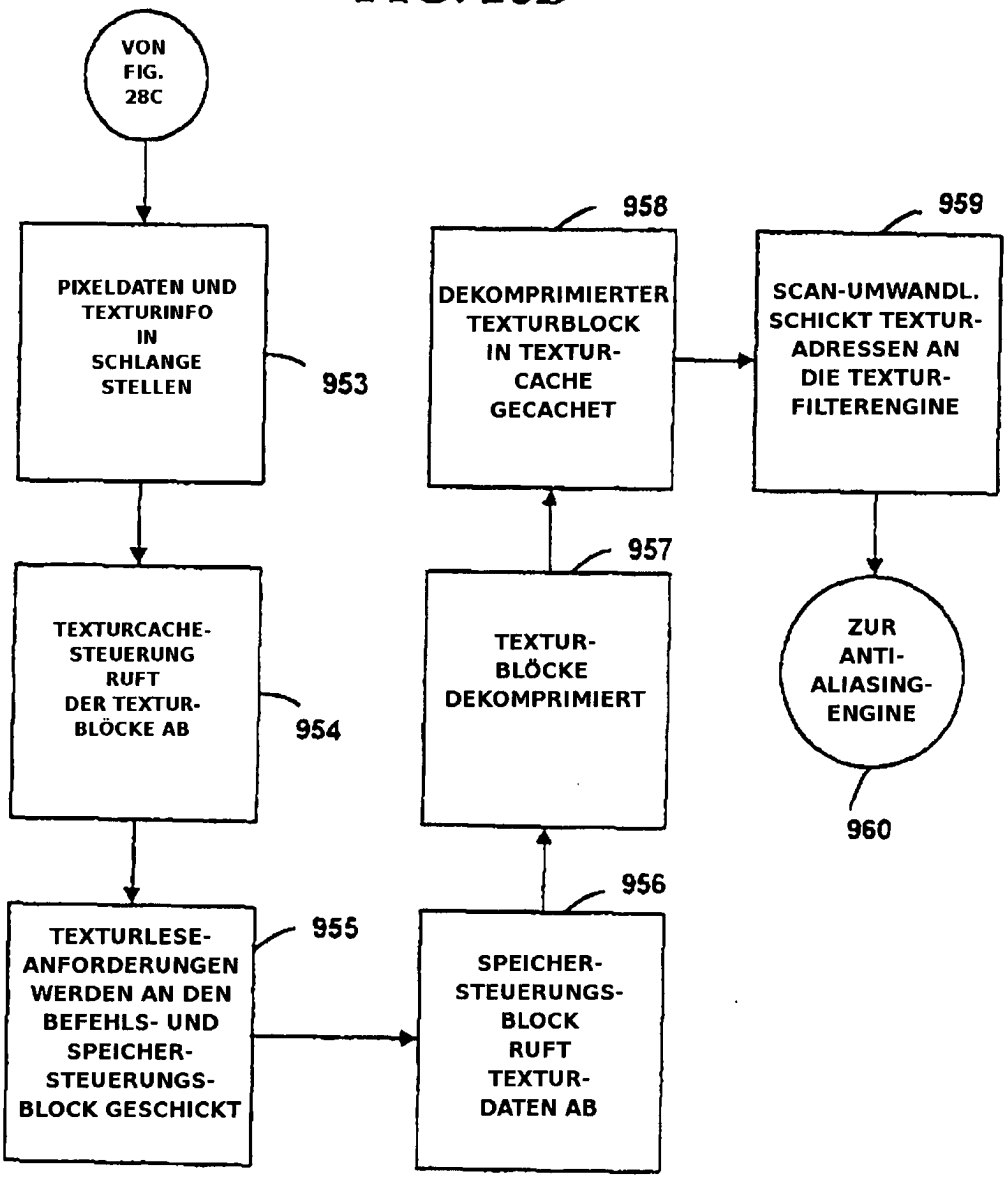

VON FIG. 28C
PIXELDATEN UND TEXTURINFO IN SCHLANGE STELLEN
953
TEXTURCACHE-STEUERUNG RUFT DER TEXTUR-BLÖCKE AB
954
TEXTURLESE-ANFORDERUNGEN WERDEN AN DEN BEFEHLS- UND SPEICHER-STEUERUNGS-BLOCK GESCHICKT
955
SPEICHER-STEUERUNGS-BLOCK RUFT TEXTUR-DATEN AB
956
TEXTUR-BLÖCKE DEKOMPRIMIERT
957
DEKOMPRIMIERTER TEXTURBLOCK IN TEXTUR-CACHE GECACHET
958
SCAN-UMWANDL. SCHICKT TEXTUR-ADRESSEN AN DIE TEXTUR-FILTERENGINE
959
ZUR ANTI-ALIASING-ENGINE
960

FIG. 28E

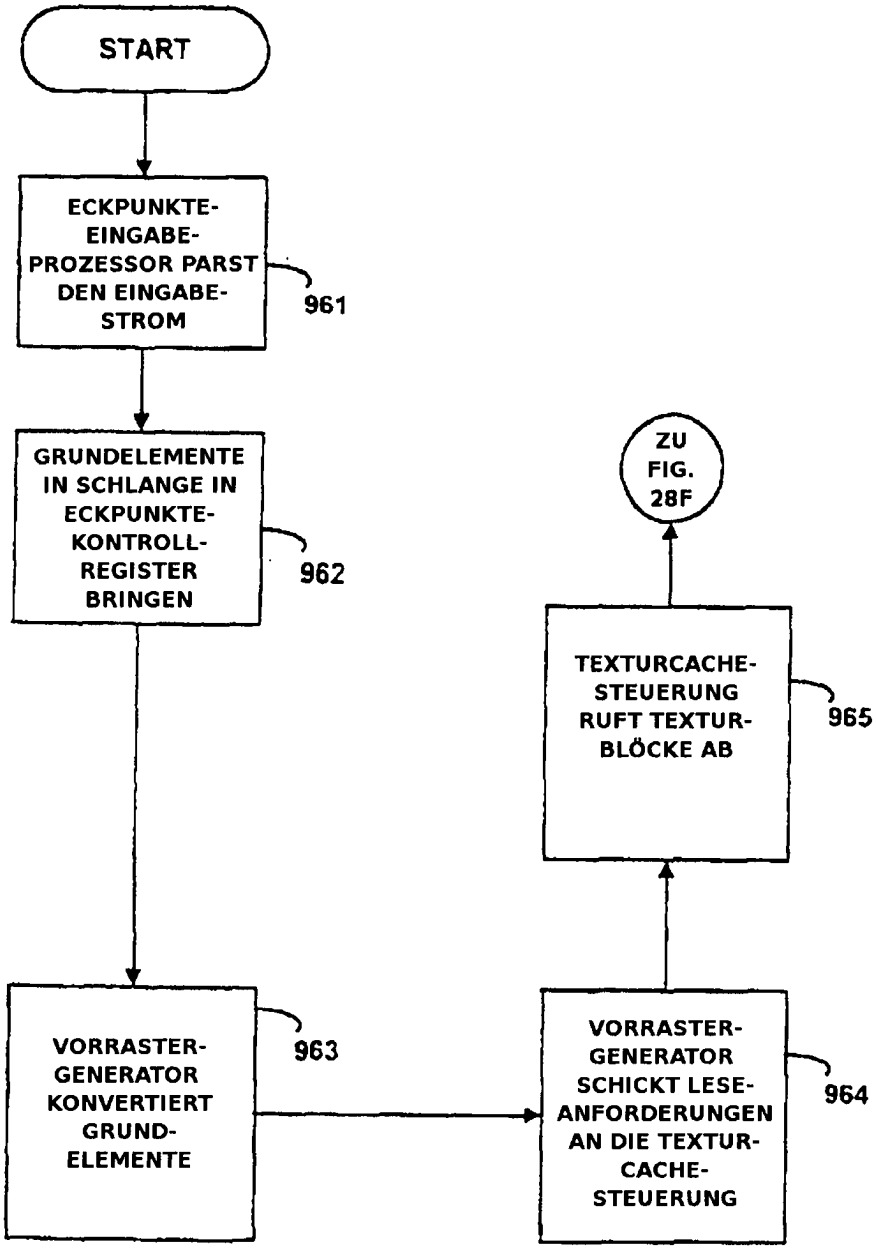

START
ECKPUNKTE-EINGABE-PROZESSOR PARST DEN EINGABE-STROM
961
GRUNDELEMENTE IN SCHLANGE IN ECKPUNKTE-KONTROLL-REGISTER BRINGEN
962
ZU FIG. 28F
TEXTURCACHE-STEUERUNG RUFT TEXTUR-BLÖCKE AB
965
VORRASTER-GENERATOR KONVERTIERT GRUND-ELEMENTE
963
VORRASTER-GENERATOR SCHICKT LESE-ANFORDERUNGEN AN DIE TEXTUR-CACHE-STEUERUNG
964

FIG. 28F

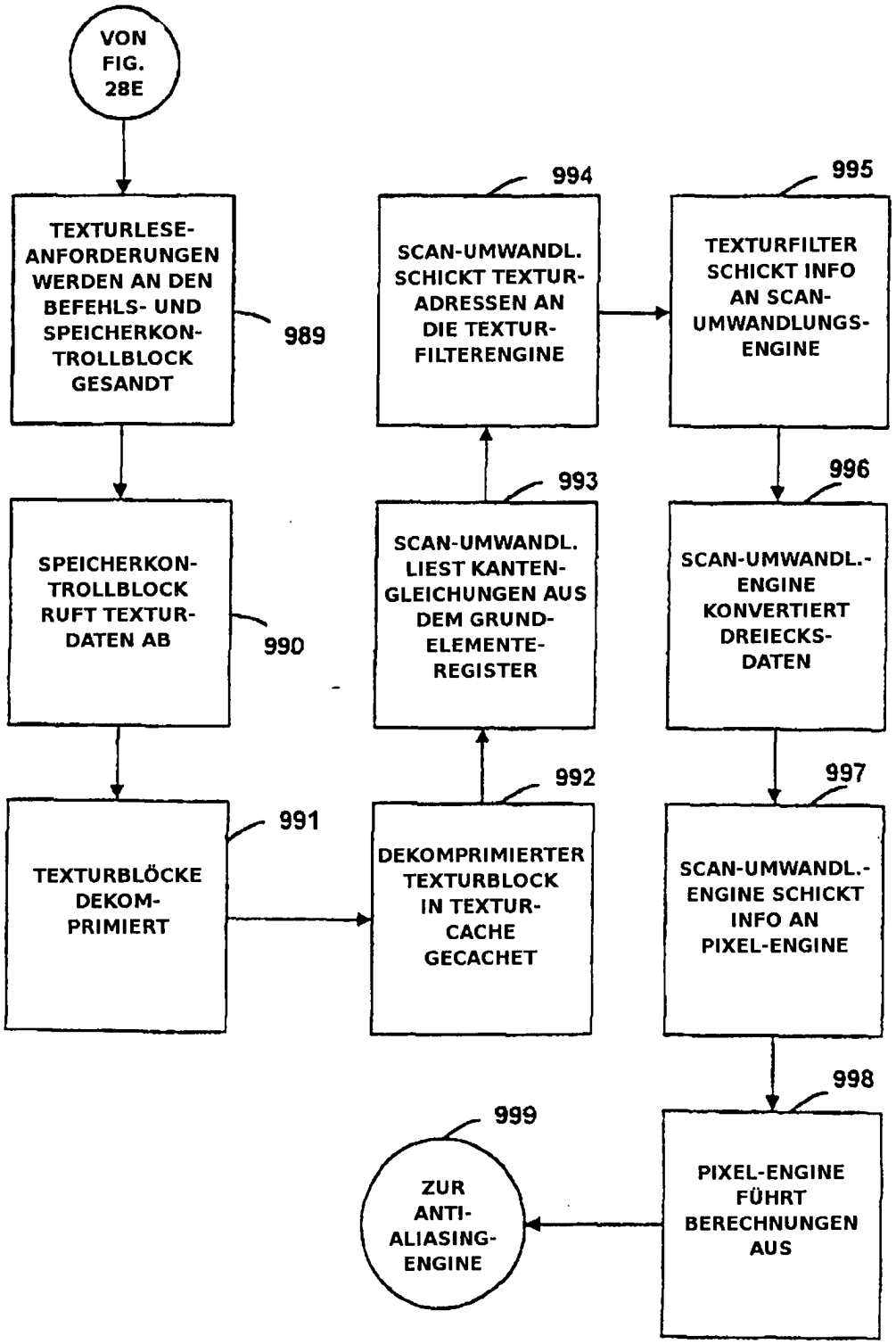

VON FIG. 28E
TEXTURLESE-ANFORDERUNGEN WERDEN AN DEN BEFEHLS- UND SPEICHERKON-TROLLBLOCK GESANDT
989
SPEICHERKON-TROLLBLOCK RUFT TEXTUR-DATEN AB
990
TEXTURBLÖCKE DEKOM-PRIMIERT
991
DEKOMPRIMIERTER TEXTURBLOCK IN TEXTUR-CACHE GECACHET
992
SCAN-UMWANDL. LIEST KANTEN-GLEICHUNGEN AUS DEM GRUND-ELEMENTE-REGISTER
993
SCAN-UMWANDL. SCHICKT TEXTUR-ADRESSEN AN DIE TEXTUR-FILTERENGINE
994
TEXTURFILTER SCHICKT INFO AN SCAN-UMWANDLUNGS-ENGINE
995
SCAN-UMWANDL.-ENGINE KONVERTIERT DREIECKS-DATEN
996
SCAN-UMWANDL.-ENGINE SCHICKT INFO AN PIXEL-ENGINE
997
PIXEL-ENGINE FÜHRT BERECHNUNGEN AUS
998
ZUR ANTI-ALIASING-ENGINE
999

FIG. 29

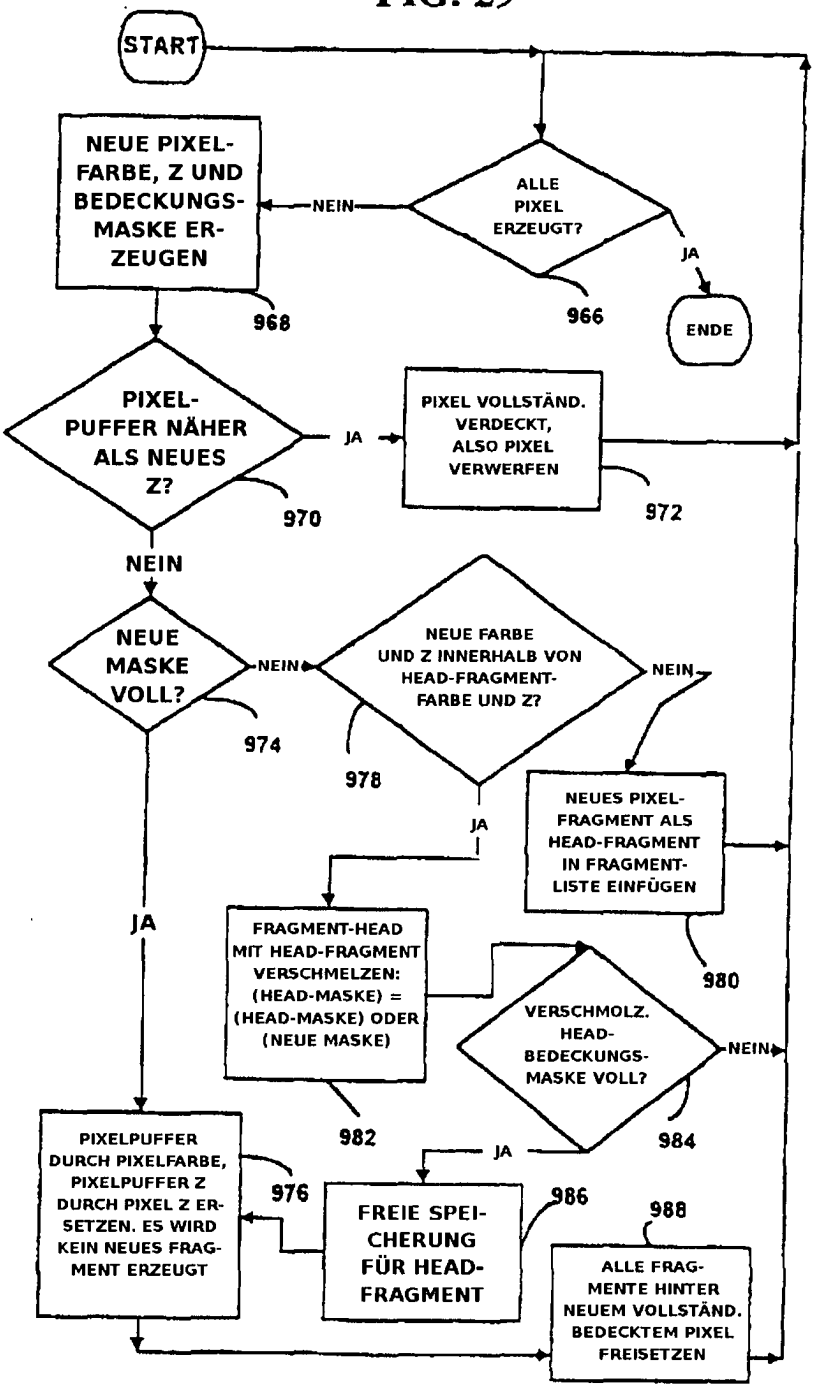
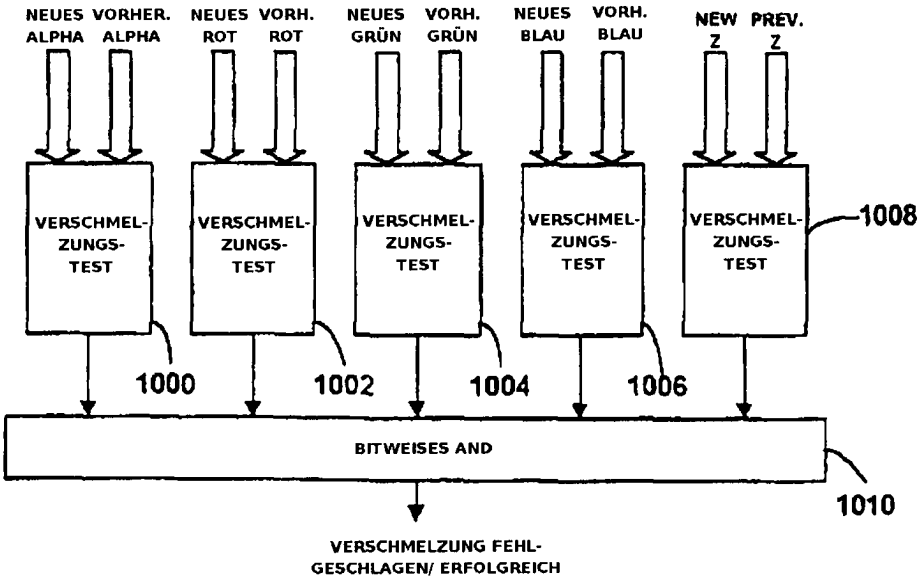

START
NEUE PIXEL-FARBE, Z UND BEDECKUNGS-MASKE ER-ZEUGEN
968
NEIN
ALLE PIXEL ERZEUGT?
966
JA
ENDE
PIXEL-PUFFER NÄHER ALS NEUES Z?
970
JA
PIXEL VOLLSTÄND. VERDECKT, ALSO PIXEL VERWERFEN
972
NEIN
NEUE MASKE VOLL?
974
NEIN
NEUE FARBE UND Z INNERHALB VON HEAD-FRAGMENT-FARBE UND Z?
978
NEIN
JA
NEUES PIXEL-FRAGMENT ALS HEAD-FRAGMENT IN FRAGMENT-LISTE EINFÜGEN
980
JA
FRAGMENT-HEAD MIT HEAD-FRAGMENT VERSCHMELZEN: (HEAD-MASKE) = (HEAD-MASKE) ODER (NEUE MASKE)
982
VERSCHMOLZ. HEAD-BEDECKUNGS-MASKE VOLL?
984
NEIN
JA
PIXELPUFFER DURCH PIXELFARBE, PIXELPUFFER Z DURCH PIXEL Z ER-SETZEN. ES WIRD KEIN NEUES FRAG-MENT ERZEUGT
976
FREIE SPEI-CHERUNG FÜR HEAD-FRAGMENT
986
ALLE FRAG-MENTE HINTER NEUEM VOLLSTÄND. BEDECKTEM PIXEL FREISETZEN
988

FIG. 30


NEUES ALPHA
VORHER. ALPHA
NEUES ROT
VORH. ROT
NEUES GRÜN
VORH. GRÜN
NEUES BLAU
VORH. BLAU
NEW Z
PREV. Z
VERSCHMEL-ZUNGS-TEST
1008
1000
1002
1004
1006
BITWEISES AND
1010
VERSCHMELZUNG FEHL-GESCHLAGEN/ ERFOLGREICH

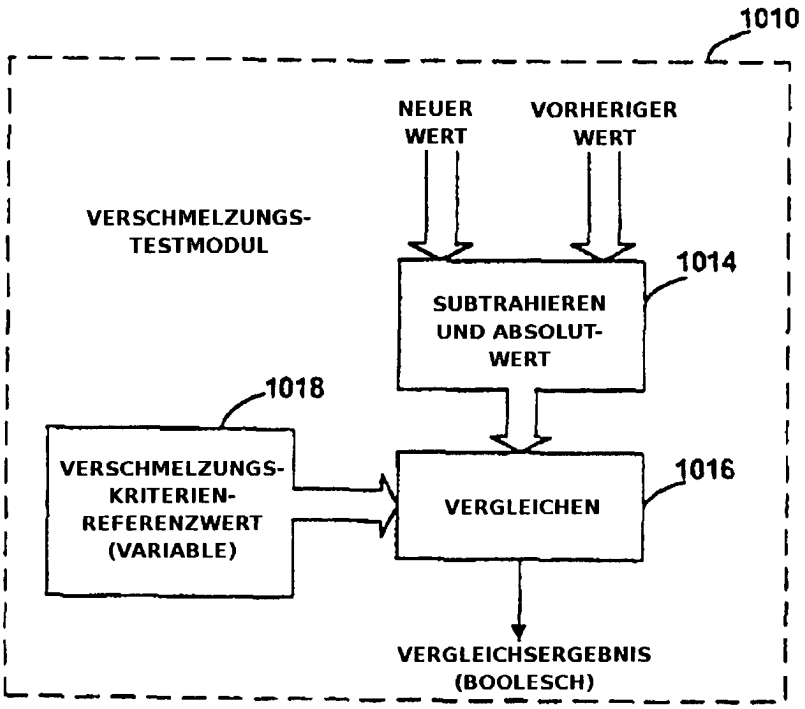

1010
NEUER WERT
VORHERIGER WERT
VERSCHMELZUNGS-TESTMODUL
1014
SUBTRAHIEREN UND ABSOLUT-WERT
1018
VERSCHMELZUNGS-KRITERIEN-REFERENZWERT (VARIABLE)
1016
VERGLEICHEN
VERGLEICHSERGEBNIS (BOOLESCH)

FIG. 31

FIG. 32

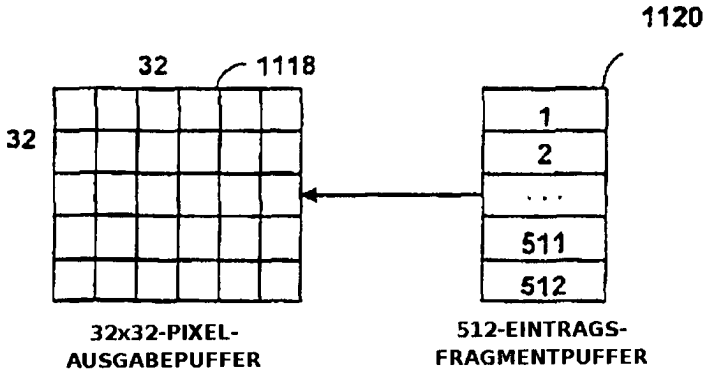

1120
32
1118
32
1
2
...
511
512
32x32-PIXEL-
AUSGABEPUFFER
512-EINTRAGS-
FRAGMENTPUFFER

FIG. 33

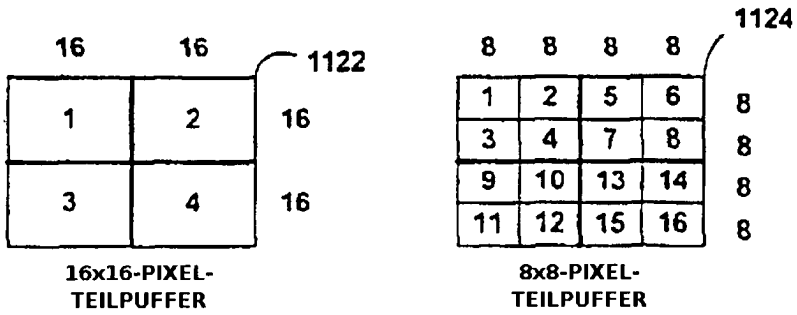

16
16
1122
1
2
16
3
4
16
16x16-PIXEL-
TEILPUFFER
1124
8
8
8
8
1
2
5
6
8
3
4
7
8
8
9
10
13
14
8
11
12
15
16
8
8x8-PIXEL-
TEILPUFFER

FIG. 34A

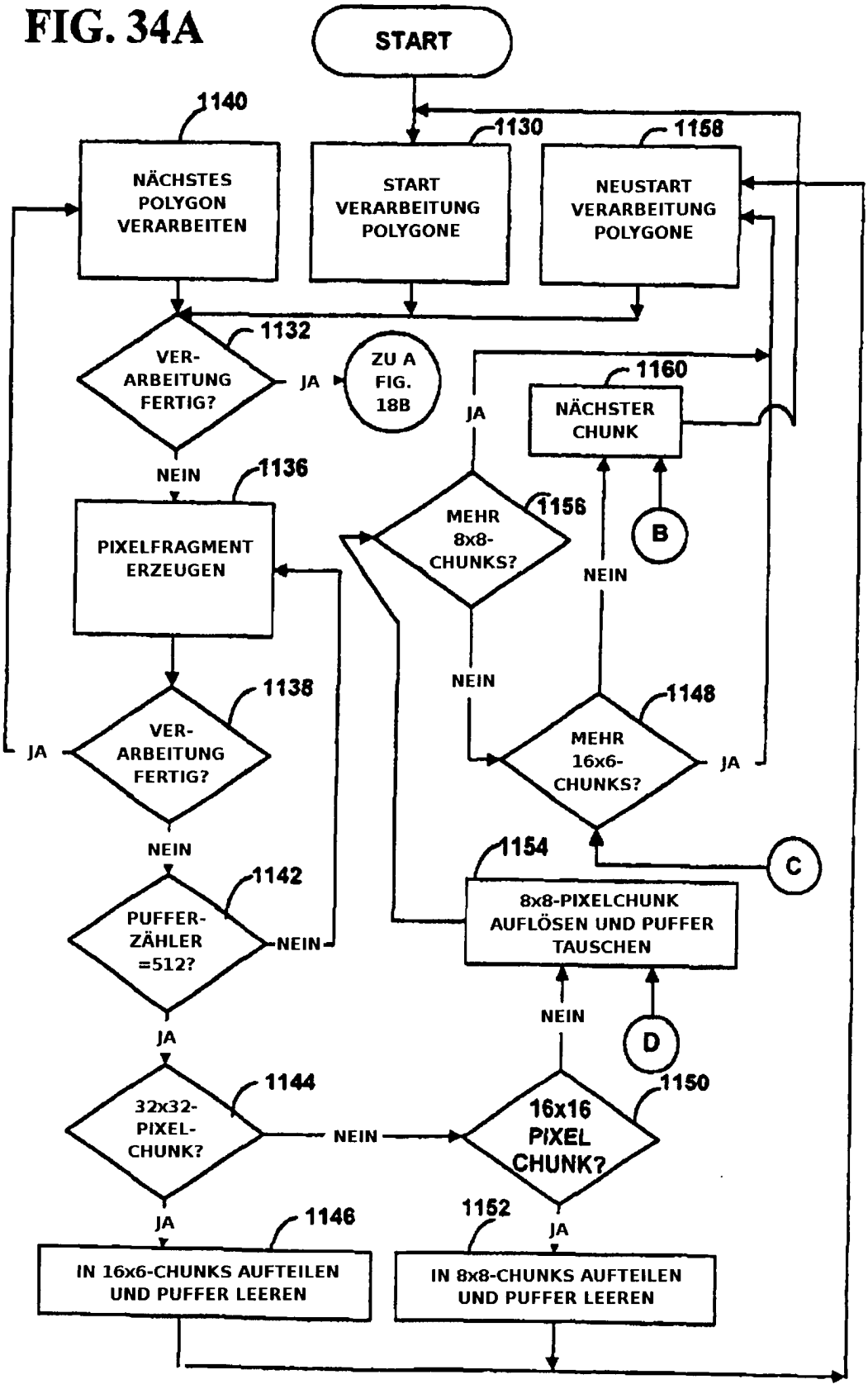

START
1140
NÄCHSTES POLYGON VERARBEITEN
1130
START VERARBEITUNG POLYGONE
1158
NEUSTART VERARBEITUNG POLYGONE
1132
VER-ARBEITUNG FERTIG?
JA
ZU A FIG. 18B
NEIN
1136
PIXELFRAGMENT ERZEUGEN
1138
VER-ARBEITUNG FERTIG?
JA
NEIN
1142
PUFFER-ZÄHLER =512?
NEIN
JA
1144
32x32-PIXEL-CHUNK?
NEIN
JA
1146
IN 16x6-CHUNKS AUFTEILEN UND PUFFER LEEREN
1160
NÄCHSTER CHUNK
JA
1156
MEHR 8x8-CHUNKS?
B
NEIN
NEIN
1148
MEHR 16x6-CHUNKS?
JA
C
1154
8x8-PIXELCHUNK AUFLÖSEN UND PUFFER TAUSCHEN
NEIN
D
1150
16x16 PIXEL CHUNK?
1152
JA
IN 8x8-CHUNKS AUFTEILEN UND PUFFER LEEREN

FIG. 34B

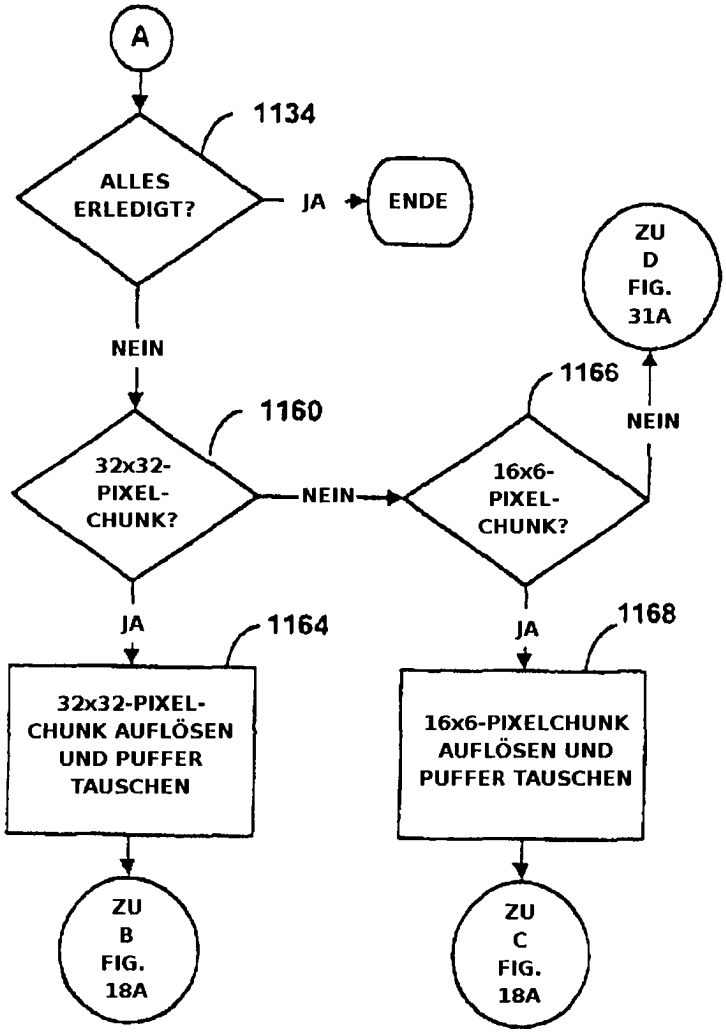
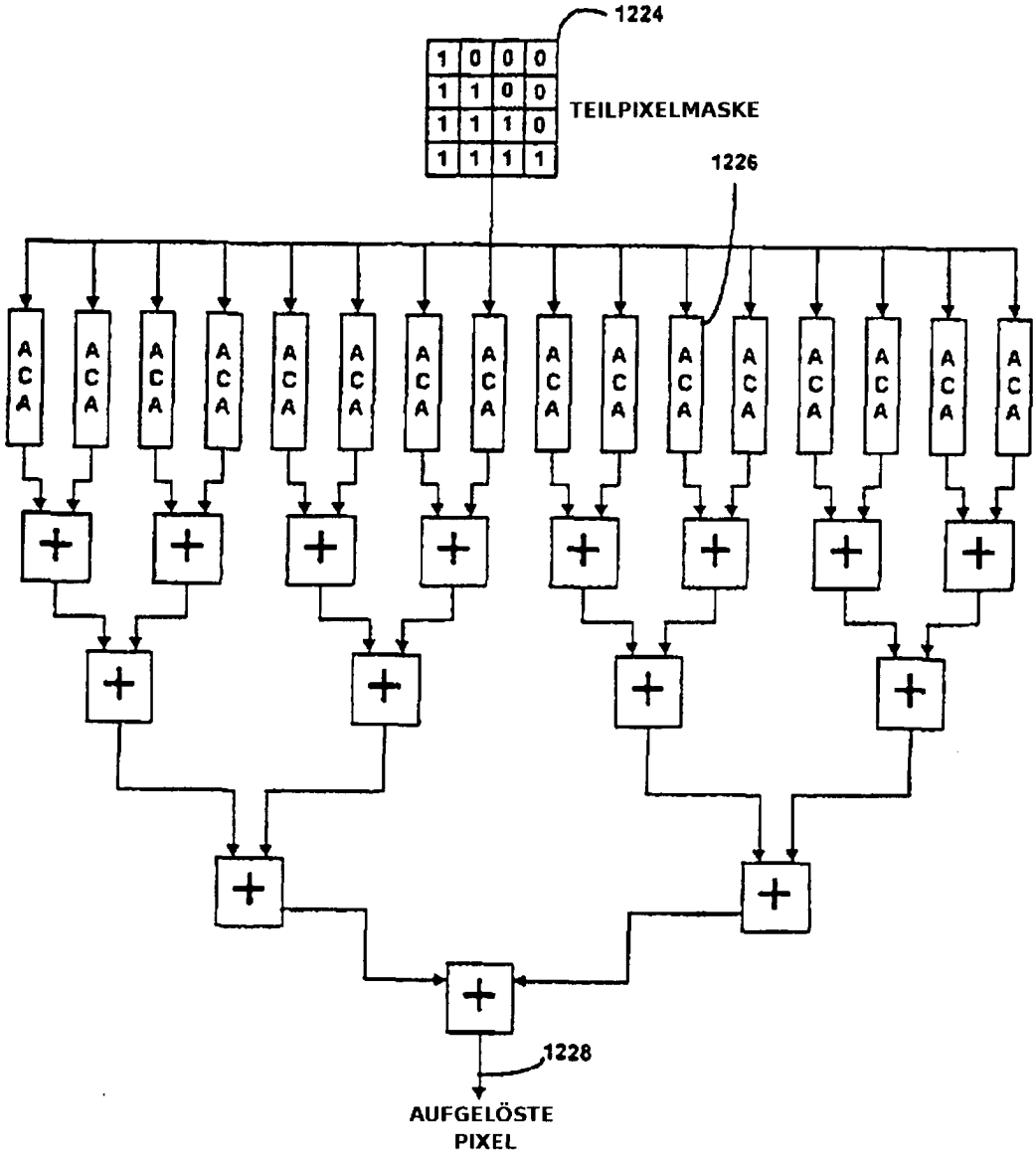

A
1134
ALLES ERLEDIGT?
JA
ENDE
NEIN
1160
32x32-PIXEL-CHUNK?
NEIN
1166
16x6-PIXEL-CHUNK?
NEIN
ZU D FIG. 31A
JA
1164
32x32-PIXEL-CHUNK AUFLÖSEN UND PUFFER TAUSCHEN
ZU B FIG. 18A
JA
1168
16x6-PIXELCHUNK AUFLÖSEN UND PUFFER TAUSCHEN
ZU C FIG. 18A

FIG. 35


1224
1 0 0 0
1 1 0 0
1 1 1 0
1 1 1 1
TEILPIXELMASKE
1226
ACA
1228
AUFGELÖSTE
PIXEL

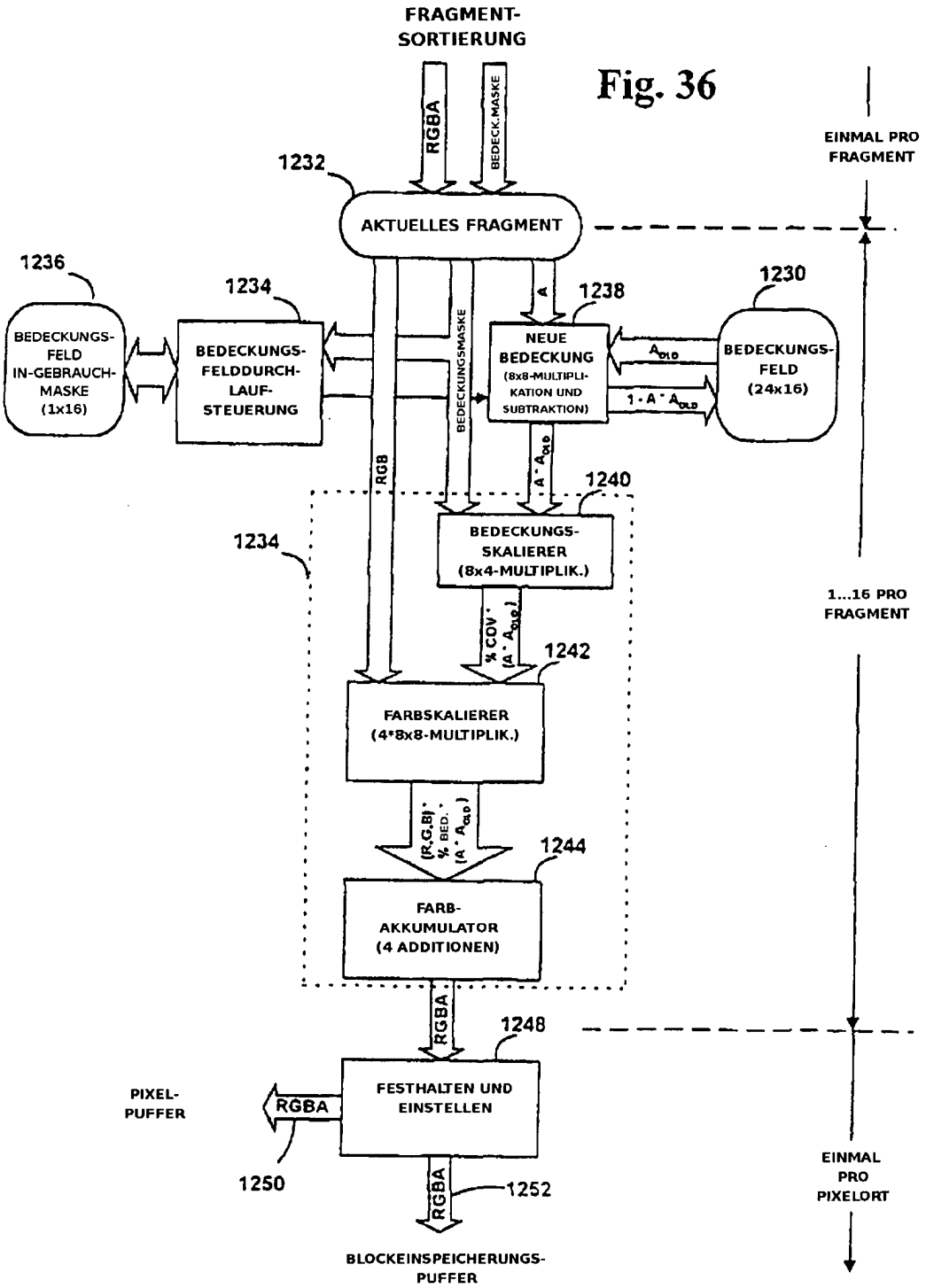

FRAGMENT-
SORTIERUNG
RGBA
BEDECK.-MASKE
1232
AKTUELLES FRAGMENT
EINMAL PRO
FRAGMENT
1236
BEDECKUNGS-
FELD
IN-GEBRAUCH-
MASKE
(1x16)
1234
BEDECKUNGS-
FELDDURCH-
LAUF-
STEUERUNG
BEDECKUNGSMASKE
A
1238
NEUE
BEDECKUNG
(8x8-MULTIPLI-
KATION UND
SUBTRAKTION)
1230
BEDECKUNGS-
FELD
(24x16)
RGB
1240
BEDECKUNGS-
SKALIERER
(8x4-MULTIPLIK.)
1234
1...16 PRO
FRAGMENT
1242
FARBSKALIERER
(4*8x8-MULTIPLIK.)
1244
FARB-
AKKUMULATOR
(4 ADDITIONEN)
RGBA
1248
PIXEL-
PUFFER
RGBA
FESTHALTEN UND
EINSTELLEN
1250
RGBA
1252
EINMAL
PRO
PIXELORT
BLOCKEINSPEICHERUNGS-
PUFFER

Fig. 36

FIG. 37

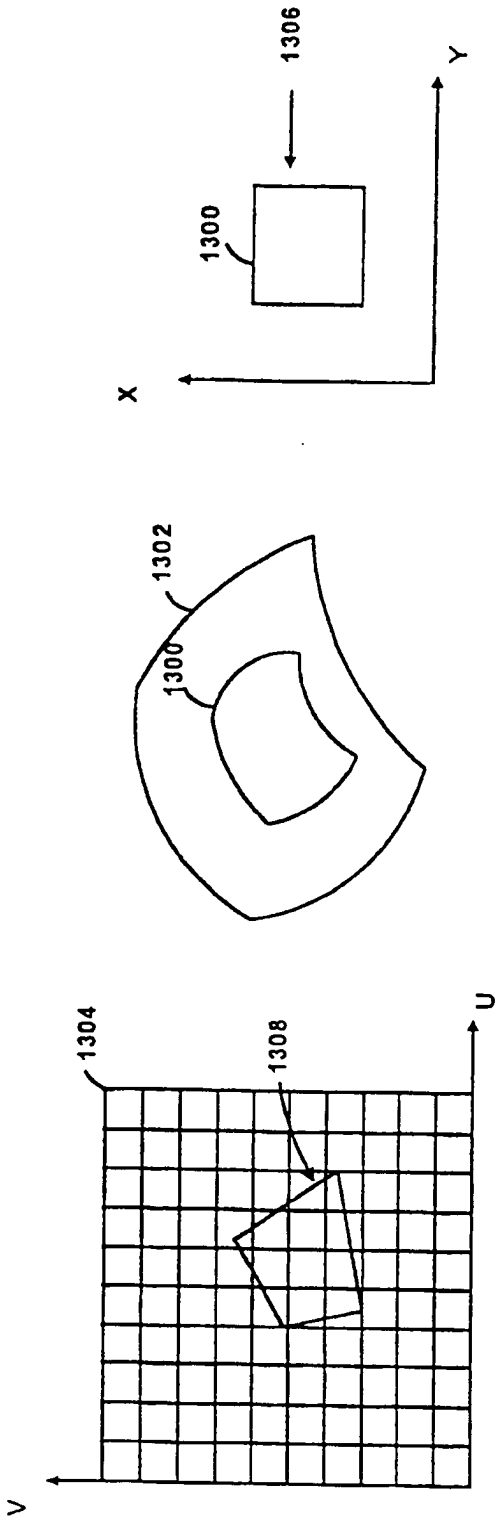


FIG. 38

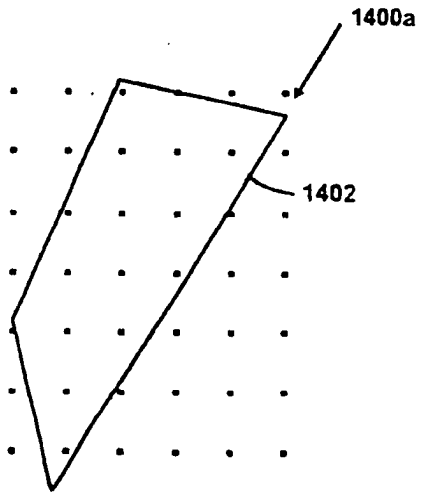


FIG. 38A

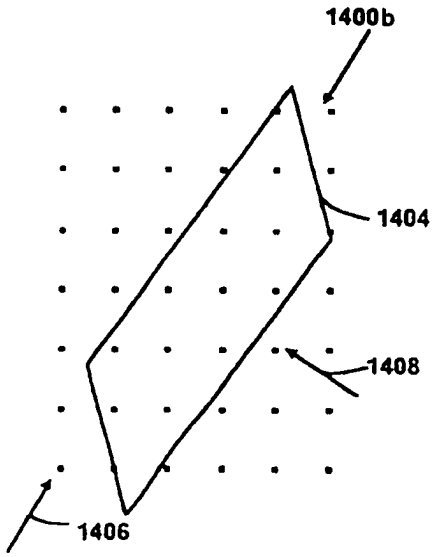


FIG. 38B

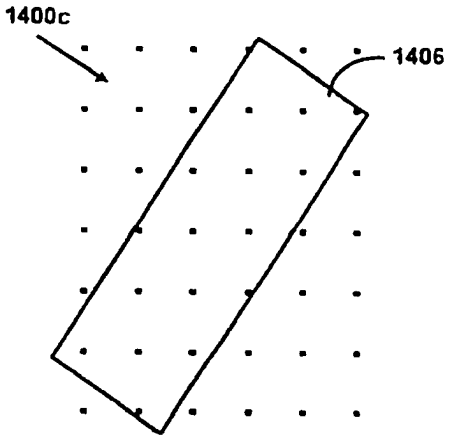


FIG. 38C

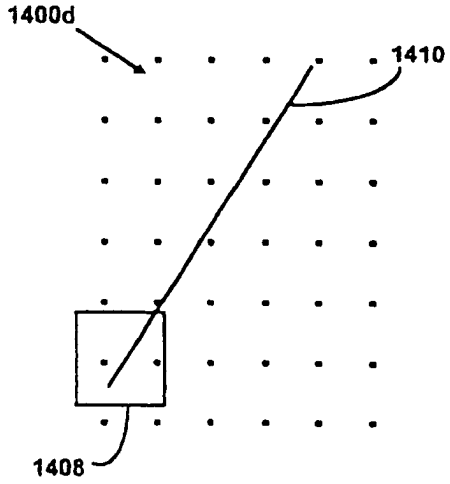


FIG. 38D

FIG. 39

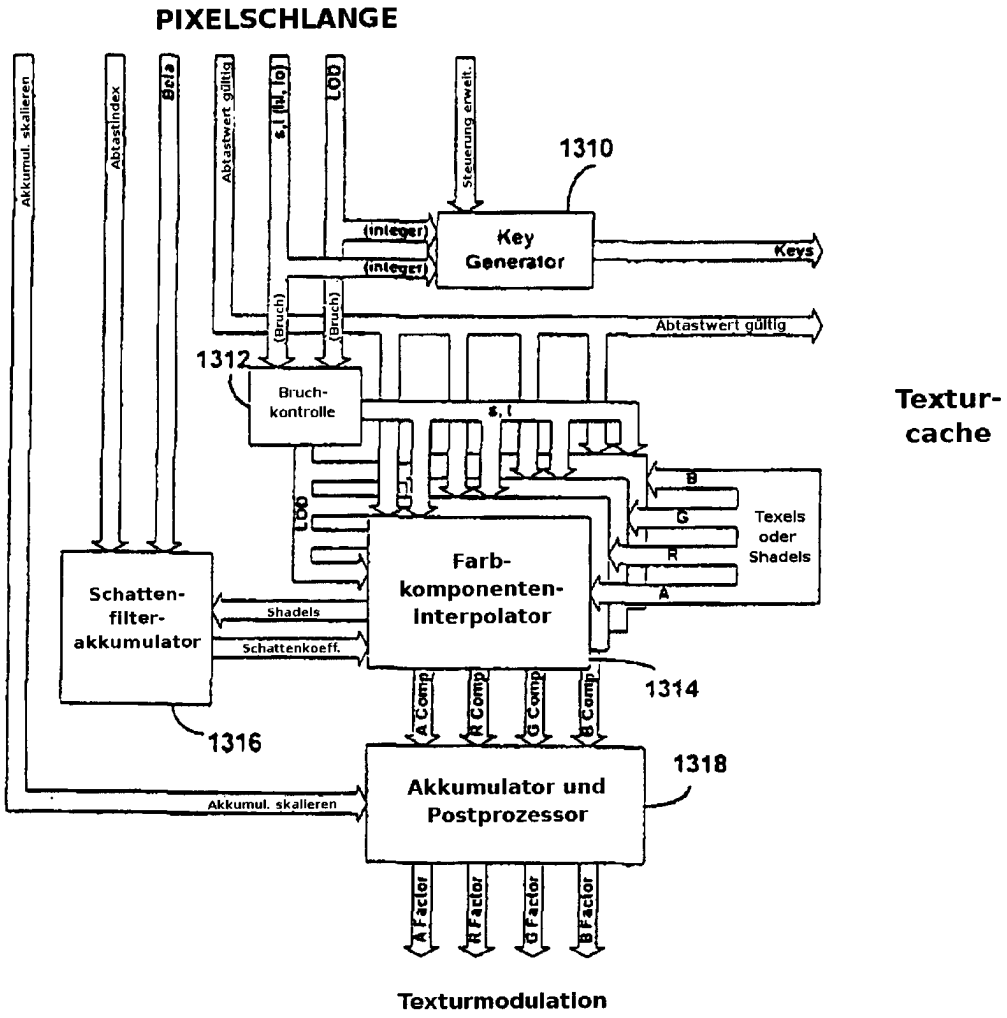
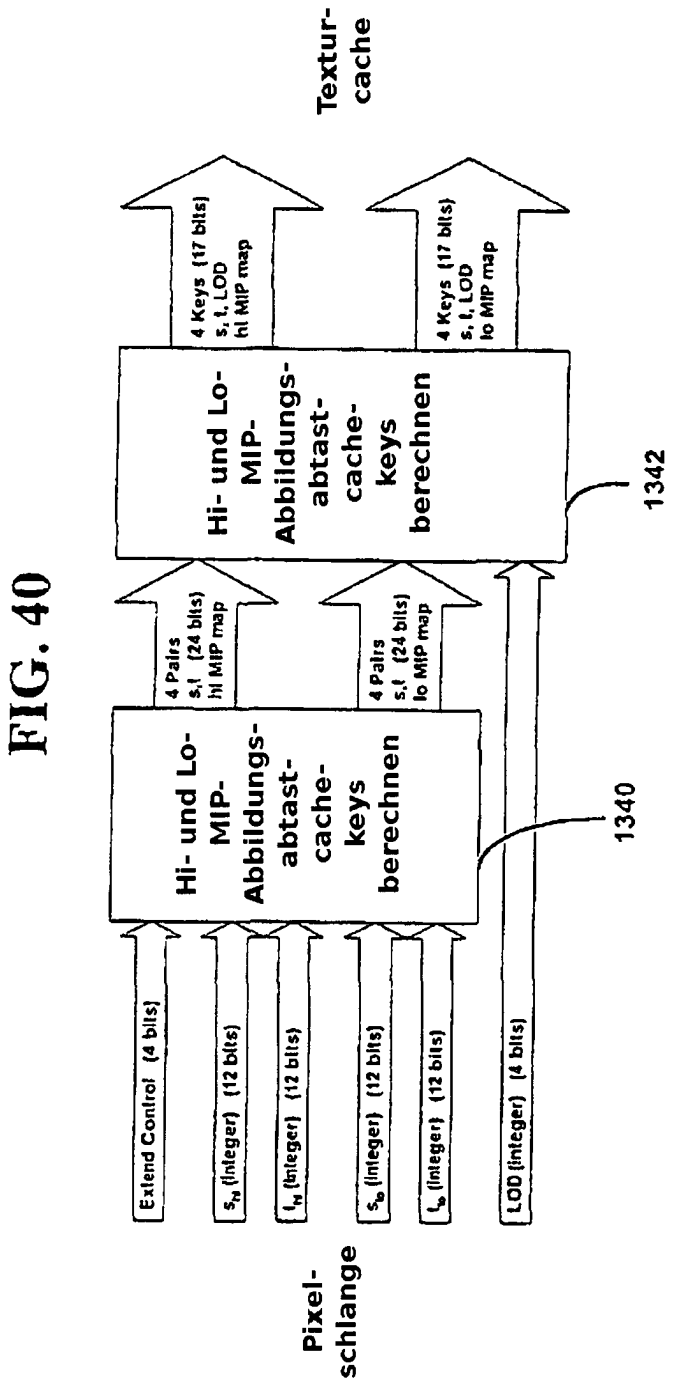

PIXELSCHLANGE
Akkumul. skalieren
Abtastindex
Beta
Abtastwert gültig
s,t [lod, lo]
LOD
Steuerung erwelt.
1310
(integer)
(integer)
Key Generator
Keys
(Bruch)
(Bruch)
Abtastwert gültig
1312
Bruch-kontrolle
s, t
LOD
B
G
R
A
Texels oder Shadels
Texur-cache
Schatten-filter-akkumulator
Shadels
Schattenkoeff.
Farb-komponenten-Interpolator
1314
A Comp
R Comp
G Comp
B Comp
1316
1318
Akkumulator und Postprozessor
Akkumul. skalieren
A Factor
R Factor
G Factor
B Factor
Texturmodulation

FIG. 40


Pixel-schlange
Extend Control (4 bits)
S_H (integer) (12 bits)
t_H (integer) (12 bits)
S_L (integer) (12 bits)
t_L (integer) (12 bits)
LOD (integer) (4 bits)
Hi- und Lo-MIP-Abbildungs-abtast-cache-keys berechnen
1340
4 Pairs s,t (24 bits) hi MIP map
4 Pairs s,t (24 bits) lo MIP map
Hi- und Lo-MIP-Abbildungs-abtast-cache-keys berechnen
1342
4 Keys (17 bits) s, t, LOD hi MIP map
4 Keys (17 bits) s, t, LOD lo MIP map
Textur-cache

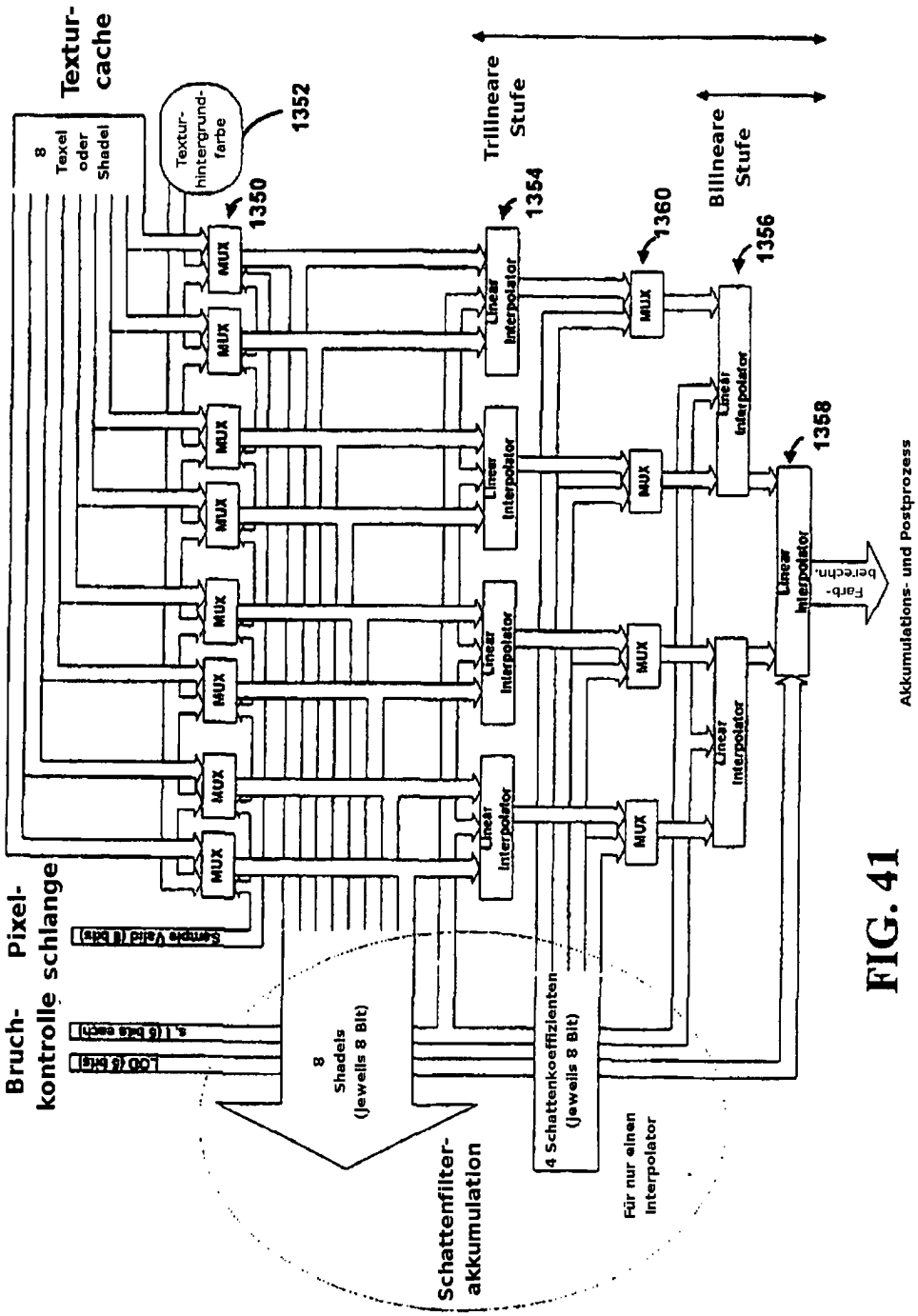

Textur-cache
8 Texel oder Shadel
Textur-hintergrund-farbe
1352
1350
Trillineare Stufe
1354
1360
Billineare Stufe
1356
1358
MUX
Linear Interpolator
Farb-berechn.
Akkumulations- und Postprozess
Pixel-kontrolle schlange
8 Shadels (jeweils 8 Bit)
4 Schattenkoeffizienten (jeweils 8 Bit)
Für nur einen Interpolator
Schattenfilter-akkumulation

FIG. 41

FIG. 42

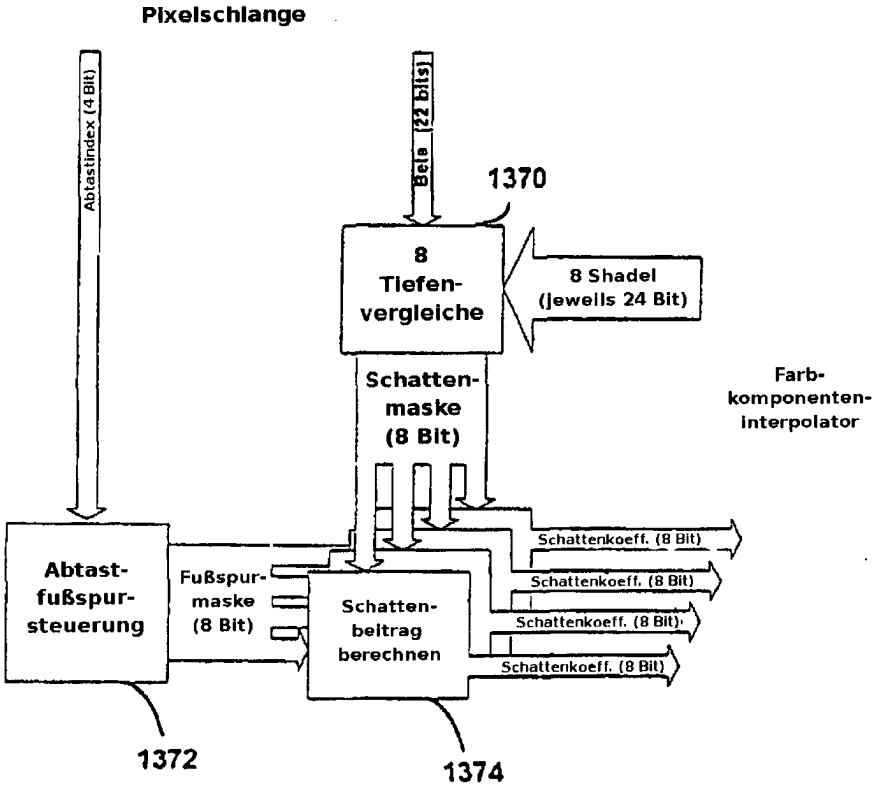
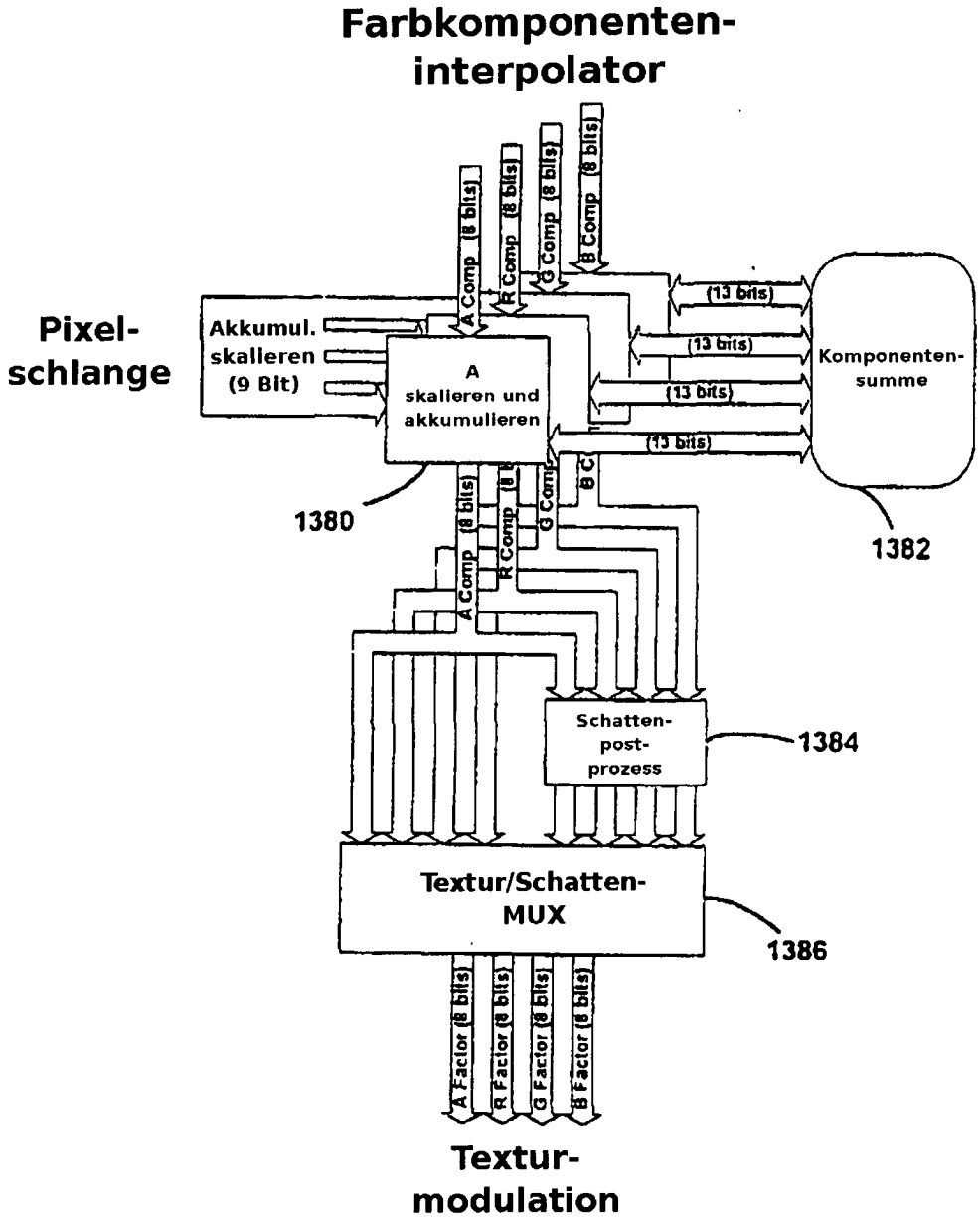

Pixelschlange
Abtastindex (4 Bit)
Bels (22 bits)
1370
8 Tiefen-vergleiche
8 Shadel (jeweils 24 Bit)
Schatten-maske (8 Bit)
Farb-komponenten-interpolator
Schattenkoeff. (8 Bit)
Schattenkoeff. (8 Bit)
Schattenkoeff. (8 Bit)
Schattenkoeff. (8 Bit)
Abtast-fußspur-steuerung
Fußspur-maske (8 Bit)
Schatten-beitrag berechnen
1372
1374

FIG. 43


Farbkomponenten-
interpolator
Pixel-
schlange
Akkumul.
skalieren
(9 Bit)
A
skalieren und
akkumulieren
A Comp (8 bits)
R Comp (8 bits)
G Comp (8 bits)
B Comp (8 bits)
(13 bits)
(13 bits)
(13 bits)
(13 bits)
Komponenten-
summe
1380
1382
Schatten-
post-
prozess
1384
Textur/Schatten-
MUX
1386
A Factor (8 bits)
R Factor (8 bits)
G Factor (8 bits)
B Factor (8 bits)
Textur-
modulation