FIG. 3A

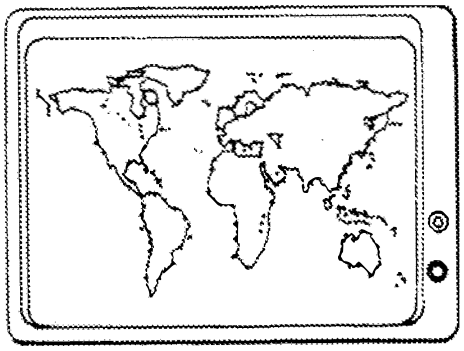


FIG. 3B

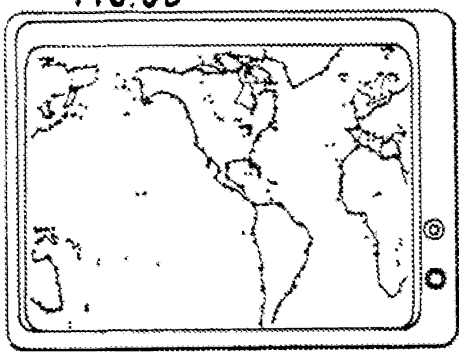


FIG. 3C

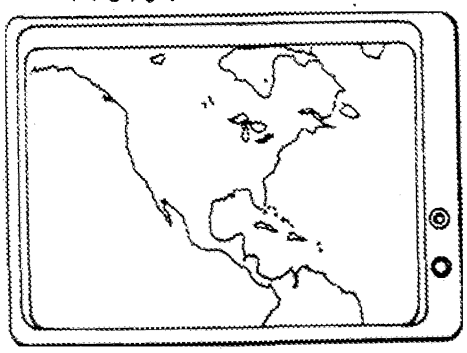


FIG. 3D

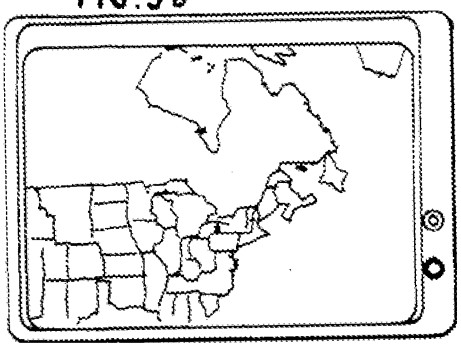


FIG. 3E

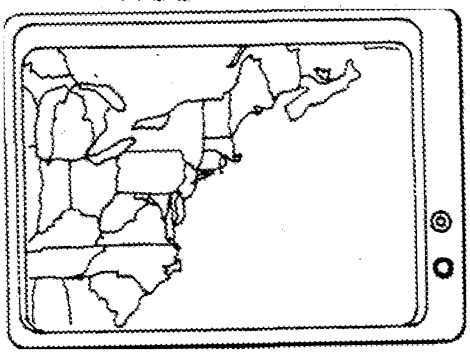


FIG. 3F

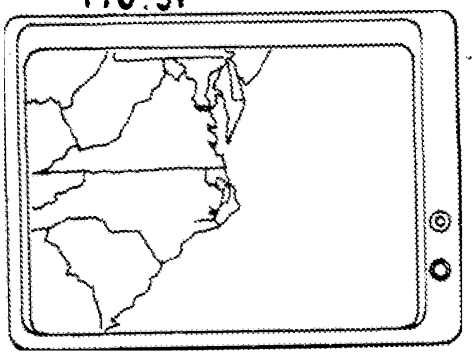


FIG.4

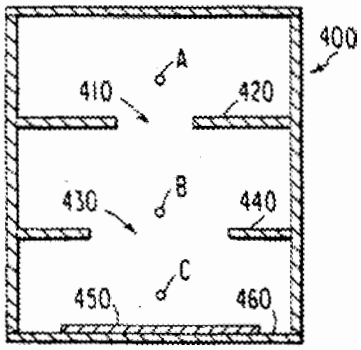


FIG.5A

FIG.5B

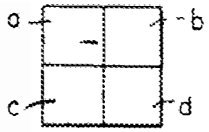


FIG.6

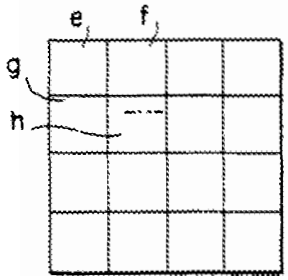


FIG.7

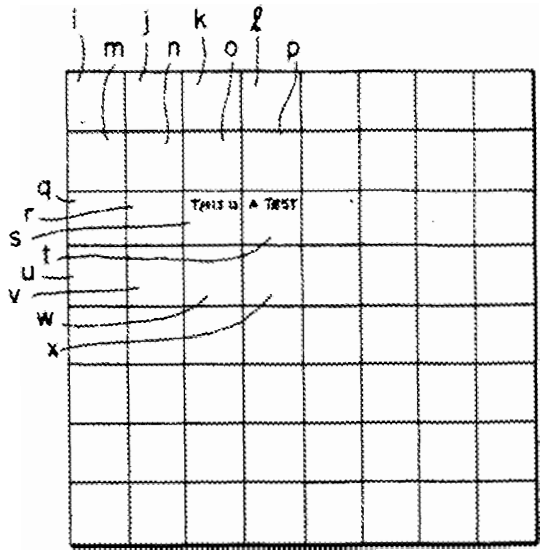
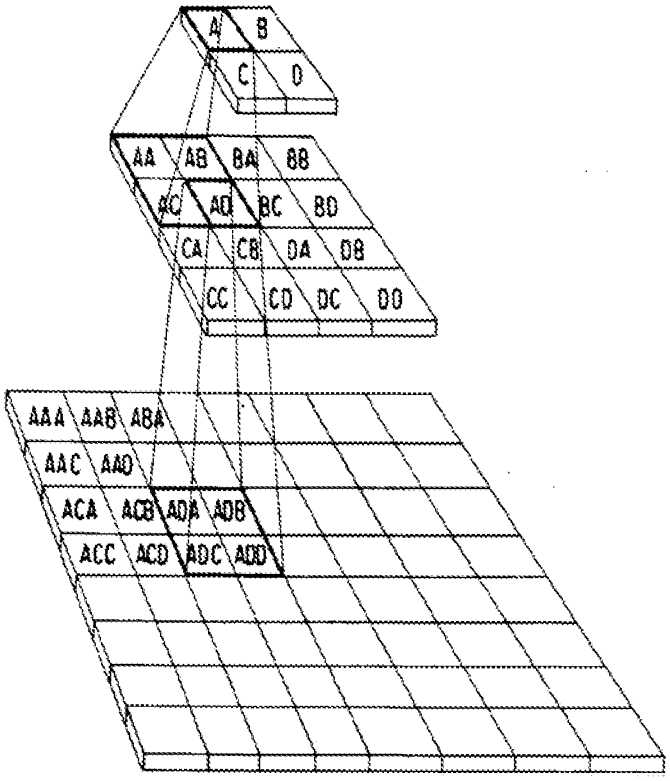


FIG. 8



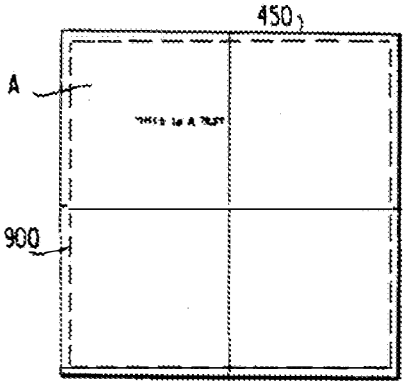
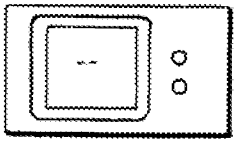


FIG. 9A

FIG. 9B



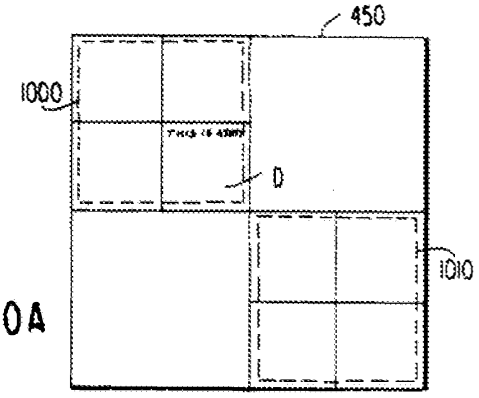
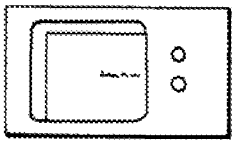


FIG. 10A

FIG. 10B



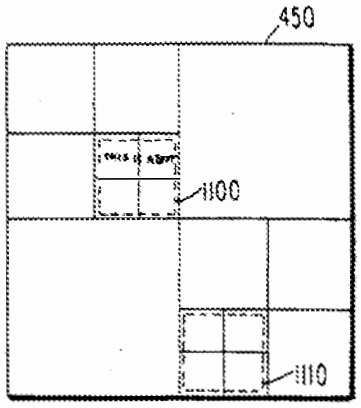


FIG. 11A

FIG. 11B

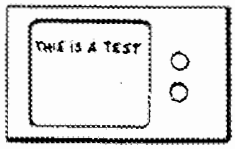


FIG. 12

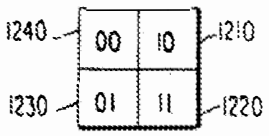


FIG. 13

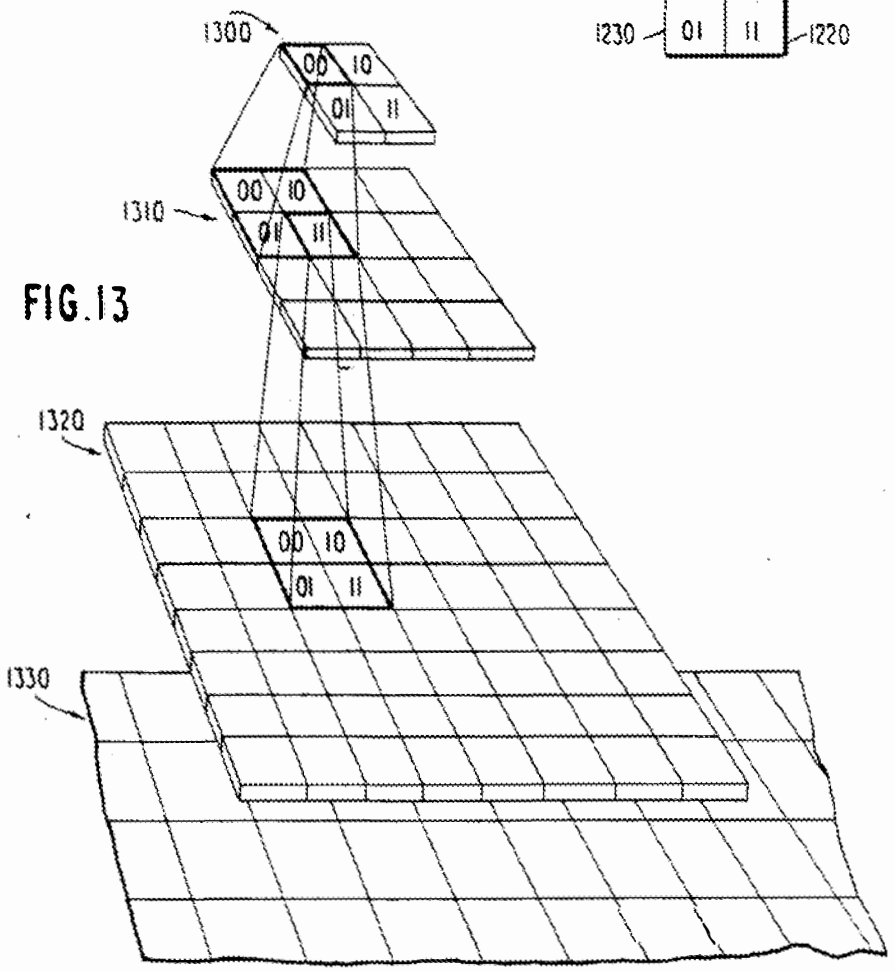


FIG.14

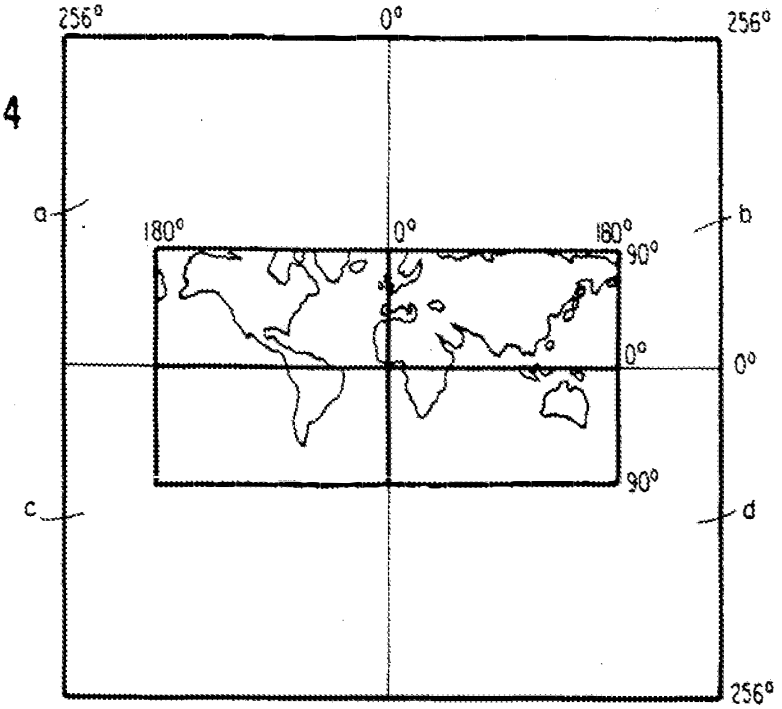

256°
0°
256°
a
b
180°
0°
180°
90°
0°
0°
90°
c
d
256°

FIG.15

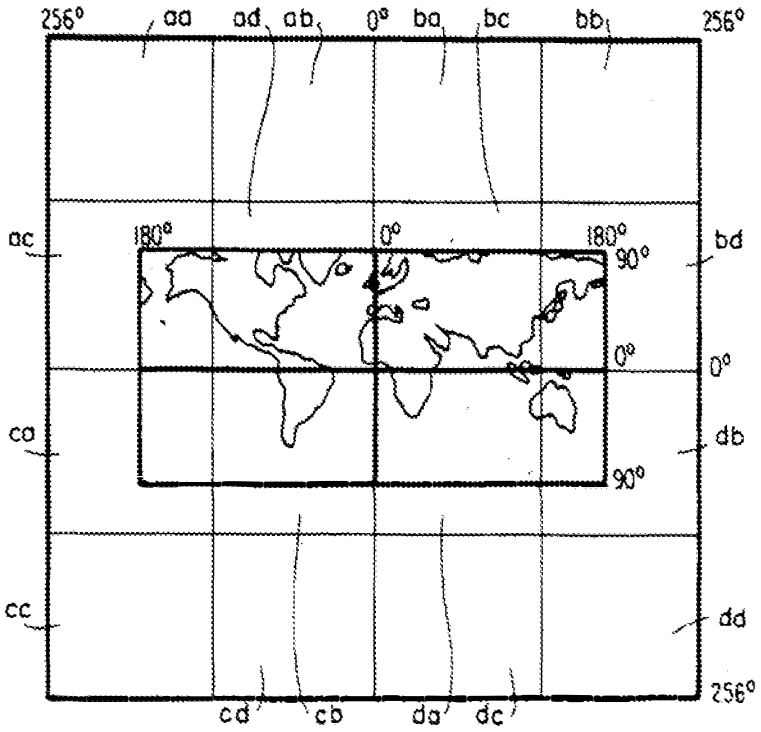

256°
aa
ad
ab
0°
ba
bc
bb
256°
ac
180°
0°
180°
90°
bd
0°
0°
ca
db
90°
cc
dd
256°
cd
cb
da
dc

FIG. 16

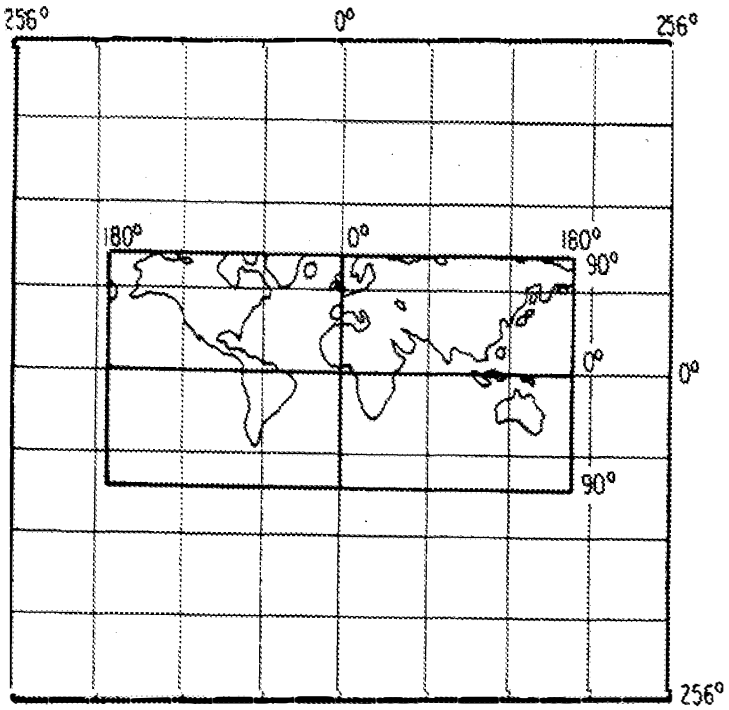


FIG. 17

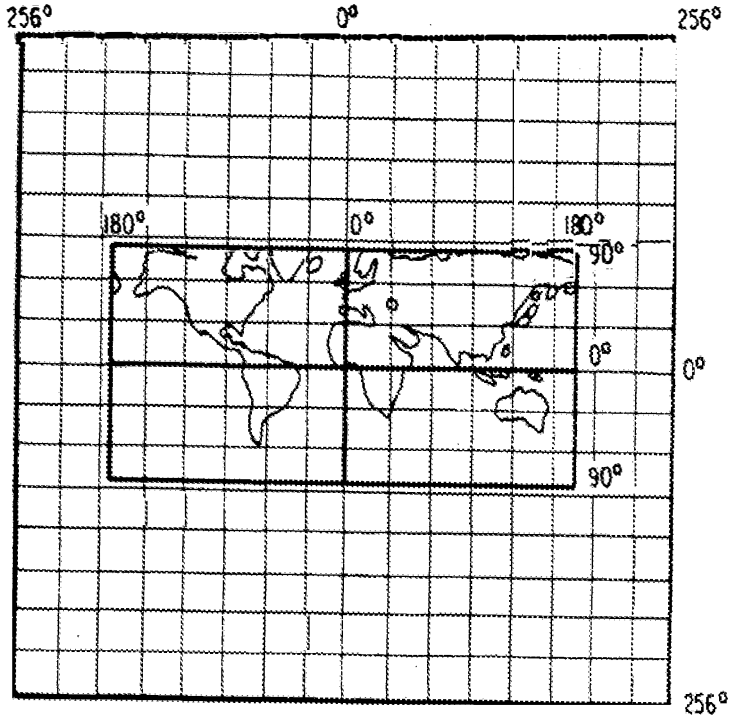


FIG. 18

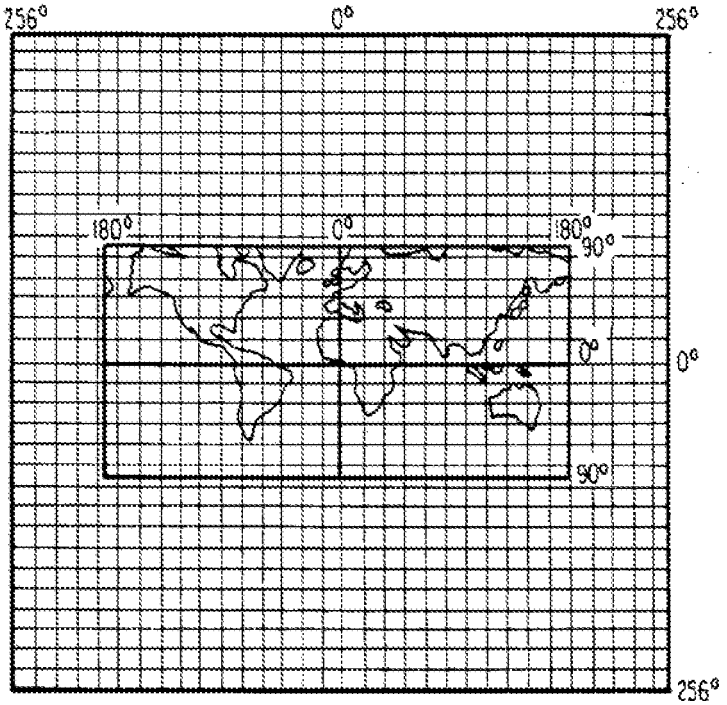


FIG. 19

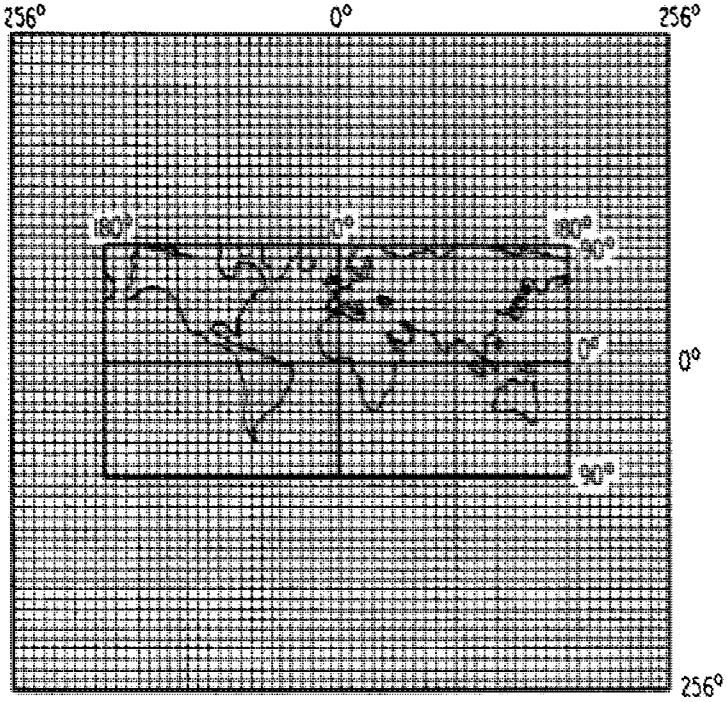
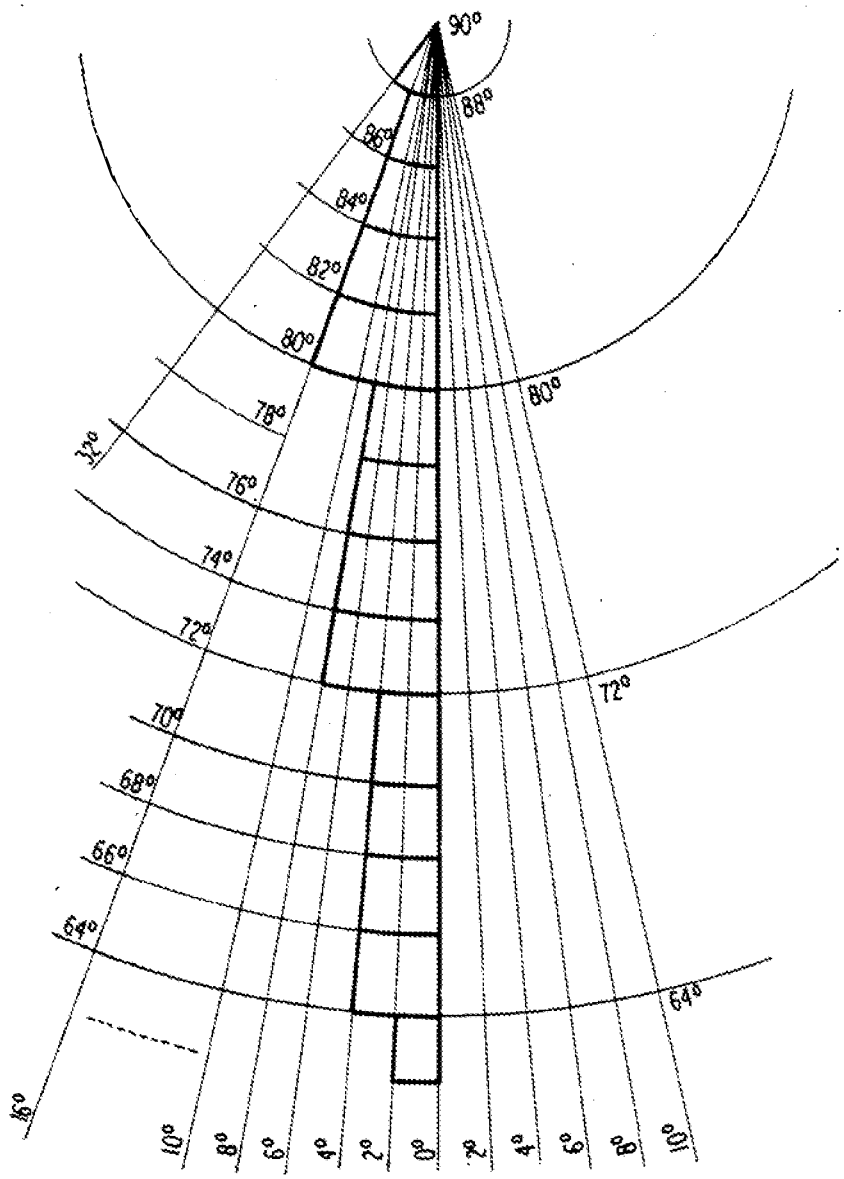


FIG. 20

ILLUSTRATION OF POLAR COMPRESSION AT THE 8th MAGNITUDE



ELECTRONIC GLOBAL MAP GENERATING SYSTEM

BACKGROUND OF THE INVENTION

1. Technical Field

This invention relates to a new variable resolution global map generating system for structuring digital mapping data in a new data base structure, managing and controlling the digital mapping data according to new mapping data access strategies, and displaying the mapping data in a new map projection of the earth.

2. Background Art

Numerous approaches have been forwarded to provide improved geographical maps, for example:

U.S. Pat. No. 4,315,747, issued to McBryde on Feb. 16, 1982, describes a new map "projection" and intersecting array of coordinate lines known as the "graticule", which is a composite of two previously known forms of projection. In particular, the equatorial portions of the world are represented by a fusiform equal area projection in which the meridian curves, if extended, would meet at points at the respective poles, referred to as "pointed poles". In contrast, the polar regions of the world map are represented by a flat polar equal area projection in which the poles are depicted as straight horizontal lines with the meridians intersecting along its length. Thus, in a flat polar projection the meridian curves converge toward the poles but do not meet at a point and, instead, intersect a horizontal linear pole. The two component portions of the flat world map are joined where the parallels are of equal length. The composite is said to be "homolinear" because all of the meridian curves are similar curves, for example, sine, cosine or tangent curves, which merge where the two forms of projection are joined where the respective parallels are equal. The flat polar projections in the polar portions of the map provide a compromise with the Mercator cylinder projections, thereby greatly reducing distortion.

U.S. Pat. No. 1,050,596, issued to Bacon on Jan. 14, 1913, describes another composite projection for world maps and charts which uses a Mercator or cylindrical projection for the central latitudes of the earth and a convergent projection at the respective poles. In the central latitudes, the grids of the Mercator projection net or graticule are rectangular. In the polar regions, the converging meridians may be either straight or curved.

U.S. Pat. No 1,620,413, issued to Balch on Dec. 14, 1926, discusses gnomic projections from a conformal sphere to a tangent plane and Mercator or cylindrical projections from the conformal sphere to a tangent cylinder. Balch is concerned with taking into account the non-spherical shape of the earth, and therefore, devises the so-called "conformal sphere" which represents the coordinates from the earth whose shape is actually that of a spheroid or ellipsoid of revolution, without material distortion.

U.S. Pat. No. 752,957, issued to Colas on Feb. 23, 1904, describes a map projection in which a map of the entire world is plotted or transcribed on an oval constructed from two adjacent side by side circles with arcs joining the two circles. The meridians are smooth curves equally spaced at the equator, while the latitude lines are non-parallel curves.

U.S. Pat. No. 400,642 issued to Beaumont on Apr. 2, 1889, describes a map of the earth on two intersecting spheres, on which the coordinate lines of latitude and longitude are all arcs of circles.

U.S. Pat. No. 751,226, issued to Grinten on Feb. 2, 1904, represents the whole world upon the plane surface of a single circle with twice the diameter of the corresponding globe, the circle being delineated by a graticule of coordinates of latitude and longitude which are also arcs of circles.

U.S. Pat. No. 3,248,806, issued to Schrader on May 3, 1966, discloses a subdivision of the earth into a system of pivotally mounted flat maps, each map segment representing only a portion of the earth's surface in spherical projection on an equilateral spherical triangle to minimize distortion.

U.S. Pat. No. 2,094,543, issued to Lackey et al on Sept. 28, 1937, describes a projector for optically producing a variety of different map projections, including orthographic, stereographic and globular projections onto flat translucent screens and a variety of other projections on shaped screens.

U.S. Pat. No. 2,650,517, issued to Falk on Sept. 1, 1953, describes a photographic method for making geographical maps.

U.S. Pat. No. 2,354,785, issued to Rohl on Aug. 1, 1944, discloses two circular maps which are mounted side by side, and an arrangement for rotating the two maps in unison so that corresponding portions of the earth's surface are at all times in proper relationship.

U.S. Pat. No. 3,724,079, issued to Jasperson et al on Apr. 3, 1973, discloses a navigational chart display device which is adapted to display a portion of a map and enable a pilot to fix his position, to plot courses and to measure distances.

U.S. Pat. No. 2,431,847 issued to Van Dusen on Dec. 2, 1947, discloses a projection arrangement, in which a portion of the surface of a spherical or curved map may be projected in exact scale and in exact proportional relationship.

McBryde and Thomas, *Equal Area Projections for World Statistical Maps*, Special Publication No. 245, Coast & Geodetic Survey 1949.

In addition to the above further teachings as to geographical mapping can be found in the *Elements of Cartography*, 4th edition which was written by Arthur Robinson, Randall Sale and Joel Morrison, and published by John Wiley & Sons (1978).

The present invention seeks to provide a low cost and efficient mapping system which allows the quick and easy manipulation of and access to an extraordinary amount of mapping information, i.e., a mapping system which allows a user to quickly and easily access a detailed map of any geographical area of the world.

Map information can be stored using at least three different approaches, i.e., paper, analog storage and digital storage, each approach having its own advantages and disadvantages as detailed below.

The paper mapping approach has been around since papyrus and will probably exist for the next thousand years.

Advantages of paper storage:

inexpensive.

once printed, no further processing is required to access the map information, so not subject to processing breakdown.

Disadvantages of paper storage:

can become bulky and unwieldy when dealing with a large geographical area, or a large amount of maps.

paper does not have the processing capabilities or "intelligence" of computers, and therefore does not support automated search or data processing capabilities.

cannot be updated cheaply and easily.

The analog mapping approach is used to provide what is commonly known as videodisc maps. The information is stored as still frames under N.T.S.C. (National Television Standards Committee) conventions. To make maps, a television camera moves across a paper map lying on a workbench. Every few inches a frame is recorded on videotape. After one row of the map is completely recorded, the camera is moved down to the next row of frames to be recorded. This process is repeated until frames representing a checkerboard pattern of the entire map are recorded. The recorded videotape could be used to view the map; however, access time to scan to different areas of the recorded map is usually excessive. As a result, a videodisc, with its quicker access time, is typically used as the medium for analog map storage. The recorded videotape is sent to a production house which "stamps" out 8 inch or 12 inch diameter, videodiscs.

Advantages of the analog storage approach:

one side of a 12 inch videodisc can hold 54,000 "frames" of a paper map. A frame is typically equal to 2½×3 inches of the paper map.

access time to any frame can be fast usually under 5 seconds.

once located on the videodisc, the recorded analog map information will be used to control the raster scan of a monitor and to produce a reproduction of the map in 1/30th of a second.

through additional hardware and software, mapping symbols, text and/or patterhsn can be overlaid on top of the recorded frame.

Disadvantages of the analog storage approach:

the "frames" are photographed from paper maps, which, as mentioned above, cannot be updated cheaply or easily.

due to paper map projections, mechanical camera movements, lens distortions and analog recording electronics, the videodisc image which is reproduced is not as accurate as the original paper map.

as a result of the immediately above phenomena, latitude and longitude information which is extracted from the reproduced image cannot be fully trusted.

if a major error is made in recording any one of the 54,000 frames, it usually requires redoing and re-stamping.

since frames cannot be scrolled, most implementations employ a 50% overlap technique. This allows the viewer to jump around the database with a degree of visual continuity; however, this is at a sacrifice of storage capacity. If the frame originally covered 2½×3 inches or approximately 8 square inches of the paper map, the redundant overlap information is 6 square inches, leaving only 2 square inches of new information in the centroid of each frame.

as a result of the immediately above deficiency, a 2×3 foot map containing 864 square inches would require 432 frames; thus, only 125 paper maps could be stored on one side of a 12 inch videodisc.

must take hundreds of video screen dumps to make a hard copy of a map area of interest and, even then, the screens do not immediately splice together because of the overlap areas.

the biggest disadvantage is that, since frames have to be arranged in a checkerboard fashion, there is no way to jump in directions other that north, south, east or west and maintain visual continuity. As an example, the visual discontinuity in viewing a "great circle" route from Alaska to New York would be unbearable for all but the most hearty.

The digital mapping approach has been around for at least 20 years and is much more frequently used than the analog approach. Digital data bases are stored in computers in a format similar to text of other databases. Unlike map information on a videodisc, the outstanding map features are stored as a list of objects to be drawn, each object being defined by a plurality of vector "dot" coordinates which define the crude outline of the object. As one example, a road is drawn by connecting a series of dots which were chosen to define the path (i.e., the "outline") of the road. Once drawn, further data and processing can be used to smooth the crude outline of the object, place text, such as the name or description of the object in a manner similar to what happens when drawing on a paper map.

Advantages of the digital approach:

digital maps are the purest form of geographical mapping data: from them, paper and analog maps can be produced.

digital maps can be quickly and easily updated in near real-time, and this updating can be in response to data input from external sources (e.g., geographical monitoring devices such as satellite photography).

digital maps can be easily modified to effect desirable mapping treatments such as uncluttering, enhancing, coloring, etc.

digital maps can be easily and accurately scaled, rotated and drawn at any perspective view point.

digital maps can be caused to reproduce maps in 3-D.

digital maps can drive pen-plotters (for easy paper reproductions), robots, etc.

digital maps can be stored on any mass storage device.

Disadvantages of the digital approach:

digital maps require the use or creation of a digital database: this is a very time-consuming and expensive process, but once it is made, the data base can be very easily copied and used for many different projects.

The digital approach is utilized with the present invention, as this approach provides overwhelming advantages over the above-described paper and analog approaches.

In designing any mapping system, several features are highly desirable:

First, it is highly desirable that the mapping system be of low cost.

Second, and probably most important, is access time. Not only is it generally desirable that the desired map section be accessible and displayed within a reasonable amount of time, but in some instances, this access time is critical.

In addition to the above, the present invention (as mentioned above), seeks to provide a third important feature,—a mapping system which allows the manipulation of and access to an extraordinary amount of mapping information, i.e., a mapping system which allows a user to quickly and easily access a detailed map of any geographical area of the world.

A tremendous barrier is encountered in any attempt to provide this third feature. In utilizing the digital approach to map a large geographical area in detail

(e.g., the earth), one should be able to appreciate that the storage of mapping data sufficient to accurately define all the geographical features would represent a tremendous data base.

While there have been digital mapping implementations which have successfully been able to manipulate a tremendous data base, these implementations involve tremendous cost (i.e., for the operation and maintenance of massive mainframe computer and data storage facilities). Furthermore, there is much room for improvement in terms of access time as these mainframe implementations result in access times which are only as quick as 20 seconds. Thus, there still exists a need for a low-cost digital mapping system which can allow the storage, manipulation and quick (i.e., "real time") access and visual display of a desired map section from a tremendous mapping data base.

There are several additional mapping system features which are attractive.

It is highly desirable that a mapping system be sensitive to and compensate for distortions caused by mapping curved geographical (i.e., earth) surfaces onto a flat, two-dimensional representation. While prior art approaches have provided numerous methods with varying degrees of success, there is a need for further improvements which are particularly applicable to the digital mapping system of the present invention.

It is additionally attractive for a mapping system to easily allow a user to change his/her "relative viewing position", and that in changing this relative position, the change in the map display should reflect a feeling of continuity. Note that the "relative viewing position should be able to be changed in a number of different ways. First, the mapping system should allow a user to selectively cause the map display to scroll or "fly" along the geographical map to view a different (i.e., "lateral") position of the geographical map while maintaining the same degree of resolution as the starting position. Second, the mapping system should allow a user to selectively vary the size of the geographical area being displayed (i.e., "zoom") while still maintaining an appropriate degree of resolution, i.e., allow a user to selectively zoom to a higher "relative viewing position" to view a larger geographical area with lower resolution regarding geographical, political and cultural characteristics, or zoom to a lower "relative viewing position" to view a smaller geographical area with higher resolution. (Note that maintaining the appropriate amount of resolution is important to avoid map displays which are effectively barren or are cluttered with geographical, political and cultural features.) Again, while prior art approaches have provided numerous methods with varying degrees of success, there is a need for further improvements which are particularly applicable to the digital mapping system of the present invention.

The final feature concerns compatibility with existing mapping formats. As mentioned above, the creation of a digital database is a very tedious, time-consuming and expensive process. Tremendous bodies of mapping data are available from many important mapping authorities, for example, the U.S. Geological Survey (USGS), Defense Mapping Agency (DMA), National Aeronautics and Space Administration (NASA), etc. In terms of both being able to easily utilize the mapping data produced by these agencies, and represent an attractive mapping system to these mapping agencies, it would be highly desirable for a mapping system to be compatible with all of the mapping formats used by these respective agencies. Prior art mapping systems have been deficient in this regard; hence, there still exists a need for such a mapping system.

SUMMARY OF THE INVENTION

The present invention provides a digital mapping method and system of a unique implementation to satisfy the aforementioned needs.

The present invention provides a computer implemented method and system for manipulating and accessing digital mapping data in a tremendous data base, and for the reproduction and display of electronic display maps which are representative of the geographical, political and cultural features of a selected geographical area. The system includes a digital computer, a mass storage device (optical or magnetic), a graphics monitor, a graphics controller, a pointing device, such as a mouse, and a unique approach for structuring, managing, controlling and displaying the digital map data.

The global map generating system organizes the mapping data into a hierarchy of successive magnitudes or levels for presentation of the mapping data with variable resolution, starting from a first or highest magnitude with lowest resolution and progressing to a last or lowest magnitude with highest resolution. The idea of this hierarchical structure can be likened to a pyramid with fewer stones or "tiles" at the top, and where each successive descending horizontal level or magnitude contains four times as many "tiles" as the level or magnitude directly above it. The top or first level of the pyramid contains 4 tiles, the second level contains 16 tiles, the third contains 64 tiles and so on, such that the base of a 16 magnitude or level pyramid would contain 4 to the 16th power or 4,294,967,296 tiles. This total includes "hyperspace" which is later clipped or ignored. Hyperspace is that excess imaginary space left over from mapping of 360 deg, space to a zero magnitude virtual or imaginary space of 512 deg, square.

A first object of the present invention is to provide a digital mapping method and system which are of low cost.

A second and more important object of the present invention is to provide a unique digital mapping method and system which allow access to a display of the geographical, political and cultural features of a selected geographical area within a minimum amount of time.

A third object of the present invention is to provide a digital mapping method and system which allow the manipulation of and access to an extraordinary amount of mapping information, i.e., a mapping method and system which allow a user to quickly and easily access a detailed map of any geographical area of the world.

Another object of the present invention is to provide a digital mapping method and system which recognize and compensate for distortion introduced by the representation of curved (i.e., earth) surfaces onto a flat two-dimensional display.

Still a further object of the present invention is to provide a digital mapping method and system which allow a user to selectively change his/her "relative viewing position", i.e., to cause the display monitor to scroll or "fly" to display a different "lateral" mapping position of the same resolution, and to cause the display monitor to "zoom" to a higher or lower position to display a greater or smaller geographical area, with an appropriate degree of resolution.

A fifth object of the present invention is to provide a digital mapping method and system utilizing a unique

mapping graticule system which allows mapping data to be compatibly adopted from several widely utilized mapping graticule systems.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, structures and features of the present invention will become more apparent from the following detailed description of the preferred mode for carrying out the invention; in the description to follow, reference will be made to the accompanying drawings in which:

FIG. **1** is an illustration corresponding to a flat projection of the earth's surface.

FIG. **2** is an illustration of a digital computer and mass storage devices which can be utilized in implementing the present invention.

FIGS. **3A-3F** are illustrations of monitor displays showing the ability of the present invention to display varying sizes of geographical areas at varying degrees of resolution.

FIG. **4** is a cross-sectional diagram of a simple building example explaining the operation of the present invention.

FIG. **5A** and B are plan view representations of a paper **450** as it is viewed from the relative viewing position A shown in FIG. **4**.

FIG. **6** is a plan view representation of a paper **450** as it is viewed from the relative viewing position B shown in FIG. **4**.

FIG. **7** is a plan view representation of a paper **450** as it is viewed from the relative viewing position C shown in FIG. **4**.

FIG. **8** is a pyramidal hierarchy of the data base file structure showing an example of the ancestry which exits between files.

FIG. **9A** is a plan view representation of a paper **450**, with the paper being divided into a first level of quadrant areas.

FIG. **9B** is an illustration of a monitor displaying a digital map of the area enclosed by the dashed portions in FIG. **9A**.

FIG. **10A** is a plan view representation of a paper **450**, with the upper-left and lower-right paper quadrant areas being further divided into quadrants.

FIG. **10B** is an illustration of a monitor displaying a digital map of the area enclosed by the upper-left dashed portion in FIG. **10A**.

FIG. **11A** is a plan view representation of a paper **450**, with several sections of the second level of quadrants being further divided into additional quadrants.

FIG. **11B** is a higher resolution display of the area enclosed within the dashed portion in FIG. **11A**.

FIG. **12** is a plan view illustration of a quadrant area division, with a two-bit naming protocol being assigned to each of the quadrant areas.

FIG. **13** is a pyramidal hierarchy of the data base files using the two-bit naming protocol of FIG. **12**, and showing an example of the ancestry which exits between files.

FIG. **14** is a plan view illustration of a 360°×180° flat projection of the earth being impressed in the 512"×512" mapping area of the present invention, with a first quadrant division dividing the mapping area into four equal 250°×256" mapping areas.

FIG. **15** is the same plan view illustration of FIG. **14**, with a second quadrant division dividing the mapping area into 16 equal 126°×128° mapping areas.

FIG. **16** is the same plan view illustration of FIG. **15**, with a third quadrant division dividing the mapping area into 64 equal 64°×64" mapping areas.

FIG. **17** is the same plan view illustration of FIG. **16**, with a fourth quadrant division dividing the mapping area into 256 equal 32°×32° mapping areas.

FIG. **18** is the same plan view illustration of FIG. **17**, with a fifth quadrant division dividing the mapping area into 1024 equal 16°×16° mapping areas.

FIG. **19** is the same plan view illustration of FIG. **18**, with a sixth quadrant division dividing the mapping area into 4096 equal 8°×8° mapping areas.

FIG. **20** is an illustration showing the application of polar compression at the 8th level or magnitude of resolution.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS OF THE INVENTION

Before turning to the detailed description of the preferred embodiments of the invention, it should be noted that the map illustrations used throughout the drawings are only crude approximations which are only being used to illustrate important features and aspects and the operation of the present invention; therefore, the geographical political and cultural outlines may very well differ from actual outlines.

FIG. **1** is a crude representation of what the earth's surface would look like if it were laid flat and viewed from a "relative viewing position" which is a great distance in space. Shown as vertical lines are: **10**, corresponding to the 0° meridian extending through Greenwich, England; **20**, corresponding to the 180° west meridian; and, **30**, corresponding to the 180° east meridian. Shown as horizontal lines are: **40**, corresponding to the equator; **50**, corresponding to 90° north (i.e., the north pole); and **60**, corresponding to 90° south (i.e., the south pole).

Note that at this "relative viewing position", not much detail as to cultural features is seen; i.e., all that is seen is the general outline of the main geographical masses of the continents.

The present invention seeks to provide a low cost and efficient computer-based mapping method and system having a unique approach for arranging and accessing a digital mapping database of unlimited size, i.e., a mapping method and system which can manipulate and access a data base having sufficient data to allow the mapping system to reproduce digital maps of any geographical area with different degrees of resolution. This can be most easily understood by viewing FIG. **2** and FIGS. **3A-F**.

Because of the overwhelming advantages over the paper and analog mapping approaches, the digital mapping approach is utilized with the present invention; thus, there is shown in FIG. **2**, a digital computer **200**, having a disk or hard drive **280**, a monitor **210**, a keyboard **220** (having a cursor control portion **230**), and a mouse device **240**. As mentioned previously, in a digital mapping approach, mapping information is stored in a format similar to the text of other databases, i.e., the outstanding map features are stored as a list of objects to be drawn, each object being defined by a plurality of vector "dot" coordinates which define the crude outline of the object. (Note: the reproduction of a digital map from a list of objects and "dot" vectors is well known the art, and is not the subject matter of the present invention; instead, the invention relates to a unique

method and system for storing and accessing the list of objects and "dot" vectors contained in a tremendous digital data base.)

Once a geographical map has been "digitized",—i.e., converted to a list of objects to be drawn and a plurality of vector "dot" coordinates which define the crude outline of the object —, the mapping database must be stored in the memory of a mass storage device. Thus, the digital computer **200**, which is to be used with the mapping method and system of the the present invention, is shown associated with the magnetic disk **260** (which represents any well-known magnetic mass storage medium, e.g., floppy disks, hard disks. magnetic tape, etc.), and the CD-ROM 270 (which represents any well-known optical storage medium, e.g. a laser-read compact disk). Alternatively, the digital mapping database can be stored on, and the digital computer can be associated with any well known electronic mass storage memory medium (e.g., ROM, RAM, etc.). Because of every increasing availability. reductions in cost, and tremendous storage capacities, the preferred memory mass storage medium is the CD-ROM, i.e., a laser-read compact disk.

The discussion now turns to FIGS. 3A–F, showing illustrations of monitor displays which provide a brief illustration of the operation of the present invention. Although the digital nature of the maps of FIGS. 3A–3F can easily be detected due to the jagged outlines, it should be understood that these geographical outlines could easily be smoothed using any of a number of "smoothing" techniques which are well-known to those skilled in the digital mapping art.

In FIG. 3A. the digital computer has retrieved relevant mapping information from the digital mapping database, and has produced a monitor display of a digital map substantially corresponding to the flat projection of the earth's surface which was shown in FIG. 1. In FIG. 3A, the monitor display reflects a "relative viewing position" which is a great distance in space, and hence, only the crude geographical outline of the continents is shown with sparse detail.

Suppose a user wishes to view a map of the states of Virginia and Maryland in greater detail. By entering the appropriate commands using the keyboard **220** or the mouse device **240**, a user can cause the monitor display to "zoom" to a lower "relative viewing position", such that the monitor displays a digital map of a smaller geographical area which is shown at a higher degree of resolution. Thus, in FIG. 3B the a digital map of the continents of the western hemisphere is displayed in greater detail.

By entering additional commands, a user can cause the monitor display to further "zoom" to the following displays: FIG. 3C showing North America in greater detail; FIG. 3D showing the eastern half of the United States in greater detail: FIG. 3E showing the east coast of the United States in greater detail; and. FIG. 3F showing Virginia and Maryland in greater detail.

Although in this example, the monitor display was caused to "zoom" to Virginia and Maryland, it should, be appreciated that the present invention allowed a user to selectively zoom into any geographical area of the earth, and once a user has reached the desired degree of mapping resolution, the mapping system of the present invention also allows the user to "scroll" or "fly" to a different lateral position on the map.

Furthermore, although the drawings illustrate the monitor display zooming to display state boundaries, and features, it should be further appreciated that the present invention is by no means limited to this degree of resolution. In fact, the degree of resolution capable with the present invention will be shown to be limited only by the operating system of the digital computer **200** with which the present invention is used. In one demonstration, the monitor display has been shown to be able to zoom to resolution where the outlines of streets were displayed. Even further degrees of resolution are possible as will be more fully understood after the discussions below.

In digitally mapping a large geographical area (e.g., the earth) in detail, —especially in the degree of resolution mentioned above —, one should be able to appreciate that the storage of digital mapping data sufficient to accurately define all the geographical, political and cultural features would represent a tremendous digital mapping database. In order to provide a low cost mapping system having quick access time and allowing a high degree of resolution, what is needed is a mapping system having an effective approach for arranging an accessing the digital database. Prior art mapping systems have been deficient in this regard.

The mapping system of the present invention utilizes a new and extremely effective approach, which can be most easily understood using the following simplified example.

In FIG. **4**, there is shown the cross-section of a building **400**, with a square hole **410** (shown in cross-section) cut through the third level floor **420**. with a larger square hole **430** (shown in cross-section) cut in the second level floor **440**, and with a large square piece of paper **450** (shown in cross-section) laid out on the first level floor **460**. Suppose it was desired to build up a digital data base which could be used to reproduce a digital map of the paper **450** with varying degrees of resolution.

First, one would take the "relative viewing position" A, and view the paper **450** through the square hole **410** in the third level floor **420**. At this level. the paper **450** appears small (FIG. 5A), and the degree of resolution is such that the message appears only as a series of dots. In order to build up a digital mapping database, the visual perception (FIG. 5A) is imagined to be divided into four equal quadrants a, b, c, d (FIG. 5B), and visual features appearing in each respective area is digitized and stored in a separate database file. Thus, four separate database files can be utilized to reproduce a digital map of the paper **450** as viewed from position A (FIG. **4**).

In order to digitize and record data corresponding to a second (or higher) degree of resolution, the next "relative viewing position" B (FIG. **4**) is taken to view the paper **450** through the square hole **430**. At this level, the paper **450** appears larger (FIG. **6**), and the degree of resolution is such that the message now appears as a series of lines. At this second level, the map is imagined as being divided into four times as many areas as the first imaginary division, and then, the visual information contained within each area is digitized and stored in a separate database file. Thus, 16 files can be used to reproduce a digital map of the paper **450**, as viewed from the relative viewing position B (FIG. **4**).

In order to digitize and record data corresponding to a third (or higher) degree of resolution. the next "relative viewing position" C (FIG. **4**) is taken to view the paper **450**. At this level, paper **450** now appears larger (FIG. **7**) and has visual features of higher resolution.

The paper **450** is imagined as being divided into four times as many areas as the second imaginary division, and the visual information is digitized and stored. Thus, 64 files could be used to reproduce a digital map of the paper **450**, as viewed from the relative viewing position C (FIG. 4).

Once digital data has been entered for the above three "relative viewing positions" A, B, C (FIG. 4), the digital mapping database contains 4+16+64 or 84 files which can be conceptually envisioned as being arranged in a pyramid structure as shown in FIG. 8. In order to allow a user to selectively display any desired map section at the desired degree of resolution, the digital computer **200** must be able to know which of the 84 files to access such that the appropriate mapping data can be obtained. The present invention accomplishes this by conceptually arranging the files in a pyramidal structure, and assigning a file name to each file which is related both to the file's position and ancestry within the pyramidal structure. This can be more specifically described as follows:

A file's ancestry can be explained using the illustrations of FIGS. 5B, 6 and 7. In FIG. 5B, the paper **450**, as viewed from "relative viewing position" A (FIG. 4), is subjected to an imaginary division into four quadrants a, b, c, and d. Quadrants a, b, c, d are related to one another in the sense that it takes all four areas to represent the paper **450**: hence quadrants a, b, c, d can be termed as brothers and sisters.

FIG. 6 is an illustration of the paper **450** as it appears from the relative viewing position B (FIG. 4), with the paper **450** being subjected to an imaginary division into 16 areas. Note that the areas e, f, g, h (FIG. 6) represent the same area of paper **450** as the quadrant a (FIG. 5B). In effect, quadrant a has been enlarged (to show a higher degree of resolution) and divided into quadrants e, f, g, h. Thus, it can be said that quadrant a (FIG. 5B) is the parent, and that quadrants e, f, g, h (FIG. 6) are brothers and sisters and the offspring of ancestor a. Similar discussions can be made for quadrants b, c and d and the remaining area of FIG. 6.

FIG. 7 is an illustration of the paper **450** as it appears from the relative viewing position C (FIG. 4), with the paper **450** being subjected to an imaginary division into 64 areas. In a manner similar to the discussion above, note that areas s, t, w, x (FIG. 7) represent the same area of paper **450** as the quadrant h (FIG. 6). In effect, quadrant h has been enlarged (to show a higher degree of resolution) and divided into quadrants s, t, w, x. Thus, it can be said that quadrant a (FIG. 5B) is the grandparent, quadrant h (FIG. 6) is the parent, and quadrants s, t, w, x (FIG. 7) are the brothers and sisters and offspring of ancestors a and h.

As described previously, once FIGS. 5B, 6 and 7 are subjected to the imaginary divisions, the visual information in each area (or quadrant) is digitized and stored in a separate file. The 84 resulting files can be conceptually envisioned as the pyramidal structure shown in FIG. 8. In FIG. 8, dashed lines are utilized to show the lineage of the files just discussed.

FIG. 8 is further exemplary of one file naming operation which can be utilized with the present invention.

At the top of the pyramidal structure (FIG. 8), each of the four quadrant files is arbitrarily assigned a different character. A, B, C, D, (Note: The characters assigned are not critical with regard to the invention and hence it should be noted that any characters can be assigned, e.g., 0,1,2,3, etc.)

In moving down one level in the pyramidal structure,, the filenames for each of the respective files on the second level is increased to two characters.

In calculating the filenames, it is convenient to first divide the second level files into groups of four, according to parentage. To maintain a record of ancestry, the ancestor filename of each file is maintained as the first part of the filename. In determining the second part, the naming protocol which was utilized to name the quadrant files of the top level, is also utilized in naming the respective quadrant files on the second level. Thus, parent file A is shown as being related to descendent (i.e., brother and sister) files AA, AB, AC, AD. Similar discussion can be made for the remaining files along these two level.

A similar process can be utilized in providing the unique filenames to the third level files. At this level, the filenames consist of three characters. Again, the ancestor filename of each file would be maintained as a first filename part, in order to maintain a record of ancestry. In the example illustrated (FIG. 8), parent file AD is shown as being related to descendent (i.e., brother and sister) files ADA, ADB, ADC, ADD. Similar discussions can be made for the remaining files along these two levels, and furthermore, similar discussions can be made each time a pyramidal level is added.

From the above discussion, one should be able to realize that the above-described naming convention is particularly useful in programming a digital computer to move through the pyramidal file structure to access the appropriate data corresponding to varying degrees of resolution. More particularly, one should be able to realize that, since file names increase one character in length each time there is a downward movement through the pyramidal structure and the protocol for naming descendent files is known, the digital computer can be programmed to quickly and easily access the appropriate files for a smaller mapping area with a greater degree of resolution. Similarly, one should be able to realize that, since the filenames decrease one character in length each time there is an upward movement through the pyramidal structure, the digital computer can be programmed to quickly and easily access the appropriate files for a greater mapping area with a smaller degree of resolution.

The following example is believed to provide an increase in the understanding of the present invention.

In the example, it is assumed that the digital database corresponding to the three resolutions of the paper **450** (as shown in FIGS. 4, 5A–B, 6, 7) have been loaded to be accessible from the memory mass storage device, and furthermore, it is assumed that the mapping system is programmed to initially access and display a digital map corresponding to the digital mapping data in the files A, B, C, D (FIG. 8). Thus, the monitor (FIG. 9B) would display (in low resolution) the entire area enclosed within dashed portion **900** illustrated on the paper **450** (FIG. 9A). (Note: The reproduction of a digital map from digital data from several different files or sources is well-known in the art and is not the subject matter of the present invention.)

Suppose the user notices the dotted area on the low resolution map and wishes to investigate this area further. By using the appropriate keys (e.g. ←, ↑, →, ↓) and/or a mouse device, a user can give the mapping system an indication that he/she wishes to see the smaller area (i.e., quadrant A) at a higher degree of resolution. Upon receiving this preference, the mapping

system can use its knowledge of the file naming operations to quickly determine the names of the files which must be accessed. More specifically, using A as the parent file name and following the existing quadrant naming protocol the mapping system is quickly and easily able to calculate that it is files AA, AB, AC, AD which it needs to access. Once these files are accessed, the monitor in FIG. 10B displays (in higher resolution) the area enclosed within the dashed portion **1000** as illustrated on the paper **450** (FIG. 10A).

If a user is still not satisfied with the degree of mapping resolution, the user can again use the appropriate keys or mouse device to indicate that he/she wishes to see the smaller area (e.g., quadrant D; FIG. 10A) in a higher degree of resolution. In using AD as the parent filename and following the existing quadrant naming protocol, the mapping system is quickly and easily able to calculate that it is files ADA, ADB, ADC, ADD which it needs to access. Once these files are accessed, the monitor (FIG. 11B) displays (in higher resolution), the area enclosed within the dashed portion **1100** as illustrated on the paper **450** (FIG. 11A).

One skilled in the digital mapping and computer programming art should recognize that "scrolling" or "flying" to different lateral "relative viewing positions" to display a different lateral portion of the map is also provided by the present invention. Instead of adding or removing filename characters as in a change of resolution, in this instance, the mapping system must be programmed to keep track of the filenames of the current position and also, the orderly arrangement of filenames so that the appropriate filenames corresponding to the desired lateral position can be determined. As an example if the user desired to scroll to the right border of the paper **450**, the mapping system would respond by accessing and causing the monitor to display the digital maps corresponding to the following sequence of files: (Note: In this example, it is assumed that it takes 4 files to provide sufficient digital data to display a full digital map on a monitor) ADA, ADB, ADC, ADD; ADB, ADD, BCA, BCC; BCA, BCB, BCC, BCD; BCB, BCD, BDA, BDC; and BDA, BDB, BDC, BDD. If the user, then desired to scroll to the bottom (right corner) of the paper **450**, the mapping system would respond by accessing and causing the monitor to display the digital maps corresponding to the following files: BDA, BDB, BDC, BDD; BDC, BDD, DBA, DBB; DBA, DBB, DBC, DBD; DBC, DBD, DDH, DDB; DDA, DDB, DDC, DDD. In effect as all of the files in the above example correspond to the same level of resolution all these files (and any group of files which exist on the same level of resolution) can be taken as being related as cousins.

FIGS. 9A, 10A, 11A can also be used to illustrate the operation of moving toward the display of a larger mapping area with a lower degree of resolution.

Assume that after lateral "scrolling" or "flying", that the monitor is now displaying (not shown) a digital map corresponding to the enclosed area **1110** shown in FIG. 11A. (Note: at this position the mapping system is accessing and display a digital map corresponding to the digital data in the files DCA, DCB, DCC, DCD). Suppose the user now wishes to cause the "relative viewing position" to zoom upward, such that the monitor will display a larger portion of the paper **450** at a lower degree of resolution. By using the appropriate keys or a mouse device, the user indicates his/her preference to the mapping system. Upon receiving this preference, the mapping system is programmed to quickly determine the names of the files which must be accessed. More specifically, the mapping system is able to look at the first portion of the filenames currently being used (i.e., DCA, DCB, DCC, DCD), to immediately determine that these files have the ancestry DC, i.e., have a grandfather D and a parent DC. The mapping system then immediately determines brother and sister files of parent file DC as being DA, DB and DD. The mapping system then accesses these files and causes the monitor to display a digital map (not shown) corresponding to the enclosed portion **1010** (FIG. 10A) of the paper **450**.

Suppose the user again indicate a preference to cause the "relative viewing position" zoom upward. Upon receiving this preference, the mapping system again goes through a process similar to that discussed immediately above. However, this time the mapping system looks at the filenames currently being used (i.e., DA, DB, DC, DD) and determines that parent file D has brother and sister files A, B and C. The mapping system then immediately accesses these files and causes the monitor to display a digital map (FIG. 9B) corresponding to the enclosed portion **900** (FIG. 9A) of the paper **450**.

The text now turns to a description of the operation for assigning unique filenames in the currently preferred embodiment, i.e., in a digital mapping system which is implemented in a DOS operating system.

As anyone skilled in the computer art will know, every computer operating system has its own unique set of rules which must be followed. In an implementation of the present invention in a DOS operating system, the DOS rules must be followed. Since a critical feature of the present invention is the division of the digital mapping database into a plurality of files (each having a unique filename), of particular concern with the present invention is the DOS rules regarding the naming of filenames.

A DOS filename may be up to eight (8) characters long, and furthermore, may contain three (3) additional trailing characters which can represent a file specification. Thus, a valid DOS filename can be represented by the following form:

---

where "-" can be replaced by any ASCII character (including blanks), except for the following ASCII characters:

. " / \ [ ] : | < > + = ,

and ASCII characters below **20H**. The currently preferred embodiment stays within these DOS filename rules by using the file naming operations which are detailed below.

Because the assigned filenames will be seen to be related to hexadecimals, a useful chart containing the hexadecimal base and also a conversion list (which will be shown to be convenient ahead), is reproduced below:

Column 1	Column 2	Column 3
0000	0	G
0001	1	H
0010	2	I
0011	3	J
0100	4	K
0101	5	L
0110	6	M

-continued

Column 1	Column 2	Column 3
0111	7	N
1000	8	O
1001	9	P
1010	A	Q
1011	B	R
1100	C	S
1101	D	T
1110	E	U
1111	F	V

The first column contains a list of all the possible 4-bit binary combinations: the second column contains the hexadecimal equivalent of these binary numbers: and the third column concerns a "mutant-hex" conversions which will be shown to be important in the discussion to follow. In the operatidns to assign unique filenames for use in a DOS operating system, the present invention looks at each of the eight DOS filename characters as hexadecimal characters rather than ASCII characters. Hence, while the following discussion will center around determining unique filenames using hexadecimal (and "mutant-hexadecimal") characters, it should be understood in an actual DOS implementation, the hexadecimal filenames must be further converted into the equivalent ASCII characters such that the appropriate DOS file naming rules are followed.

At this point, it is also useful to note that the file naming operation of the preferred embodiment is not concerned with the trailing three character filename extension. However, it should be further noted that this three character filename extension may prove useful in specifying data from different sources, and allowing the different types of data to reside in the same database. As examples, the filename extension ".spm" might specify data from scanned paper maps, the filename extension ".si" might specify data from satellite imagery, the filename extension ".ged" might specify gridded elevation data, etc.

As a result of the foregoing and following discussions, it will be seen that the naming operation of the preferred embodiment is concerned only with a filename of the following form:

where each "~" represents a character which is a hexadecimal character within the character set of "0~9" and "A~F", or is a "mutant-hexadecimal" character within the character set of "G~V".

Several more important file naming details should be discussed.

First, it should be pointed out that the first four (4) filename characters is designated as corresponding to the "x" coordinate characters, and the last four (4) filename characters are designated as corresponding to the "y" coordinate characters.

Second, during the file naming operations, often it is necessary to convert the filename characters into the equivalent binary representation. As each hexadecimal character can be converted into a four bit binary number, it can be seen that the first four (4) filename characters (designated as "x" coordinate characters) can be converted into sixteen (16) binary bits designated as "x" bits, and similarly, that the last four (4) filename characters (designated as "y" coordinate characters) can be converted into sixteen (16) binary bits designated as "y" bits. As will become more apparent ahead, each of these sixteen (16) "x" and "y" bits corresponds to a filename bit which can be manipulated when assigning filenames at a corresponding magnitude or level of mapping resolution, e.g., the first "x" and first "y" bits correspond to filename bits which can be manipulated when assigning unique filenames at the first magnitude, the second "x" and second "y" bits correspond to filename bits which can be manipulated when assigning unique filenames at the second magnitude, etc.

Third, FIG. 12 corresponds to the naming protocols which are utilized to modify and relate a parent filename to four (4) quadrant filenames. Note that there is a two-bit naming protocol in each of the quadrant files. As will become more clear ahead, the first bit of each protocol determines whether the current "x" filename bit will be modified (i.e., if the first protocol bit is a "1", the current "x" filename bit is changed to a "1", and if first protocol bit is a "0", the current "x" filename bit is maintained as a "0"), and the second bit determines whether the current "y" filename bit will be modified (in a similar manner).

The text now turns to a file naming example which is believed to provide further teachings and clarity to the currently preferred file naming operation.

FIG. 13 is an illustration of a portion of the preferred digital data base, with the plurality of files (partially shown) being arranged in a conceptual pyramidal manner in a manner similar to that which was described with reference to FIG. 8. More specifically, there are shown four files **1300** having digital data corresponding to a first level or magnitude of mapping resolution, sixteen files **1310** having digital data corresponding to a second level or magnitude of mapping resolution, sixty-four files **1320** having digital data corresponding to a third level or magnitude of mapping resolution, and a partial cut-away of a plurality of files **1330** having data corresponding to a fourth level or magnitude of mapping resolution. Although not shown, it is to be understood that, in the preferred embodiment, additional pyramidal structure corresponding to levles magnitudes five through sixteen similarly exist. As examples of the file naming operation, filenames will now be calculated for the files which essentially occupy the same positions as the files which were outlined in FIG. 8.

We begin with the initializing eight (8) character filename:

0	0	0	0	0	0	0	0

which can be converted to the binary equivalent:

0000	0000	0000	0000	0000	0000	0000	0000

This binary representation is the basic foundation which will be used to calculate all of the filenames for the files on the first level (**1300**). Note, that the first and last four filename characters, and the first and last sixteen bits are slightly separated in order to conveniently distinguish the "x" and "y" coordinate characters and bits. Both the first (leftmost) "x" bit and the first (leftmost) "y" bit are the bits which can be manipulated in assigning a unique filename to the files on the first level.

File naming begins with the first (upper-rightmost) file on the first level **1300**. The naming protocol assigned to this quadrant file is the two-bit protocol "10".

As the first protocol bit is a "1", this means that the current "x" bit must be changed to a "1". As the second protocol bit is a "0", this means that the current "y" bit is maintained as a "0". As a result of the foregoing, the first (upper-rightmost) file is assigned the filename having the binary equivalent of:

1000	0000	0000	0000	0000	0000	0000	0000

which can be converted to the hex characters:

8	0	0	0	0	0	0	0.

In proceeding clockwise, next is the second (lower-rightmost) file on the first level **1300**. The naming protocol assigned to this quadrant file is the two-bit protocol "11". As the first protocol bit is a "1", the current "x" bit is changed to a "1": similarly, as the second protocol bit is a "1", the current "y" bit is changed to a "1". As a result of the foregoing, the second (lower-rightmost) file is assigned the filename having the binary equivalent of:

1000	0000	0000	0000	1000	0000	0000	0000

which can be converted to the hex characters:

8	0	0	0	8	0	0	0.

Continuing clockwise, next is the third (lower-leftmost) file on the first level **1300**. The naming protocol assigned to this quadrant file is the two-bit protocol "01". As the first protocol bit is a "0", the current "x" bit is maintained at 0. As the second protocol bit is a "1", the current "y" bit is changed to a "1". As a result of the foregoing, the third (lower-leftmost) file is assigned the filename having the binary equivalent of:

0000	0000	0000	0000	1000	0000	0000	0000

which can be converted to the hex characters:

0	0	0	0	8	0	0	0.

Finally, there is the fourth (upper-leftmost) file on the first level **1300**. The naming protocol assigned to this quadrant is the two-bit protocol "00". As neither of the protocol bits is a "1", it can be easily seen that neither of the current "x" and "y" bits changes, and hence, the fourth (upper-leftmost) file is assigned the filename having the binary equivalent of:

0000	0000	0000	0000	0000	0000	0000	0000

which can be converted to the hex characters:

0	0	0	0	0	0	0	0.

In further discussions of the example, it is important to note that the initializing (8) character filename of 0000 0000 (which was utilized to calculate the filenames of the files on the first level 1300) is not utilized in assigning filenames on subsequent levels. In naming files from the second level or magnitude downward, the binary equivalent of the parent file's name is utilized as the foundation from which the descendent file's name is derived. It is only coincidental that the filename of the parent file 00000000 (located in the user-left most corner of the first level **1300**) is the same as the initializing filename. Use of the parent's filename to calculate the descendent's filename will become more readily apparent ahead in the example.

In continuing the file naming example, the fourth (upper-leftmost) file (having filename 00000000) in the first level **1300** can be viewed as being the parent file of the four (highlighted) quadrant files in the second level **1310**. As stated above, the binary equivalent of parent file's 00000000 name is utilized as the foundation for calculating the descendent file's filenames. At this second level or magnitude, the second "x" and "y" bits from the left in the parent's binary filename are taken as the "current" bits which can be manipulated to provide a unique filename for the descendent files.

As the calculation of the filename for the fourth (upper-leftmost) file of the second level **1310** illustrates a very important modification in the file naming operation, the example will first continue with discussions corresponding to this file.

As the naming protocol assigned to the fourth (upper-leftmost) file of the second level **1310** is two-bit protocol "00", it can be seen that neither of the current "x" and "y" bit would be changed. Hence the parent's filename 00000000 is unchanged, and is attempted to be adopted as the descendent's filename. However, note that this is extremely undesirable as the operation of the present invention is based on assigning each data file a unique filename, and furthermore, a DOS operation system will not allow the same filename to be assigned to two different files. To avoid this clash, the preferred file naming operation of the present invention incorporates a further step which can be detailed as follows:

First calculate the filename as explained above. Once the binary filename is obtained, convert to the eight character hexadecimal equivalent.

Next, take the decimal number of the current level or magnitude and subtract one (1) to result in a decimal magnitude modifier. Convert the decimal magnitude modifier into a four-bit binary magnitude modifier, and line these four bits up with the four hexadecimal "x" filename characters. Whenever a "1" appears in the binary magnitude modifier, the corresponding aligned "x" filename character is converted to a "mutant-hexadecimal" character, i.e., a decimal 16 value is added to convert the aligned filename character into a one of the "mutant-hexadecimal" characters in the character set of "G-V".

Conversions from a hexadecimal character to a "mutant-hexadecimal" character can be most readily made using the chart detailed above. As an example, if decimal 16 is added to the hex character "0" (Column 2), there is a conversion to the "mutant-hexadecimal" character "G" (Column 3). Similarly, if decimal 16 is added to the hex character "1" (Column 2), there is a conversion to the "mutant-hexadecimal" character "H" (Column 3). Similar discussion can be made for the remaining hex and "mutant-hexadecimal characters in the chart.

Once correspondingly aligned filename characters are converted to "mutant-hexadecimal", the resultant

eight (8) characters correspond to the file's unique filename.

The above processing will now be applied to the fourth (upper-rightmost) file of the second level **1310** (which was recently discussed above). The resultant binary filename:

0000	0000	0000	0000	0000	0000	0000	0000

is converted to the hex characters:

0	0	0	0	0	0	0	0.

The level or magnitude two (2) minus one (1) results in a decimal magnitude modifier of one (1). The decimal magnitude modifier is converted to the four-bit binary equivalent and is aligned with the "x" filename characters above, as follows:

0	0	0	1

Only the fourth bit of the binary magnitude modifier is a "1", so only the fourth "x" filename character needs to be converted to "mutant-hexadecimal". From the chart, the hexadecimal character "0" is shown to convert to a "mutant-hexadecimal" character "G". Thus, the unique filename which is assigned to the fourth (upper-leftmost) file of the second level **1310**, is:

0	0	0	G	0	0	0	0.

In continuing the example to calculate the filename for the first (upper-right-quadrant) file of the second level **1310**, it can be seen that this file is assigned the two-bit naming protocol "10". The first protocol bit is a "1" which indicates that the current (second from the left) "x" bit of the parent file's binary filename must be changed to a "1", In contrast, the second protocol bit is a "0", which indicates that the current (second from the left) "y" bit is maintained as "0" Thus the parent filename:

0000	0000	0000	0000	0000	0000	0000	0000

is converted to:

0100	0000	0000	0000	0000	0000	0000	0000

which results in the hex characters:

4	0	0	0	0	0	0	0.

The level or magnitude two (2) minus one (1) results in a decimal magnitude modifier of one (1). The decimal magnitude modifier is converted to the four-bit binary equivalent and is aligned with the "x" filename characters above, as follows:

0	0	0	1

Only the fourth bit of the binary magnitude modifier is a "1", so only the fourth "x" filename character needs to be converted to "mutant-hexadecimal". From the chart, the hexadecimal character "0" is shown to convert to a "mutant-hexadecimal" character "G". Thus, the unique filename which is assigned to the first (upper-right-quadrant) file of the second level **1310**, is:

4	0	0	G	0	0	0	0.

Turning now to the second (lower-right-quadrant) file, this file is assigned the two-bit naming protocol "11". The first protocol bit is a "1" which indicates that the current (second from the left) "x" bit of the parent file's binary filename must be changed to a "1", and similarly, the second protocol bit is a "1", which indicates that the current (second from the left) "y" bit of the parent file's binary filename must be changed to a "1" Thus the parent filename:

0000	0000	0000	0000	0000	0000	0000	0000

is converted to:

0100	0000	0000	0000	0100	0000	0000	0000

which results in the hex characters:

4	0	0	0	4	0	0	0.

The level or magnitude two (2) minus one (1) results in a decimal magnitude modifier of one (1). The decimal magnitude modifier is converted to the four-bit binary equivalent and is aligned with the "x" filename characters above, as follows:

0	0	0	1

Only the fourth bit of the binary magnitude modifier is a "1", so only the fourth "x" filename character needs to be converted to "mutant-hexadecimal". From the chart, the hexadecimal character "4" is shown to convert to a "mutant-hexadecimal" character "G". Thus, the unique filename which is assigned to the second (lower-right quadrant) file of the second level **1310**, is:

4	0	0	G	4	0	0	0.

In applying the above operations to the third (lower-left-quadrant) file of the second level **1310**, it can be easily calculated that the resultant filename is:

4	0	0	G	4	0	0	0.

The example of the file naming operation is further extended to the third level or magnitude, as this example is illustrative of both the use of the parent file's binary filename to calculate the descendent's filename, and the removal of "mutant-hexadecimal" conversions before calculating the descendent's filename.

In FIG. 13. the third (lower-right-quadrant) file of the second level **1310** is shown as being the parent of the four (4) quadrant files highlighted in the third level or magnitude **1320**.

The discussion centers on the calculation of the unique filename for the second (lower-right-quadrant) file in the third level **1320**. Before the parent filename can be used as the foundation for calculating the descendent's filename. all "mutant-hexadecimal" conversions must be removed. Thus the parent filename:

4	0	0	G	4	0	0	0.

is converted back to:

4	0	0	0	4	0	0	0.

which is further converted to the binary equivalent:

0100	0000	0000	0000	0100	0000	0000	0000

In continuing the calculation, this second (lower-right-quadrant) file is assigned the two-bit naming protocol "11". The first protocol bit is a "1" which indicates that the current (third from the left) "x" bit of the parent file's binary filename must be changed to a "1", and similarly, the second protocol bit is a "1", which indicates that the current (third from the left) "y" bit of the parent file's binary filename must be changed to a "1". Thus the parent filename:

0100	0000	0000	0000	0100	0000	0000	0000

is converted to:

0110	0000	0000	0000	0110	0000	0000	0000

which results in the hex characters:

6	0	0	0	6	0	0	0.

The level or magnitude three (3) minus one (1) results in a decimal magnitude modifier of two (2). The decimal magnitude modifier is converted to the four-bit binary equivalent and is aligned with the "x" filename characters above, as follows:

0	0	1	0

Only the third bit of the binary magnitude modifier is a "1", so only the third "x" filename character needs to be converted to "mutant-hexadecimal". From the chart, the hexadecimal character "0" is shown to convert to a "mutant-hexadecimal" character "G". Thus, the unique filename which is assigned to the second (lower-right-quadrant) file of the third level **1320**, is:

6	0	G	0	6	0	0	0.

The filenames for several additional third level files will be given to give the patent reader further practice.

In applying the above operations to the first (upper-right-quadrant) file of the third level **1320**, it can be easily calculated that the resultant filename is:

6	0	G	0	4	0	0	0.

In applying the above operations to the third (lower-left-quadrant) file of the third level **1320**, it can be easily calculated that the resultant filename is:

4	0	G	0	6	0	0	0.

Finally, in applying the above operations to the fourth (upper-left-quadrant) file of the third level **1320**, it can be easily calculated that the resultant filename is:

4	0	G	0	4	0	0	0.

As a result of all of the foregoing teachings, one skilled in the art should now be able to calculate the filename of any other of the 1.4 billion files which would be required to provide digital maps corresponding to sixteen (16) resolutions of any geographical area on earth. Furthermore, once a file is being accessed, by understanding the rules and operations of the file naming operation one skilled in the are should be able to calculate any other related files, i.e., parent files. and brother/sister/cousin files.

While the unique approach for storing and accessing files in the pyramidal file structure has been particularly pointed out. further discussion is needed as to an additional advantageous feature of the present invention.

As mentioned previously, the creation of a digital database is a very tedious, time consuming and expensive process. Tremendous bodies of mapping data are available from many important mapping authorities, for example, the U.S. Geological Survey (USGS), Defense Mapping Agency (DMA), National Aeronautics and Space Administration (NASA), etc.

The maps and mapping information produced by the above recited agencies, is always based on well established mapping area divisions. As a few examples, the Defense Mapping Agency (DMA) produces maps and mapping information based on the following mapping areas: GNC maps which are 2°×2°: JNC maps which are 1°×1°; ONC maps which are 30'×30': TPC maps which are 15'×15'; and JOG maps which are 7.5'×7.5'. As a further example, the U.S. Geological Survey (USGS) also produces maps and utilizes mapping information based on 15'×15' and 7.5'×7.5'.

In terms of both being able to easily utilize the mapping data produced by these agencies, and represent an attractive mapping system to these mapping agencies, it would be highly desirable for the mapping system of the present invention to be compatible with all of the mapping formats used by these respective agencies. Such is not the case when the mapping database is based on a graticule system corresponding to 360°

If one were to apply multiple quadrant divisions to the 360°×180° flat map projection of the earth (FIG. 1). one would result in the following mapping area subdivisions:

Level of quadrant div.:	Resultant mapping area:
1	(4) 180° × 90°
2	(16) 90° × 45°
3	(64) 45° × 22.5°
4	(256) 22.5° × 11.25°
5	(1024) 11.25° × 5.625°
etc.	

Note that these mapping area subdivisions are very awkward, and do not match any of the well settled mapping area subdivisions. (It should be further noted that no better results are obtained if the initial map projection is imagined as being a 360°×360° square instead of a rectangle.)

In order to avoid these awkward mapping subdivisions, and result in quadrant divisions which precisely match widely used mapping area subdivisions, the present invention utilizes a unique initial map projection.

More specifically, as can be seen in FIG. **14**, the present invention initially begins with a unique 512°×512° initial map projection. Shown centered in the 512°×512° map projection is the now familiar 360°×180° flat projection of the surface of the earth. Although the 512°×512° projection initially appears awkward and a waste of map projection space, the great advantages which are resultant from the use of this projection will become more apparent in the discussions to follow.

To aid in this discussion, provided on the next page is a chart which details these important advantages as well as other useful information regarding the use of this map projection.

less complicated, the non-DOS file naming operation will be used in the discussion.

The digital mapping of the earth surfaces begins in FIG. **14**. The visual perception of the earth surfaces is experienced as being centered, and occupying only a portion of the 512°×512° projection. A first quadrant division is applied to result in four equal 256°×256° mapping areas. The visual information in each of the areas is digitized, and stored in a separate file, Thus, it can be seen that one would have to access four files a, b, c, d in order to reproduce a digital map corresponding to the earth surfaces as viewed from this "relative viewing position."

One skilled in the art, might, at this point, wonder if the massive blank portions of the 512°×512° projections result in large blank portions on the digital map display. The preferred embodiment avoid this phenomena, through a simple watchdog operation, i.e., the computer is programmed to keep track of longitudinal and latitudinal movements from an initial position of 0° longitude and 0° latitude, and the computer does not allow scrolling of the monitor display beyond 90° north or south.

As to side to side movements, the computer allows scrolling beyond 180° east or west by patching the appropriate data files together to perform a "wrap around" operation. Note that, with the knowledge of the logical file naming operation, the computer can quickly and easily calculate the appropriate files to access.

Before moving to the next level or magnitude of mapping resolution, it is beneficial to note the correspondence between our findings and the enties in the

MAGNITUDE EQUIVALENCY CHART FOR DELORME PROJECTION
Chart assumes 69 statute miles per degree at equator

MAG-NI-TUDE	Window Size without overlap	Ht of window statute miles	Ht of window kilometers	# Windows per MAG	# Windows/ MAG w/polar com-pression	Pixel resolution 480 monitor (ft)	Data reso-lution (ft) 1024-based window	Equivalent Paper Map Scales	Size of paper map image at equator (in)
1	256° × 256°	17664	28421	4	4		91080		
2	128° × 128°	8832	14211	8	8		45540		
3	64° × 64°	4416	7105	24	24	48576	22770	1:100 million	2.8 × 2.8
4	32° × 32°	2208	3553	72	72	24288	11385	1:50 million	2.8 × 2.8
5	16° × 16°	1104	1776	288	288	12144	5693	1:30 million	2.3 × 2.3
6	8° × 8°	552	888	1152	858	6072	2846	1:16 million	2.2 × 2.2
7	4° × 4°	276	444	4232	3432	3036	1423	1:10 million	1.7 × 1.7
8	2° × 2°	138	222	16200	12808	1518	712	1:5 million	1.7 × 1.7
9	1° × 1°	69	111	64800	51210	759	356	1:2 million	2.2 × 2.2
10	30' × 30'	34.5	55.5	259000	204840	380	178	1:1 million	2.2 × 2.2
11	15' × 15'	17.25	27.8	1036800	813600	190	89	1:500,000	2.2 × 2.2
12	7.5' × 7.5'	8.625	13.9	4147200	3277440	95	44	1:250,000	2.2 × 2.2
13	3.75' × 3.75'	4.312	6.9	16588800	13109760	47.4	22	1:125,000	2.2 × 2.2
								1:100,000	2.73 × 2.73
								1:80,000	3.4 × 3.4
14	1.875' × 1.875'	2.156	3.5	66355200	52439040	23.7	11.1	1:62,500	2.2 × 2.2
								1:50,000	2.73 × 2.73
								1:40,000	3.4 × 3.4
15	0.9375' × 0.9375'	1.078	1.7	265420800	209756160	11.9	5.6	1:24,000	2.8 × 2.8
								1:20,000	3.4 × 3.4
16	0.46875' × 0.46875'	0.539	0.9	1061683200	839024640	5.9	2.8	1:12,000	2.8 × 2.8

The best way to see the advantages of the 512°×512° mapping projection, is to use it with the previously, taught, quadrant division and pyrimidal file structure to show how this unique mapping projection can provide digital maps of any geographical areas of the earth, with 16 levels or magnitudes of resolution. As it is slightly

above-indicated chart.

In looking at the left-most column, and tracing down to magnitude **1**, note that the 256°×256° window size exactly matches our determination. Furthermore, note that our findings is also in agreement with the number of widows i.e., 4. It is also interesting to note from the third column, that the height or "relative viewing posi-

tion" of this magnitude or level would be 17, 664 statute miles above the earth's surface.

Turning now to the second level or magnitude of resolution (FIG. 15). a further quadrant division is applied, resulting in sixteen (16) mapping areas of 128°×128°. The respective filenames which are assigned to each of the mapping areas is shown. In viewing FIG. 15, note that there are eight (8) mapping areas which are not intersected by the earth's surface. In order to save valuable memory space, the preferred embodiment will ignore, and in fact will never create these files. Note that there is no use for these files as they do not contain any digital mapping data nor will they ever have any descendents which hold mapping data. In order to implement this "file selectivity", the preferred embodiment again utilizes a watchdog approach. More specifically, as the computer already knows the degree (°) size of the earth's surface and the degree (°) size of each of the mapping areas (i.e., at each level or magnitude of resolution), it can be seen that the computer can easily calculate the filenames which will not intersect the earth's surface.

Again it is useful to correspond our findings with the entries in the chart.

Our findings are substantiated, as, at a magnitude of 2, the window size is shown as being 128°×128°, and there are shown to be eight (8) pertinent windows or files at this magnitude. Again, it is interesting to note that the height or "relative viewing position" of this window would be 8,832 statute miles above the earths earth's surface.

It is important to note that, although the "relative viewing position" of each level or magnitude is moving closer to the earth, the visual perception of the earth (as seen in FIGS. 14-19 is not illustrated as getting larger with a greater degree of detail. This is because or the paper size limitations.

In the third level or magnitude of resolution (FIG. 16). a further quadrant division is applied, resulting in sixty-four (64) mapping areas of of 64°×64°. As the projection is beginning to represent a large plurality of mapping areas, the filenames have been ommitted. However, it should be understood that the filename assigned to a respective file in this and subsequent degrees of resolution, can easily be calculated by following the previously described file naming operation. In this projection, it can be seen that 40 mapping areas or files are not used, resulting in 24 files which contain the digital mapping data of this resolution. Note that the observed window, and used files again correlates to the entries in the chart. Furthermore, it can be seen that the height or "relative viewing position" is at 4,416 statute miles above the earth.

Further quadrant divisions and the corresponding data can be seen in the FIGS. 17-19 and the chart. From the foregoing discussions, prior teachings, and data from the chart, one skilled in the art should be able to quickly appreciate that a mapping system can be constructed which can provide digital maps corresponding to a plurality of resolutions, of any geographical area of the world.

The chart can now be used to observe the tremendous advantage provided by the 512°×512° projection. In the second column of the chart, one can view the sizes of the mapping area divisions which are produced as a result of the continued quadrant division of the 512°×512° projection. One skilled in the mapping art will be able to fully appreciate that the resultant mapping area divisions exactly correspond to well settled and widely used mapping area formats.

Having described all of the important operations of the present invention, the following further conclusions, comments and teachings can be made.

With the mapping system of the present invention, the mapping data are structured at each magnitude or level into windows, frames or tiles representing subdivisions or partitions of the surface area at the specified magnitude. The windows, frames or tiles of all magnitudes for whatever resolution are structured to receive substantially the same amount or quantity of mapping data for segmented visual presentation of the mapping data by window.

As a further improvement, the lapping system of the present invention can further store and organize mapping data into attributed or coded geographical and cultural features according to the classification and level or resolution or magnitude for presentation on the map display. Several examples of this was previously discussed with regard to the use of the filename extension. If this further improvement is used, the computer can be programmed and arranged for managing and accessing the mapping data, and excluding or including coded features in tiles of a particular magnitude according to the resolution and density of mapping data appropriate to the particular magnitude of the window. The selective display of attributed geographical and cultural features according to resolution maintains or limits the mapping data entered in each tile to no greater than a specified full complement of mapping data for whatever magnitude.

In reviewing the file naming operations which were described, one can see that the global map generating system data base structure relates tiles of the same magnitude by tile position coordinates that are keyed to the control corner of each tile and maintained in the name of the "tile-file". Continuity of same scale tiles is maintained during scrolling between adjacent or neighboring tiles in any direction. The new data base structure also relates tiles of different magnitudes by vertical lineage through successive magnitudes. Each tile of a higher magnitude and lower resolution is an "ancestor tile" encompassing a lineage of "descendant tiles" of lower magnitude and high resolution in the next lower magnitude. Thus the present invention permits accessing, displaying and presenting the structured mapping data by tile, by scrolling between adjacent or neighboring tiles of different magnitude in the same vertical lineage for varying the resolution.

In its simplest form the coordinate system is Cartesian, but the invention contemplates a variety of virtual tile manifestations of windowing the mapping data at each magnitude: for example: tilting the axes; scaling one axis relative to another; having one or both axes logarithmic; or rendering the coordinate space as non-Euclidean all together.

When dealing with vector or point information and gridded data, the most common method is to describe individual points as an x-y offset from the control corner of the tile. In this way the mapping data exist as pre-processed relative points on a spherical surface in a de-projected space. The mapping data can then be projected at the user interface with an application program. When projected, all data ultimately represent points of latitude and longitude. Tiles may also contain mapping data as variable offsets of arc in the x and y directions. The tile header may carry an internal descriptor defin-

ing what type of mapping data is contained. The application or display program may then decode and project the data to the appropriate latitude or longitude positions.

The map generating system contemplates storing analog mapping data in electronic mapping frames in which the raw analog data would be scanned and converted digitally to the tile structure and then later accessed and projected for the purpose of displaying continuous analog mapping data.

In the preferred example embodiment, the digital mapping data are structured by window or tile in a substantially rectangular configuration encompassing defined widths and heights in degrees of latitude and longitude for each magnitude. The mapping data representing each magnitude or level are stored in a deprojected format according to mapping on an imaginary cylindrical surface. For display of the maps, however, the data base manager accesses and presents the tiles in a projected form, according to the real configuration of the mapped surface, by varying the aspect ratio of latitude to longitude dimensions of the tiles according to the absolute position of the window on the surface area.

For example, for a spherical or spheroidal globe having an equator and poles, such as the earth, the mapping data are accessed and displayed by aspecting or narrowing the width in the west-east dimension of the tiles of the same magnitude, while scrolling from the equator to the poles. This is accomplished by altering the width of the tile relative to the height. In the graphics display of each window or tile on the monitor, the tiles are presented essentially as rectangles having an aspect ratio substantially equal to the center latitude encompassed by the tile. Thus, the width of the visual display windows is corrected in two respects. First, the overall width is corrected by aspecting to a narrower width, during scrolling in the direction of the poles, and to a wider width during scrolling in the direction of the equator. Second, the width of the tile is averaged to the center latitude width encompassed by the tile throughout the tile height to conserve the rectangular configuration. Alternatively, or in addition, further compensation may be provided by increasing the number of degrees of longitude encompassed by the tiles during scrolling from the equator to the poles to compensate for the compound curvature of the globe.

A feature and advantage of this new method and new system of map projection are that the dramatic and perverse distortion of the globe near the poles, introduced by the traditional and conventional Mercator projection is substantially eliminated. According to the invention, the compensating aspect ratio of latitudinal to longitudinal dimension of aspecting is a function of the distance from the equator, where the aspect ratio is one, to the poles where the aspect ratio approaches zero, all as described for example in Elements of Cartography, 4th edition, John Wiley & Sons (1978) by Arthur Robinson, Randall Sale and Joel Morrison.

The new system contemplates "polar compression" (FIG. 20) in the following manner. Starting at 64 degrees latitude, the width of each tile doubles for every eight degrees of latitude. From 72 degrees to 80 degrees latitude, there are 4 degrees of longitude for 1 degree of latitude. From 80 degrees to 88 degrees latitude, it becomes eight to one, and from 88 degrees to the pole (90 degrees) it becomes 16 to one (see illustration of polar compression). (FIG. 20)

Another feature and advantage of the way in which the new map system and new projection handle polar mapping data are in the speed required to access and display polar data. The new polar compression method drastically minimizes tile or window seeks and standard I/O time. Also, without compressing the poles, the Creation/Edit Software would have to work on increasingly narrow tiles as the aspect ratio approached zero at the poles.

The invention embodies an entirely new cartographic organization for an automated atlas of the earth or other generally spherical or spheroidal globe with 360 degrees of longitude and 180 degrees of latitude, an equator and poles. The digital mapping data for the earth is structured on an imaginary surface space having 512 degrees of latitude and longitude. The imaginary 512 degree square surface represents the zero magnitude or root node at the highest level above the earth for a hierarchial type quadtree data base structure. In fact, the 512 degree square plane at the zero magnitude encompasses the entire earth in a single tile. The map of the earth, of course, fills only a portion of the root node window of 512 degrees square, and the remainder may be deemed imaginary space or "hyperspace".

In the preferred example embodiment from a zero magnitude virtual or imaginary space 512 degrees square, the data base structure of the global map generating system descends to a first magnitude of mapping data in four tiles, windows or quadrants, each comprising 256 degrees of latitude and longitude. Each quadrant represents mapping data for one-quarter of the earth thereby mapping 180 degrees of longitude and 90 degrees of latitude in the imaginary surface of the tile or frame comprising 256 degrees square, leaving excess imaginary space or "hyperspace". In the second magnitude, the digital mapping data are virtually mapped and stored in an organization of 16 tiles or windows each comprising 128 degrees of latitude and longitude.

The map generating system supports two windowing formats, one based on the binary system of the 512 degree square zero magnitude root node with hyperspace and the other based on a system of a 360 degree square root node without hyperspace. A feature and advantage of the virtual 512 degree data base structure with hyperspace are that the tiles or windows to be displayed at respective magnitudes are consistent with conventional mapping scale divisions, for example, those followed by the U.S. Geological Survey (USGS), Defense Mapping Agency (DMA), National Aeronautics and Space Administration (NASA) and other government mapping agencies. Thus, typical mapping scale divisions of the USGS and military mapping agencies include scale divisions in the same range of 1 deg, 30 minutes, 15 minutes, 7.5 minutes of arc on the earth s surface. This common subdivision of mapping space does not exist in a data structure based on a 360 degree model without hyperspace (see chart).

Thus, according to the present invention, the world is represented in an assemblage of magnitudes, with each magnitude divided into adjacent tiles or windows on a virtual or imaginary two-dimensional plane or cylinder. At higher magnitudes the quadtree tiles of mapping data do not fill the imaginary projection space. However, from the seventh magnitude down, the mapping data fills a virtual closed cylinder, and no hyperspace exists at these levels.

In the preferred example embodiment the invention (running on a 16 bit computer) has sixteen magnitudes

or levels (with extensions to 20 levels) representing sixteen altitudes or distances above the surface of the earth. At the lowest (16th) magnitude of highest resolution and closest to the earth, the data base structure contains over one billion tiles or windows (excluding hyperspace), each encompassing a tile height of approximately one half statute mile. At this level of resolution, one pixel on a monitor of 480 pixels in height represents approximately 6 feet on the ground. Mapping data are positioned within each tile using a 0 to 1023 offset coordinate structure, resulting in a data resolution of approximately 3 feet at this level of magnitude (see chart). The contemplated 20th magnitude tile or window height is approximately 175 feet, which results in a pixel resolution of about 4 inches on a monitor of 480 pixels in height and a data resolution of about 2 inches, when utilizing the 0 to 1023 offset coordinate structure. Alternatively, the map-generating system contemplates an extended offset from 10 bits (0 to 1023) to an offset of 16 bits (0 to 65,535). In this case, the extended 20th magnitude results in a data resolution of 3 hundredths of an inch.

For still more resolution, the map generating system contemplates 32 magnitudes on a 32 bit computer and representing 32 altitudes or distances about the surface of the earth. Each level of magnitude may define mapping data within each tile using a 32 bit offset coordinate structure, thereby giving relative mathematical accuracy to a billionth of an inch. In all practicality, 20 separate magnitudes or levels are more than sufficient to carry the necessary levels of resolution and accuracy.

The new invention provides users with the ability graphically to view mapping data from any part of the world-wide data base graphically on a monitor, either by entering coordinates and a level of zoom (or magnitude) on the keyboard, or by "flying" to that location in the "step-zoom" mode using consecutive clicks of the mouse or other pointing device. Once a location has been chosen (this point becomes the user-defined screen center), the mapping software accesses all adjacent tiles needed to fill the entire view window of the monitor and, then, projects the data to the screen. Same scale scrolling is accomplished by simply choosing a new screen center and maintaining the same magnitude.

Vertical zooming up or down is accomplished by choosing another magnitude or level from the menu area with the pointing device or by directly entering location and magnitude on the keyboard. An advantage of this vertical lineage of tiles organized in a quadtree structure is that it affords the efficient and easily followed zooming continuity inherent in the present invention. Further discussion of such quadtree data organization is found in the article, "The Quadtree and Related Hierarchical Data Structures", by Hannan Samet, Computer Surveys, Volume 16 , No. 2, (June 1984), Pages 187 et seq.

The map-generating system also supports many types of descriptive information such as that contained in tabular or relational data bases. This descriptive information can be linked to the mapping data with a latitude and longitude coordinate position but may need to be displayed in alternate ways. Descriptive information is better suited for storage in a relational format and can be linked to the map with a "spatial hook".

In summary, the present invention provides a new automated world atlas and global map generating system having a multi-level hierarchial quadtree data base structure and a data base manager or controller which permits scrolling, through mapping tiles or windows of a particular magnitude, and zooming between magnitudes for varying resolution. While the data base organization is hierarchial between levels or magnitudes, it is relational within each level, resulting in a three dimensional network of mapping and descriptive information. The present invention also provides a new mapping projection that has similarities to the Mercator projection but eliminates drastic distortions near the poles for the purpose of presentation through a method of "aspecting" tile widths as a function of the latitudinal distance from the equator.

While the invention has been particularly shown and described with reference to the preferred embodiment thereof, it will be understood by those skilled in the art that various changes in form and details of the device and the method may be made therein without departing from the spirit and scope of the invention.

What is claimed is:

1. A computer implemented method for generating, displaying and presenting an electronic map from digital mapping data for a surface area having geographical and cultural features, said method comprising the steps of:

organizing the mapping data into a hierarchy of a plurality of successive magnitudes or levels for presentation of said mapping data with variable degrees of mapping resolution, each magnitude for presentation of said mapping data with a different degree of mapping resolution from a first or highest magnitude with lowest resolution to a last or lowest magnitude with highest resolution;

structuring said mapping data at each magnitude into a plurality of windows, frames or files representing subdivisions or partitions of said surface area, said windows of a respective magnitude including mapping data which are appropriate to a degree of mapping resolution being afforded at said magnitude while excluding mapping data which are not appropriate to said degree of mapping resolution, and at least a portion of said windows of each magnitude being structured to receive substantially a same predetermined amount or quantity of mapping data for segmented presentation of the mapping data by window;

organizing said mapping data into records of geographical or cultural features for presentation within said windows, and coding said features;

managing said mapping data for each window by excluding or including coded features appropriate to the degree of mapping resolution and density being afforded by said window, such that a quantity of mapping data entered in each window is no greater than said predetermined amount;

relating windows of a same magnitude by window position coordinates or names and structuring said windows with overlap or mapping data between adjacent or neighboring windows of a magnitude or achieve display continuity during generation, display and presentation of an electronic map;

relating windows of different magnitude by vertical lineage through successive magnitudes, each window of a higher magnitude and lower resolution being an ancestor window being related to a plurality of descendant windows of lower magnitude and higher resolution in a next lower magnitude;

accessing and displaying or presenting mapping data for different positions of a selected magnitude by

scrolling between adjacent or neighboring windows of a same magnitude in predetermined north, south, each and west directions;

and accessing and displaying or presenting mapping data for different selected magnitudes having different resolutions by zooming between windows of different magnitudes in a same vertical lineage.

2. The method of claim 1 further comprising:

organizing said mapping data of said surface area by degrees of latitude and longitude;

structuring each said window of mapping data to represent a substantially rectangular surface area configuration encompassing defined degrees of latitude and longitude for each magnitude, and storing the mapping data for each magnitude in a vertical Mercator projection format;

accessing and presenting said windows of mapping data in a corrected or compensated projection format departing from said Mercator projection format according to a real configuration of said surface area, by varying an aspect ratio of latitude to longitudinal dimensions of each window according to a coordinate position of said window with respect to a coordinate layout of said surface area.

3. The method of claim 2 wherein said surface area comprises a spherical or spheroidal globe having an equator and poles, said method comprising the further steps of:

accessing and presenting mapping data in a corrected projection format by aspecting or narrowing, in a direction from an equator to pole, the width or latitudinal dimension of windows, of a same magnitude, which encompass the same number of degrees of latitude and longitude;

and periodically increasing a number of degrees of longitude encompassed by said windows in said direction from equator to pole to compensate for compound curvature of said globe.

4. The method of claim 1 wherein said surface area comprises a generally spherical or spheroidal globe with 360 degrees of longitudinal, 180 degrees of latitude and an equator and poles, said method comprising the further steps of:

relating windows of different magnitudes by vertical lineage in a hierarchical quadtree database structure, by successively partitioning or subdividing ancestor windows of a vertical lineage into four descent windows or quadrants at a next lower magnitude or level, and incorporating additional records of features in said descendant windows to incorporate mapping data for a next higher resolution.

5. The method of claim 4 wherein said hierarchical quadtree database structure comprises at least sixteen degrees of magnitudes or levels.

6. The method of claim 4 comprising the further steps of:

mapping and storing mapping data for said globe in a virtual Mercator projection format representing an imaginary surface having 512 degrees of longitude and latitude comprising a zero magnitude or root node of said hierarchical quadtree database structure;

mapping and storing a first degree or highest magnitude of mapping data in four windows or quadrants each comprising 256 degrees of longitude and latitude, each window of said first degree of magnitude comprising mapping data for one quarter of said globe thereby mapping 180 degrees of surface area longitude and 90 degrees of surface area latitude in said imaginary surface of 256 degrees of longitude and latitude and leaving excess imaginary space;

mapping and storing a second degree of magnitude of mapping data in sixteen windows each comprising 128 degrees of longitude and latitude of said imaginary surface, each window of said second degree of magnitude comprising mapping data for a further subdivision or partition of said globe;

and mapping and storing third through twelfth degrees of magnitude thereby forming additional levels of a hierarchical quadtree database structure so that an eleventh magnitude comprises windows encompassing 15 seconds of latitude and a twelfth magnitude comprises windows encompassing seven and a half seconds of latitude;

whereby, as a result of the foregoing, windows of said electronic map at respective magnitudes or levels are consistent with conventional mapping scale divisions.

7. The method of claim 6 wherein said hierarchical quadtree database structure comprises sixteen degree of magnitudes or levels including a sixteenth magnitude comprising over 1.4 billion windows, each encompassing approximately a fraction of a minute of a degree of latitude.

8. The method of claim 6 wherein each said window corresponds to a trapezoidal surface area configuration.

9. The method of claim 6 comprising the step of floating mapping data records of selected features from a window of one magnitude to a window of the same vertical lineage in another magnitude.

10. The method of claim 6 comprising the further steps of: generating analog mapping data, structuring said analog mapping data according to a same format as digital mapping data, and overlaying and presenting said digital mapping data and analog mapping data during generation, display and presentation of an electronic map.

11. The method of claim 6 comprising the further step of selectively filling said windows with mapping data so that some windows contain a full complement of mapping data appropriate to a degree of mapping resolution being afforded at said magnitude, and other windows, each of which correspond to a subdivision of surface area containing few or no geographical or cultural features, contain less than a full complement of mapping data.

12. The method of claim 6 comprising the further steps of:

accessing and presenting mapping data in a corrected projection format by aspecting or narrowing, in a direction from an equator to pole, a width or latitudinal dimension of windows, of a same magnitude, which encompass the same number of degrees of latitude and longitude;

and periodically increasing a number of degrees of longitude encompassed by said windows in said direction from equator to pole to compensate for a compound curvature of said globe.

13. The method of claim 12 comprising the further steps of accessing and presenting mapping data in corrected projection format, with each window having a width substantially equal to a center latitude width of said window throughout said window, so that said window is of rectangular configuration.

14. An electronic map generating system including a digital computer, a mass storage device, a display monitor, graphics controller, and system software for structuring, managing, controlling and displaying digital mapping data for a surface area having cultural and geographical features, said system comprising:

a database structure comprising a hierarchical database structure programmed and arranged for organizing said digital mapping data into a hierarchy of a plurality of successive magnitudes or levels for presentation of mapping data with variable resolution, each magnitude for presentation of said mapping data with a different degree of mapping resolution from a first or highest magnitude of lowest resolution to a last or lowest magnitude of lowest resolution to a last or lowest magnitude of highest resolution, and for structuring said digital mapping data at each magnitude into a plurality of windows, frames or files representing subdivisions or partitions of said surface area, said windows of a respective magnitude including mapping data which are appropriate to a degree of mapping resolution being afforded at said magnitude while excluding mapping data which are not appropriate to said degree of mapping resolution, at least a portion of said windows of all magnitudes being structured to receive substantially a same predetermined amount of mapping data for segmented presentation of said mapping data by window, said mapping data being organized into coded records of geographical and cultural features within each window;

a database manager or controller programmed and arranged for managing said mapping data by magnitude or level by excluding or including coded records of features in each window of a particular magnitude according to a resolution and density of mapping data appropriate to the particular magnitude of said each window, and maintaining a quantity of mapping data entered in each window to no greater than a specified full complement whatever the magnitude of the window;

said database structure being programmed to relate windows of a same magnitude by position coordinates or names, and to structure windows of a same magnitude with overlap of mapping data between adjacent or neighboring windows of a magnitude to achieve display continuity during generation, display and presentation of an electronic map, and to relate windows of different magnitude by vertical lineage through successive magnitudes, each window of a higher magnitude and lower resolution being an ancestor window of a plurality of descendant windows of lower magnitude and higher resolution in a next lower magnitude;

said database manager being programmed to access and display or present mapping data for different positions of a selected magnitude by scrolling between adjacent or neighboring windows of a same magnitude in predetermined north, south, east and west directions, and being programmed to access and display or present mapping data for different magnitudes having different resolutions by zooming between windows of different magnitudes in a same vertical lineage.

15. The system of claim 14 wherein said hierarchical database structure is programmed to organize said mapping data by degrees of latitude and longitude and to structure each window of mapping data to represent a substantially rectangular surface area configuration encompassing predetermined degrees of latitude and longitude, said windows for each magnitude being stored in virtual Mercator projection format, said database manager being programmed to access and present windows of mapping data in a corrected or compensated projection format departing from Mercator projection format according to a real configuration of said surface area by varying an aspect ratio of latitude and longitude dimensions of each window according to a coordinate position of said each window with respect to a coordinate layout of said surface area.

16. The system of claim 15 wherein said surface area comprises a spherical or spheroidal globe having an equator and poles, and wherein said database manager is programmed to access and present mapping data in a corrected projection format by aspecting or narrowing, in a direction from an equator to pole, the width or latitudinal dimension of windows, of a same magnitude, which encompass the same number of degrees of longitude, said database manager being further programmed to periodically increase a number of degrees of longitude encompassed by said windows in said direction from equator to pole to compensate for compound curvature of said globe.

17. The system of claim 16 wherein said hierarchical database structure comprises a hierarchical quadtree database structure successively partitioning or subdividing ancestor windows of a vertical lineage into four descendant windows or quadrants at a next lower magnitude or level, and incorporating additional coded records of features in said descendant windows to incorporate mapping data for a next higher resolution.

18. The system of claim 17 wherein said database structure is programmed and arranged to store the mapping data in a virtual Mercator projection representing an imaginary surface having 512 degrees of longitude and latitude comprising a zero magnitude or root node of said hierarchical quadtree database structure, wherein a first degree or first magnitude of mapping data comprises four windows, each window of said first magnitude comprising mapping data for one quarter of said globe on an imaginary surface area of 256 degrees of longitude and latitude, said hierarchical quadtree database structure comprising, in addition to first through tenth magnitudes each having windows which are predetermined subdivisions of said imaginary surface having 512 degrees of longitude and latitude, at least an eleventh magnitude having windows encompassing 15 minutes of latitude, and a twelfth magnitude having windows encompassing 7.5 minutes of latitude, so that windows of a resultant electronic map at respective said eleventh and twelfth magnitudes or levels are consistent with conventional mapping scale divisions.

19. The system of claim 18 wherein said hierarchical quadtree database structure comprises at least 16 degrees of magnitudes or levels, said sixteenth magnitude comprising over 1.4 billion windows, each encompassing degrees of latitude of approximately a fraction of a second of a degree.

20. The system of claim 19 further comprising a database of digital mapping data selectively entered in said database structure, such that some of said windows contain a full complement of mapping data appropriate to a degree of mapping resolution being afforded at said magnitude, and other windows, each of which correspond to a subdivision of surface area containing few or

no geographical or cultural features, contain less than a full complement of mapping data.

21. The system of claim 19 further comprising a database of analog data structured according to a same format as said digital data, and means for overlaying said digital and analog data for electronic map presentation.

22. An electronic map generating system for generating reproductions of a map with selectable degrees of mapping resolution, said map generating system comprising:

database means storing a plurality of computer files containing mapping data corresponding to respective surface areas of a mapping surface, wherein said plurality of computer files is organized into a plurality of successive magnitudes, each magnitude for presentation of said mapping data with a different degree of mapping resolution from a first or highest magnitude with lowest resolution to a last or lowest magnitude with highest resolution, files of a respective magnitude including mapping data which are appropriate to a degree of mapping resolution being afforded at said respective magnitude while excluding mapping data which are not appropriate to said degree of mapping resolution, and wherein a predetermined file naming procedure is utilized to assign, to each respective computer file, a unique filename which:

relates said respective computer file to all other computer files having mapping data corresponding to a same magnitude or degree of mapping resolution; and

relates said respective computer file to any computer file comprising mapping data corresponding to a same surface area of a mapping surface as said respective computer file; and

database manager means for accessing said plurality of computer files using said predetermined file naming procedure, to generate a reproduction of a selected area of a map at a selected degree of mapping resolution.

23. An electronic map generating system as claimed in claim 22,

wherein each said unique filename is represented by a value contained in a plurality of bits, and

wherein said predetermined file naming procedure:

utilizes a first predetermined subset of said plurality of bits to relate said respective files having mapping data corresponding to a same magnitude or degree of mapping resolution; and

utilizes a second predetermined subset of said plurality of bits to relate said respective computer file to any computer file comprising mapping data corresponding to a same surface area of a mapping surface as said respective computer file.

24. An electronic map generating system as claimed in claim 23, wherein said unique filename also includes geographical information which can be used to relate a geographical coordinate position of a respective computer file with respect to a coordinate layout of surface areas of said mapping surface.

25. An electronic map generating system as claimed in claim 22,

wherein an assignment of said unique filenames using said predetermined file naming procedure results in said respective computer files of said plurality to be related in a quadtree database structure.

26. An electronic map generating system as claimed in claim 25, wherein the respective area of a mapping surface covered within the computer files of consecutive magnitudes or degrees of mapping resolution changes at a predetermined rate in that, when a computer file at a reference magnitude or degree of mapping resolution contains mapping data corresponding to an $N \times N$ area of a mapping surface (where N is a real number, and is associated with one of the conventional degree °, minute ', or second " mapping scale divisions), then a computer file at a next consecutive magnitude having a higher degree of mapping resolution contains mapping data corresponding to an $(N/2) \times (N/2)$ area of said mapping surface.

27. An electronic map generating system as claimed in claim 26, wherein the value of N at said reference magnitude or degree of mapping resolution, corresponds to one of the following values: 512°, 256°, 128°, 64°, 32°, 16°, 8°, 4°, 2°, 1°, 30', 15', 7.5', 3.75', 1.875', 0.9375' and 0.46875'.

28. A method for providing an electronic map generating system for generating reproductions of a map with selectable degrees of mapping resolution, said method comprising the steps of:

storing a plurality of computer files containing mapping data corresponding to respective surface areas of a mapping surface, wherein said plurality of computer files is organized into a plurality of successive magnitudes, each magnitude for presentation of said mapping data with a different degree of mapping resolution from a first or highest magnitude with lowest resolution to a last or lowest magnitude with highest resolution, files of a respective magnitude including mapping data which are appropriate to a degree of mapping resolution being afforded at said respective magnitude while excluding mapping data which are not appropriate to said degree of mapping resolution, and wherein a predetermined file naming procedure is utilized to assign, to each respective computer file, a unique filename which:

relates said respective computer file to all other computer files having mapping data corresponding to a same magnitude or degree of mapping resolution; and

relates said respective computer file to any computer file comprising mapping data corresponding to a same surface area of a mapping surface as said respective computer file; and

accessing said plurality of computer files using said predetermined file naming procedure, to generate a reproduction of a selected area of a map at a selected degree of mapping resolution.

29. A method as claimed in claim 28,

wherein each said unique filename is represented by a value contained in a plurality of bits, and

wherein said predetermined file naming procedure;

utilizes a first predetermined subset of said plurality of bits to relate said respective computer file to all other computer files having mapping data corresponding to a same magnitude or degree of mapping resolution; and

utilizes a second predetermined subset of said plurality of bits to relate said respective computer file to any computer file comprising mapping data corresponding to a same surface area of a mapping surface as said respective computer file.

30. A method as claimed in claim 29, wherein said unique filename also includes geographical information which can be used to relate a geographical coordinate position of a respective computer file with respect to a coordinate layout of surface areas of said mapping surface.

31. A method as claimed in claim 28,

wherein an assignment of said unique filenames using said predetermined file naming procedure results in said respective computer files of said plurality to be related in a quadtree database structure.

32. A method as claimed in claim 31, wherein the respective area of a mapping surface covered within the computer files of consecutive magnitudes or degrees of mapping resolution changes at a predetermined rate in that, when a computer file at a reference magnitude or degree of mapping resolution contains mapping data corresponding to an N×N area of a mapping surface (where N is a real number, and is associated with one of the conventional degree °, minute ', or second " mapping scale divisions), then a computer file at a next consecutive magnitude having a higher degree of mapping resolution contains mapping data corresponding to an (N/2)×(N/2) area of said mapping surface.

33. A method as claimed in claim 32, wherein the value of N at said reference magnitude or degree of mapping resolution, corresponds to one of the following values: 512°, 256°, 128°, 64°, 32°, 16°, 8°, 4°, 2°, 1°, 30', 15', 7.5', 3.75', 1.875', 0.9375' and 0.46875'.

* * * * *

US006182114B1

(12) **United States Patent**
Yap et al.

(10) **Patent No.: US 6,182,114 B1**
(45) **Date of Patent: Jan. 30, 2001**

(54) **APPARATUS AND METHOD FOR REALTIME VISUALIZATION USING USER-DEFINED DYNAMIC, MULTI-FOVEATED IMAGES**

(75) Inventors: **Chee K. Yap**; **Ee-Chien Chang**, both of New York, NY (US); **Ting-Jen Yen**, Jersey City, NJ (US)

(73) Assignee: **New York University**, New York, NY (US)

(*) Notice: Under 35 U.S.C. 154(b), the term of this patent shall be extended for 0 days.

(21) Appl. No.: **09/005,174**

(22) Filed: **Jan. 9, 1998**

(51) **Int. Cl.**[7] **G06F 15/16**
(52) **U.S. Cl.** **709/203**; 709/246
(58) **Field of Search** 709/217, 219, 709/246, 247, 203; 707/10; 382/103, 233, 235, 232, 240, 302

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,622,632		11/1986	Tanimoto .	
5,341,466		8/1994	Perlin .	
5,481,622	*	1/1996	Gerhardt et al.	382/103
5,568,598	*	10/1996	Mack et al.	382/302 X
5,710,835	*	1/1998	Bradley	382/233
5,724,070	*	3/1998	Denninghoff et al.	382/235 X
5,861,920	*	1/1999	Mead et al.	382/232 X
5,880,856	*	3/1999	Ferriere	382/240 X
5,920,865	*	7/1999	Ariga	707/10

OTHER PUBLICATIONS

Tams Frajka et al., Progressive Image Coding with Spatially Variable Resolution, IEEE, Proceedings International Conference on Image Processing 1997, Oct. 1997, vol. 1, pp. 53–56.*

E. C. Chang et al., "Realtime Visualization of Large . . . " Mar. 31, 11997,pp. 1–9, Courant Institute of Mathematical Sciences, New York University, N.Y. U.S.A.

E. C. Chang et al., "A Wavelet Approach to Foveating Images", Jan. 10, 1997,pp 1–11, Courant Institute of Mathematical Sciences, New York University, N.Y. U.S.A.

S.G. Mallat, "A Theory for Multiresolutional Signal Decomposition . . . ", IEEE Transactions on Pattern Analysis and Machine Intelligence,pp. 3–23, Jul. 1989, vol. 11, No. 7, IEEE Computer Society.

News Release, "Wavelet Image Features",Summus'Wavelet Image Compression,Summus 14 pages.

R.L. White et al., "Compression and Progressive Transmission of Astronomical Images", SPIE Technical Conference 2199, 1994.

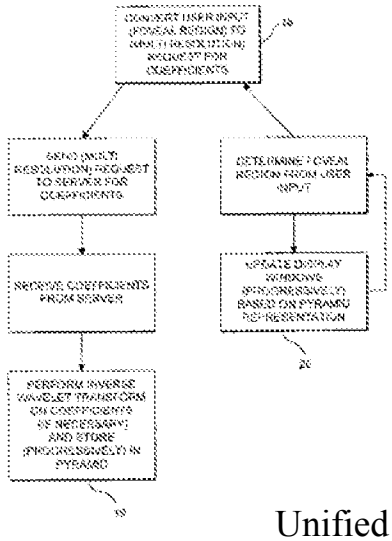
(List continued on next page.)

Primary Examiner—Zarni Maung
Assistant Examiner—Patrice Winder
(74) *Attorney, Agent, or Firm*—Baker Botts, L.L.P.

(57) **ABSTRACT**

A client apparatus which enables a realtime visualization of at least one image. The client apparatus includes a storage device which stores first data corresponding to a multifoveated representation of an original image, and a user input device which providing second data corresponding to at least one visualization command of at least one user. In addition, the client apparatus includes a processing arrangement which generates third data corresponding to a multifoveated image using the first data, the second data and a foveation operator.

8 Claims, 6 Drawing Sheets



OTHER PUBLICATIONS

E.L. Schwartz, "The Development of Specific Visual . . . " Journal of Theoretical Biology, 69:655–685, 1977.
F.S. Hill Jr. et al.,"Interactive Image Query . . . " Computer Graphics, 17(3), 1983.
T.H. Reeves et al., "Adaptive Foveation of MPEG Video", Proceedings of the 4th ACM International Multimedia Conference, 1996.
R.S. Wallace et al., "Space–variant image processing". Int'l. J. of Computer Vision, 13:1(1994) 71–90.
E.L. Schwartz A quantitative model of the functional architecture: Biological cybernetics, 37(1980) 63–76.
P. Kortum et al., "Implementation of a Foveated Image . . . " Human Vision and Electronic Imagining, SPIE Proceedings vol. 2657, 350–360, 1996.
M.H. Gross et al., "Efficient triangular surface . . . ", IEEE Trans on Visualization and Computer Graphics, 2(2) 1996.

* cited by examiner

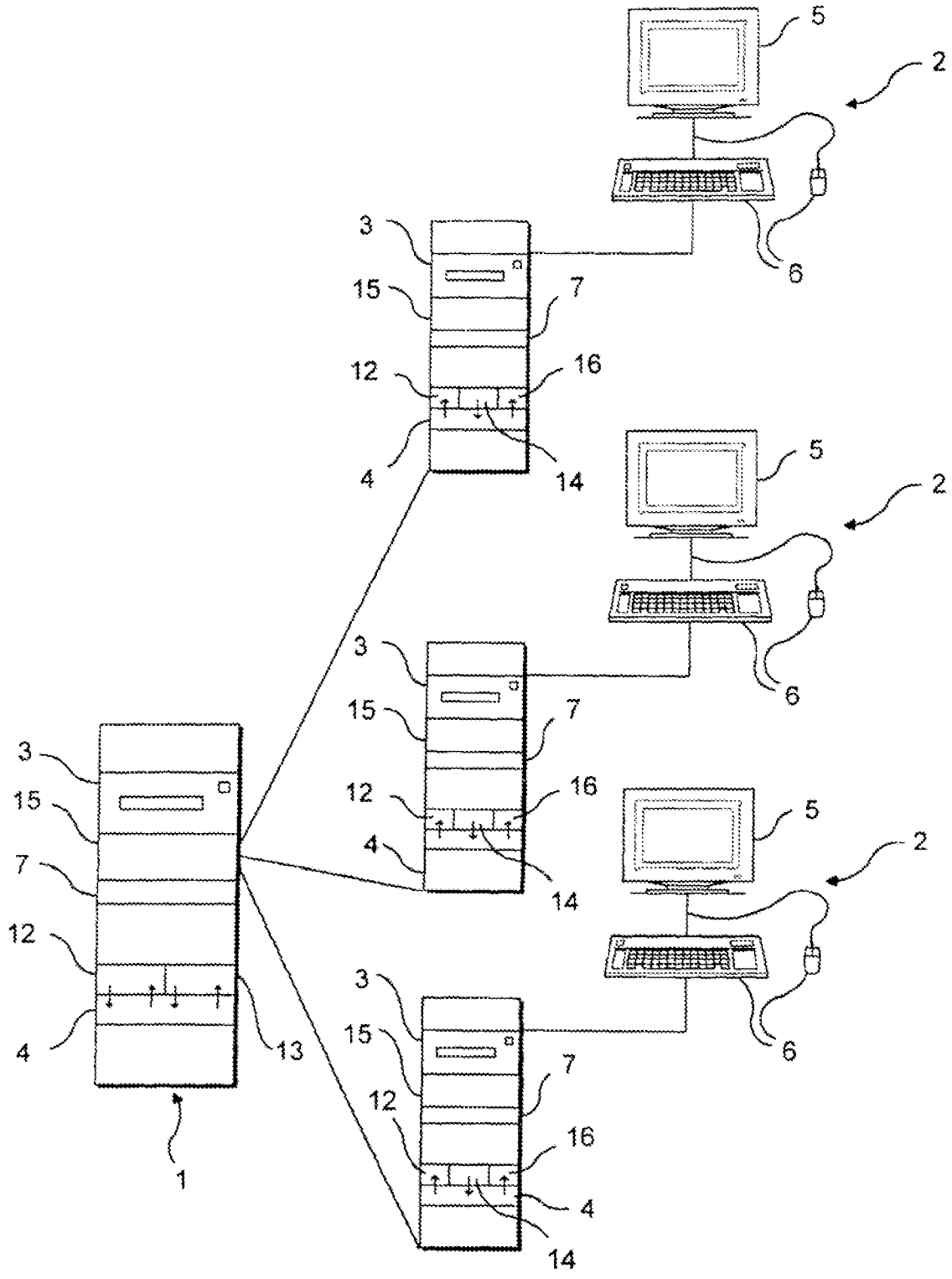


FIG. 1

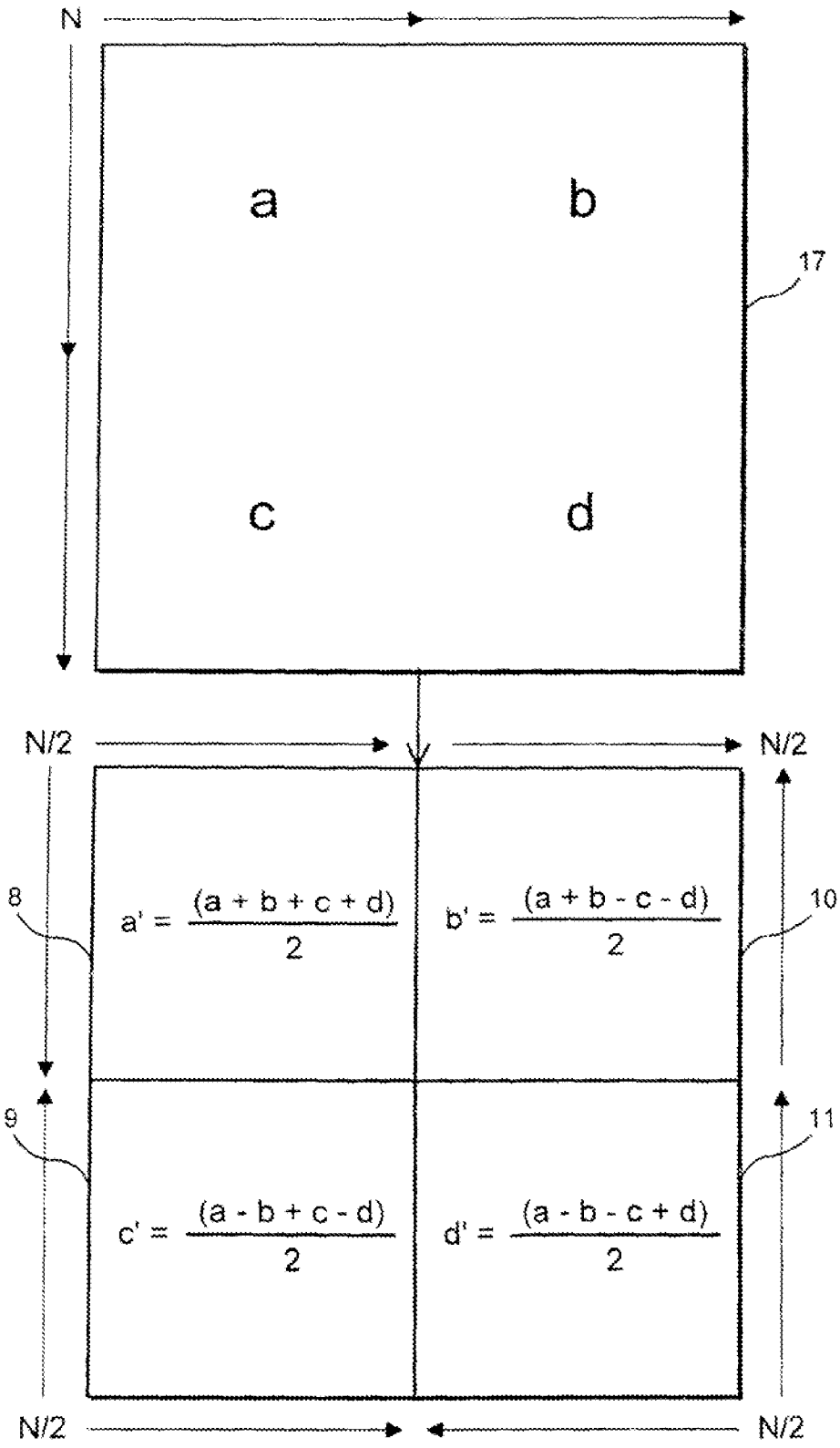

N
a
b
c
d
17
N/2
N/2
8
$a' = \frac{(a + b + c + d)}{2}$
$b' = \frac{(a + b - c - d)}{2}$
10
9
11
$c' = \frac{(a - b + c - d)}{2}$
$d' = \frac{(a - b - c + d)}{2}$
N/2
N/2

FIG. 2A

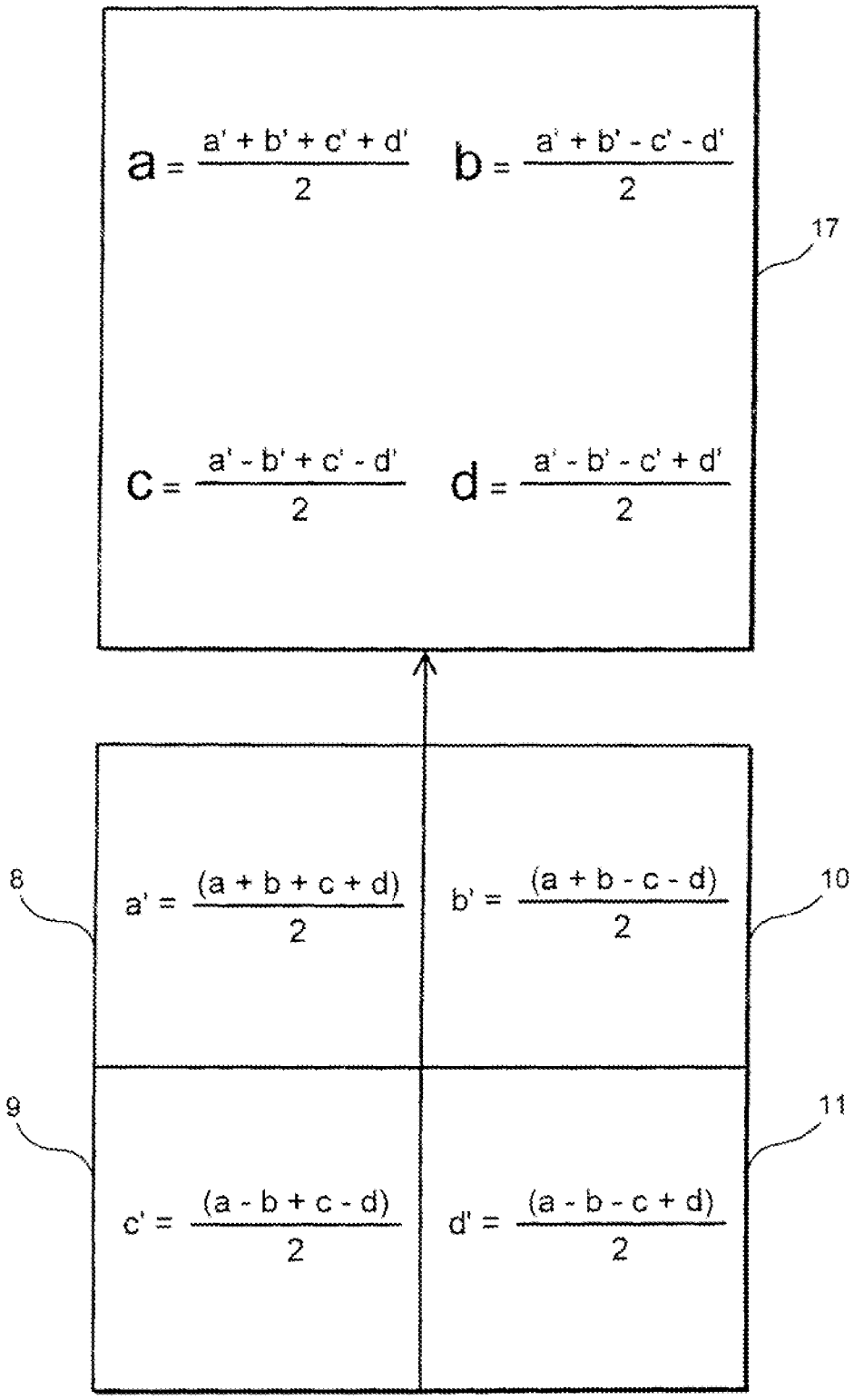

$a = \frac{a' + b' + c' + d'}{2}$
$b = \frac{a' + b' - c' - d'}{2}$
$c = \frac{a' - b' + c' - d'}{2}$
$d = \frac{a' - b' - c' + d'}{2}$
17
$a' = \frac{(a + b + c + d)}{2}$
$b' = \frac{(a + b - c - d)}{2}$
$c' = \frac{(a - b + c - d)}{2}$
$d' = \frac{(a - b - c + d)}{2}$
8
10
9
11

FIG. 2B

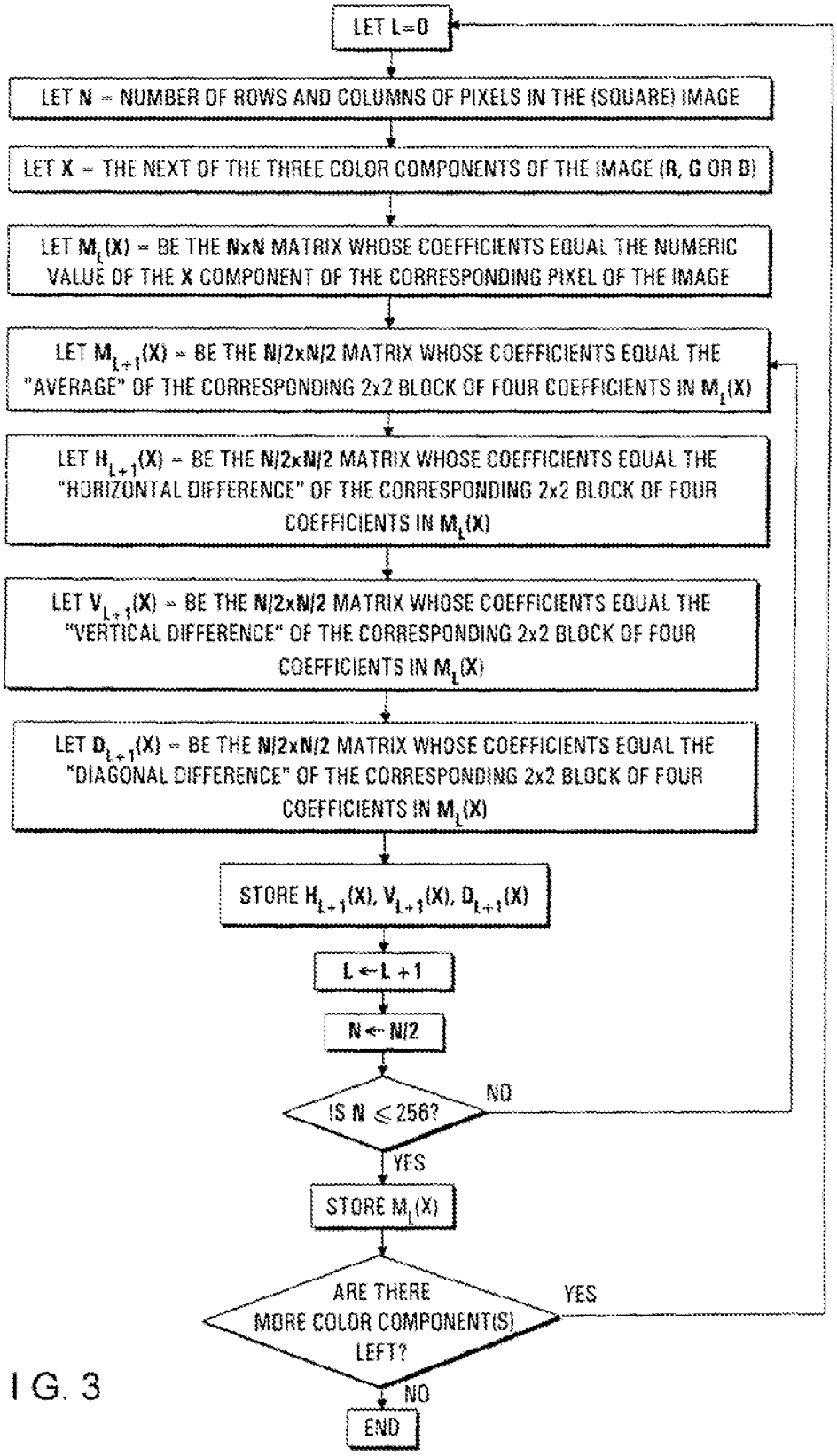

LET L=0
LET N = NUMBER OF ROWS AND COLUMNS OF PIXELS IN THE (SQUARE) IMAGE
LET X = THE NEXT OF THE THREE COLOR COMPONENTS OF THE IMAGE (R, G OR B)
LET $M_L(X)$ = BE THE NxN MATRIX WHOSE COEFFICIENTS EQUAL THE NUMERIC VALUE OF THE X COMPONENT OF THE CORRESPONDING PIXEL OF THE IMAGE
LET $M_{L+1}(X)$ = BE THE N/2xN/2 MATRIX WHOSE COEFFICIENTS EQUAL THE "AVERAGE" OF THE CORRESPONDING 2x2 BLOCK OF FOUR COEFFICIENTS IN $M_L(X)$
LET $H_{L+1}(X)$ = BE THE N/2xN/2 MATRIX WHOSE COEFFICIENTS EQUAL THE "HORIZONTAL DIFFERENCE" OF THE CORRESPONDING 2x2 BLOCK OF FOUR COEFFICIENTS IN $M_L(X)$
LET $V_{L+1}(X)$ = BE THE N/2xN/2 MATRIX WHOSE COEFFICIENTS EQUAL THE "VERTICAL DIFFERENCE" OF THE CORRESPONDING 2x2 BLOCK OF FOUR COEFFICIENTS IN $M_L(X)$
LET $D_{L+1}(X)$ = BE THE N/2xN/2 MATRIX WHOSE COEFFICIENTS EQUAL THE "DIAGONAL DIFFERENCE" OF THE CORRESPONDING 2x2 BLOCK OF FOUR COEFFICIENTS IN $M_L(X)$
STORE $H_{L+1}(X)$, $V_{L+1}(X)$, $D_{L+1}(X)$
L ← L + 1
N ← N/2
IS N ≤ 256?
NO
YES
STORE $M_L(X)$
ARE THERE MORE COLOR COMPONENT(S) LEFT?
YES
NO
END

FIG. 3

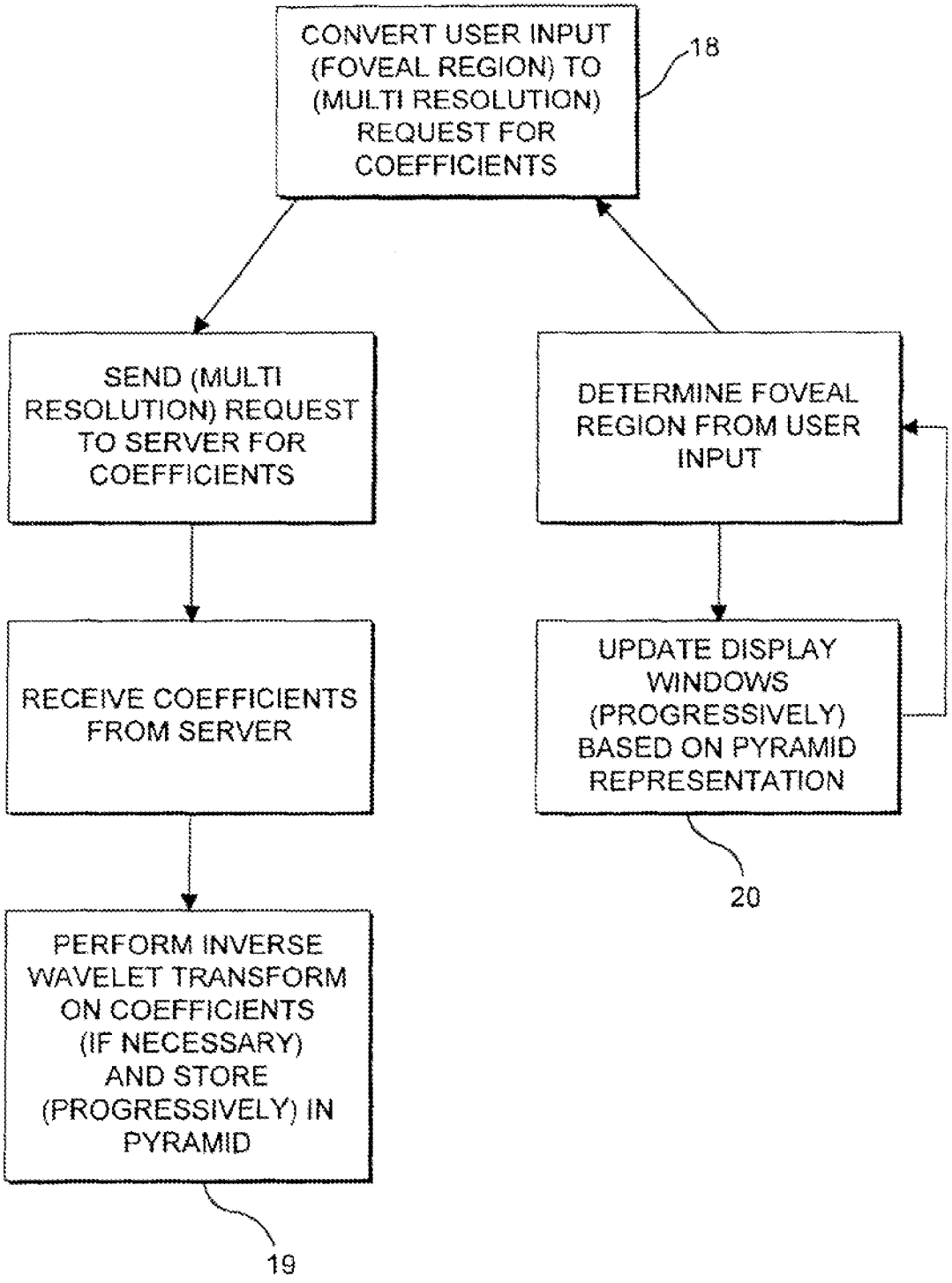

CONVERT USER INPUT (FOVEAL REGION) TO (MULTI RESOLUTION) REQUEST FOR COEFFICIENTS
18
SEND (MULTI RESOLUTION) REQUEST TO SERVER FOR COEFFICIENTS
DETERMINE FOVEAL REGION FROM USER INPUT
RECEIVE COEFFICIENTS FROM SERVER
UPDATE DISPLAY WINDOWS (PROGRESSIVELY) BASED ON PYRAMID REPRESENTATION
20
PERFORM INVERSE WAVELET TRANSFORM ON COEFFICIENTS (IF NECESSARY) AND STORE (PROGRESSIVELY) IN PYRAMID
19

FIG. 4

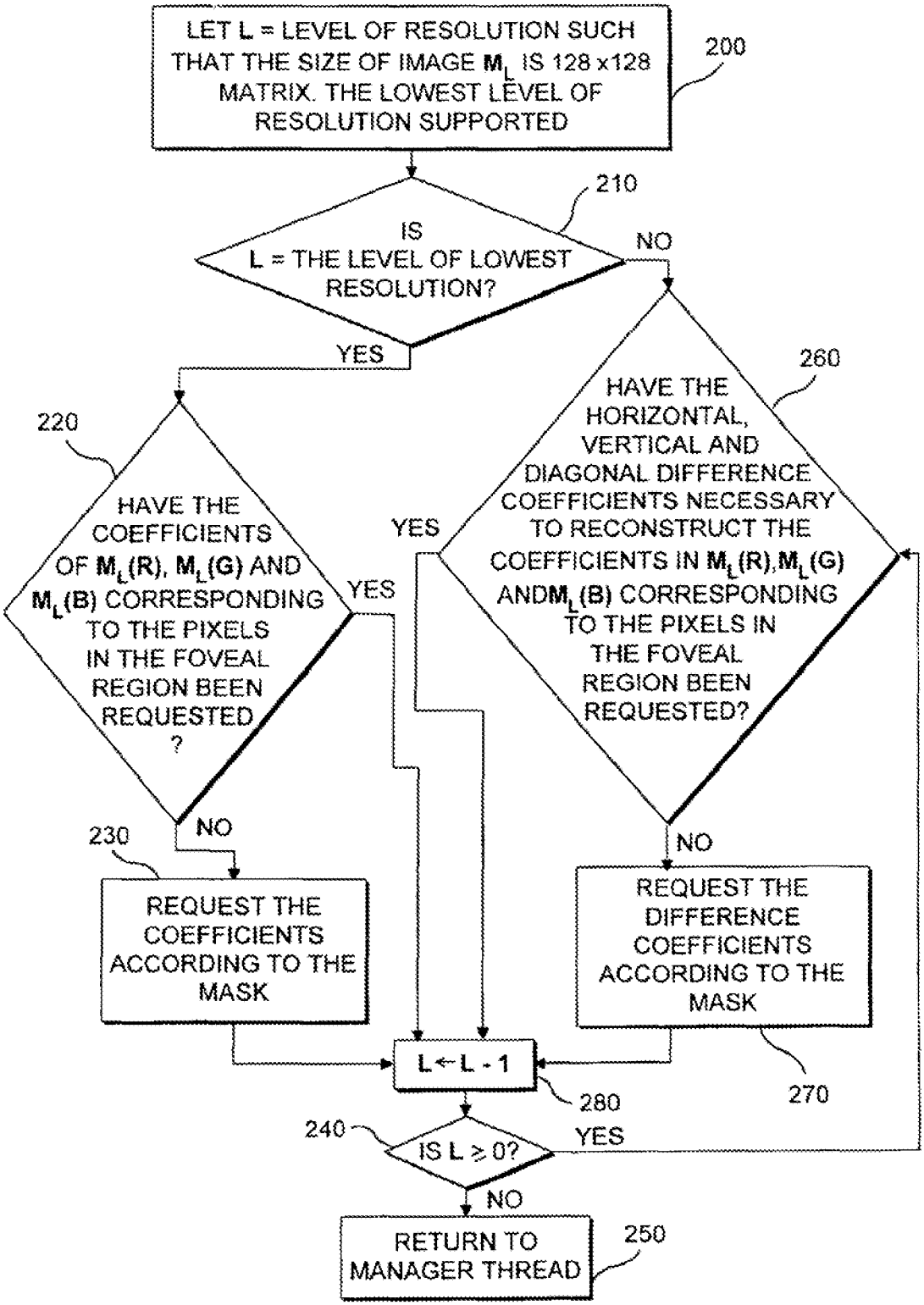

LET L = LEVEL OF RESOLUTION SUCH THAT THE SIZE OF IMAGE M_L IS 128 x128 MATRIX. THE LOWEST LEVEL OF RESOLUTION SUPPORTED
200
IS L = THE LEVEL OF LOWEST RESOLUTION?
210
NO
YES
HAVE THE COEFFICIENTS OF $M_L(R)$, $M_L(G)$ AND $M_L(B)$ CORRESPONDING TO THE PIXELS IN THE FOVEAL REGION BEEN REQUESTED ?
220
YES
NO
REQUEST THE COEFFICIENTS ACCORDING TO THE MASK
230
HAVE THE HORIZONTAL, VERTICAL AND DIAGONAL DIFFERENCE COEFFICIENTS NECESSARY TO RECONSTRUCT THE COEFFICIENTS IN $M_L(R)$,$M_L(G)$ AND$M_L(B)$ CORRESPONDING TO THE PIXELS IN THE FOVEAL REGION BEEN REQUESTED?
260
YES
NO
REQUEST THE DIFFERENCE COEFFICIENTS ACCORDING TO THE MASK
270
L←L - 1
280
IS L ≥ 0?
240
YES
NO
RETURN TO MANAGER THREAD
250

FIG. 5

APPARATUS AND METHOD FOR REALTIME VISUALIZATION USING USER-DEFINED DYNAMIC, MULTI-FOVEATED IMAGES

FIELD OF THE INVENTION

The present invention relates to a method and apparatus for serving images, even very large images, over a "thin-wire" (e.g., over the Internet or any other network or application having bandwidth limitations).

BACKGROUND INFORMATION

The Internet, including the World Wide Web, has gained in popularity in recent years. The Internet enables clients/users to access information in ways never before possible over existing communications lines.

Often, a client/viewer desires to view and have access to relatively large images. For example, a client/viewer may wish to explore a map of a particular geographic location. The whole map, at highest (full) level of resolution will likely require a pixel representation beyond the size of the viewer screen in highest resolution mode.

One response to this restriction is for an Internet server to pre-compute many smaller images of the original image. The smaller images may be lower resolution (zoomed-out) views and/or portions of the original image. Most image archives use this approach. Clearly this is a sub-optimal approach since no preselected set of views can anticipate the needs of all users.

Some map servers (see, e.g., URLs http://www.mapquest.com and http://www.MapOnUs.com) use an improved approach in which the user may zoom and pan over a large image. However, transmission over the Internet involves significant bandwidth limitations (i.e transmission is relatively slow). Accordingly, such map servers suffer from at least three problems:

- Since a brand new image is served up for each zoom or pan request, visual discontinuities in the zooming and panning result. Another reason for this is the discrete nature of the zoom/pan interface controls.
- Significantly less than realtime response.
- The necessarily small fixed size of the viewing window (typically about 3"x4.5"). This does not allow much of a perspective.

To generalize, what is needed is an apparatus and method which allows realtime visualization of large scale images over a "thinwire" model of computation. To put it another way, it is desirable to optimize the model which comprises an image server and a client viewer connected by a low bandwidth line.

One approach to the problem is by means of progressive transmission. Progressive transmission involves sending a relatively low resolution version of an image and then successively transmitting better resolution versions. Because the first, low resolution version of the image requires far less data than the full resolution version, it can be viewed quickly upon transmission. In this way, the viewer is allowed to see lower resolution versions of the image while waiting for the desired resolution version. This gives the transmission the appearance of continuity. In addition, in some instances, the lower resolution version may be sufficient or may in any event exhaust the display capabilities of the viewer display device (e.g., monitor).

Thus, R. L. White and J. W. Percival, "Compression and Progressive Transmission of Astronomical Images," *SPIE Technical Conference* 2199, 1994, describes a progressive transmission technique based on bit planes that is effective for astronomical data.

However, utilizing progressive transmission barely begins to solve the "thinwire" problem. A viewer zooming or panning over a large image (e.g., map) desires realtime response. This of course is not achieved if the viewer must wait for display of the desired resolution of a new quadrant or view of the map each time a zoom and pan is initiated. Progressive transmission does not achieve this realtime response when it is the higher resolution versions of the image which are desired or needed, as these are transmitted later.

The problem could be effectively solved, if, in addition to variable resolution over time (i.e, progressive transmission), resolution is also varied over the physical extent of the image.

Specifically, using foveation techniques, high resolution data is transmitted at the user's gaze point but with lower resolution as one moves away from that point. The very simple rationale underlying these foveation techniques is that the human field of vision (centered at the gaze point) is limited. Most of the pixels rendered at uniform resolution are wasted for visualization purposes. In fact, it has been shown that the spatial resolution of the human eye decreases exponentially away from the center gaze point. E. L. Schwartz, "The Development of Specific Visual Projections in the Monkey and the Goldfish: Outline of a Geometric Theory of Receptotopic Structure," *Journal of Theoretical Biology*, 69:655–685, 1977

The key then is to mimic the movements and spatial resolution of the eye. If the user's gaze point can be tracked in realtime and a truly multi-foveated image transmitted (i.e., a variable resolution image mimicking the spatial resolution of the user's eye from the gaze point), all data necessary or useful to the user would be sent, and nothing more. In this way, the "thinwire" model is optimized, whatever the associated transmission capabilities and bandwidth limitations.

In practice, in part because eye tracking is imperfect, using multi-foveated images is superior to attempting display of an image portion of uniform resolution at the gaze point.

There have in fact been attempts to achieve multifoveated images in a "thinwire" environment.

F. S. Hill Jr., Sheldon Walker Jr. and Fuwen Gao, "Interactive Image Query System Using Progressive Transmission," *Computer Graphics*, 17(3), 1983, describes progressive transmission and a form of foveation for a browser of images in an archive. The realtime requirement does not appear to be a concern.

T. H. Reeves and J. A. Robinson, "Adaptive Foveation of MPEG Video," *Proceedings of the 4th ACM International Multimedia Conference*, 1996, gives a method to foveate MPEG-standard video in a thin-wire environment. MPEG-standard could provide a few levels of resolution but they consider only a 2-level foveation. The client/viewer can interactively specify the region of interest to the server/sender.

R. S. Wallace and P. W. Ong and B. B. Bederson and E. L. Schwartz, "Space-variant image processing", Intl. J. Of Computer Vision, 13:1 (1994) 71–90 discusses space-variant images in computer vision. "Space-Variant" may be regarded as synonymous with the term "multifoveated" used above. A biological motivation for such images is the complex logmap model of the transformation from the retina to the visual cortex (E. L. Schwartz, "A quantitative model of the functional architecture of human striate cortex with

application to visual illusion and cortical texture analysis", Biological Cybernetics, 37(1980) 63–76).

Philip Kortum and Wilson S. Geisler, "Implementation of a Foveated Image Coding System For Image Bandwidth Reduction," *Human Vision and Electronic Imaging, SPIE Proceedings Vol.* 2657, 350–360, 1996, implement a real time system for foveation-based visualization. They also noted the possibility of using foveated images to reduce bandwidth of transmission.

M. H. Gross, O. G. Staadt and R. Gatti, "Efficient triangular surface approximations using wavelets and quadtree data structures", IEEE Trans, On Visualization and Computer Graphics, 2(2), 1996, uses wavelets to produce multifoveated images.

Unfortunately, each of the above attempts are essentially based upon fixed super-pixel geometries, which amount to partitioning the visual field into regions of varying (predetermined) sizes called super-pixels, and assigning the average value of the color in the region to the super-pixel. The smaller pixels (higher resolution) are of course intended to be at the gaze point, with progressively larger super-pixels (lower resolution) about the gaze point.

However, effective real-time visulization over a "thin wire" requires precision and flexibility. This cannot be achieved with a geometry of predetermined pixel size. What is needed is a flexible foveation technique which allows one to modify the position and shape of the basic foveal regions, the maximum resolution at the foveal region and the rate at which the resolution falls away. This will allow the "thinwire" model to be optimized.

In addition, none of the above noted references addresses the issue of providing multifoveated images that can be dynamically (incrementally) updated as a function of user input. This property is crucial to the solution of the thinwire problem, since it is essential that information be "streamed" at a rate that optimally matches the bandwidth of the network with the human capacity to absorb the visual information.

SUMMARY OF THE INVENTION

The present invention overcomes the disadvantages of the prior art by utilizing means for tracking or approximating the user's gaze point in realtime and, based on the approximation, transmitting dynamic multifoveated image (s) (i.e., a variable resolution image over its physical extent mimicking the spatial resolution of the user's eye about the approximated gaze point) updated in realtime.

"Dynamic" means that the image resolution is also varying over time. The user interface component of the present invention may provide a variety of means for the user to direct this multifoveation process in real time.

Thus, the invention addresses the model which comprises an image server and a client viewer connected by a low bandwidth line. In effect, the invention reduces the bandwidth from server to client, in exchange for a very modest increase of bandwidth from the client to the server

Another object of the invention is that it allows realtime visualization of large scale images over a "thinwire" model of computation.

An additional advantage is the new degree of user control provided for realtime, active, visualization of images (mainly by way of foveation techniques). The invention allows the user to determine and change in realtime, via input means (for example, without limitation, a mouse pointer or eye tracking technology), the variable resolution over the space of the served up image(s).

An additional advantage is that the invention demonstrates a new standard of performance that can be achieved by large-scale image servers on the World Wide Web at current bandwidth or even in the near future.

Note also, the invention has advantages over the traditional notion of progressive transmission, which has no interactivity. Instead, the progressive transmission of an image has been traditionally predetermined when the image file is prepared. The invention's use of dynamic (constantly changing in realtime based on the user's input) multifoveated images allows the user to determine how the data are progressively transmitted.

Other advantages of the invention include that it allows the creation of the first dynamic and a more general class of multifoveated images. The present invention can use wavelet technology. The flexibility of the foveation approach based on wavelets allows one to easily modify the following parameters of a multifoveated image: the position and shape of the basic foveal region(s), the maximum resolution at the foveal region(s), and the rate at which the resolution falls away. Wavelets can be replaced by any multi resolution pyramid schemes. But it seems that wavelet-based approaches are preferred as they are more flexible and have the best compression properties.

Another advantage is the present invention's use of dynamic data structures and associated algorithms. This helps optimize the "effective real time behavior" of the system. The dynamic data structures allow the use of "partial information" effectively. Here information is partial in the sense that the resolution at each pixel is only partially known. But as additional information is streamed in, the partial information can be augmented. Of course, this principle is a corollary to progressive transmission.

Another advantage is that the dynamic data structures may be well exploited by the special architecture of the client program. For example, the client program may be multi-threaded with one thread (the "manager thread") designed to manage resources (especially bandwidth resources). This manager is able to assess network congestion, and other relevant parameters, and translate any literal user request into the appropriate level of demand for the network. For example, when the user's gaze point is focused on a region of an image, this may be translated into requesting a certain amount, say, X bytes of data. But the manager can reduce this to a request over the network of (say) X/2 bytes of data if the traffic is congested, or if the user is panning very quickly.

Another advantage of the present invention is that the server need send only that information which has not yet been served. This has the advantage of reducing communication traffic.

Further objects and advantages of the invention will become apparent from a consideration of the drawings and ensuing description.

BRIEF DESRIPTION OF DRAWINGS

FIG. **1** shows an embodiment of the present invention including a server, and client(s) as well as their respective components.

FIG. **2***a* illustrates one level of a particular wavelet transform, the Haar wavelet transform, which the server may execute in one embodiment of the present invention.

FIG. **2***b* illustrates one level of the Haar inverse wavelet transform.

FIG. **3** is a flowchart showing an algorithm the server may execute to perform a Haar wavelet transform in one embodiment of the present invention.

FIG. 4 shows Manager, Display and Network threads, which the client(s) may execute in one embodiment of the present invention.

FIG. 5 is a more detailed illustration of a portion of the Manager thread depicted in FIG. 4.

DETAILED DESCRIPTION OF THE INVENTION

FIG. 1 depicts an overview of the components in an exemplary embodiment of the present invention. A server 1 is comprised of a storage device 3, a memory device 7 and a computer processing device 4. The storage device 3 can be implemented as, for example, an internal hard disk, Tape Cartridge, or CD-ROM. The faster access and greater storage capacity the storage device 3 provides, the more preferable the embodiment of the present invention. The memory device 7 can be implemented as, for example, a collection of RAM chips.

The processing device 4 on the server 1 has network protocol processing element 12 and wavelet transform element 13 running off it. The processing device 4 can be implemented with a single microprocessor chip (such as an Intel Pentium chip), printed circuit board, several boards or other device. Again, the faster the speed of the processing device 4, the more preferable the embodiment. The network protocol processing element 12 can be implemented as a separate "software" (i.e., a program, sub-process) whose instructions are executed by the processing device 4. Typical examples of such protocols include TCP/IP (the Internet Protocol) or UDP (User Datagram Protocol). The wavelet transform element 13 can also be implemented as separate "software" (i.e., a program, sub-process) whose instructions are executed by the processing device 4.

In a preferred embodiment of the present invention, the server 1 is a standard workstation or Pentium class system. Also, TCP/IP processing may be used to implement the network protocol processing element 12 because it reduces complexity of implementation. Although a TCP/IP implementation is simplest, it is possible to use the UDP protocol subject to some basic design changes. The relative advantage of using TCP/IP as against UDP is to be determined empirically. An additional advantage of using modern, standard network protocols is that the server 1 can be constructed without knowing anything about the construction of its client(s) 2.

According to the common design of modern computer systems, the most common embodiments of the present invention will also include an operating system running off the processing means device 4 of the server 1. Examples of operating systems include, without limitation, Windows 95, Unix and Windows NT. However, there is no reason a processing device 4 could not provide the functions of an "operating system" itself.

The server 1 is connected to a client(s) 2 in a network. Typical examples of such servers 1 include image archive servers and map servers on the World Wide Web.

The client(s) 2 is comprised of a storage device 3, memory device 7, display 5, user input device 6 and processing device 4. The storage device 3 can be implemented as, for example, an internal hard disks, Tape Cartridge, or CD-ROM. The faster access and greater storage capacity the storage device 3 provides, the more preferable the embodiment of the present invention. The memory device 7 can be implemented as, for example, a collection of RAM chips. The display 5 can be implemented as, for example, any monitor, whether analog or digital. The user input device 6 can be implemented as, for example, a keyboard, mouse, scanner or eye-tracking device.

The client 2 also includes a processing device 4 with network protocol processing element 12 and inverse wavelet transform element means 14 running off it. The processing device 4 can be implemented as, for example, a single microprocessor chip (such as an Intel Pentium chip), printed circuit board, several boards or other device. Again, the faster the run time of the processing device 4, the more preferable the embodiment. The network protocol processing element 12 again can be implemented as a separate "software" (i.e., a program, sub-process) whose instructions are executed by the processing device 4. Again, TCP/IP processing may be used to implement the network protocol processing element 12. The inverse wavelet transform element 14 also may be implemented as separate "software." Also running off the processing device 4 is a user input conversion mechanism 16, which also can be implemented as "software."

As with the server 1, according to the common design of modern computer systems, the most common embodiments of the present invention will also include an operating system running off the processing device 4 of the client(s) 2.

In addition, if the server 1 is connected to the client(s) 2 via a telephone system line or other systems/lines not carrying digital pulses, the server 1 and client(s) 2 both also include a communications converter device 15. A communications converter device 15 can be implemented as, for example, a modem. The communications converter device 15 converts digital pulses into the frequency/signals carried by the line and also converts the frequency/signals back into digital pulses, allowing digital communication.

In the operation of the present invention, the extent of computational resources (e.g., storage capacity, speed) is a more important consideration for the server 1, which is generally shared by more than one client 2, than for the client(s) 2.

In typical practice of the present invention, the storage device 3 of the server 1 holds an image file, even a very large image file. A number of client 2 users will want to view the image.

Prior to any communication in this regard between the server 1 and client(s) 2, the wavelet transform element 13 on the server 1 obtains a wavelet transform on the image and stores it in the storage device 3.

There has been extensive research in the area of wavelet theory. However, briefly, to illustrate, "wavelets" are defined by a group of basis functions which, together with coefficients dependant on an input function, can be used to approximate that function over varying scales, as well as represent the function exactly in the limit. Accordingly, wavelet coefficients can be categorized as "average" or "approximating coefficients" (which approximate the function) and "difference coefficients" (which can be used to reconstruct the original function exactly). The particular approximation used as well as the scale of approximation depend upon the wavelet bases chosen. Once a group of basis functions is chosen, the process of obtaining the relevant wavelet coefficients is called a wavelet transform.

In the preferred embodiment, the Haar wavelet basis functions are used. Accordingly, in the preferred embodiment, the wavelet transform element 13 on the server 1 performs a Haar wavelet transform on a file representation of the image stored in the storage device 3, and then stores the transform on the storage device 3. However, it is readily apparent to anyone skilled in the art that any of the wavelet

family of transforms may be chosen to implement the present invention.

Note that once the wavelet transform is stored, the original image file need not be kept, as it can be reconstructed exactly from the transform.

FIG. **2** illustrates one step of the Haar wavelet transform. Start with an n by n matrix of coefficients **17** whose entries correspond to the numeric value of a color component (say, Red, Green or Blue) of a square screen image of n by n pixels. Divide the original matrix **17** into 2 by 2 blocks of four coefficients, and for each 2×2 block, label the coefficient in the first column, first row "a," second column, first row "b"; second row, first column "c"; and second row, second column "d."

Then one step of the Haar wavelet transform creates four n/2 by n/2 matrices. The first is an n/2 by n/2 approximation matrix **8** whose entries equal the "average" of the corresponding 2 by 2 block of four coefficients in the original matrix **17**. As is illustrated in FIG. **2**, the coefficient entries in the approximation matrix **8** are not necessarily equal to the average of the corresponding four coefficients a, b, c and d (i.e., a'=(a+b+c+d)/4) in the original matrix **17**. Instead, here, the "average" is defined as (a+b+c+d)/2.

The second is an n/2 by n/2 horizontal difference matrix **10** whose entries equal b'=(a+b−c−d)/2, where a, b, c and d are, respectively, the corresponding 2×2 block of four coefficients in the original matrix **17**. The third is an n/2 by n/2 vertical difference matrix **9** whose entries equal c'=(a−b+c−d)/2, where a, b, c and d are, respectively, the corresponding 2×2 block of four coefficients in the original matrix **17**. The fourth is an n/2 by n/2 diagonal difference matrix **11** whose entries equal d'=(a−b−c+d)/2, where a, b, c and d are, respectively, the corresponding 2×2 block of four coefficients in the original matrix **17**.

A few notes are worthy of consideration. First, the entries a', b', c', d' are the wavelet coefficients. The approximation matrix **8** is an approximation of the original matrix **17** (using the "average" of each 2×2 group of 4 pixels) and is one fourth the size of the original matrix **17**.

Second, each of the 2×2 blocks of four entries in the original matrix **17** has one corresponding entry in each of the four n/2 by n/2 matrices. Accordingly, it can readily be seen from FIG. **2** that each of the 2×2 blocks of four entries in the original matrix **17** can be reconstructed exactly, and the transformation is invertible. Therefore, the original matrix **17** representation of an image can be discarded during processing once the transform is obtained.

Third, the transform can be repeated, each time starting with the last approximation matrix **8** obtained, and then discarding that approximation matrix **8** (which can be reconstructed) once the next wavelet step is obtained. Each step of the transform results in approximation and difference matrices ½ the size of the approximation matrix **8** of the prior step.

Retracing each step to synthesize the original matrix **17** is called the inverse wavelet transform, one step of which is depicted in FIG. **2***b*.

Finally, it can readily be seen that the approximation matrix **8** at varying levels of the wavelet transform can be used as a representation of the relevant color component of the image at varying levels of resolution.

Conceptually then, the wavelet transform is a series of approximation and difference matrices at various levels (or resolutions). The number of coefficients stored in a wavelet transform is equal to the number of pixels in the original matrix **17** image representation. (However, the number of bits in all the coefficients may differ from the number of bits in the pixels. Applying data compression to coefficients turns out to be generally more effective on coefficients.) If we assume the image is very large, the transform matrices must be further decomposed into blocks when stored on the storage means **3**.

FIG. **3** is a flowchart showing one possible implementation of the wavelet transform element **13** which performs a wavelet transform on each color component of the original image. As can be seen from the flowchart, the transform is halted when the size of the approximation matrix is 256x 256, as this may be considered the lowest useful level of resolution.

Once the wavelet transform element **13** stores a transform of the image(s) in the storage means **3** of the server **1**, the server **1** is ready to communicate with client(s) **2**.

In typical practice of the invention the client **2** user initiates a session with an image server **1** and indicates an image the user wishes to view via user input means **6**. The client **2** initiates a request for the 256 by 256 approximation matrix **8** for each color component of the image and sends the request to the server **1** via network protocol processing element **12**. The server **1** receives and processes the request via network protocol processing element **12**. The server **1** sends the 256 by 256 approximation matrices **8** for each color component of the image, which the client **2** receives in similar fashion. The processing device **4** of the client **2** stores the matrices in the storage device **3** and causes a display of the 256 by 256 version of the image on the display **5**. It should be appreciated that the this low level of resolution requires little data and can be displayed quickly. In a map server application, the 256 by 256, coarse resolution version of the image may be useful in a navigation window of the display **5**, as it can provide the user with a position indicator with respect to the overall image.

A more detailed understanding of the operation of the client **2** will become apparent from the discussion of the further, continuous operation of the client **2** below.

Continuous operation of the client(s) **2** is depicted in FIG. **4**. In the preferred embodiment, the client(s) **2** processing device may be constructed using three "threads," the Manager thread **18**, the Network Thread **19** and the Display Thread **20**. Thread programming technology is a common feature of modern computers and is supported by a variety of platforms. Briefly, "threads" are processes that may share a common data space. In this way, the processing means can perform more than one task at a time. Thus, once a session is initiated, the Manager Thread **18**, Network Thread **19** and Display Thread **20** run simultaneously, independently and continually until the session is terminated. However, while "thread technology" is preferred, it is unnecessary to implement the client(s) **2** of the present invention.

The Display Thread **20** can be based on any modern windowing system running off the processing device **4**. One function of the Display Thread **20** is to continuously monitor user input device **6**. In the preferred embodiment, the user input device **6** consists of a mouse or an eye-tracking device, though there are other possible implementations. In a typical embodiment, as the user moves the mouse position, the current position of the mouse pointer on the display **5** determines the foveal region. In other words, it is presumed the user gaze point follows the mouse pointer, since it is the user that is directing the mouse pointer. Accordingly, the display thread **20** continuously monitors the position of the mouse pointer.

In one possible implementation, the Display Thread **20** places user input requests (i.e., foveal regions determined from user input device **6**) as they are obtained in a request queue. Queue's are data structures with first-in-first-out characteristics that are generally known in the art.

The Manager Thread **18** can be thought of as the brain of the client **2**. The Manager Thread **18** converts the user input request in the request queue into requests in the manager request queue, to be processed by the Network Thread **19**. The user input conversion mechanism **16** converts the user determined request into a request for coefficients.

A possible implementation of user input conversion mechanism **16** is depicted in the flow chart in FIG. **5**. Essentially, the user input conversion mechanism **16** requests all the coefficient entries corresponding to the foveal region in the horizontal difference **10** matrices, vertical difference **9** matrices, diagonal difference matrices **11** and approximation matrix **8** of the wavelet transform of the image at each level of resolution. (Recall that only the last level approximation matrix **8** needs to be stored by the server **1**.) That is, wavelet coefficients are requested such that it is possible to reconstruct the coefficients in the original matrix **17** corresponding to the foveal region.

As the coefficients are included in the request, they are masked out. The use of a mask is commonly understood in the art. The mask is maintained to determine which coefficients have been requested so they are not requested again. Each mask can be represented by an array of linked lists (one linked list for each row of the image at each level of resolution).

As shown in FIG. **5**, the input conversion mechanism **16** determines the current level of resolution ("L") of an image ("M_L") such that the image M_L is, e.g., 128×128 pixel matrix (for example, the lowest supported resolution), as shown in Step **200**. Then, the input conversion mechanism **16** determines if the current level L is the lowest resolution level (Step **210**). If so, it is determined if the three color coefficients (i.e., $M_L(R)$, $M_L(G)$, and $M_L(B)$) correspond to the foveal region that has been requested (Step **220**). If that is the case, then the input conversion mechanism **16** confirms that the current region L is indeed the lowest resolution region (Step **240**), and returns the control to the Manager Thread **18** (Step **250**). If, in Step **220**, it is determined that the three color coefficients have not been requested, these coefficients are requested using the mask described above, and the process continues to Step **240**, and the control is returned to the Manager Thread **18** (Step **250**).

If, in Step **210**, it is determined that the current level L is not the lowest resolution level, then the input conversion mechanism **16** determines whether the horizontal, vertical and diagonal difference coefficients (which are necessary to reconstruct the three color coefficients) have been requested (Step **260**). If so, then the input conversion mechanism **16** skips to Step **280** to decrease the current level L by 1. Otherwise a set of difference coefficients may be requested. This set depends on the mask and the foveal parameters (e.g., a shape of the foveal region, a maximum resolution, a rate of decay of the resolution, etc.). The user may select "formal" values for these foveal parameters, but the Manager Thread **18** may, at this point, select the "effective" values for these parameters to ensure a trade-off between (1) achieving a reasonable response time over the estimated current network bandwidth, and (2) achieving a maximum throughput in the transmission of data. The process then continues to Step **280**. Thereafter, the input conversion mechanism **16** determines whether the current level L is greater or equal to zero (Step **240**). If that is the case, the process loops back to step **260**. Otherwise, the control is returned to the Manager Thread **18** (Step **250**).

The Network Thread **19** includes the network protocol processing element **12**. The Network Thread obtains the (next) multi-resolution request for coefficients corresponding to the foveal region from request queue and processes and sends the request to the server **1** via network protocol processing element **12**.

Notice that the data requested is "local" because it represents visual information in the neighborhood of the indicated part of the image. The data is incremental because it represents only the additional information necessary to increase the resolution of the local visual information. (Information already available locally is masked out).

The server **1** receives and processes the request via network protocol processing element **12**, and sends the coefficients requested. When the coefficients are sent, they are masked out. The mask is maintained to determine which coefficients have been sent and for deciding which blocks of data can be released from main memory. Thus, an identical version of the mask is maintained on both the client **2** side and server **1** side.

The Network Thread **19** of the client **2** receives and processes the coefficients. The Network Thread **19** also includes inverse wavelet transform element **14**. The inverse wavelet transform element **14** performs an inverse wavelet transform on the received coefficients and stores the resulting portion of an approximation matrix **8** each time one is obtained (i.e., at each level of resolution) in the storage device **3** of the client **2**. The sub-image is stored at each (progressively higher, larger and less course) level of its resolution.

Note that as the client **2** knows nothing about the image until it is gradually filled in as coefficients are requested. Thus, sparse matrices (sparse, dynamic data structures) and associated algorithms can be used to store parts of the image received from the server **1**. Sparse matrices are known in the art and behave like normal matrices except that the memory space of the matrix are not allocated all at once. Instead the memory is allocated in blocks of sub-matrices. This is reasonable as the whole image may require a considerable amount of space.

Simultaneously, the Display thread **20** (which can be implemented using any modern operating system or windowing system) updates the display **5** based on the pyramid representation stored in the storage device **3**.

Of course, the Display thread **20** continues its monitoring of the user input device **6** and the whole of client **2** processing continues until the session is terminated.

A few points are worthy of mention. Notice that since lower, coarser resolution images will be stored on the client **2** first, they are displayed first Also, the use of foveated images ensures that the incremental data to update the view is small, and the requested data can arrive within the round trip time of a few messages using, for example, the TCP/IP protocol.

Also notice, that a wavelet coefficient at a relatively coarser level of resolution corresponding to the foveal region affects a proportionately larger part of the viewer's screen than a coefficient at a relatively finer level of resolution corresponding to the foveal region (in fact, the resolution on the display **5** exponentially away from the mouse pointer). Also notice the invention takes advantage of progressive transmission, which gives the image perceptual continuity, but unlike the traditional notion of progressive

transmission, it is the client 2 user that is determining transmission ordering, which is not pre-computed because the server 1 doesn't know what the client(s) 2 next request will be. Thus, as noted in the objects and advantages section, the "thinwire" model is optimized.

Note that in the event the thread technology is utilized to implement the present invention, semaphores data structures are useful if the threads share the same data structures (e.g., the request queue). Semaphores are well known in the art and ensure that only one simultaneous process (or "thread") can access and modify a shared data structure at one time. Semaphores are supported by modern operating systems.

CONCLUSION

It is apparent that various useful modifications can be made to the above description while remaining within the scope of the invention.

For example, without limitation, the user can be provided with two modes for display: to always fill the pixels to the highest resolution that is currently available locally or to fill them up to some user specified level. The client 2 display 5 may include a re-sizable viewing window with minimal penalty on the realtime performance of the system. This is not true of previous approaches. There also may be an auxiliary navigation window (which can be re-sized but is best kept fairly small because it displays the entire image at a low resolution). The main purpose of such a navigation window would be to let the viewer know the size and position of the viewing window in relation to the whole image.

It is readily seen that further modifications within the scope of the invention provide further advantages to the user. For example, without limitation, the invention may have the following capabilities: continuous realtime panning, continuous realtime zooming, foveating, varying the foveal resolution and modification of the shape and size of the foveal region. A variable resolution feature may also allow the server 1 to dynamically adjust the amount of transmitted data to match the effective bandwidth of the network.

While the above description contains many specificities, these should not be construed as limitations on the scope of the invention, but rather as an exemplification of one preferred embodiment thereof. Many other variations are possible. Accordingly, the scope of the invention should be determined not by the embodiment(s) illustrated, but by the appended claims and their legal equivalents.

What is claimed is:

1. A client apparatus for enabling a realtime visualization of at least one image, the client apparatus comprising:

a storage device storing first data corresponding to a multifoveated representation of an original image,

a user input device providing second data corresponding to at least one visualization command of at least one user; and

a processing arrangement generating third data corresponding to a multifoveated image using the first data, the second data and a foveation operator.

2. The client apparatus of claim 1, further comprising a network protocol processing element which provides the third data using a TCP/IP protocol.

3. The client apparatus of claim 1, wherein the processing element transmits the third data to the at least one client via the Internet.

4. The client apparatus of claim 1, wherein the user input device includes a mouse device.

5. The client apparatus of claim 1, wherein the user input device includes at least one of an eye-tracking device and a keyboard.

6. The client apparatus of claim 1, wherein the foveation operator is specified using parameters that include at least one of:

a set of foveation points,

a shape of a foveated region,

a maximum resolution of the foveated region, and

a rate at which a maximum resolution of the foveal region decays.

7. The client apparatus of claim 1,

wherein the processing arrangement receives the original image from a server, and

wherein the memory arrangement stores a data structure representing the multifoveated image, the data structure that is optimized for the client apparatus being independent of an image representation provided by a server.

8. The client apparatus of claim 1, wherein the third data corresponding to the multifoveated image is generated for at least one of

a first arbitrary-shaped foveal region,

a second arbitrarily-fine foveal region, and

an arbitrary union of the first and second foveal regions.

* * * * *

US005179638A

United States Patent [19]

Dawson et al.

[11] Patent Number: 5,179,638

[45] Date of Patent: Jan. 12, 1993

[54] METHOD AND APPARATUS FOR GENERATING A TEXTURE MAPPED PERSPECTIVE VIEW

[75] Inventors: John F. Dawson; Thomas D. Snodgrass; James A. Cousens, all of Albuquerque, N. Mex.

[73] Assignee: Honeywell Inc., Minneapolis, Minn.

[21] Appl. No.: 514,598

[22] Filed: Apr. 26, 1990

[51] Int. Cl.[5] G06F 15/62

[52] U.S. Cl. 395/125; 395/127; 395/130

[58] Field of Search 395/125, 126, 127, 130; 364/443, 723; 340/729

[56] References Cited

U.S. PATENT DOCUMENTS

4,677,576	6/1987	Berlin, Jr. et al.	395/127 X
4,876,651	10/1989	Dawson et al.	395/126 X
4,884,220	11/1989	Dawson et al.	395/125
4,899,293	2/1990	Dawson et al.	395/125 X
4,940,972	7/1990	Mouchot et al.	395/125 X
4,985,854	1/1991	Wittenburg	395/126 X
5,020,014	5/1991	Miller et al.	364/723

Primary Examiner—Gary V. Harkcom
Assistant Examiner—Mark K. Zimmerman
Attorney, Agent, or Firm—Ronald E. Champion; George A. Leone, Sr.

[57] ABSTRACT

A method and apparatus for providing a texture mapped perspective view for digital map systems. The system includes apparatus for storing elevation data, apparatus for storing texture data, apparatus for scanning a projected view volume from the elevation data storing apparatus, apparatus for processing, apparatus for generating a plurality of planar polygons and apparatus for rendering images. The processing apparatus further includes apparatus for receiving the scanned projected view volume from the scanning apparatus, transforming the scanned projected view volume from object space to screen space, and computing surface normals at each vertex of each polygon so as to modulate texture space pixel intensity. The generating apparatus generates the plurality of planar polygons from the transformed vertices and supplies them to the rendering apparatus which then shades each of the planar polygons. In one alternate embodiment of the invention, the polygons are shaded by apparatus of the rendering apparatus assigning one color across the surface of each polygon. In yet another alternate embodiment of the invention, the rendering apparatus interpolates the intensities between the vertices of each polygon in a linear fashion as in Gouraud shading.

8 Claims, 7 Drawing Sheets

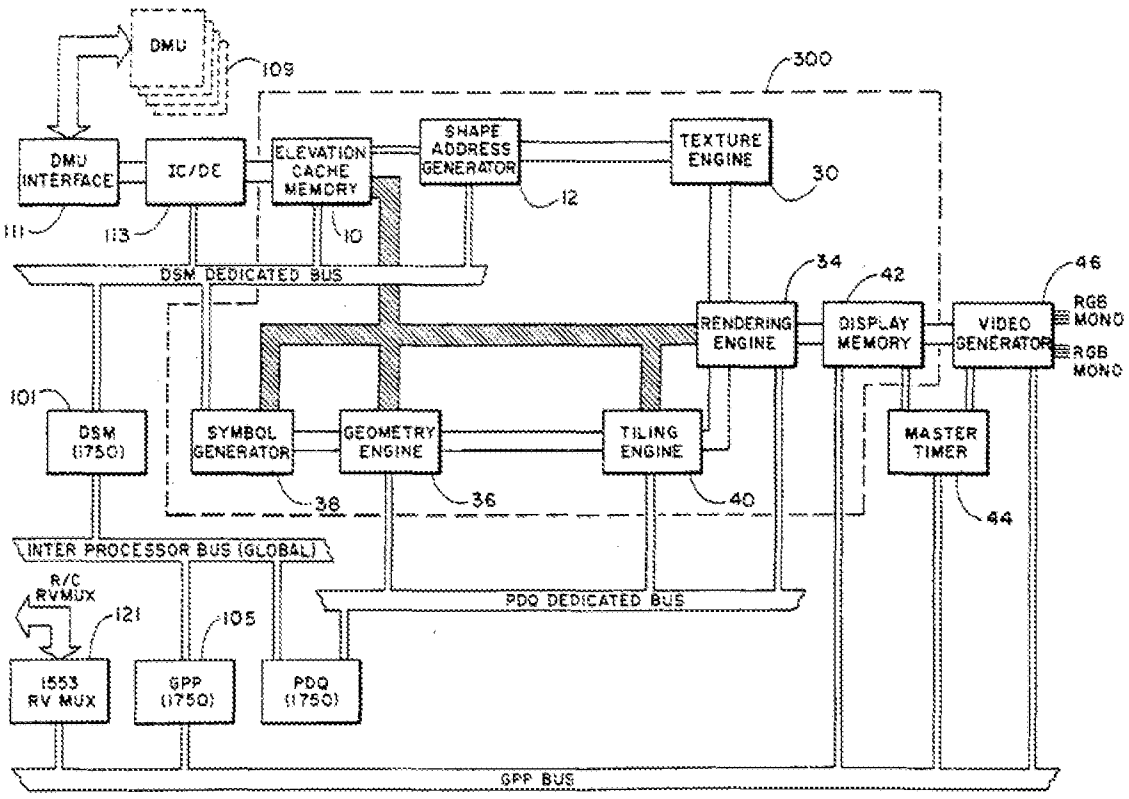


Fig.-1

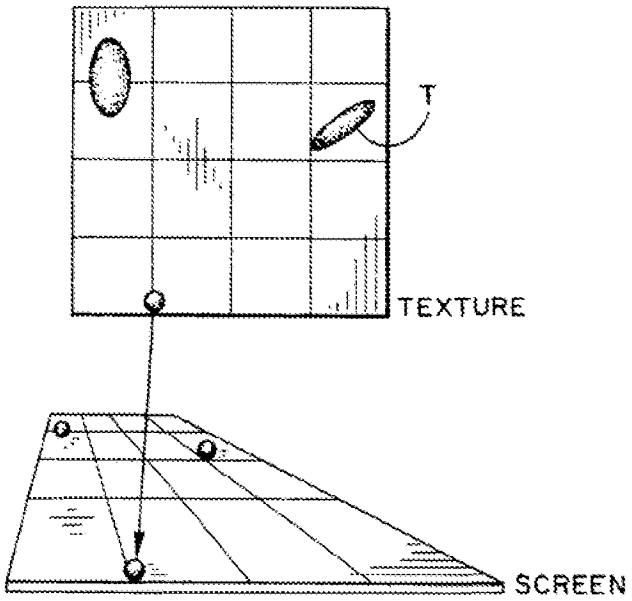

T
TEXTURE
SCREEN

Fig.-2

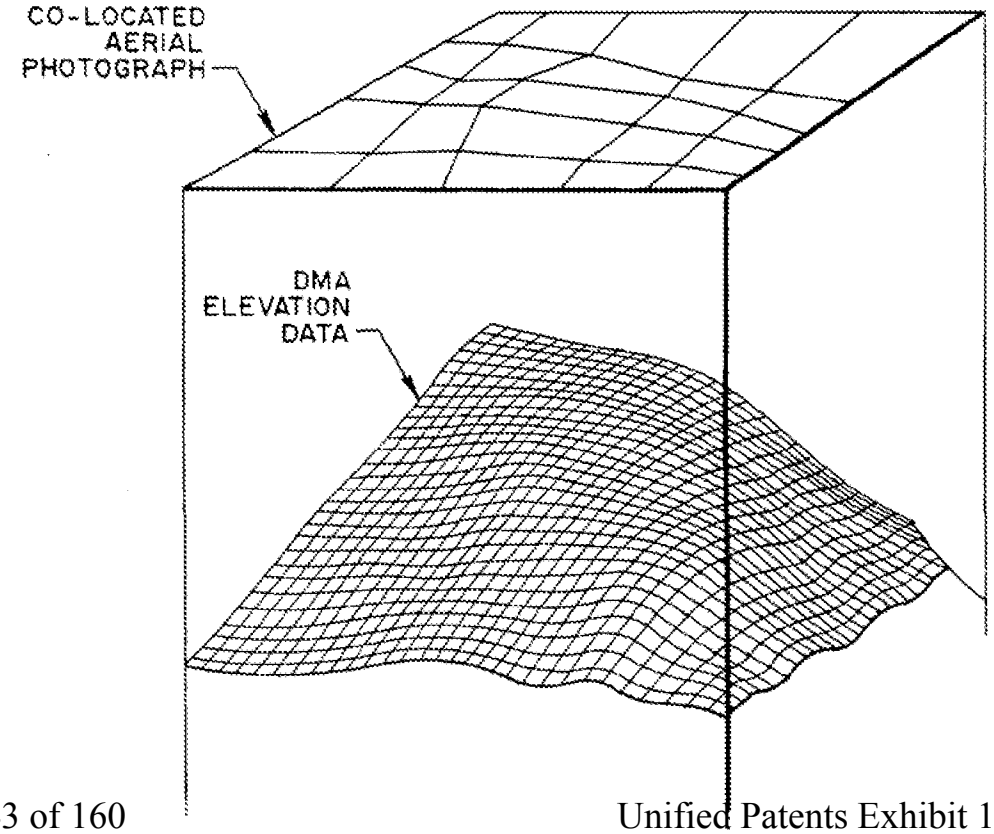

CO-LOCATED
AERIAL
PHOTOGRAPH
DMA
ELEVATION
DATA

Fig.-3

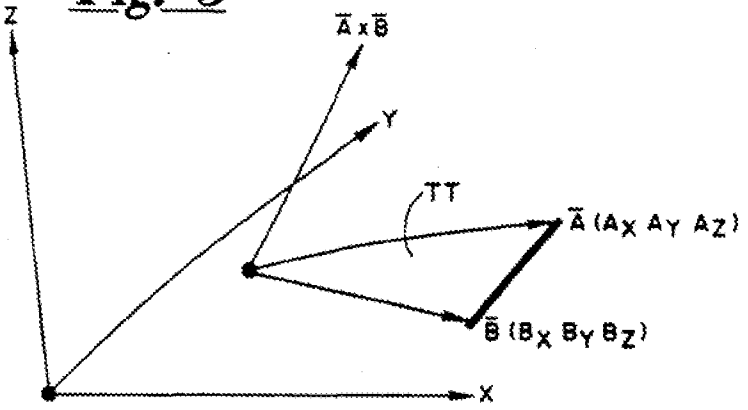
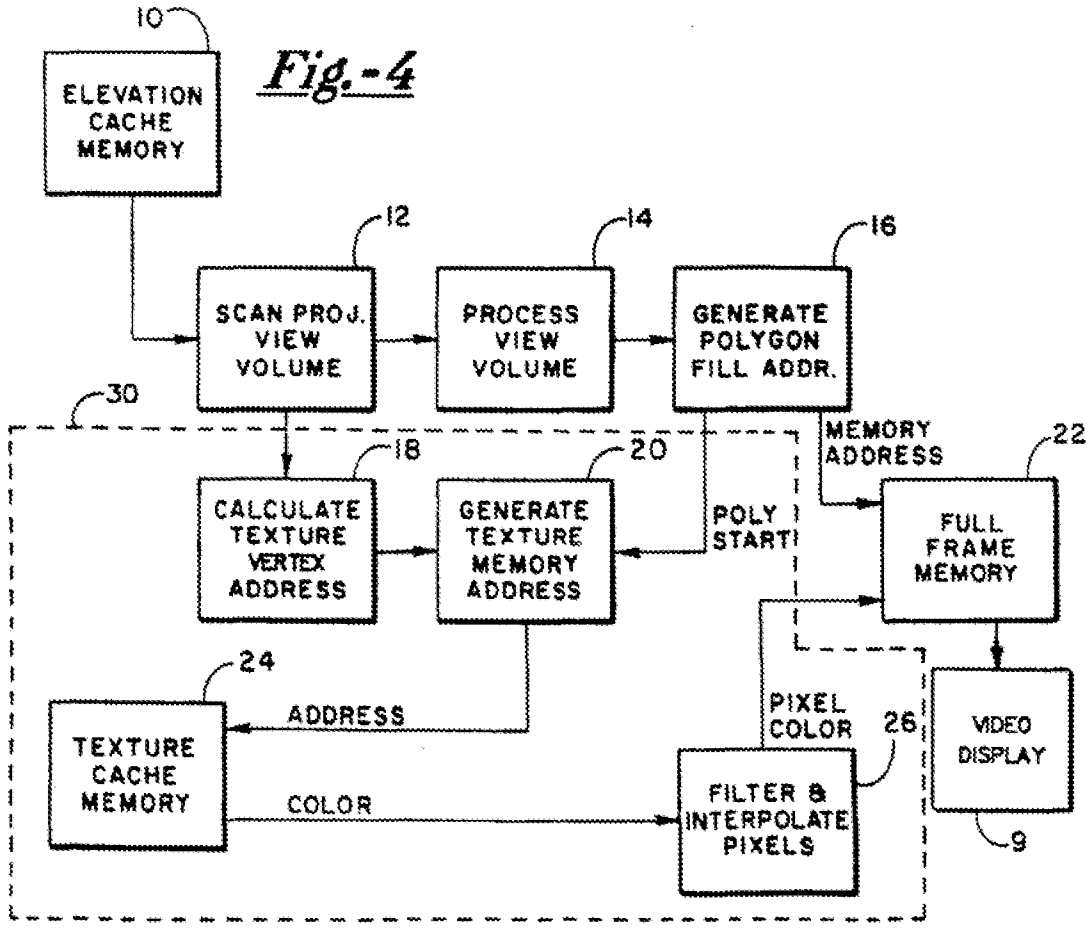

Z
$\bar{A} \times \bar{B}$
Y
TT
$\bar{A}$ (A_X A_Y A_Z)
$\bar{B}$ (B_X B_Y B_Z)
X

Fig.-4


10
ELEVATION CACHE MEMORY
12
SCAN PROJ. VIEW VOLUME
14
PROCESS VIEW VOLUME
16
GENERATE POLYGON FILL ADDR.
30
18
CALCULATE TEXTURE VERTEX ADDRESS
20
GENERATE TEXTURE MEMORY ADDRESS
POLY START
MEMORY ADDRESS
22
FULL FRAME MEMORY
24
TEXTURE CACHE MEMORY
ADDRESS
PIXEL COLOR
26
FILTER & INTERPOLATE PIXELS
COLOR
VIDEO DISPLAY
9

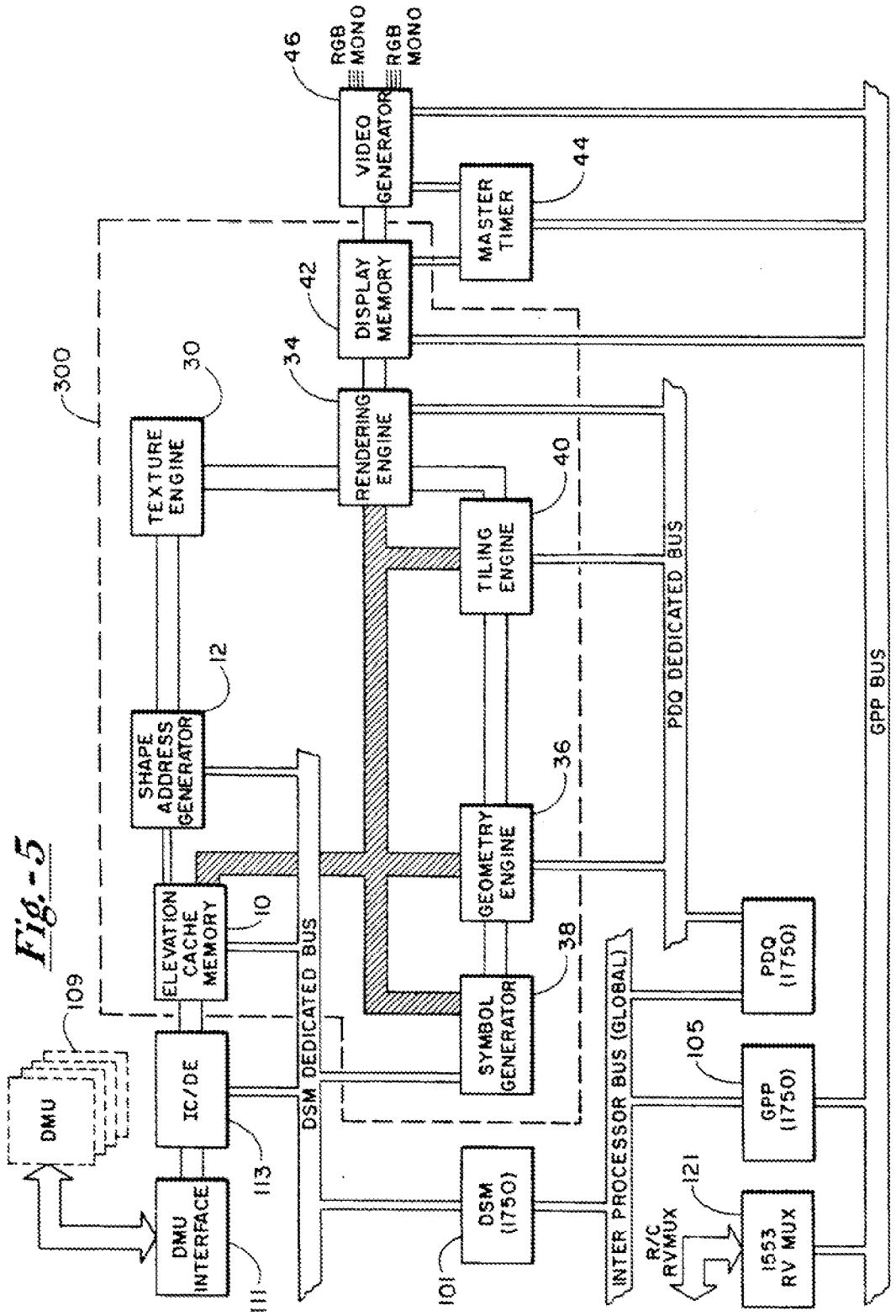

46
RGB MONO
RGB MONO
VIDEO GENERATOR
MASTER TIMER
44
DISPLAY MEMORY
42
34
RENDERING ENGINE
300
30
TEXTURE ENGINE
40
TILING ENGINE
12
SHAPE ADDRESS GENERATOR
36
GEOMETRY ENGINE
ELEVATION CACHE MEMORY
10
38
SYMBOL GENERATOR
109
DMU
IC/DE
113
DMU INTERFACE
111
DSM DEDICATED BUS
DSM (1750)
101
INTER PROCESSOR BUS (GLOBAL)
PDQ DEDICATED BUS
PDQ (1750)
105
GPP (1750)
121
R/C RVMUX
1553 RV MUX
GPP BUS

Fig. 5

Fig.-6

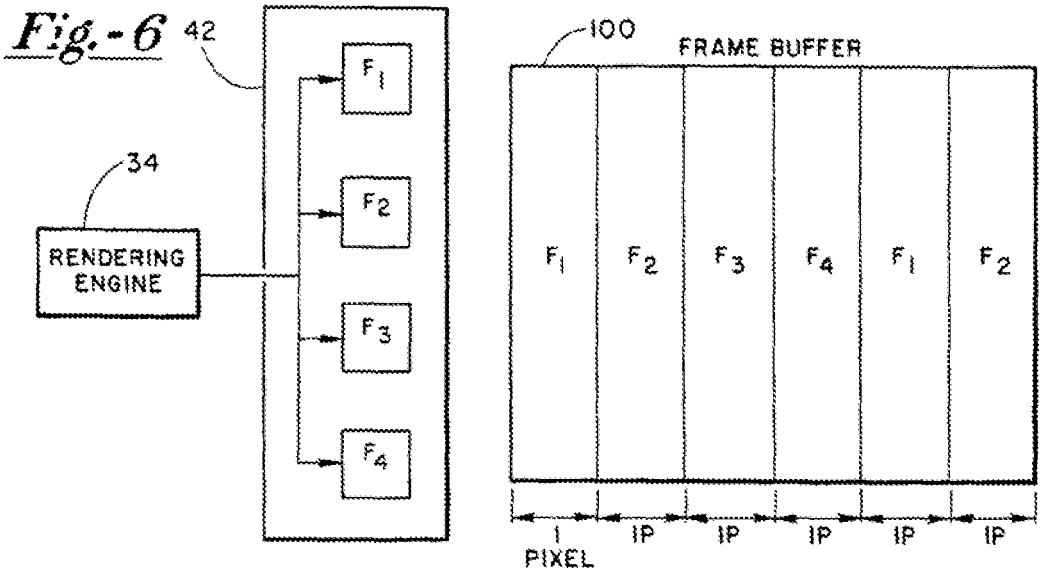


Fig.-7A

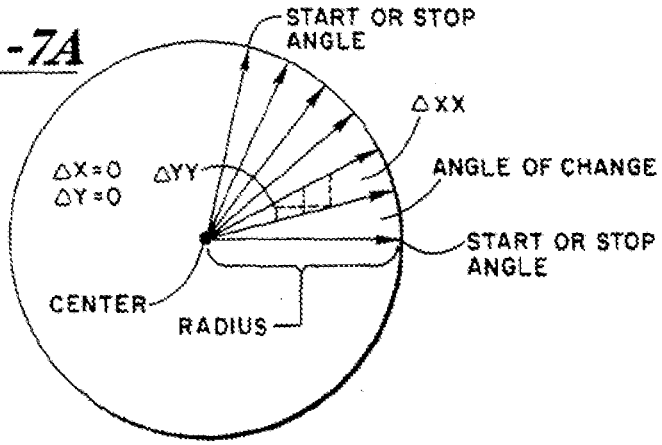


Fig.-7B

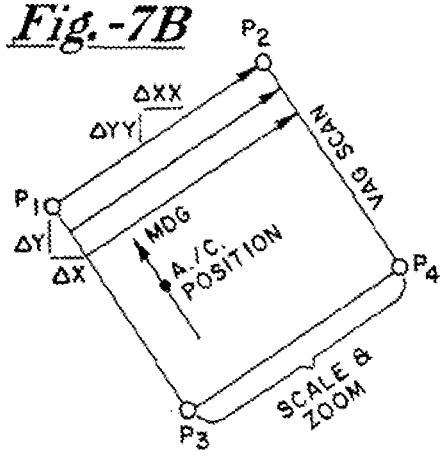


Fig.-7C

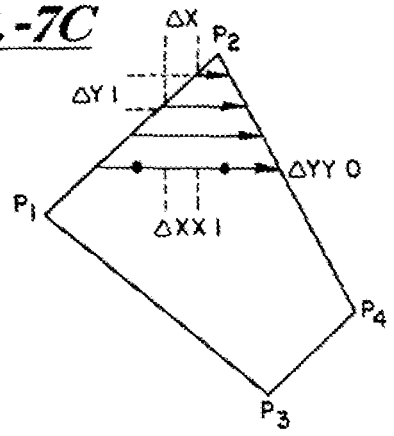
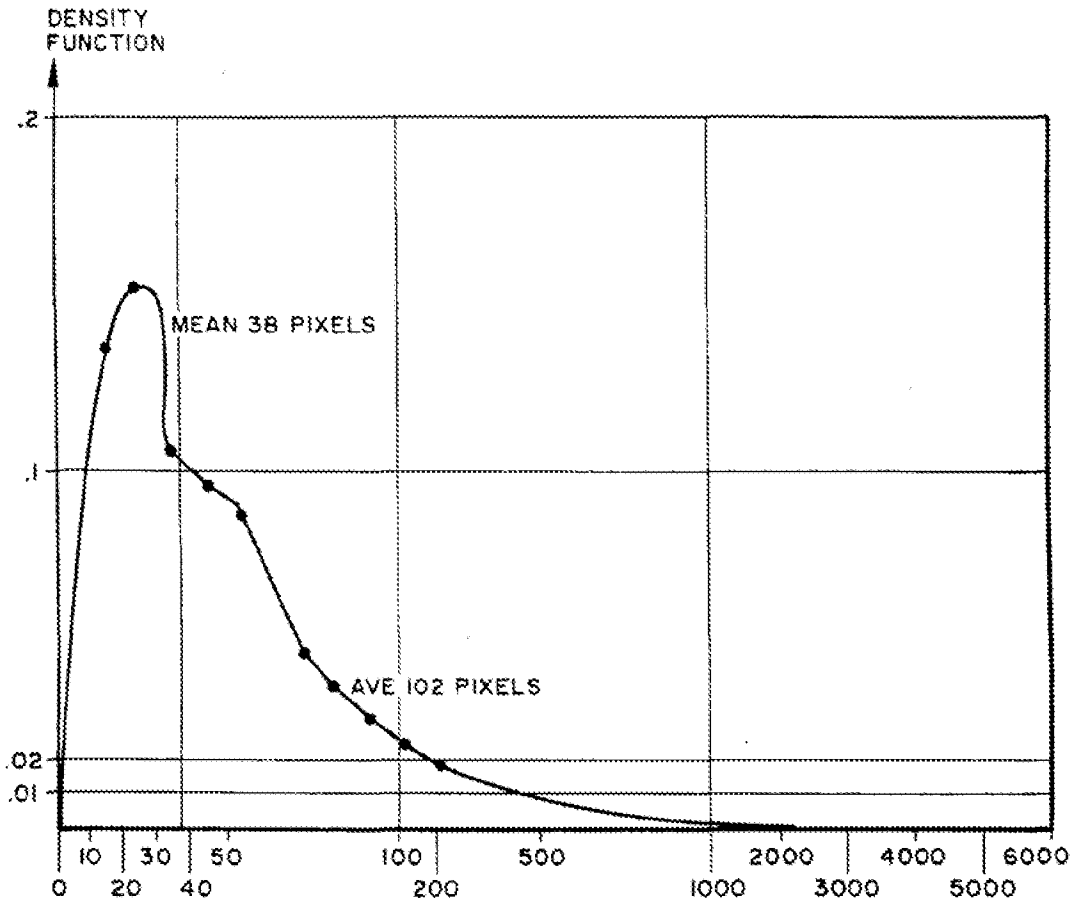


Fig.-8


DENSITY FUNCTION
.2
.1
.02
.01
MEAN 38 PIXELS
AVE 102 PIXELS
0
10
20
30
40
50
100
200
500
1000
2000
3000
4000
5000
6000

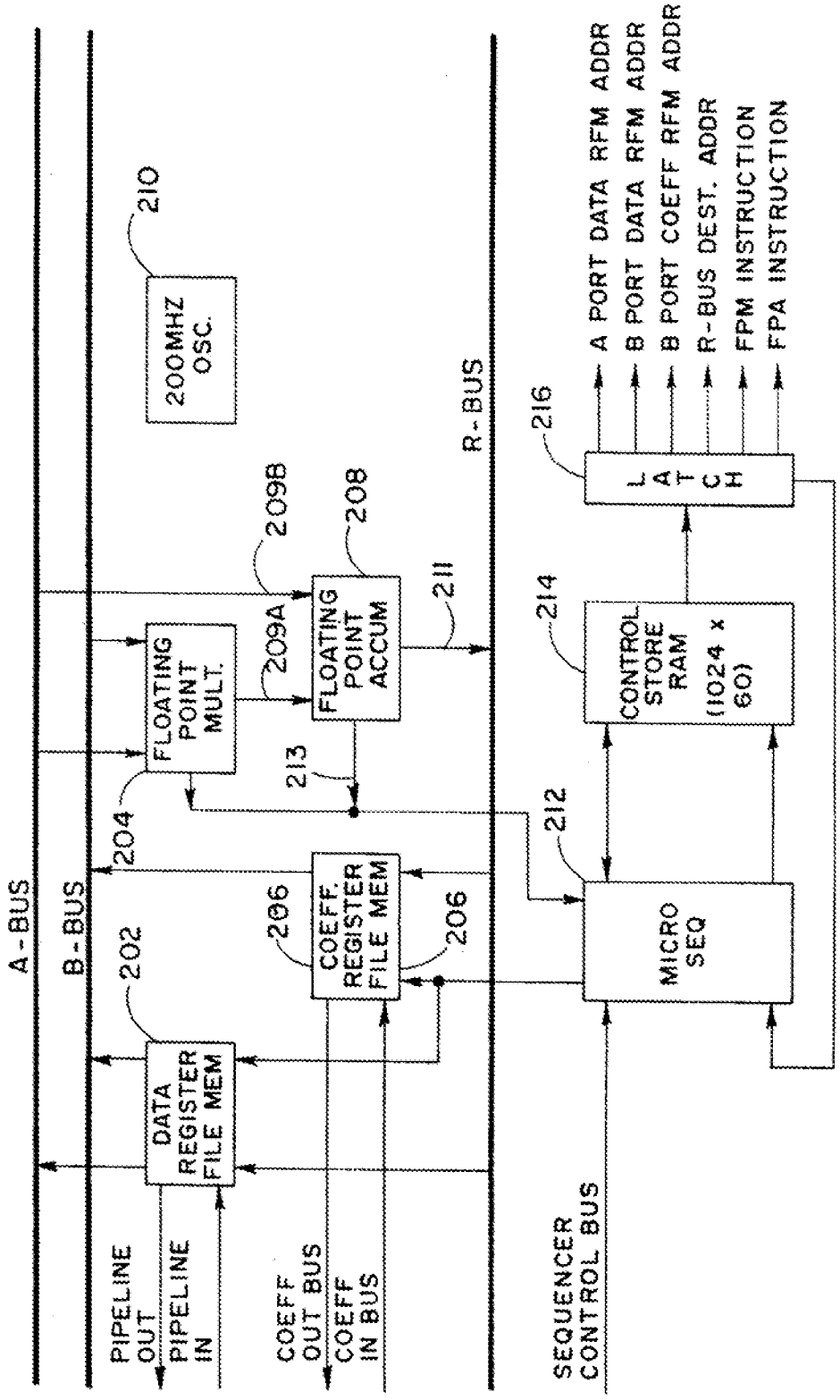

A-BUS
B-BUS
R-BUS
200MHZ OSC.
210
FLOATING POINT MULT.
209A
209B
FLOATING POINT ACCUM
208
211
213
204
202
DATA REGISTER FILE MEM
206
206
COEFF. REGISTER FILE MEM
212
MICRO SEQ
214
CONTROL STORE RAM (1024 x 60)
216
LATCH
A PORT DATA RFM ADDR
B PORT DATA RFM ADDR
B PORT COEFF RFM ADDR
R-BUS DEST. ADDR
FPM INSTRUCTION
FPA INSTRUCTION
PIPELINE OUT
PIPELINE IN
COEFF OUT BUS
COEFF IN BUS
SEQUENCER CONTROL BUS

Fig.-9

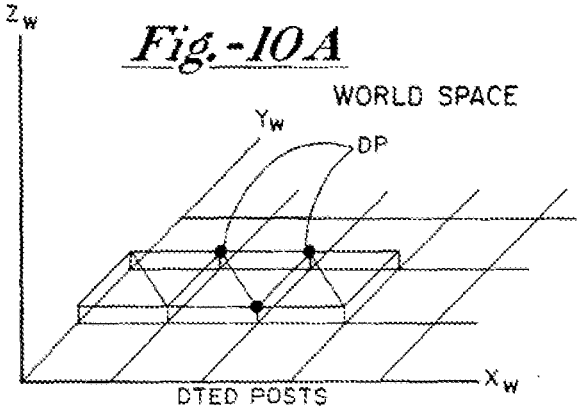


Fig.-10A
Z_W
WORLD SPACE
Y_W
DP
DTED POSTS
X_W

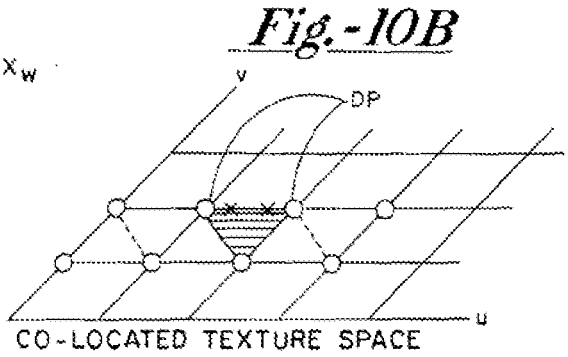


Fig.-10B
v
DP
CO-LOCATED TEXTURE SPACE
u

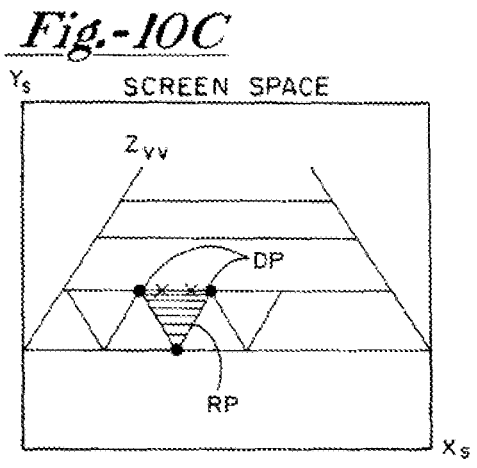


Fig.-10C
Y_S
SCREEN SPACE
Z_{VV}
DP
RP
X_S

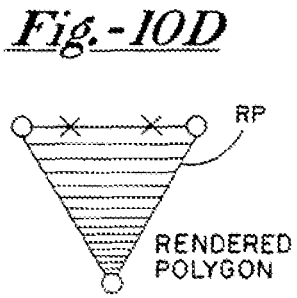


Fig.-10D
RP
RENDERED POLYGON

METHOD AND APPARATUS FOR GENERATING A TEXTURE MAPPED PERSPECTIVE VIEW

The present invention is directed generally to graphic display systems and, more particularly, to a method and apparatus for generating texture mapped perspective views for a digital map system.

RELATED APPLICATIONS

The following applications are included herein by reference:

(1) U.S. Pat. No. 4,876,651 filed May 11, 1988, issued Oct. 24, 1989 entitled "Digital Map System" which was assigned to the assignee of the present invention;

(2) Assignee copending application Ser. No. 09/514,685 filed Apr. 26, 1990, entitled "High Speed Processor for Digital Signal Processing";

(3) U.S. Pat. No. 4,884,220 entitled "Generator with Variable Scan Patterns" filed Jun. 7, 1988, issued Nov. 28, 1989, which is assigned to the assignee of the present invention;

(4) U.S. Pat. No. 4,899,293 entitled "A method of Storage and Retrieval of Digital Map Data Based Upon a Tessellated Geoid System", filed Dec. 14, 1988, issued Feb. 6, 1990;

(5) U.S. Pat. No. 5,020,014 entitled "Generic Interpolation Pipeline Processor", filed Feb. 7, 1989, issued May 28, 1991, which is assigned to the assignee of the present invention;

(6) Assignee's copending patent application Ser. No. 07/732,725 filed Jul. 18, 1991 entitled "Parallel Polygon/Pixel Rendering Engine Architecture for Computer Graphics" which is a continuation of patent application 07/419,722 filed Oct. 11, 1989 now abandoned;

(7) Assignee's copending patent application Ser. No. 07/514,724 filed Apr. 26, 1990 entitled "Polygon Tiling Engine";

(8) Assignee's copending patent application Ser. No. 07/514,723 filed Apr. 26, 1990 entitled "Polygon Sort Engine"; and

(9) Assignee's copending patent application Ser. No. 07/514,742 filed Apr. 26, 1990 entitled "Three Dimensional Computer Graphic Symbol Generator".

BACKGROUND OF THE INVENTION

Texture mapping is a computer graphics technique which comprises a process of overlaying aerial reconnaissance photographs onto computer generated three dimensional terrain images. It enhances the visual reality of raster scan images substantially while incurring a relatively small increase in computational expense. A frequent criticism of known computer-generated synthesized imagery has been directed to the extreme smoothness of the image. Prior art methods of generating images provide no texture, bumps, outcroppings, or natural abnormalities in the display of digital terrain elevation data (DTED).

In general, texture mapping maps a multidimensional image to a multidimensional space. A texture may be thought of in the usual sense such as sandpaper, a plowed field, a roadbed, a lake, woodgrain and so forth or as the pattern of pixels (picture elements) on a sheet of paper or photographic film. The pixels may be arranged in a regular pattern such as a checkerboard or may exhibit high frequencies as in a detailed photograph of high resolution LandSat imagery. Texture may also be three dimensional in nature as in marble or woodgrain surfaces. For the purposes of the invention, texture mapping is defined to be the mapping of a texture onto a surface in three dimensional object space. As is illustrated schematically in FIG. 1, a texture space object T is mapped to a display screen by means of a perspective transformation.

The implementation of the method of the invention comprises two processes. The first process is geometric warping and the second process is filtering. FIG. 2 illustrates graphically the geometric warping process of the invention for applying texture onto a surface. This process applies the texture onto an object to be mapped analogously to a rubber sheet being stretched over a surface. In a digital map system application, the texture typically comprises an aerial reconnaissance photograph and the object mapped is the surface of the digital terrain data base as shown in FIG. 2. After the geometric warping has been completed, the second process of filtering is performed. In the second process, the image is resampled on the screen grid.

The invention provides a texture mapped perspective view architecture which addresses the need for increased aircraft crew effectiveness, consequently reducing workload, in low altitude flight regimes characterized by the simultaneous requirement to avoid certain terrain and threats. The particular emphasis of the invention is to increase crew situational awareness. Crew situational awareness has been increased to some degree through the addition of a perspective view map display to a plan view capability which already exists in digital map systems. See, for example, assignee's copending application Ser. No. 07/192,798, for a DIGITAL MAP SYSTEM, filed May 11, 1988, issued Oct. 24, 1989 as U.S. Pat. No. 4,876,651 which is incorporated herein by reference in its entirety. The present invention improves the digital map system capability by providing a means for overlaying aerial reconnaissance photographs over the computer generated three dimensional terrain image resulting in a one-to-one correspondence from the digital map image to the real world. In this way the invention provides visually realistic cues which augment the informational display of such a computer generated terrain image. Using these cues an aircraft crew can rapidly make a correlation between the display and the real world.

The architectural challenge presented by texture mapping is that of distributing the processing load to achieve high data throughput using parallel pipelines and then recombining the parallel pixel flow into a single memory module known as a frame buffer. The resulting contention for access to the frame buffer reduces the effective throughput of the pipelines in addition to requiring increased hardware and board space to implement the additional pipelines. The method and apparatus of the invention addresses this challenge by effectively combining the low contention attributes of a single high speed pipeline with the increased processing throughput of parallel pipelines.

SUMMARY OF THE INVENTION

A method and apparatus for providing a texture mapped perspective view for digital map systems is provided. The invention comprises means for storing elevation data, means for storing texture data, means for scanning a projected view volume from the elevation data storing means, means for processing the projected view volume, means for generating a plurality of planar polygons and means for rendering images. The process-

ing means further includes means for receiving the scanned projected view volume from the scanning means, transforming the scanned projected view volume from object space to screen space, and computing surface normals at each vertex of each polygon so as to modulate texture space pixel intensity. The generating means generates the plurality of planar polygons from the transformed vertices and supplies them to the rendering means which then shades each of the planar polygons.

A primary object of the invention is to provide a technology capable of accomplishing a fully integrated digital map display system in an aircraft cockpit.

In one alternate embodiment of the invention, the polygons are shaded by means of the rendering means assigning one color across the surface of each polygon.

In yet another alternate embodiment of the invention, the rendering means interpolates the intensities between the vertices of each polygon in a linear fashion as in Gouraud shading.

It is yet another object of the invention to provide a digital map system including capabilities for perspective view, transparency, texture mapping, hidden line removal, and secondary visual effects such as depth cues and artifact (i.e., anti-aliasing) control.

It is yet another object of the invention to provide the capability for displaying forward looking infrared (FLIR) data and radar return images overlaid onto a plan and perspective view digital map image by fusing images through combining or subtracting other sensor video signals with the digital map terrain display.

It is yet another object of the invention to provide a digital map system with an arbitrary warping capability of one data base onto another data base which is accommodated by the perspective view texture mapping capability of the invention.

Other objects, features and advantages of the invention will become apparent to those skilled in the art through the drawings, description of the preferred embodiment and claims herein. In the drawings, like numerals refer to like elements.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** shows the mapping of a textured object to a display screen by a perspective transformation.

FIG. **2** illustrates graphically the geometric warping process of the invention for applying texture onto a surface.

FIG. **3** illustrates the surface normal calculation as employed by the invention.

FIG. **4** presents a functional block diagram of one embodiment of the invention.

FIG. **5** illustrates a top level block diagram of one embodiment of the texture mapped perspective view architecture of the invention.

FIG. **6** schematically illustrates the frame buffer configuration as employed by one embodiment of the invention.

FIGS. *7a*, *7b* and *7c* illustrate three examples of display format shapes.

FIG. **8** graphs the density function for maximum pixel counts.

FIG. **9** is a block diagram of one embodiment of the geometry array processor as employed by the invention.

FIGS. **10A**, **10B**, **10C** and **10D** illustrated the tagged architectural texture mapping as provided by the invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT

Generally, perspective transformation from texture space having coordinates U, V to screen space having coordinates X, Y requires an intermediate transformation from texture space to object space having coordinates X_0, Y_0, Z_0. Perspective transformation is accomplished through the general perspective transform equation as follows:

$$[X\ Y\ Z\ H] = [X\ Y\ Z\ 1]\,X\left[\begin{array}{ccc|c} A & B & C & P \\ D & E & F & Q \\ G & H & I & R \\ \hline L & M & N & S \end{array}\right]$$

where a point (X,Y,Z) in 3-space is represented by a four dimensional position vector [X Y Z H] in homogeneous coordinates.

The 3×3 sub-matrix

$$\left[\begin{array}{ccc} A & B & C \\ D & E & F \\ G & H & I \end{array}\right]$$

accomplishes scaling, shearing, and rotation.

The 1×3 row matrix [L M N] produces translation.

The 3×1 column matrix

$$\left[\begin{array}{c} P \\ Q \\ R \end{array}\right]$$

produces perspective transformation.

The 1×1 scalar [S] produces overall scaling.

The Cartesian cross-product needed for surface normal requires a square root. As shown in FIG. **3**, the surface normal shown is a vector $A\times B$ perpendicular to the plane formed by edges of a polygon as represented by vectors A and B, where $A\times B$ is the Cartesian cross-product of the two vectors. Normalizing the vector allows calculation for sun angle shading in a perfectly diffusing Lambertian surface. This is accomplished by taking the vector dot product of the surface normal vector with the sun position vector. The resulting angle is inversely proportional to the intensity of the pixel of the surface regardless of the viewing angle. This intensity is used to modulate the texture hue and intensity value.

$$\frac{A\times B}{\|A\|\,\|B\|} \quad \text{where} \quad \begin{array}{l} A = Ax^2 + Ay^2 + Az^2 \\ B = Bx^2 + By^2 + Bz^2 \end{array}$$

A terrain triangle TT is formed by connecting the endpoints of vectors A and B, from point B_X, B_Y, B_Z to point A_X, A_Y, A_Z.

Having described some of the fundamental basis for the invention, a description of the method of the invention will now be set out in more detail below.

Referring now to FIG. **4**, a functional block diagram of one embodiment of the invention is shown. The invention functionally comprises a means for storing ele-

vation data **10**, a means for storing texture data **24**, a means for scanning a projected view volume from the elevation data storing means **12**, means for processing view volume **14** including means for receiving the scanned projected view volume from the scanning means **12**, means for generating polygon fill addresses **16**, means for calculating texture vertices addresses **18**, means for generating texture memory addresses **20**, means for filtering and interpolating pixels **26**, a full-frame memory **22**, and video display **9**. The processing means **14** further includes means for transforming the scanned projected view volume from object space to screen space and means for computing surface normals at each vertex of each polygon so as to calculate pixel intensity.

The means for storing elevation data **10** may preferably be a cache memory having at least a 50 nsec access time to achieve 20 Hz bi-linear interpolation of a 512×512 pixel resolution screen. The cache memory further may advantageously include a 256×256 bit buffer segment with 2K bytes of shadow RAM used for the display list. The cache memory may arbitrarily be reconfigured from 8 bits deep (data frame) to 64 bits (i.e., comprising the sum of texture map data (24 bits)+DTED (16 bits)+aeronautical chart data (24 bits)). A buffer segment may start at any cache address and may be written horizontally or vertically. Means for storing texture data **24** may advantageously be a texture cache memory which is identical to the elevation cache memory except that it stores pixel information for warping onto the elevation data cache. Referring now to FIG. **5**, a top level block diagram of the texture mapped perspective view architecture is shown. The architecture implements the functions as shown in FIG. **4** and the discussion which follows shall refer to functional blocks in FIG. **4** and corresponding elements in FIG. **5**. In some cases, such as element **14**, there is a one-to-one correspondence between the functional blocks in FIG. **4** and the architectural elements of FIG. **5**. In other cases, as explained hereinbelow, the functions depicted in FIG. **4** are carried out by a plurality of elements shown in FIG. **5**. The elements shown in FIG. **5** comprising the texture mapped perspective view system **300** of the invention include elevation cache memory **10**, shape address generator (SHAG) **12**, texture engine **30**, rendering engine **34**, geometry engine **36**, symbol generator **38**, tiling engine **40**, and display memory **42**. These elements are typically part of a larger digital map system including a digital map unit (DMU) **109**, DMU interface **111**, IC/DE **113**, a display stream manager (DSM) **101**, a general purpose processor (GPP) **105**, RV MUX **121**, PDQ **123**, master time **44**, video generator **46** and a plurality of data bases. The latter elements are described in assignee's Digital Map System U.S. Pat. No. 4,876,651.

GEOMETRY ENGINE

The geometry engine **36** is comprised of one or more geometry array processors (GAPs) which process the 4×4 Euler matrix transformation from object space (sometimes referred to as "world" space) to screen space. The GAPs generate X and Y values in screen coordinates and Zvv values in range depth. The GAPs also compute surface normals at each vertex of a polygon representing an image in object space via Cartesian cross-products for Gouraud shading, or they may assign one surface normal to the entire polygon for flat shading and wire mesh. Intensity calculations are performed using a vector dot product between the surface normal or normals and the illumination source to implement a Lambertian diffusely reflecting surface. Hue and intensity values are then assigned to the polygon. The method and apparatus of the invention also provides a dot rendering scheme wherein the GAPs only transform one vertex of each polygon and the tiling engine **40**, explained in more detail below, is inhibited. In this dot rendering format, hue and intensity are assigned based on the planar polygon containing the vertex and the rendering engine is inhibited. Dot polygons may appear in the same image as multiple vertex polygons or may comprise the entire image itself. The "dots" are passed through the polygon rendering engine **34**. A range to the vertices or polygon (Zvv) is used if a fog or "DaVinci" effect are invoked as explained below. The GAPs also transform three dimensional overlay symbols from world space to screen space.

Referring now to FIG. **9**, a block diagram of one example embodiment of a geometry array processor (GAP) is shown. The GAP comprises a data register file memory **202**, a floating point multiplier **204**, a coefficient register file memory **206**, a floating point accumulator **208**, a 200 MHz oscillator **210**, a microsequencer **212**, a control store RAM **214**, and latch **216**.

The register file memory may advantageously have a capacity of 512 by 32 bits. The floating point accumulator **208** includes two input ports **209A** and **209B** with independent enables, one output port **211**, and a condition code interface **212** responsive to error codes. The floating point accumulator operates on four instructions, namely, multiply, no-op, pass A, and pass B. The microsequencer **212** operates on seven instructions including loop on count, loop on condition, jump, continue, call, return and load counter. The microsequencer includes a debug interface having a read/write (R/W) internal register, R/W control store memory, halt on address, and single step, and further includes a processor interface including a signal interrupt, status register and control register. The GAP is fully explained in the assignee's co-pending application No. 07/514,685 filed Apr. 26, 1990 entitled High Speed Processor for Digital Signal Processing which is incorporated herein by reference in its entirety.

In one alternative embodiment of the invention, it is possible to give the viewer of the display the visual effect of an environment enshrouded in fog. The fog option is implemented by interpolating the color of the triangle vertices toward the fog color. As the triangles get smaller with distance, the fog particles become denser. By using the known relationship between distance and fog density, the fog thickness can be "dialed" or adjusted as needed. The vertex assignment interpolates the vertex color toward the fog color as a function of range toward the horizon. The fog technique may be implemented in the hardware version of the GAP such as may be embodied in a GaAs semiconductor chip. If a linear color space (typically referred to as "RGB" to reflect the primary colors, red, green and blue) is assumed, the amount of fog is added as a function of range to the polygon vertices' color computation by well known techniques. Thus, as the hue is assigned by elevation banding or monochrome default value, the fog color is tacked on. The rendering engine **34**, explained in more detail below, then straight forwardly interpolates the interior points.

In another alternative embodiment of the invention, a DaVinci effect is implemented. The DaVinci effect

causes the terrain to fade into the distance and blend with the horizon. It is implemented as a function of range of the polygon vertices by the GAP. The horizon color is added to the vertices similarly to the fog effect.

SHAPE ADDRESS GENERATOR (SHAG)

The SHAG **12** receives the orthographically projected view volume outline onto cache from the DSM. It calculates the individual line lengths of the scans and the delta x and delta y components. It also scans the elevation posts out of the elevation cache memory and passes them to the GAPs for transformation. In one embodiment of the invention, the SHAG preferably includes two arithmetic logic units (ALUs) to support the **50** nsec cache **10**. In the SHAG, data is generated for the GAPs and control signals are passed to the tiling engine **40**. DFAD data is downloaded into overlay RAM (not shown) and three dimensional symbols are passed to the GAPs from symbol generator **38**. Elevation color banding hue assignment is performed in this function. The SHAG generates shapes for plan view, perspective view, intervisibility, and radar simulation. These are illustrated in FIG. 7. The SHAG is more fully explained in assignee's copending application, Ser. No. 203,660, Generator With Variable Scan Patterns, filed Jun. 7, 1988 issued as U.S. Pat. No. 4,884,220 on Nov. 28, 1989 which is incorporated herein by reference in its entirety.

A simple Lambertian lighting diffusion model has proved adequate for generating depth cueing in one embodiment of the invention. The sun angle position is completely programmable in azimuth and zenith. It may also be self-positioning based on time of day, time of year, latitude and longitude. A programmable intensity with gray scale instead of color implements the moon angle position algorithm. The display stream manager (DSM) programs the sun angle registers. The illumination intensities of the moon angle position may be varied with the lunar waxing and waning cycles.

TILING ENGINE AND TEXTURE ENGINE

Still referring to FIGS. **4** and **5**, the means for calculating texture vertex address **18** may include the tiling engine **40**. Elevation posts are vertices of planar triangles modeling the surface of the terrain. These posts are "tagged" with the corresponding U,V coordinate address calculated in texture space. This tagging eliminates the need for interpolation by substituting an address lookup. Referring to FIGS. **10A**, **10B**, **10C** and **10D**, with continuing reference to FIGS. **4** and **5**, the tagged architectural texture mapping as employed by the invention is illustrated. FIG. **10A** shows an example of DTED data posts, DP, in world space. FIG. **10B** shows the co-located texture space for the data posts. FIG. **10C** shows the data posts and rendered polygon in screen space. FIG. **10D** illustrates conceptually the interpolation of tagged addresses into a rendered polygon RP. The texture engine **30** performs the tagged data structure management and filtering processes. When the triangles are passed to the rendering engine by the tiling engine for filling with texture, the tagged texture address from the elevation post is used to generate the texture memory address. The texture value is filtered by filtering and interpolation means **26** before being written to full-frame memory **22** prior to display.

The tiling engine generates the planar polygons from the transformed vertices in screen coordinates and passes them to the rendering engine. For terrain polygons, a connectivity offset from one line scan to the next is used to configure the polygons. For overlay symbols, a connectivity list is resident in a buffer memory (not shown) and is utilized for polygon generation. The tiling engine also informs the GAP if it is busy. In one embodiment 512 vertices are resident in a 1K buffer.

All polygons having surface normals more than 90 degrees from LOS are eliminated from rendering. This is known in the art as backface removal. Such polygons do not have to be transformed since they will not be visible on the display screen. Additional connectivity information must be generated if the polygons are non-planar as the transformation process generates implied edges. This requires that the connectivity information be dynamically generated. Thus, only planar polygons with less than 513 vertices are implemented. Non-planar polygons and dynamic connectivity algorithms are not implemented by the tiling engine. The tiling engine is further detailed in assignee's copending applications of even filing date herewith entitled Polygon Tiling Engine, as referenced hereinabove and Polygon Sort Engine, as referenced hereinabove, both of which are incorporated herein by reference.

RENDERING ENGINE

Referring again to FIG. **5**, the rendering engine **34** of the invention provides a means of drawing polygons in a plurality of modes. The rendering engine features may include interpolation algorithms for processing coordinates and color, hidden surface removal, contour lines, aircraft relative color bands, flat shading, Gourand shading, phong shading, mesh format or screen door effects, ridgeline display, transverse slice, backface removal and RECE (aerial reconnaissance) photo modes. With most known methods of image synthesis, the image is generated by breaking the surfaces of the object into polygons, calculating the color and intensity at each vertex of the polygon, and drawing the results into a frame buffer while interpolating the colors across the polygon. The color information at the vertices is calculated from light source data, surface normal, elevation and/or cultural features.

The interpolation of coordinate and color (or intensity) across each polygon must be performed quickly and accurately. This is accomplished by interpolating the coordinate and color at each quantized point or pixel on the edges of the polygon and subsequently interpolating from edge to edge to generate the fill lines. For hidden surface removal, such as is provided by a Z-buffer in a well-known manner, the depth or Z-value for each pixel is also calculated. Furthermore, since color components can vary independently across a surface or set of surfaces, red, green and blue intensities are interpolated independently. Thus, a minimum of six different parameters (X,Y,Z,R,G,B) are independently calculated when rendering polygons with Gouraud shading and interpolated Z-values.

Additional features of the rendering engine include a means of providing contour lines and aircraft relative color bands. For these features the elevation also is interpolated at each pixel. Transparency features dictate that an alpha channel be maintained and similarly interpolated. These requirements imply two additional axes of interpolation bringing the total to eight. The rendering engine is capable of processing polygons of one vertex in its dot mode, two vertices in its line mode, and three to 512 coplanar vertices in its polygon mode.

In the flat shading mode the rendering engine assigns the polygon a single color across its entire surface. An arbitrary vertex is selected to assign both hue and intensity for the entire polygon. This is accomplished by assigning identical RGB values to all vertices. Interpolation is performed normally but results in a constant value. This approach will not speed up the rendering process but will perform the algorithm with no hardware impact.

The Gouraud shading algorithm included in the rendering engine interpolates the intensities between the vertices of each polygon rendered in a linear fashion. This is the default mode. The Phong shading algorithm interpolates the surface normals between the vertices of the polygon between applying the intensity calculations. The rendering engine would thus have to perform an illumination calculation at each pixel after interpolation. This approach would significantly impact the hardware design. This algorithm may be simulated, however, using a weighing function (typically a function of cosine (Θ)) around a narrow band of the intensities. This results in a non-linear interpolation scheme and provides for a simulated specular reflectance. In an alternative embodiment, the GAP may be used to assign the vertices of the polygon this non-linear weighing via the look-up table and the rendering engine would interpolate as in Gouraud shading.

Transparency is implemented in the classical sense using an alpha channel or may be simulated with a screen door effect. The screen door effect simply renders the transparent polygon as normal but then only outputs every other or every third pixel. The mesh format appears as a wire frame overlay with the option of rendering either hidden lines removed or not. In the case of a threat dome symbol, all polygon edges must be displayed as well as the background terrain. In such a case, the fill algorithm of the rendering engine is inhibited and only the polygon edges are rendered. The intensity interpolation is performed on the edges which may have to be two pixels wide to eliminate strobing. In one embodiment, an option for terrain mesh includes the capability for tagging edges for rendering so that the mesh appears as a regular orthogonal grid.

Typical of the heads up display (HUD) format used in aircraft is the ridgeline display and the transverse slice. In the ridgeline format, a line drawing is produced from polygon edges whose slopes change sign relative to the viewpoint. All polygons are transformed, tiled, and then the surface normals are computed and compared to the viewpoint. The tiling engine strips away the vertices of non-ridge contributing edges and passes only the ridge polygons to the rendering engine. In transverse slice mode, fixed range bins relative to the aircraft are defined. A plane orthogonal to the view LOS is then passed through for rendering. The ridges then appear to roll over the terrain as the aircraft flies along. These algorithms are similar to backface removal. They rely upon the polygon surface normal being passed to the tiling engine.

One current implementation of the invention guarantees non-intersecting polygon sides by restricting the polygons rendered to be planar. They may have up to 512 vertices. Polygons may also consist of one or two vertices. The polygon "end" bit is set at the last vertex and processed by the rendering engine. The polygon is tagged with a two bit rendering code to select mesh, transparent, or Gouraud shading. The rendering engine also accomplishes a fine clip to the screen for the polygon and implements a smoothing function for lines.

An optional aerial reconnaissance (RECE) photo mode causes the GAP to texture map an aerial reconnaissance photograph onto the DTED data base. In this mode the hue interpolation of the rendering engine is inhibited as each pixel of the warping is assigned a color from the RECE photo. The intensity component of the color is dithered in a well known manner as a function of the surface normal as well as the Z-depth. These pixels are then processed by the rendering engine for Z-buffer rectification so that other overlays such as threats may be accommodated. The RECE photos used in this mode have been previously warped onto a tessellated geoid data base and thus correspond pixel-for-pixel to the DTED data. See assignee's aforereferenced copending application for A Method of Storage and Retrieval of Digital Map Data Based Upon A Tessellated Geoid System, which is hereby incorporated by reference in its entirety. The photos may be denser than the terrain data. This implies a deeper cache memory to hold the RECE photos. Aeronautical chart warping mode is identical to RECE photos except that aeronautical charts are used in the second cache. DTED warping mode utilizes DTED data to elevation color band aeronautical charts.

The polygon rendering engine may preferably be implemented in a generic interpolation pipeline processor (GIPP) of the type as disclosed in assignee's aforereferenced patent entitled Generic Interpolation Pipeline Processor, which is incorporated herein by reference in its entirety. In one embodiment of the invention, the GIPPs fill in the transformed polygons using a bi-linear interpolation scheme with six axes (X,Y,Z,R,G,B). The primitive will interpolate a 16 bit pair and 8 bit pair of values simultaneously, thus requiring 3 chips for a polygon edge. One embodiment of the system of the invention has been sized to process one million pixels each frame time. This is sufficient to produce a 1K×1K high resolution chart, or a 512×512 DTED frame with an average of four overwrites per pixel during hidden surface removal with GIPPs outputting data at a 60 nsec rate, each FIFO, F1-F4, as shown in FIG. 6, will receive data on the average of every 240 nsec. An even distribution can be assumed by decoding on the lower 2X address bits. Thus, the memory is divided into one pixel wide columns FIG. 6 is discussed in more detail below.

Referring again to FIGS. 4 and 5, the "dots" are passed through the GIPPs without further processing. Thus, the end of each polygon's bit is set. A ZB buffer is needed to change the color of a dot at a given pixel for hidden dot removal. Perspective depth cuing is obtained as the dots get closer together as the range from the viewpoint increases.

Bi-linear interpolation mode operates in plan view on either DLMS or aeronautical charts. It achieves 20 Hz interpolation on a 512×512 display. The GIPPs perform the interpolation function.

DATA BASES

A Level I DTED data base is included in one embodiment of the invention and is advantageously sampled on three arc second intervals. Buffer segments are preferably stored at the highest scales (104.24 nm) and the densest data (13.03 nm). With such a scheme, all other scales can be created. A Level II DTED data base is also included and is sampled at one arc second inter-

vals. Buffer segments are preferably stored only at the densest data (5.21 nm).

A DFAD cultural feature data base is stored in a display list of 2K words for each buffer segment. The data structure consists of an icon font call, a location in cache, and transformation coefficients from model space to world space consisting of scaling, rotation, and position (translation). A second data structure comprises a list of polygon vertices in world coordinates and a color or texture. The DFAD data may also be rasterized and overlaid on a terrain similar to aerial reconnaissance photos.

Aeronautical charts at the various scales are warped into the tessellated geoid. This data is 24 bits deep. Pixel data such as LandSat, FLIR, data frames and other scanned in source data may range from one bit up to 24 bits in powers of two (1,2,4,8,16,24).

FRAME BUFFER CONFIGURATION

Referring again to FIG. **6**, the frame buffer configuration of one embodiment of the invention is shown schematically. The frame buffer configuration is implemented by one embodiment of the invention comprises a polygon rendering chip **34** which supplies data to full-frame memory **42**. The full-frame memory **42** advantageously includes first-in, first-out buffers (FIFO) F_1, F_2, F_3 and F_4. As indicated above with respect to the discussion of the rendering engine, the memory is divided up into one pixel wide columns as shown in FIG. 6. By doing so, however, chip select must changed on every pixel when the master timer **44** shown in FIG. **5** reads the memory. However, by orienting the SHAG scan lines at 90 degrees to the master timer scan lines, the chip select will change on every line. The SHAG starts scanning at the bottom left corner of the display and proceeds to the upper left corner of the display.

With the image broken up in this way, the probability that the GIPP will write to the same FIFO two times in a row, three times, four, and so on can be calculated to determine how deep the FIFO must be. Decoding on the lower order address bits means that the only time the rendering engine will write to the same FIFO twice in a row is when a new scan line is started. At four deep as shown in the frame buffer graph **100**, the chances of the FIFO filling up are approximately one in 6.4K. With an image of 1 million pixels, this will occur an acceptably small number of times for most applications. The perspective view transformations for 10,000 polygons with the power and board area constraints that are imposed by an avionics environment is significant. The data throughput for a given scene complexity can be achieved by adding more pipeline in parallel to the architecture. It is desirable to have as few pipelines as possible, preferably one, so that the image reconstruction at the end of the pipeline does not suffer from an arbitration bottleneck for a Z-buffered display memory.

In one embodiment of the invention, the processing throughput required has been achieved through the use of GaAs VSLI technology for parallel pipelines and a parallel frame buffer design has eliminated contention bottlenecks. A modular architecture allows for additional functions to be added to further the integration of the digital map into the avionics suite. The system architecture of the invention has high flexibility while maintaining speed and data throughput. The polygonal data base structure approach accommodate arbitrary scene complexity and a diversity of data base types.

The data structure of the invention is tagged so that any polygon may be rendered via any of the implemented schemes in a single frame. Thus, a particular image may have Gouraud shaded terrain, transparent threat domes, flat shaded cultural features, lines, and dots. In addition, since each polygon is tagged, a single icon can be comprised of differently shaded polygons. The invention embodies a 24 bit color system, although a production map would be scaled to 12 bits. A 12 bit system provides 4K colors and would require a 32K by 8 RGB RAM look-up table (LUT).

MISCELLANEOUS FEATURES

The display formats in one example of the invention are switchable at less than 600 milliseconds between paper chart, DLMS plan and perspective view. A large cache (1 megabit D-RAMs) is required for texture mapping. Other format displays warp chart data over DTED, or use DTED to pseudo-color the map. For example, change the color palate LUT for transparency. The GAP is used for creating a true orthographic projection of the chart data.

An edit mode for three dimensions is supported by the apparatus of the invention. A three dimensional object such as a "pathway in the sky" may be tagged for editing. This is accomplished by first, moving in two dimensions at a given AGL, secondly, updating the AGL in the three dimensional view, and finally, updating the data base.

The overlay memory from the DMC may be video mixed with the perspective view display memory.

Freeze frame capability is supported by the invention. In this mode, the aircraft position is updated using the cursor. If the aircraft flies off the screen, the display will snap back in at the appropriate place. This capability is implemented in plan view only. There is data frame software included to enable roaming through cache memory. This feature requires a two axis roam joystick or similar control. Resolution of the Z-buffer is 16 bits. This allows 64K meters down range.

The computer generated imagery has an update rate of 20 Hz. The major cycle is programmable and variable with no frame extend invoked. The system will run as fast as it can but will not switch ping-pong display memories until each functional unit issues a "pipeline empty" message to the display memory. The major cycle may also be locked to a fixed frame in multiples of 16.6 milliseconds. In the variable frame mode, the processor clock is used for a smooth frame interpolation for roam or zoom. The frame extend of the DMC is eliminated in perspective view mode. Plan view is implemented in the same pipeline as the perspective view. The GPP **105** loads the countdown register on the master timer to control the update rate.

The slowest update rate is 8.57 Hz. The image must be generated in this time or the memories will switch. This implies a pipeline speed of 40 million pixels per second. In a 512×512 image, it is estimated that there would be 4 million pixels rendered worst case with heavy hidden surface removal. In most cases, only million pixels need be rendered. FIG. **8** illustrates the analysis of pixel over-writes. The minimum requirement for surface normal resolution so that the best image is achieved is 16 bits. Tied to this is the way in which the normal is calculated. Averaging from surrounding tiles gives a smoother image on scale change or zoom. Using one tile is less complex, but results in poorer image

quality. Surface normal is calculated on the fly in accordance with known techniques.

DISPLAY MEMORY

This memory is a combination of scene and overlay with a Z-buffer. It is distributed or partitioned for optimal loading during write, and configured as a frame buffer during read-out. The master time speed required is approximately 50 MHz. The display memory resolution can be configured as $512\times512\times12$ or as $1024\times1024\times12$. The Z-buffer is 16 bits deep and $1K\times1K$ resolution. At the start of each major cycle, the Z-values are set to plus infinity (FF Hex). Infinity (Zmax) is programmable. The back clipping plane is set by the DSM over the control bus.

At the start of each major cycle, the display memory is set to a background color. In certain modes such as mesh or dot, this color will change. A background color register is loaded by the DSM over the configuration bus and used to fill in the memory.

VIDEO GENERATOR/MASTER TIMER

The video generator **46** performs the digital to analog conversion of the image data in the display memory to send to the display head. It combines the data stream from the overlay memory of the DMC with the display memory from the perspective view. The configuration bus loads the color map.

A 30 Hz interlaced refresh rate may be implemented in a system employing the present invention. Color pallets are loadable by the GPP. The invention assumes a linear color space in RGB. All colors at zero intensity go to black.

THREE DIMENSIONAL SYMBOL GENERATOR

The three-dimensional symbol generator **38** performs the following tasks:

1. It places the model to world transformation coefficients in the GAP.

2. It operates in cooperation with the geometry engine to multiply the world to screen transformation matrix by the model to world transformation matrix to form a model to screen transformation matrix. This matrix is stored over the model to world transformation matrix.

3. It operates in cooperation with the model to screen transformation matrix to each point of the symbol from the vertex list to transform the generic icon to the particular symbol.

4. It processes the connectivity list in the tiling engine and forms the screen polygons and passes them to the rendering engine.

One example of a three-dimensional symbol generator is described in detail in the assignee's aforereferenced patent application entitled "Three Dimensional Computer Graphic Symbol Generator".

The symbol generator data base consists of vertex list library and 64K bytes of overlay RAM and a connectivity list. Up to 18K bytes of DFAD (i.e., 2K bytes display list from cache shadow RAM $\times$ 9 buffer segments) are loaded into the overlay RAM for cultural feature processing. The rest of the memory holds the threat/intelligence file and the mission planning file for the entire gaming area. The overlay RAM is loaded over the control bus from the DSM processor with the threat and mission planning files. The SHAG loads the DFAD files. The symbol libraries are updated via the configuration bus.

The vertex list contains the relative vertex positions of the generic library icons. In addition, it contains a 16 bit surface normal, a one bit end of polygon flag, and a one bit end of symbol flag. The table is $32K\times16$ bits. A maximum of 512 vertices may be associated with any given icon. The connectivity list contains the connectivity information of the vertices of the symbol. A 64K by 12 bit table holds this information.

A pathway in the sky format may be implemented in this system. It consists of either a wire frame tunnel or an elevated roadbed for flight path purposes. The wire frame tunnel is a series of connected transparent rectangles generated by the tiling engine of which only the edges are visible (wire mesh). Alternatively, the polygons may be precomputed in world coordinates and stored in a mission planning file. The roadbed is similarly comprised of polygons generated by the tiler along a designated pathway. In either case, the geometry engine must transform these polygons from object space (world coordinate system) to screen space. The transformed vertices are then passed to the rendering engine. The parameters (height, width, frequency) of the tunnel and roadbed polygons are programmable.

Another symbol used in the system is a waypoint flag. Waypoint flags are markers consisting of a transparent or opaque triangle on a vertical staff rendered in perspective. The waypoint flag icon is generated by the symbol generator as a macro from a mission planning file. Alternatively, they may be precomputed as polygons and stored. The geometry engine receives the vertices from the symbol generator and performs the perspective transformation on them. The geometry engine passes the rendering engine the polygons of the flag staff and the scaled font call of the alphanumeric symbol. Plan view format consists of a circle with a number inside and is not passed through the geometry engine.

DFAD data processing consists of a generalized polygon renderer which maps 32K points possible down to 256 polygons or less for a given buffer segment. These polygons are then passed to the rendering engine. This approach may redundantly render terrain and DFAD for the same pixels but easily accommodates declutter of individual features. Another approach is to rasterize the DFAD and use a texture warp function to color the terrain. This would not permit declutter of individual features but only classes (by color). Terrain color show-through in sparse overlay areas would be handled by a transparent color code (screen door effect). No verticality is achieved.

There are 298 categories of aerial, linear, and point features. Linear features must be expanded to a double line to prevent interlace strobing. A point feature contains a length, width, and height which can be used by the symbol generator for expansion. A typical lake contains 900 vertices and produces 10 to 20 active edges for rendering at any given scan line. The number of vertices is limited to 512. The display list is 64K bytes for a 1:250K buffer segment. Any given feature could have 32K vertices.

Up to 2K bytes of display list per buffer segment DTED is accommodated for DFAD. The DSM can tag the classes or individual features for clutter/declutter by toggling bits in the overlay RAM of the SHAG.

The symbol generator processes macros and graphic primitives which are passed to the rendering engine. These primitives include lines, arcs, alphanumerics, and two dimensional symbology. The rendering engine

draws these primitives and outputs pixels which are anti-aliased. The GAP transforms these polygons and passes them to the rendering engine. A complete 4×4 Euler transformation is performed. Typical macros include compass rose and range scale symbols. Given a macro command, the symbol generator produces the primitive graphics calls to the rendering engine. This mode operates in plan view only and implements two dimensional symbols. Those skilled in the art will appreciate that the invention is not limited to specific fonts.

Three dimensional symbology presents the problem of clipping to the view volume. A gross clip is handled by the DSM in the cache memory at scan out time. The base of a threat dome, for example, may lie outside the orthographic projection of the view volume onto cache, yet a part of its dome may end up visible on the screen. The classical implementation performs the functions of tiling, transforming, clipping to the view volume (which generates new polygons), and then rendering. A gross clip boundary is implemented in cache around the view volume projection to guarantee inclusion of the entire symbol. The anomaly under animation to be avoided is that of having symbology sporadically appear and disappear in and out of the frame at the frame boundaries. A fine clip to the screen is performed downstream by the rendering engine. There is a 4K boundary around the screen which is rendered. Outside of this boundary, the symbol will not be rendered. This causes extra rendering which is clipped away.

Threat domes are represented graphically in one embodiment by an inverted conic volume. A threat/intelligence file contains the location and scaling factors for the generic model to be transformed to the specific threats. The tiling engine contains the connectivity information between the vertices and generates the planar polygons. The threat polygons are passed to the rendering engine with various viewing parameters such as mesh, opaque, dot, transparent, and so forth.

Graticles represent latitude and longitude lines, UTM klicks, and so forth which are warped onto the map in perspective. The symbol generator produces these lines.

Freeze frame is implemented in plan view only. The cursor is flown around the screen, and is generated by the symbol generator.

Programmable blink capability is accommodated in the invention. The DSM updates the overlay RAM toggle for display. The processor clock is used during variable frame update rate to control the blink rate.

A generic threat symbol is modeled and stored in the three dimensional symbol generation library. Parameters such as position, threat range, and angular threat view are passed to the symbol generator as a macro call (similar to a compass rose). The symbol generator creates a polygon list for each threat instance by using the parameters to modify the generic model and place it in the world coordinate system of the terrain data base. The polygons are transformed and rendered into screen space by the perspective view pipeline. These polygons form only the outside envelope of the threat cone.

This invention has been described herein in considerable detail in order to comply with the Patent Statues and to provide those skilled in the art with the information needed to apply the novel principles and to construct and use such specialized components as are required. However, it is to be understood that the invention can be carried out by specifically different equipment and devices, and that various modifications, both as to the equipment details and operating procedures, can be accomplished without departing from the scope of the invention itself.

What is claimed is:

1. A system for providing a texture mapped perspective view for a digital map system wherein objects are transformed from texture space having U, V coordinates to screen space having X, Y coordinates comprising:

(a) a cache memory means for storing terrain data including elevation posts, wherein the cache memory means includes an output and an address bus;

(b) a shape address generator means for scanning cache memory having an ADDRESS SIGNAL coupled to the cache memory means address bus wherein the shape address generator means scans the elevation posts out of the cache memory means;

(c) a geometry engine coupled to the cache memory means output to receive the elevation posts scanned from the cache memory by the shape address generator means, the geometry engine including means for

i. transformation of the scanned elevation posts from object space to screen space so as to generate transformed vertices in screen coordinates for each elevation post, and

ii. generating three dimensional coordinates;

(d) a tilling engine coupled to the geometry engine for generating planar polygons from the generated three dimensional coordinates;

(e) a symbol generator to the geometry engine for transmitting a vertex list to the geometry engine wherein the geometry engine operates on the vertex list to transform the vertex list into screen space X, Y coordinates and passes the screen space X, Y coordinates to the tilling engine for generating planar polygons which form icons for display and processing information from the tilling engine into symbols,

(f) a texture engine means coupled to receive the ADDRESS SIGNAL from the shape address generator means including a texture memory and including a means for generating a texture vertex address to texture space correlated to an elevation post address and further including a means for generating a texture memory address for scanning the texture memory wherein the texture memory provides texture data on a texture memory data bus in response to being scanned by the texture memory address;

(g) a rendering engine having an input coupled to the tilling engine and the texture memory data bus for generating image data from the planar polygons; and

(h) a display memory for receiving image data from the rendering engine output wherein the display memory includes at least four first-in, first-out memory buffers.

2. The apparatus of claim 1 wherein each polygon has a surface and the rendering means assigns one color across the surface of each polygon.

3. The apparatus of claim 1 wherein the vertices of each polygon have an intensity and the rendering means interpolates the intensities between the vertices of each polygon in a linear fashion.

4. The apparatus of claim 1 wherein the rendering means further includes means for generating transparent polygons and passing the transparent polygon to the display memory.

5. A method for providing a texture mapped perspective view for a digital map system having a cache memory, a geometry engine coupled to the cache memory, a shape address generator coupled to the cache memory, a tiling engine coupled to the geometry engine, a symbol generator coupled to the geometry engine and the tiling engine, a texture engine coupled to the cache memory, a rendering engine coupled to the tiling engine and the texture engine, and a display memory coupled to the rendering engine, wherein objects are transformed from texture space having U, V coordinates to screen space having X, Y coordinates, the method comprising the steps of:

(a) storing terrain data, including elevation posts, in the cache memory;

(b) scanning the cache memory to retrieve the elevation posts;

(c) transforming the terrain data from elevation posts in object space to transformed vertices in screen space, and

(d) generating planar polygons from the generated three dimensional coordinates;

(e) transmitting a vertex list to the geometry engine, operating the geometry engine to transform the vertex list into screen space X, Y coordinates and passing the screen space X, Y coordinates to the tiling engine for generating planar polygons which form icons for display;

(f) tagging elevation posts with corresponding addresses in texture space;

(g) generating image data in the rendering engine from the planar polygons and the tagged elevation posts; and

(h) storing the generated image data in the display memory wherein the display memory comprises at least four first-in, first-out memory buffers and the step of storing the generated images includes storing the generated image data in the at least four First-in, First-out memory buffers.

6. The method of claim 5 wherein each polygon has a surface and wherein the step of generating image data further includes the steps of assigning one color across the surface of each polygon.

7. The method of claim 5 wherein the vertices of each polygon have an intensity and the step of generating image data further includes the step of interpolating the intensities between the vertices of each polygon in a linear fashion.

8. The method of claim 5 wherein the step of generating image data further includes the step of generating transparent polygons and passing the transparent polygons to the display memory.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,179,638
DATED : January 12, 1993
INVENTOR(S) : John F. Dawson, Thomas D. Snodgrass, and James A. Cousens

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Column 17, line 21, after "and" insert --generating three dimensional coordinates for the transformed vertices in screen space--.

Signed and Sealed this

Twenty-second Day of March, 1994

Attest:

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

US005798770A

United States Patent [19]

Baldwin

[11] Patent Number: 5,798,770

[45] Date of Patent: Aug. 25, 1998

[54] **GRAPHICS RENDERING SYSTEM WITH RECONFIGURABLE PIPELINE SEQUENCE**

[75] Inventor: **David Robert Baldwin**, Weybridge, United Kingdom

[73] Assignee: **3DLabs Inc. Ltd.**, Hamilton, Bermuda

[21] Appl. No.: **640,620**

[22] Filed: **May 1, 1996**

Related U.S. Application Data

[60] Provisional application No. 60/008,803 Dec. 18, 1995.

[63] Continuation-in-part of Ser. No. 410,345, Mar. 24, 1995.

[51] **Int. Cl.**[6] **G06T 1/20**

[52] **U.S. Cl.** **345/506**; 345/519; 345/509

[58] **Field of Search** 395/506, 502, 395/507, 509, 519, 122, 130, 132, 125, 503; 345/506, 507, 502, 509, 519, 422, 430–432, 425, 503

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,866,637	9/1989	Gonzalez-Lopez	395/506
5,392,391	2/1995	Caulk, Jr. et al.	395/503

OTHER PUBLICATIONS

Foley *et al.*, "Computer Graphics, Principles and Practice", 2 ed in C,1996, Chapter 18, pp. 855–920.

Kogge, P.M., "The Microprogramming of Pipelined Processors", 1977, Proc. 4th Ann. Conf Parallel Procesing, IEEE, March, pp. 63–69.

Computer Graphics, vol. 22, No. 4, "A display system for the Stellar graphics Supercomputer Model GS1000", Brian Apgar et al., Aug. 1988.

Primary Examiner—Kee M. Tung

Attorney, Agent, or Firm—Robert Groover; Betty Formby; Matthew S. Anderson

[57] **ABSTRACT**

The preferred embodiment discloses a pipelined graphics processor in which the sequence can be dynamically reconfigured (e.g. between primitives) in a rendering sequence. The pipeline sequence can be configured for compliance with specifications such as OpenGL, but may also be optimized by reconfiguring the pipeline sequence to eliminate unnecessary processing. In a preferred embodiment, pixel elimination sequences such as depth and stencil tests are performed before texturing calculations are performed, so that unneeded pixel data is discarded before said texturing calculations are performed.

26 Claims, 12 Drawing Sheets

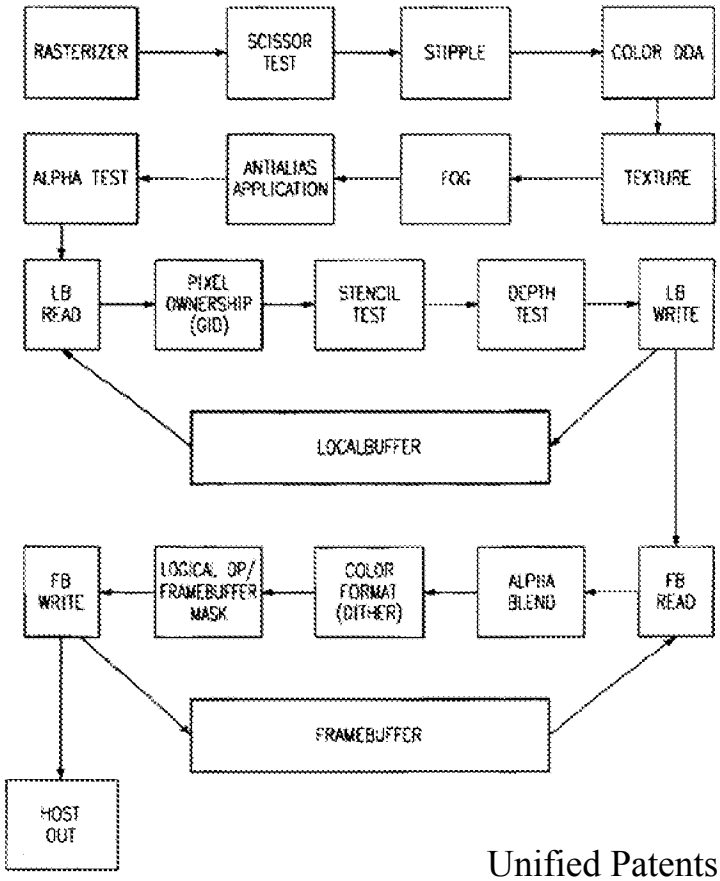


FIG. 1A

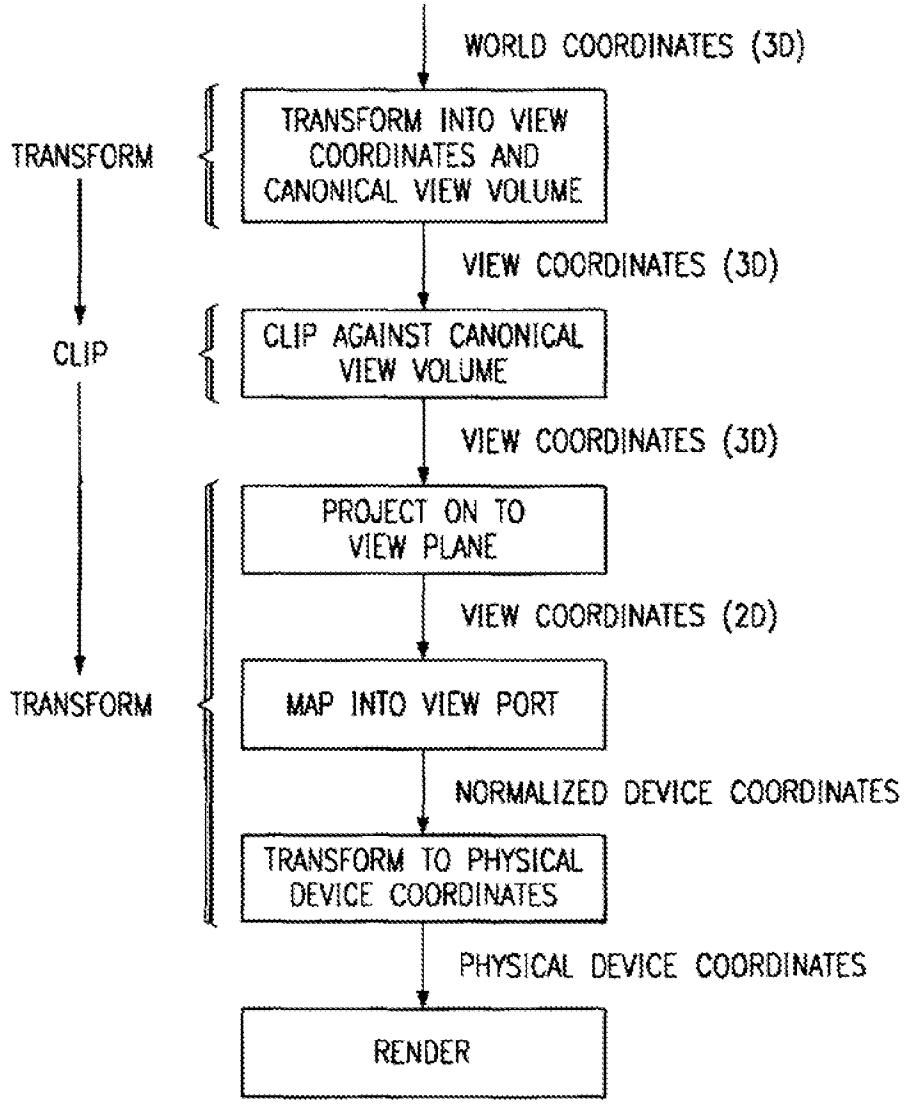
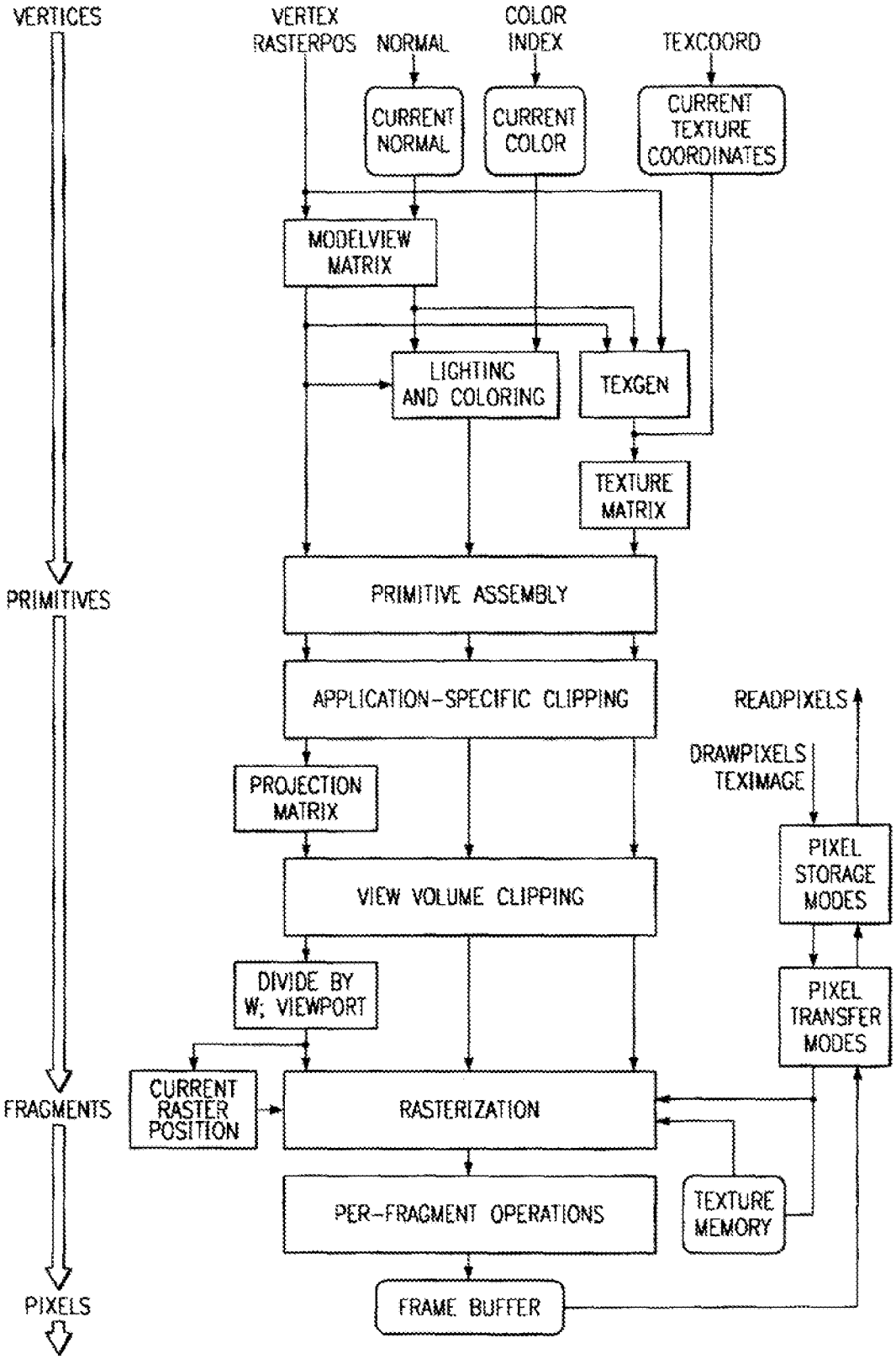

WORLD COORDINATES (3D)
TRANSFORM
TRANSFORM INTO VIEW COORDINATES AND CANONICAL VIEW VOLUME
VIEW COORDINATES (3D)
CLIP
CLIP AGAINST CANONICAL VIEW VOLUME
VIEW COORDINATES (3D)
PROJECT ON TO VIEW PLANE
VIEW COORDINATES (2D)
TRANSFORM
MAP INTO VIEW PORT
NORMALIZED DEVICE COORDINATES
TRANSFORM TO PHYSICAL DEVICE COORDINATES
PHYSICAL DEVICE COORDINATES
RENDER

FIG. 1B


VERTICES
VERTEX
RASTERPOS
NORMAL
COLOR
INDEX
TEXCOORD
CURRENT NORMAL
CURRENT COLOR
CURRENT TEXTURE COORDINATES
MODELVIEW MATRIX
LIGHTING AND COLORING
TEXGEN
TEXTURE MATRIX
PRIMITIVES
PRIMITIVE ASSEMBLY
APPLICATION-SPECIFIC CLIPPING
READPIXELS
DRAWPIXELS
TEXIMAGE
PROJECTION MATRIX
PIXEL STORAGE MODES
VIEW VOLUME CLIPPING
DIVIDE BY W; VIEWPORT
PIXEL TRANSFER MODES
CURRENT RASTER POSITION
FRAGMENTS
RASTERIZATION
PER-FRAGMENT OPERATIONS
TEXTURE MEMORY
PIXELS
FRAME BUFFER

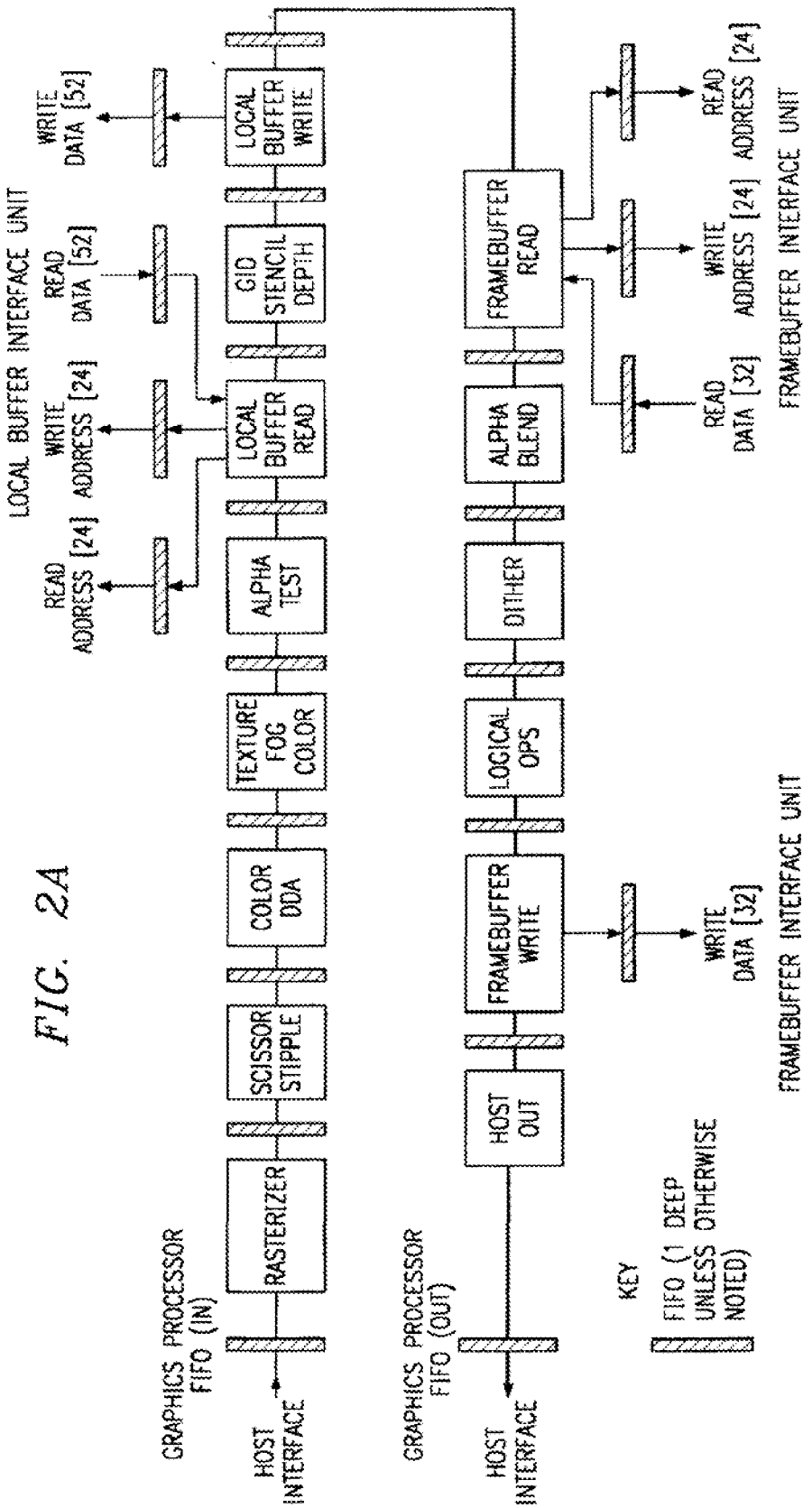

HOST INTERFACE
GRAPHICS PROCESSOR FIFO (IN)
RASTERIZER
SCISSOR STIPPLE
COLOR DDA
TEXTURE FOG COLOR
ALPHA TEST
LOCAL BUFFER READ
GID STENCIL DEPTH
LOCAL BUFFER WRITE
LOCAL BUFFER INTERFACE UNIT
READ ADDRESS [24]
WRITE ADDRESS [24]
READ DATA [52]
WRITE DATA [52]
HOST INTERFACE
GRAPHICS PROCESSOR FIFO (OUT)
HOST OUT
FRAMEBUFFER WRITE
LOGICAL OPS
DITHER
ALPHA BLEND
FRAMEBUFFER READ
FRAMEBUFFER INTERFACE UNIT
WRITE DATA [32]
READ DATA [32]
WRITE ADDRESS [24]
READ ADDRESS [24]
KEY
FIFO (1 DEEP UNLESS OTHERWISE NOTED)

FIG. 2A

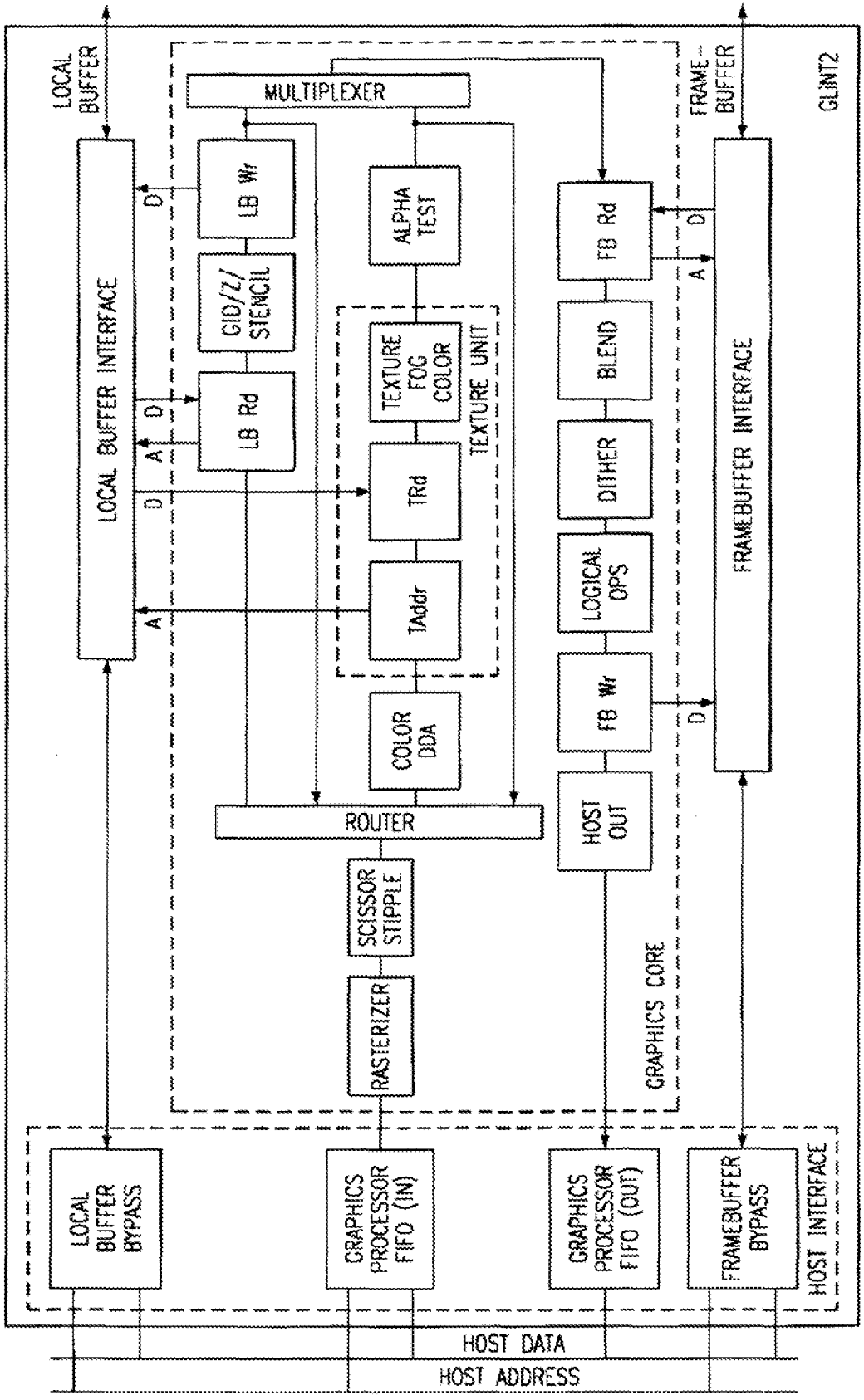
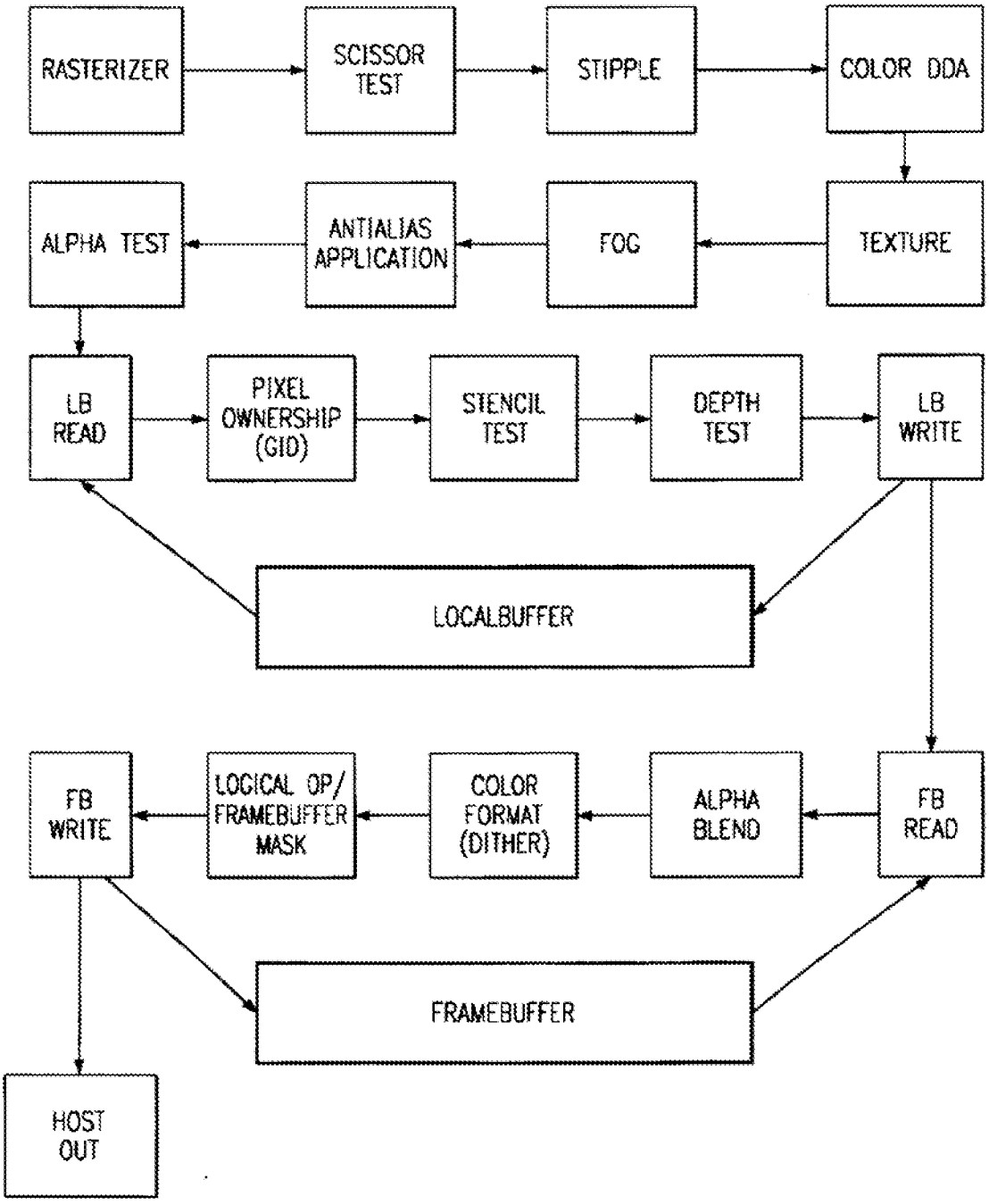

LOCAL BUFFER
MULTIPLEXER
FRAME-BUFFER
GLINT2
LOCAL BUFFER INTERFACE
LB Wr
CID/Z/STENCIL
LB Rd
ALPHA TEST
TEXTURE FOG COLOR
TRd
TAddr
COLOR DDA
TEXTURE UNIT
FB Rd
BLEND
DITHER
LOGICAL OPS
FB Wr
HOST OUT
FRAMEBUFFER INTERFACE
ROUTER
SCISSOR STIPPLE
RASTERIZER
GRAPHICS CORE
LOCAL BUFFER BYPASS
GRAPHICS PROCESSOR FIFO (IN)
GRAPHICS PROCESSOR FIFO (OUT)
FRAMEBUFFER BYPASS
HOST INTERFACE
HOST DATA
HOST ADDRESS
D
A

FIG. 2B

FIG. 2C


RASTERIZER
SCISSOR TEST
STIPPLE
COLOR DDA
ALPHA TEST
ANTIALIAS APPLICATION
FOG
TEXTURE
LB READ
PIXEL OWNERSHIP (GID)
STENCIL TEST
DEPTH TEST
LB WRITE
LOCALBUFFER
FB WRITE
LOGICAL OP/ FRAMEBUFFER MASK
COLOR FORMAT (DITHER)
ALPHA BLEND
FB READ
FRAMEBUFFER
HOST OUT

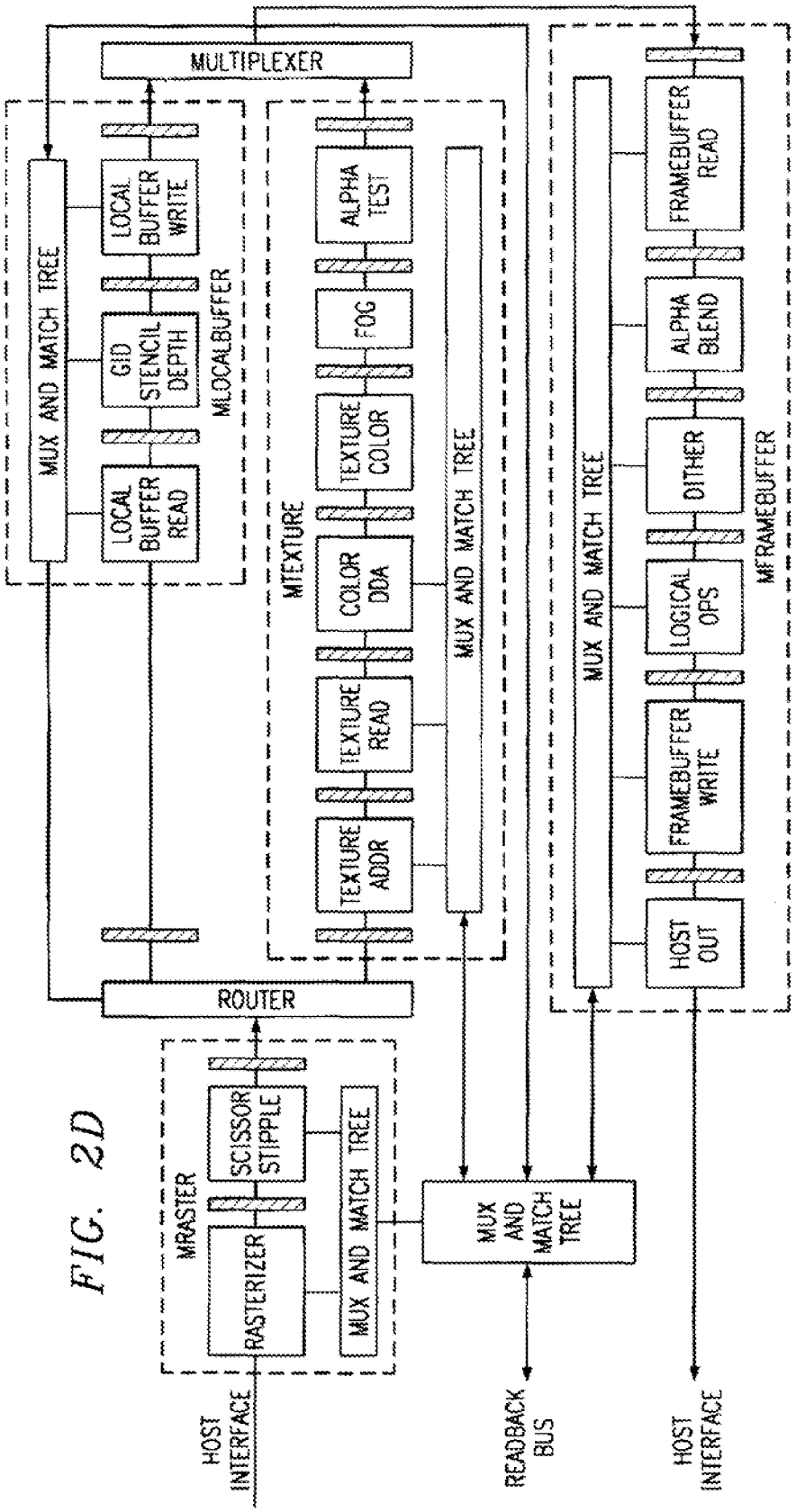

MULTIPLEXER
MUX AND MATCH TREE
LOCAL BUFFER WRITE
GID STENCIL DEPTH
LOCAL BUFFER READ
MLOCALBUFFER
MTEXTURE
TEXTURE ADDR
TEXTURE READ
COLOR DDA
TEXTURE COLOR
FOG
ALPHA TEST
MUX AND MATCH TREE
ROUTER
MUX AND MATCH TREE
HOST OUT
FRAMEBUFFER WRITE
LOGICAL OPS
DITHER
ALPHA BLEND
FRAMEBUFFER READ
MFRAMEBUFFER
MRASTER
SCISSOR STIPPLE
RASTERIZER
MUX AND MATCH TREE
MUX AND MATCH TREE
HOST INTERFACE
READBACK BUS
HOST INTERFACE

FIG. 2D

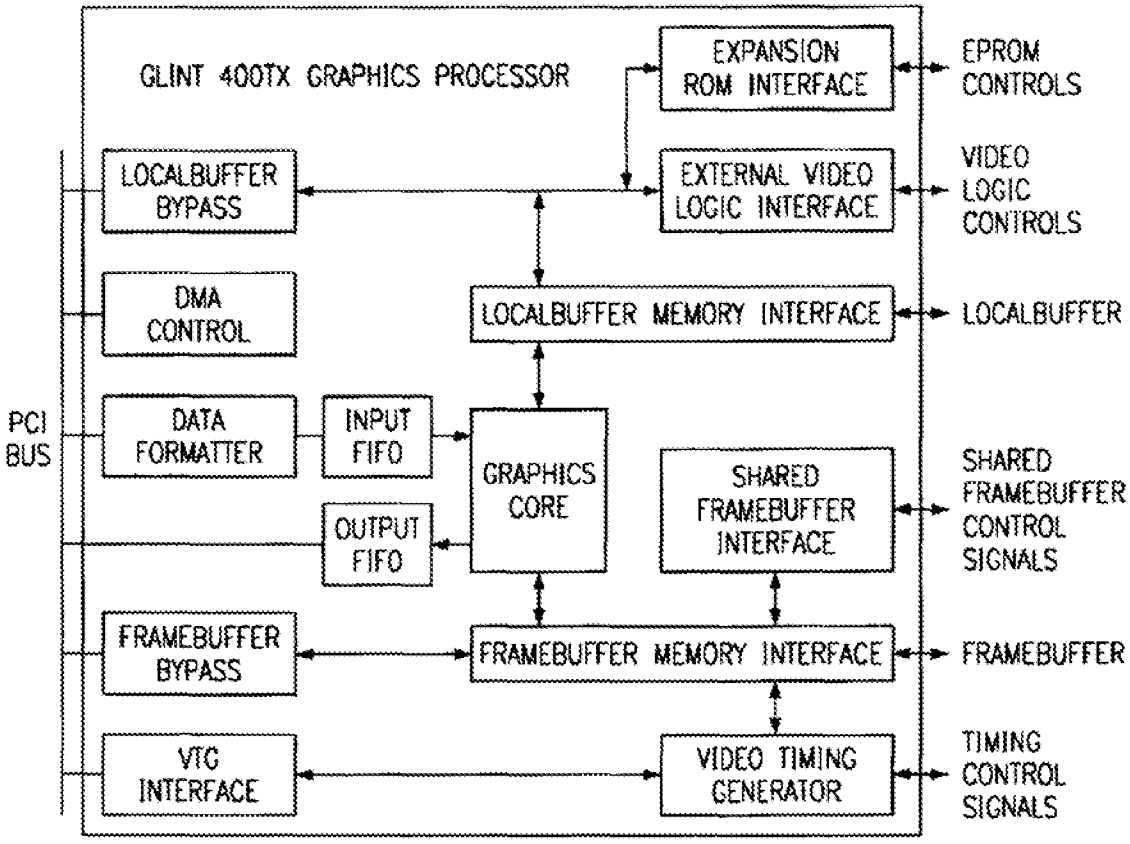

GLINT 400TX GRAPHICS PROCESSOR
EXPANSION ROM INTERFACE
EPROM CONTROLS
LOCALBUFFER BYPASS
EXTERNAL VIDEO LOGIC INTERFACE
VIDEO LOGIC CONTROLS
DMA CONTROL
LOCALBUFFER MEMORY INTERFACE
LOCALBUFFER
PCI BUS
DATA FORMATTER
INPUT FIFO
GRAPHICS CORE
SHARED FRAMEBUFFER INTERFACE
SHARED FRAMEBUFFER CONTROL SIGNALS
OUTPUT FIFO
FRAMEBUFFER BYPASS
FRAMEBUFFER MEMORY INTERFACE
FRAMEBUFFER
VTC INTERFACE
VIDEO TIMING GENERATOR
TIMING CONTROL SIGNALS

FIG. 2E

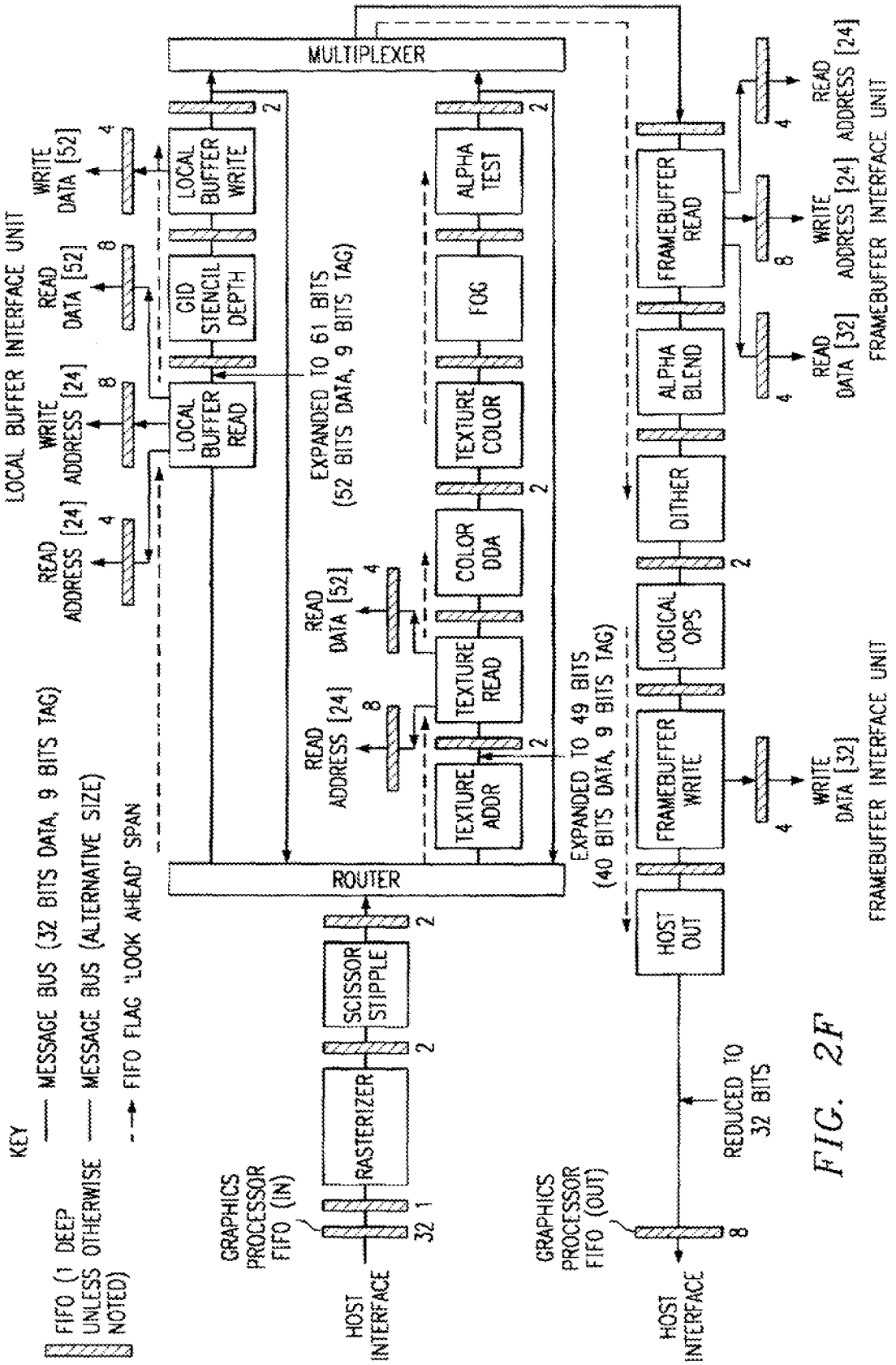

KEY
FIFO (1 DEEP UNLESS OTHERWISE NOTED)
MESSAGE BUS (32 BITS DATA, 9 BITS TAG)
MESSAGE BUS (ALTERNATIVE SIZE)
FIFO FLAG 'LOOK AHEAD' SPAN
LOCAL BUFFER INTERFACE UNIT
MULTIPLEXER
ROUTER
EXPANDED TO 61 BITS (52 BITS DATA, 9 BITS TAG)
EXPANDED TO 49 BITS (40 BITS DATA, 9 BITS TAG)
READ ADDRESS [24]
WRITE ADDRESS [24]
READ DATA [52]
WRITE DATA [52]
LOCAL BUFFER READ
LOCAL BUFFER WRITE
GID STENCIL DEPTH
TEXTURE ADDR
TEXTURE READ
COLOR DDA
TEXTURE COLOR
FOG
ALPHA TEST
SCISSOR STIPPLE
RASTERIZER
GRAPHICS PROCESSOR FIFO (IN)
HOST INTERFACE
LOGICAL OPS
DITHER
ALPHA BLEND
FRAMEBUFFER WRITE
FRAMEBUFFER READ
HOST OUT
REDUCED TO 32 BITS
GRAPHICS PROCESSOR FIFO (OUT)
READ DATA [32]
WRITE DATA [32]
READ ADDRESS [24]
WRITE ADDRESS [24]
FRAMEBUFFER INTERFACE UNIT

FIG. 2F

FIG. 3A

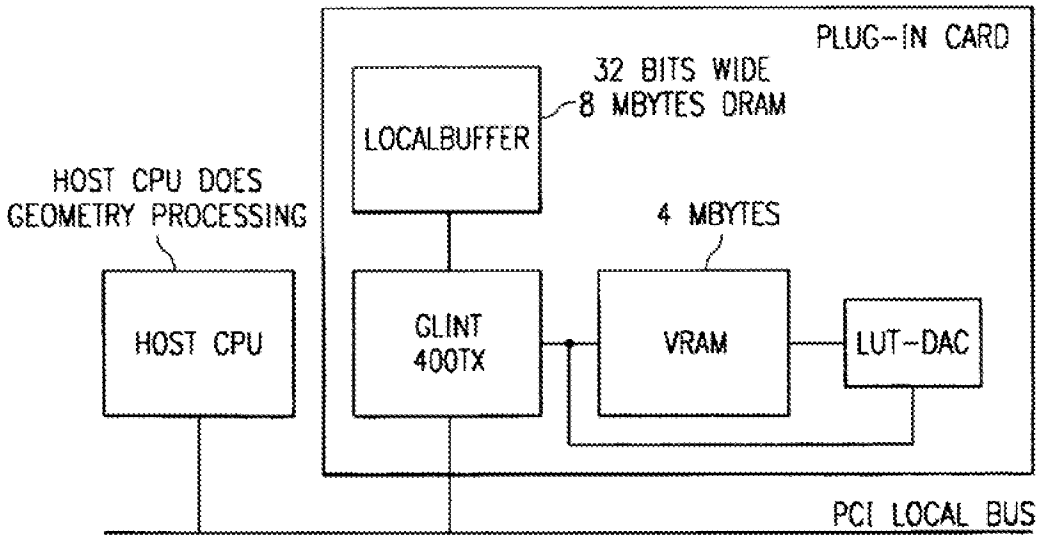

PLUG-IN CARD
32 BITS WIDE
8 MBYTES DRAM
LOCALBUFFER
HOST CPU DOES
GEOMETRY PROCESSING
4 MBYTES
HOST CPU
GLINT
400TX
VRAM
LUT-DAC
PCI LOCAL BUS

FIG. 3B

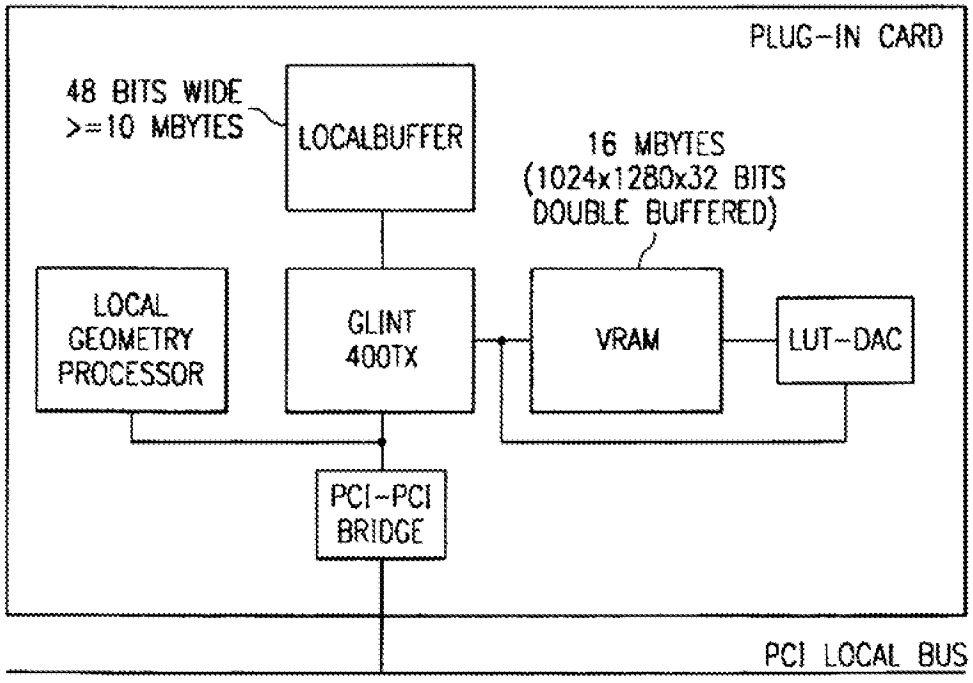

PLUG-IN CARD
48 BITS WIDE
>=10 MBYTES
LOCALBUFFER
16 MBYTES
(1024x1280x32 BITS
DOUBLE BUFFERED)
LOCAL
GEOMETRY
PROCESSOR
GLINT
400TX
VRAM
LUT-DAC
PCI-PCI
BRIDGE
PCI LOCAL BUS

FIG. 3C

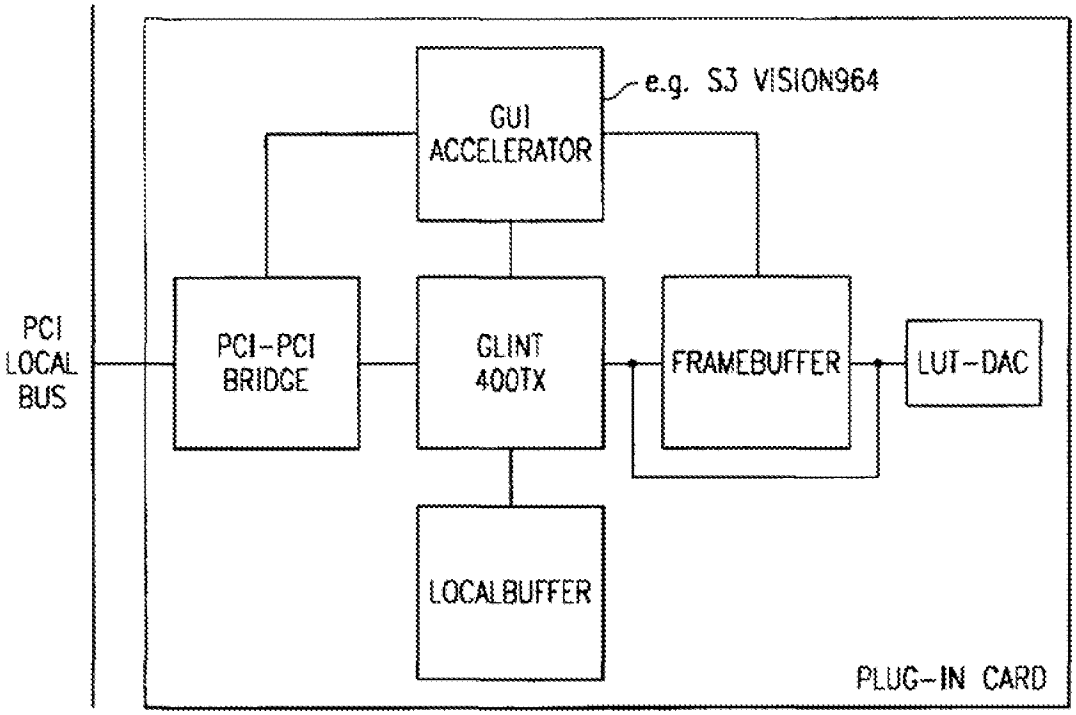

GUI
ACCELERATOR
e.g. S3 VISION964
PCI
LOCAL
BUS
PCI-PCI
BRIDGE
GLINT
400TX
FRAMEBUFFER
LUT-DAC
LOCALBUFFER
PLUG-IN CARD

FIG. 3D

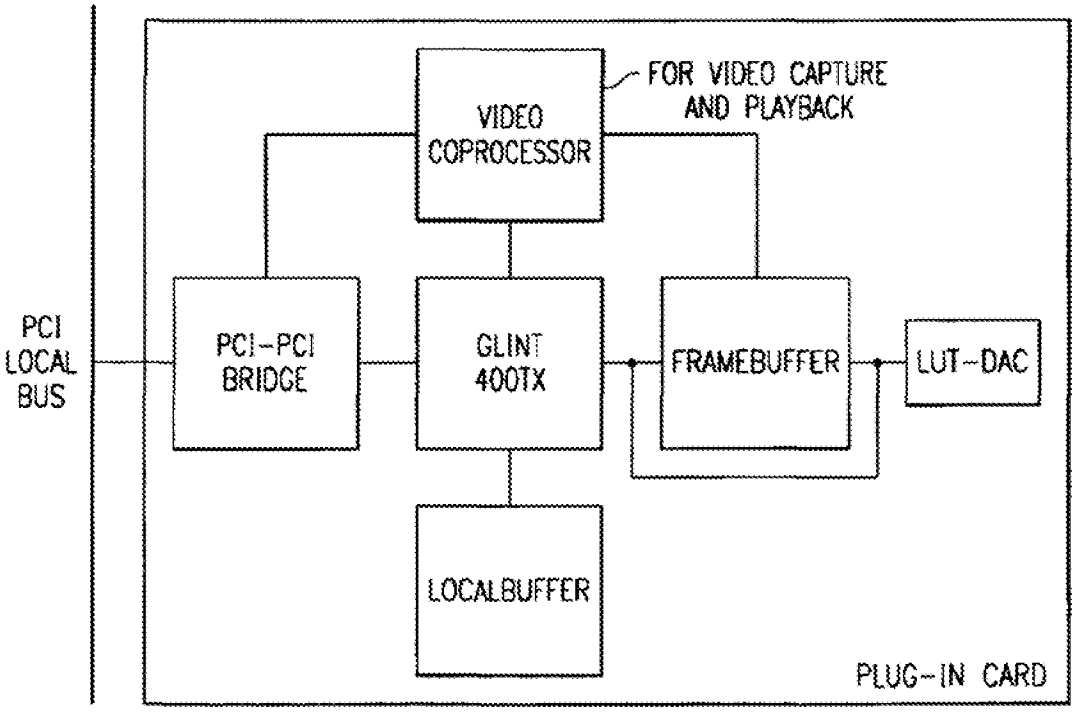

VIDEO
COPROCESSOR
FOR VIDEO CAPTURE
AND PLAYBACK
PCI
LOCAL
BUS
PCI-PCI
BRIDGE
GLINT
400TX
FRAMEBUFFER
LUT-DAC
LOCALBUFFER
PLUG-IN CARD

FIG. 4A

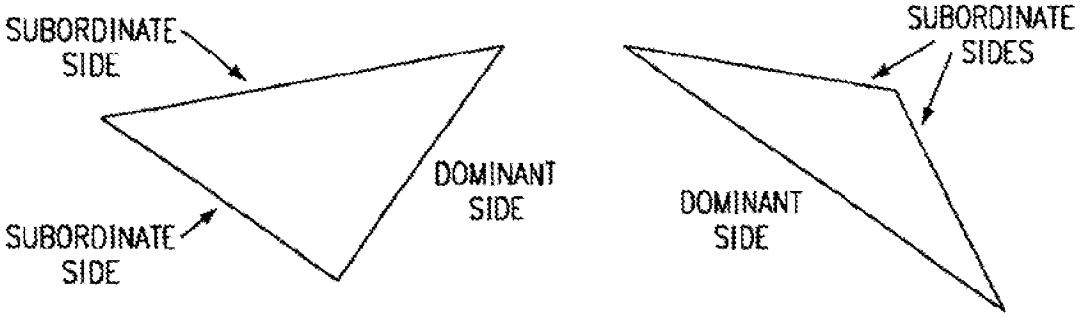
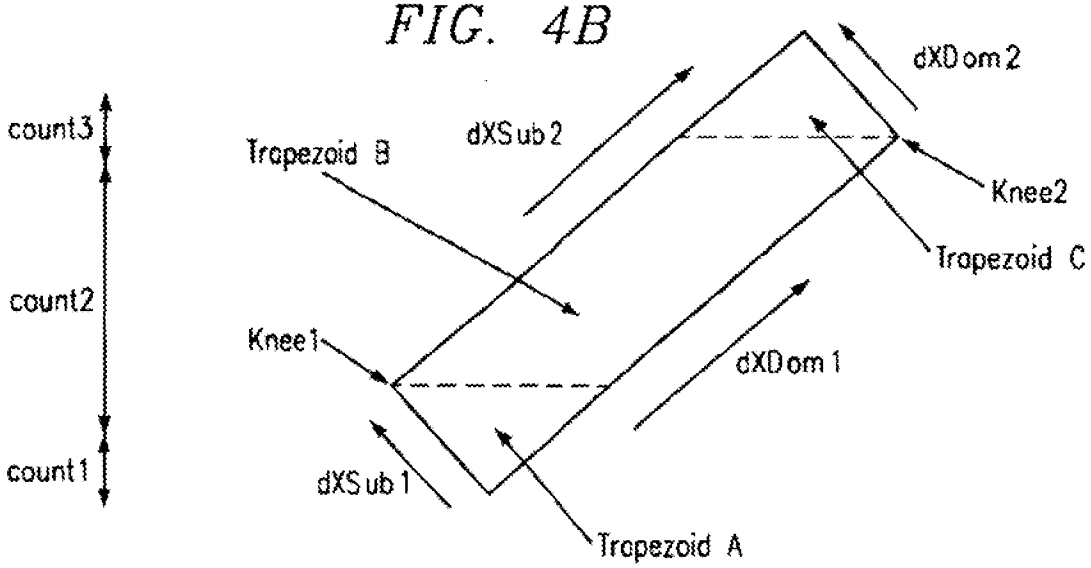

SUBORDINATE SIDE
DOMINANT SIDE
SUBORDINATE SIDE
SUBORDINATE SIDES
DOMINANT SIDE

FIG. 4B


count3
count2
count1
Trapezoid B
dXSub2
dXDom2
Knee2
Trapezoid C
Knee1
dXDom1
dXSub1
Trapezoid A

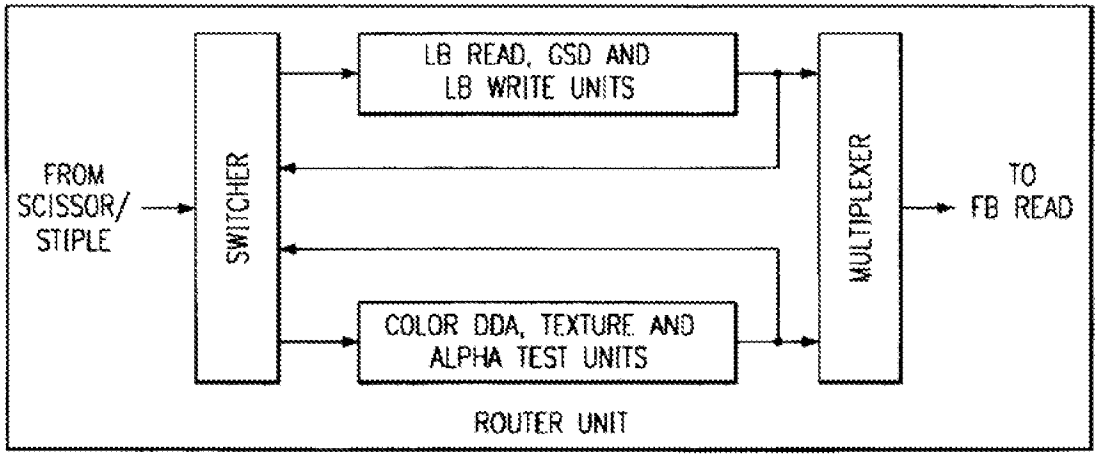


FIG. 5A

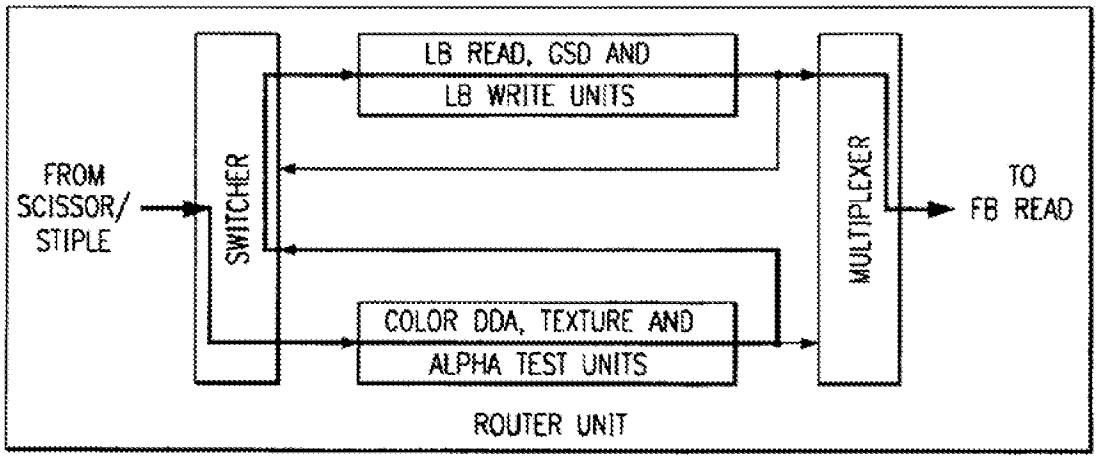


FIG. 5B

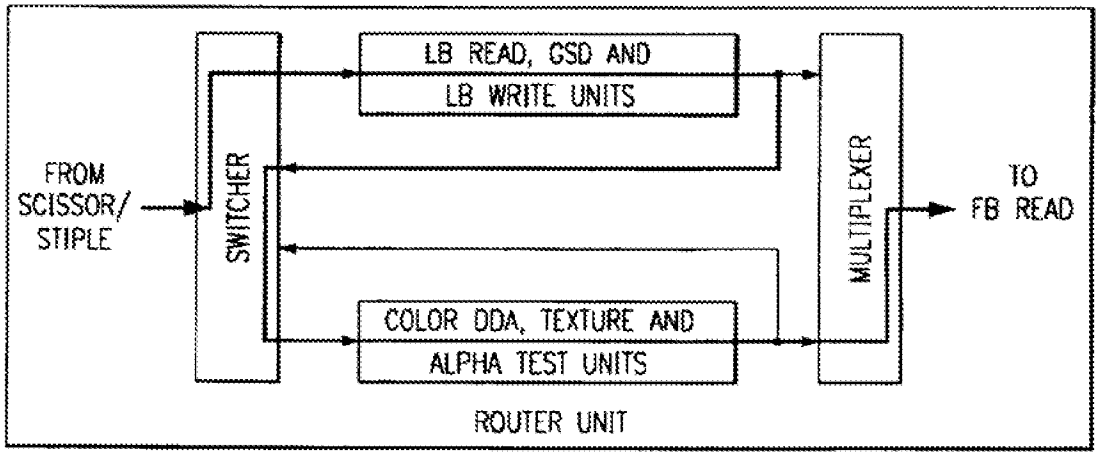


FIG. 5C

GRAPHICS RENDERING SYSTEM WITH RECONFIGURABLE PIPELINE SEQUENCE

CROSS-REFERENCE TO RELATED APPLICATION

This application is a continuation-in-part of 08/410,345 filed Mar. 24, 1995, and claims priority from provisional 60/008,803 filed Dec. 18, 1995, which is hereby incorporated by reference.

BACKGROUND AND SUMMARY OF THE INVENTION

The present application relates to computer graphics and animation systems, and particularly to 3D graphics rendering hardware. Background of the art and the prior embodiment, according to the parent application, is described below. Some of the distinctions of the presently preferred embodiment are particularly noted beginning on page 8.

COMPUTER GRAPHICS AND RENDERING

Modern computer systems normally manipulate graphical objects as high-level entities. For example, a solid body may be described as a collection of triangles with specified vertices, or a straight line segment may be described by listing its two endpoints with three-dimensional or two-dimensional coordinates. Such high-level descriptions are a necessary basis for high-level geometric manipulations, and also have the advantage of providing a compact format which does not consume memory space unnecessarily.

Such higher-level representations are very convenient for performing the many required computations. For example, ray-tracing or other lighting calculations may be performed, and a projective transformation can be used to reduce a three-dimensional scene to its two-dimensional appearance from a given viewpoint. However, when an image containing graphical objects is to be displayed, a very low-level description is needed. For example, in a conventional CRT display, a "flying spot" is moved across the screen (one line at a time), and the beam from each of three electron guns is switched to a desired level of intensity as the flying spot passes each pixel location. Thus at some point the image model must be translated into a data set which can be used by a conventional display. This operation is known as "rendering."

The graphics-processing system typically interfaces to the display controller through a "frame store" or "frame buffer" of special two-port memory, which can be written to randomly by the graphics processing system, but also provides the synchronous data output needed by the video output driver. (Digital-to-analog conversion is also provided after the frame buffer.) Such a frame buffer is usually implemented using VRAM memory chips (or sometimes with DRAM and special DRAM controllers). This interface relieves the graphics processing system of most of the burden of synchronization for video output. Nevertheless, the amounts of data which must be moved around are very sizable, and the computational and data-transfer burden of placing the correct data into the frame buffer can still be very large.

Even if the computational operations required are quite simple, they must be performed repeatedly on a large number of data points. For example, in a typical 1995 high-end configuration, a display of 1280×1024 elements is refreshed at 72 Hz, with a color resolution

of 24 bits per pixel. If blending is desired, additional bits (e.g. another 8 bits per pixel) will be required to store an "alpha" or transparency value for each pixel. This implies manipulation of more than 3 billion bits per second, without allowing for any of the actual computations being performed. Thus it may be seen that this is an environment with unique data manipulation requirements.

If the display is unchanging, no demand is placed on the rendering operations. However, some common operations (such as zooming or rotation) will require every object in the image space to be re-rendered. Slow rendering will make the rotation or zoom appear jerky. This is highly undesirable. Thus efficient rendering is an essential step in translating an image representation into the correct pixel values. This is particularly true in animation applications, where newly rendered updates to a computer graphics display must be generated at regular intervals.

The rendering requirements of three-dimensional graphics are particularly heavy. One reason for this is that, even after the three-dimensional model has been translated to a two-dimensional model, some computational tasks may be bequeathed to the rendering process. (For example, color values will need to be interpolated across a triangle or other primitive.) These computational tasks tend to burden the rendering process. Another reason is that since three-dimensional graphics are much more lifelike, users are more likely to demand a fully rendered image. (By contrast, in the two-dimensional images created e.g. by a GUI or simple game, users will learn not to expect all areas of the scene to be active or filled with information.)

FIG. 1A is a very high-level view of other processes performed in a 3D graphics computer system. A three dimensional image which is defined in some fixed 3D coordinate system (a "world" coordinate system) is transformed into a viewing volume (determined by a view position and direction), and the parts of the image which fall outside the viewing volume are discarded. The visible portion of the image volume is then projected onto a viewing plane, in accordance with the familiar rules of perspective. This produces a two-dimensional image, which is now mapped into device coordinates. It is important to understand that all of these operations occur prior to the operations performed by the rendering subsystem of the present invention. FIG. 1B is an expanded version of FIG. 1A, and shows the flow of operations defined by the OpenGL standard.

A vast amount of engineering effort has been invested in computer graphics systems, and this area is one of increasing activity and demands. Numerous books have discussed the requirements of this area; see, e.g., ADVANCES IN COMPUTER GRAPHICS (ed. Enderle 1990-); Chellappa and Sawchuk, DIGITAL IMAGE PROCESSING AND ANALYSIS (1985); COMPUTER GRAPHICS HARDWARE (ed. Reghbati and Lee 1988); COMPUTER GRAPHICS: IMAGE SYNTHESIS (ed. Joy et al.); Foley et al., FUNDAMENTALS OF INTERACTIVE COMPUTER GRAPHICS (2.ed. 1984); Foley, COMPUTER GRAPHICS PRINCIPLES & PRACTICE (2.ed. 1990); Foley, INTRODUCTION TO COMPUTER GRAPHICS (1994); Giloi, Interactive Computer Graphics (1978); Hearn and Baker, COMPUTER GRAPHICS (2.ed. 1994); Hill, COMPUTER GRAPHICS (1990); Latham, DICTIONARY OF COMPUTER GRAPHICS (1991); Magnenat-Thalma, IMAGE SYNTHESIS THEORY & PRACTICE (1988); Newman and Sproull, PRINCIPLES OF INTERACTIVE COMPUTER GRAPHICS (2.ed. 1979); PICTURE ENGINEERING (ed. Fu and Kunii 1982); PICTURE PROCESSING & DIGITAL FILTERING (2.ed. Huang 1979); Prosise, HOW COMPUTER GRAPHICS WORK (1994); Rimmer, BIT MAPPED GRAPHICS (2.ed. 1993); Salmon, [illegible]

(1987); Schachter, COMPUTER IMAGE GENERATION (1990); Watt, THREE-DIMENSIONAL COMPUTER GRAPHICS (2.ed. 1994); Scott Whitman, MULTIPROCESSOR METHODS FOR COMPUTER GRAPHICS RENDERING; the SIGGRAPH PROCEEDINGS for the years 1980–1994; and the *IEEE Computer Graphics and Applications* magazine for the years 1990–1994.

Background: Graphics Animation

In many areas of computer graphics a succession of slowly changing pictures are displayed rapidly one after the other, to give the impression of smooth movement, in much the same way as for cartoon animation. In general the higher the speed of the animation, the smoother (and better) the result.

When an application is generating animation images, it is normally necessary not only to draw each picture into the frame buffer, but also to first clear down the frame buffer, and to clear down auxiliary buffers such as depth (Z) buffers, stencil buffers, alpha buffers and others. A good treatment of the general principles may be found in *Computer Graphics: Principles and Practice*, James D. Foley et al., Reading Mass.: Addison-Wesley. A specific description of the various auxiliary buffers may be found in *The OpenGL Graphics System: A Specification (Version 1.0)*, Mark Segal and Kurt Akeley, SGI.

In most applications the value written, when clearing any given buffer, is the same at every pixel location, though different values may be used in different auxiliary buffers. Thus the frame buffer is often cleared to the value which corresponds to black, while the depth (Z) buffer is typically cleared to a value corresponding to infinity.

The time taken to clear down the buffers is often a significant portion of the total time taken to draw a frame, so it is important to minimize it.

Background: Parallelism in Graphics Processing

Due to the large number of at least partially independent operations which are performed in rendering, many proposals have been made to use some form of parallel architecture for graphics (and particularly for rendering). See, for example, the special issue of *Computer Graphics* on parallel rendering (September 1994). Other approaches may be found in earlier patent filings by the assignee of the present application and its predecessors, e.g. U.S. Pat. No. 5,195,186, and published PCT applications PCT/GB90/00987, PCT/GB90/01209, PCT/GB90/01210, PCT/GB90/01212, PCT/GB90/01213, PCT/GB90/01214, PCT/GB90/01215, and PCT/GB90/01216.

Background: Pipelined Processing Generally

There are several general approaches to parallel processing. One of the basic approaches to achieving parallelism in computer processing is a technique known as pipelining. In this technique the individual processors are, in effect, connected in series in an assembly-line configuration: one processor performs a first set of operations on one chunk of data, and then passes that chunk along to another processor which performs a second set of operations, while at the same time the first processor performs the first set operations again on another chunk of data. Such architectures are generally discussed in Kogge, THE ARCHITECTURE OF PIPELINED COMPUTERS (1981).

Background: The OpenGL™ Standard

The "OpenGL" standard is a very important software standard for graphics applications. In any computer system which supports this standard, the operating system(s) and application software programs can make calls according to the OpenGL standards, without knowing exactly what the hardware configuration of the system is.

The OpenGL standard provides a complete library of low-level graphics manipulation commands, which can be used to implement three-dimensional graphics operations. This standard was originally based on the proprietary standards of Silicon Graphics, Inc., but was later transformed into an open standard. It is now becoming extremely important, not only in high-end graphics-intensive workstations, but also in high-end PCs. OpenGL is supported by Windows NT™, which makes it accessible to many PC applications.

The OpenGL specification provides some constraints on the sequence of operations. For instance, the color DDA operations must be performed before the texturing operations, which must be performed before the alpha operations. (A "DDA" or digital differential analyzer, is a conventional piece of hardware used to produce linear gradation of color (or other) values over an image area.)

Other graphics interfaces (or "APIs"), such as PHIGS or XGL, are also current as of 1995; but at the lowest level, OpenGL is a superset of most of these.

The OpenGL standard is described in the *OPENGL PROGRAMMING GUIDE* (1993), the OPENGL REFERENCE MANUAL (1993), and a book by Segal and Akeley (of SGI) entitled THE OPENGL GRAPHICS SYSTEM: A SPECIFICATION (Version 1.0).

FIG. 1B is an expanded version of FIG. 1A, and shows the flow of operations defined by the OpenGL standard. Note that the most basic model is carried in terms of vertices, and these vertices are then assembled into primitives (such as triangles, lines, etc.). After all manipulation of the primitives has been completed, the rendering operations will translate each primitive into a set of "fragments." (A fragment is the portion of a primitive which affects a single pixel.) Again, it should be noted that all operations above the block marked "Rasterization" would be performed by a host processor, or possibly by a "geometry engine" (i.e. a dedicated processor which performs rapid matrix multiplies and related data manipulations), but would normally not be performed by a dedicated rendering processor such as that of the presently preferred embodiment.

One disadvantage of standards such as OpenGL is that they require that texturing or other processor-intensive operations be performed on data before pixel elimination tests, e.g. depth testing, is performed, which wastes processor time by performing costly texturing calculations on pixels which will be eliminated later in the pipeline. When the OpenGL specification is not required or when the current OpenGl state vector cannot eliminate pixels as a result of the alpha test, however, it would be much more efficient to eliminate as many pixels as possible before doing these calculations. The present application discloses a method and device for reordering the processing steps in the rendering pipeline to either accommodate order-specific specifications such as OpenGL, or to provide for an optimized throughput by only performing processor-intensive operations on pixels which will actually be displayed.

Background: Texturing

Texture patterns are commonly used as a way to apply realistic visual detail at the sub-polygon level. See Foley et al., COMPUTER GRAPHICS: PRINCIPLES AND PRACTICE (2.ed. 1990, corr. 1995), especially at pages 741–744; Paul S. Heckbert, "Fundamentals of Texture Mapping and Image Warping," Thesis submitted to Dept. of EE and Computer Science, University of California, Berkeley, Jun. 17, 1994; Heckbert, "Survey of Computer Graphics," IEEE Computer Graphics, November 1986, pp.56ff. Since the surfaces are transformed [illegible]

2D view, the textures will need to be similarly transformed by a linear transform (normally projective or "affine"). (In conventional terminology, the coordinates of the object surface, i.e. the primitive being rendered, are referred to as an (s,t) coordinate space, and the map of the stored texture is referred to a (u,v) coordinate space.) The transformation in the resulting mapping means that a horizontal line in the (x,y) display space is very likely to correspond to a slanted line in the (u,v) space of the texture map, and hence many page breaks will occur, due to the texturing operation, as rendering walks along a horizontal line of pixels.

Innovative System and Methods

The preferred embodiment discloses a pipelined graphics processor in which the sequence can be dynamically reconfigured (e.g. between primitives) in a rendering sequence. The pipeline sequence can be configured for compliance with specifications such as OpenGL, but may also be optimized by reconfiguring the pipeline sequence to eliminate unnecessary processing. In a preferred embodiment, pixel elimination sequences such as depth and stencil tests are performed before texturing calculations are performed, so that unneeded pixel data is discarded before said texturing calculations are performed.

It is noted that the texturing operations become more computation-intense, early elimination of unneeded pixels becomes even more valuable. For example, Phong shading and bump mapping both require many more operations than more common shading and texture mapping techniques, thus making the system of the present application even more valuable in real-time rendering systems.

An overhead cost is that the reconfigurable portion of the pipeline must be flushed at each reconfiguration—but since reconfiguration is normally done only on a per-primitive basis, or even less frequently, this is a relatively small cost.

BRIEF DESCRIPTION OF THE DRAWING

The disclosed inventions will be described with reference to the accompanying drawings, which show important sample embodiments of the invention and which are incorporated in the specification hereof by reference, wherein:

FIG. 1A, described above, is an overview of key elements and processes in a 3D graphics computer system.

FIG. 1B is an expanded version of FIG. 1A, and shows the flow of operations defined by the OpenGL standard.

FIG. 2A is an overview of the graphics rendering chip of the preferred embodiment of the parent case.

FIG. 2B is an overview of the graphics rendering chip of the presently preferred embodiment.

FIG. 2C is a more schematic view of the sequence of operations performed in the graphics rendering chip of FIG. 2B, when operating in a first mode.

FIG. 2D is a different view of the graphics rendering chip of FIG. 2B, showing the connections of a readback bus which provides a diagnostic pathway.

FIG. 2E is yet another view of the graphics rendering chip of FIG. 2B, showing how the functions of the core pipeline of FIG. 2C are combined with various external interface functions.

FIG. 2F is yet another view of the graphics rendering chip of FIG. 2B, showing how the details of FIFO depth and lookahead are implemented, in the presently preferred embodiment.

FIG. 3A shows a sample graphics board which incorporates the chip of FIG. 2B.

FIG. 3B shows another sample graphics board implementation, which differs from the board of FIG. 3A in that more memory and an additional component is used to achieve higher performance.

FIG. 3C shows another graphics board, in which the chip of FIG. 2B shares access to a common frame store with GUI accelerator chip.

FIG. 3D shows another graphics board, in which the chip of FIG. 2B shares access to a common frame store with a video coprocessor (which may be used for video capture and playback functions.

FIG. 4A illustrates the definition of the dominant side and the subordinate sides of a triangle.

FIG. 4B illustrates the sequence of rendering an Anti-aliased Line primitive.

FIG. 5A is a detailed view of the router unit of the presently preferred embodiment.

FIG. 5B is a detailed view of the data path through the router unit of the presently preferred embodiment when operating in a first mode.

FIG. 5C is a detailed view of the data path through the router unit of the presently preferred embodiment when operating in a second mode.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The numerous innovative teachings of the present application will be described with particular reference to the presently preferred embodiment (by way of example, and not of limitation). The presently preferred embodiment is a GLINT™ 400TX™ 3D rendering chip. The Hardware Reference Manual and Programmer's Reference Manual for this chip describe further details of this sample embodiment. Both are available, as of the effective filing date of this application, from 3Dlabs Inc. Ltd., 181 Metro Drive, Suite 520, San Jose Calif. 95110.

Definitions

The following definitions may help in understanding the exact meaning of terms used in the text of this application:

application: a computer program which uses graphics animation.

depth (Z) buffer: A memory buffer containing the depth component of a pixel. Used to, for example, eliminate hidden surfaces.

blt double-buffering: A technique for achieving smooth animation, by rendering only to an undisplayed back buffer, and then copying the back buffer to the front once drawing is complete.

FrameCount Planes: Used to allow higher animation rates by enabling DRAM local buffer pixel data, such as depth (Z), to be cleared down quickly.

frame buffer: An area of memory containing the displayable color buffers (front, back, left, right, overlay, underlay). This memory is typically separate from the local buffer.

local buffer: An area of memory which may be used to store non-displayable pixel information: depth(Z), stencil, FrameCount and GID planes. This memory is typically separate from the framebuffer.

pixel: Picture element. A pixel comprises the bits in all the buffers (whether stored in the local buffer or framebuffer), corresponding to a particular location in the framebuffer.

stencil buffer: A buffer used to store information about a pixel which controls how subsequent stencilled pixels at the same [illegible]

in the framebuffer. Typically used to mask complex two-dimensional shapes.

Preferred Chip Embodiment—Overview

The GLINT™ high performance graphics processors combine workstation class 3D graphics acceleration, and state-of-the-art 2D performance in a single chip. All 3D rendering operations are accelerated by GLINT, including Gouraud shading, texture mapping, depth buffering, anti-aliasing, and alpha blending.

The scalable memory architecture of GLINT makes it ideal for a wide range of graphics products, from PC boards to high-end workstation accelerators.

There will be several of the GLINT family of graphics processors: the GLINT 300SX™ is the embodiment of the parent case, and the GLINT 400TX™ is a presently preferred embodiment which is which is described herein in great detail. The two devices are generally compatible, with the 400TX adding local texture storage and texel address generation for all texture modes.

FIG. 2B is an overview of the graphics rendering chip of the presently preferred embodiment (i.e. the GLINT 400TX™).

General Concept

The overall architecture of the GLINT chip is best viewed using the software paradigm of a message passing system. In this system all the processing blocks are connected in a long pipeline with communication with the adjacent blocks being done through message passing. Between each block there is a small amount of buffering, the size being specific to the local communications requirements and speed of the two blocks.

The message rate is variable and depends on the rendering mode. The messages do not propagate through the system at a fixed rate typical of a more traditional pipeline system. If the receiving block can not accept a message, because its input buffer is full, then the sending block stalls until space is available.

The message structure is fundamental to the whole system as the messages are used to control, synchronize and inform each block about the processing it is to undertake. Each message has two fields—a 32 bit data field and a 9 bit tag field. (This is the minimum width guaranteed, but some local block to block connections may be wider to accommodate more data.) The data field will hold color information, coordinate information, local state information, etc. The tag field is used by each block to identify the message type so it knows how to act on it.

Each block, on receiving a message, can do one of several things:

- Not recognize the message so it just passes it on to the next block.
- Recognize it as updating some local state (to the block) so the local state is updated and the message terminated, i.e. not passed on to the next block.
- Recognize it as a processing action, and if appropriate to the unit, the processing work specific to the unit is done. This may entail sending out new messages such as Color and/or modifying the initial message before sending it on. Any new messages are injected into the message stream before the initial message is forwarded on. Some examples will clarify this.

When the Depth Block receives a message 'new fragment', it will calculate the corresponding depth and do the depth test. If the test passes then the 'new fragment' message is passed to the next unit. If the test fails then the message is modified and passed on. The temptation is not to pass the message on when the test fails (because the pixel is not going to be updated), but other units downstream need to keep their local DDA units in step.

(In the present application, the messages are being described in general terms so as not to be bogged down in detail at this stage. The details of what a 'new fragment' message actually specifies (i.e. coordinate, color information) is left till later. In general, the term "pixel" is used to describe the picture element on the screen or in memory. The term "fragment" is used to describe the part of a polygon or other primitive which projects onto a pixel. Note that a fragment may only cover a part of a pixel.) When the Texture Read Unit (if enabled) gets a 'new fragment' message, it will calculate the texture map addresses, and will accordingly provide 1, 2, 4 or 8 texels to the next unit together with the appropriate number of interpolation coefficients.

Each unit and the message passing are conceptually running asynchronous to all the others. However, in the presently preferred embodiment there is considerable synchrony because of the common clock.

How does the host process send messages? The message data field is the 32 bit data written by the host, and the message tag is the bottom 9 bits of the address (excluding the byte resolution address lines). Writing to a specific address causes the message type associated with that address to be inserted into the message queue. Alternatively, the on-chip DMA controller may fetch the messages from the host's memory.

The message throughput, in the presently preferred embodiment, is 50M messages per second and this gives a fragment throughput of up to 50M per second, depending on what is being rendered. Of course, this rate will predictably be further increased over time, with advances in process technology and clock rates.

Linkage

The block diagram of FIG. 2A shows how the units are connected together in the GLINT 300SX embodiment, and the block diagram of FIG. 2B shows how the units are connected together in the presently preferred embodiment. Some general points are:

- The following functionality is present in the 400TX, but missing from the 300SX: The Texture Address (TAddr) and Texture Read (TRd) Units are missing. Also, the router and multiplexer are missing from this section, so the unit ordering is Scissor/Stipple, Color DDA, Texture Fog Color, Alpha Test, LB Rd, etc.
- In the embodiment of FIG. 2B, the order of the units can be configured in two ways. The most general order (Router, Color DDA, Texture Unit, Alpha Test, LB Rd, GID/Z/Stencil, LB Wr, Multiplexer) and will work in all modes of OpenGL. However, when the alpha test is disabled it is much better to do the Graphics ID, depth and stencil tests before the texture operations rather than after. This is because the texture operations have a high processing cost and this should not be spent on fragments which are later rejected because of window, depth or stencil tests.
- The loop back to the host at the bottom provides a simple synchronization mechanism. The host can insert a Sync command and when all the preceding rendering has finished the sync command will reach the bottom host interface which will notify the host the sync event has occurred.

Benefits

The very modular nature of this architecture gives great benefits [illegible]

has a very well defined set of input and output messages. This allows the internal structure of a unit (or group of units) to be changed to make algorithmic/speed/gate count trade-offs.

The isolation and well defined logical and behavioral interface to each unit allows much better testing and verification of the correctness of a unit.

The message passing paradigm is easy to simulate with software, and the hardware design is nicely partitioned. The architecture is self synchronizing for mode or primitive changes.

The host can mimic any block in the chain by inserting messages which that block would normally generate. These message would pass through the earlier blocks to the mimicked block unchanged and from then onwards to the rest of the blocks which cannot tell the message did not originate from the expected block. This allows for an easy work around mechanism to correct any flaws in the chip. It also allows other rasterization paradigms to be implemented outside of the chip, while still using the chip for the low level pixel operations.

"A Day in the Life of a Triangle"

Before we get too detailed in what each unit does it is worth while looking in general terms at how a primitive (e.g. triangle) passes through the pipeline, what messages are generated, and what happens in each unit. Some simplifications have been made in the description to avoid detail which would otherwise complicate what is really a very simple process. The primitive we are going to look at is the familiar Gouraud shaded Z buffered triangle, with dithering. It is assumed any other state (i.e. depth compare mode) has been set up, but (for simplicity) such other states will be mentioned as they become relevant.

The application generates the triangle vertex information and makes the necessary OpenGL calls to draw it.

The OpenGL server/library gets the vertex information, transforms, clips and lights it. It calculates the initial values and derivatives for the values to interpolate (X_{left}, X_{right}, red, green, blue and depth) for unit change in dx and $dxdy_{left}$. All these values are in fixed point integer and have unique message tags. Some of the values (the depth derivatives) have more than 32 bits to cope with the dynamic range and resolution so are sent in two halves. Finally, once the derivatives, start and end values have been sent to GLINT the 'render triangle' message is sent.

On GLINT: The derivative, start and end parameter messages are received and filter down the message stream to the appropriate blocks. The depth parameters and derivatives to the Depth Unit; the RGB parameters and derivative to the Color DDA Unit; the edge values and derivatives to the Rasterizer Unit.

The 'render triangle' message is received by the rasterizer unit and all subsequent messages (from the host) are blocked until the triangle has been rasterized (but not necessarily written to the frame store). A 'prepare to render' message is passed on so any other blocks can prepare themselves.

The Rasterizer Unit walks the left and right edges of the triangle and fills in the spans between. As the walk progresses messages are send to indicate the direction of the next step: StepX or StepYDomEdge. The data field holds the current (x, y) coordinate. One message is sent per pixel within the triangle boundary. The step messages are duplicated into two groups: an active group and a passive group. The messages always start off in the active group but may be changed to the passive group if this unit fails one of the tests (e.g. depth) on its path down the message stream. The two groups are distinguished by a single bit in the message tag. The step messages (in either form) are always passed throughout the length of the message stream, and are used by all the DDA units to keep their interpolation values in step. The step message effectively identifies the fragment and any other messages pertaining to this fragment will always precede the step message in the message stream.

The Scissor and Stipple Unit. This unit does 4 tests on the fragment (as embodied by the active step message). The screen scissor test takes the coordinates associated with the step message, converts them to be screen relative (if necessary) and compares them against the screen boundaries. The other three tests (user scissor, line stipple and area stipple) are disabled for this example. If the enabled tests pass then the active step is forwarded onto the next unit, otherwise it is changed into a passive step and then forwarded.

The Color DDA unit responds to an active step message by generating a Color message and sending this onto the next unit. The active step message is then forwarded to the next unit. The Color message holds, in the data field, the current RGBA value from the DDA. If the step message is passive then no Color message is generated. After the Color message is sent (or would have been sent) the step message is acted on to increment the DDA in the correct direction, ready for the next pixel.

Texturing, Fog and Alpha Tests Units are disabled so the messages just pass through these blocks.

In general terms the Local Buffer Read Unit reads the Graphic ID, Stencil and Depth information from the Local Buffer and passes it onto the next unit. More specifically it does:

1. If the step message is passive then no further action occurs.
2. On an active step message it calculates the linear address in the local buffer of the required data. This is done using the (X, Y) position recorded in the step message and locally stored information on the 'screen width' and window base address. Separate read and write addresses are calculated.
3. The addresses are passed to the Local Buffer Interface Unit and the identified local buffer location read. The write address is held for use later.
4. Sometime later the local buffer data is returned and is formatted into a consistent internal format and inserted into a 'Local Buffer Data' message and passed on to the next unit.

   The message data field is made wider to accommodate the maximum Local Buffer width of 52 bits (32 depth, 8 stencil, 4 graphic ID, 8 frame count) and this extra width just extends to the Local Buffer Write block.

   The actual data read from the local buffer can be in several formats to allow narrower width memories to be used in cost sensitive systems. The narrower data is formatted into a consistent internal format in this block.

The Graphic ID, Stencil and Depth Unit just passes the Color message through and stores the LBData message until the step message arrives. A passive step message would just pass straight through. When the active step message is received the internal Graphic ID, stencil and depth values are compared with the ones in the LBData message as specified by this unit's mode information. If the enabled tests pass then the new local buffer data is sent in the LBWrite Data message to the next unit. If the

active step message forwarded. If any of the enabled tests fail then an LBCancelWrite message is sent followed by the equivalent passive step message. The depth DDA is stepped to update the local depth value.

The Local Buffer Write Unit performs any writes which are necessary. The LBWriteData message has its data formatted into the external local buffer format and this is posted to the Local Buffer Interface Unit to be written into the memory (the write address is already waiting in the Local Buffer Interface Unit). The LBWriteCancel message just informs the Local Buffer Interface Unit that the pending write address is no longer needed and can be discarded. The step message is just passed through.

In general terms the Framebuffer Read Unit reads the color information from the framebuffer and passes it onto the next unit. More specifically it does:

1. If the step message is passive then no further action occurs.
2. On an active step message it calculates the linear address in the framebuffer of the required data. This is done using the (X, Y) position recorded in the step message and locally stored information on the 'screen width' and window base address. Separate read and write addresses are calculated.
3. The addresses are passed to the Framebuffer Interface Unit and the identified framebuffer location read. The write address is held for use later.
4. Sometime later the color data is returned and inserted into a 'Frame Buffer Data' message and passed on to the next unit.

The actual data read from the framestore can be in several formats to allow narrower width memories to be used in cost sensitive systems. The formatting of the data is deferred until the Alpha Blend Unit as it is the only unit which needs to match it up with the internal formats. In this example no alpha blending or logical operations are taking place, so reads are disabled and hence no read address is sent to the Framebuffer Interface Unit. The Color and step messages just pass through.

The Alpha Blend Unit is disabled so just passes the messages through.

The Dither Unit stores the Color message internally until an active step is received. On receiving this it uses the least significant bits of the (X, Y) coordinate information to dither the contents of the Color message. Part of the dithering process is to convert from the internal color format into the format of the framebuffer. The new color is inserted into the Color message and passed on, followed by the step message.

The Logical Operations are disabled so the Color message is just converted into the FBWriteData message (just the tag changes) and forwarded on to the next unit. The step message just passes through.

The Framebuffer Write Unit performs any writes which are necessary.

The FBWriteData message has its data posted to the Framebuffer Interface Unit to be written into the memory (the write address is already waiting in the Framebuffer Interface Unit).

The step message is just passed through.

The Host Out Unit is mainly concerned with synchronization with the host so for this example will just consume any messages which reach this point in the message stream.

This description has concentrated on what happens as one fragment flows down the message stream. It is important to remember that at any instant in time there are many fragments flowing down the message stream and the further down they reach the more processing has occurred.

Interfacing Between Blocks FIG. 2B shows the FIFO buffering and lookahead connections which are used in the presently preferred embodiment. The FIFOs are used to provide an asynchronous interface between blocks, but are expensive in terms of gate count. Note that most of these FIFOs are only one stage deep (except where indicated), which reduces their area. To maintain performance, lookahead connections are used to accelerate the "startup" of the pipeline. For example, when the Local-Buffer-Read block issues a data request, the Texture/Fog/Color blocks also receive this, and begin to transfer data accordingly. Normally a single-entry deep FIFO cannot be read and written in the same cycle, as the writing side doesn't know that the FIFO is going to be read in that cycle (and hence become eligible to be written). The look-ahead feature give the writing side this insight, so that single-cycle transfer can be achieved. This accelerates the throughput of the pipeline.

Programming Model

The following text describes the programming model for GLINT.

GLINT as a Register file

The simplest way to view the interface to GLINT is as a flat block of memory-mapped registers (i.e. a register file). This register file appears as part of Region 0 of the PCI address map for GLINT. See the GLINT Hardware Reference Manual for details of this address map.

When a GLINT host software driver is initialized it can map the register file into its address space. Each register has an associated address tag, giving its offset from the base of the register file (since all registers reside on a 64-bit boundary, the tag offset is measured in multiples of 8 bytes). The most straightforward way to load a value into a register is to write the data to its mapped address. In reality the chip interface comprises a 16 entry deep FIFO, and each write to a register causes the written value and the register's address tag to be written as a new entry in the FIFO.

Programming GLINT to draw a primitive consists of writing initial values to the appropriate registers followed by a write to a command register. The last write triggers the start of rendering.

GLINT has approximately 200 registers. All registers are 32 bits wide and should be 32-bit addressed. Many registers are split into bit fields, and it should be noted that bit 0 is the least significant bit.

Register Types

GLINT has three main types of register:

Control Registers

Command Registers

Internal Registers

Control Registers are updated only by the host—the chip effectively uses them as read-only registers. Examples of control registers are the Scissor Clip unit min and max registers. Once initialized by the host, the chip only reads these registers to determine the scissor clip extents.

Command Registers are those which, when written to, typically cause the chip to start rendering (some command registers such as ResetPickResult or Sync do not initiate rendering). Normally, the host will initialize the appropriate control registers and then write to a command register to initiate drawing. There are two types of command registers: begin-draw and continue-draw. Begin-draw commands cause rendering to start from the specified X, Y position

control registers. Continue-draw commands cause drawing to continue with internal register values as they were when the previous drawing operation completed. Making use of continue-draw commands can significantly reduce the amount of data that has to be loaded into GLINT when drawing multiple connected objects such as polylines. Examples of command registers include the Render and ContinueNewLine registers.

For convenience this application will usually refer to "sending a Render command to GLINT" rather than saying (more precisely) "the Render Command register is written to, which initiates drawing".

Internal Registers are not accessible to host software. They are used internally by the chip to keep track of changing values. Some control registers have corresponding internal registers. When a begindraw command is sent and before rendering starts, the internal registers are updated with the values in the corresponding control registers. If a continue-draw command is sent then this update does not happen and drawing continues with the current values in the internal registers. For example, if a line is being drawn then the StartXDom and StartY control registers specify the (x, y) coordinates of the first point in the line. When a begin-draw command is sent these values are copied into internal registers. As the line drawing progresses these internal registers are updated to contain the (x, y) coordinates of the pixel being drawn. When drawing has completed the internal registers contain the (x, y) coordinates of the next point that would have been drawn. If a continue-draw command is now given these final (x, y) internal values are not modified and further drawing uses these values. If a begin-draw command had been used the internal registers would have been reloaded from the StartXDom and StartY registers.

For the most part internal registers can be ignored. It is helpful to appreciate that they exist in order to understand the continue-draw commands.

GLINT I/O Interface

There are a number of ways of loading GLINT registers for a given context:

- The host writes a value to the mapped address of the register
- The host writes address-tag/data pairs into a host memory buffer and uses the on-chip DMA to transfer this data to the FIFO.
- The host can perform a Block Command Transfer by writing address and data values to the FIFO interface registers.

In all cases where the host writes data values directly to the chip (via the register file) it has to worry about FIFO overflow. The InFIFOSpace register indicates how many free entries remain in the FIFO. Before writing to any register the host must ensure that there is enough space left in the FIFO. The values in this register can be read at any time. When using DMA, the DMA controller will automatically ensure that there is room in the FIFO before it performs further transfers. Thus a buffer of any size can be passed to the DMA controller.

FIFO Control

The description above considered the GLINT interface to be a register file. More precisely, when a data value is written to a register this value and the address tag for that register are combined and put into the FIFO as a new entry. The actual register is not updated until GLINT processes this entry. In the case where GLINT is busy performing a time consuming operation (e.g. drawing a large texture mapped polygon), and not draining the FIFO very quickly, it is possible for the FIFO to become full. If a write to a register is performed when the FIFO is full no entry is put into the FIFO and that write is effectively lost.

The input FIFO is **16** entries deep and each entry consists of a tag/data pair. The InFIFOSpace register can be read to determine how many entries are free. The value returned by this register will never be greater than **16**.

To check the status of the FIFO before every write is very inefficient, so it is preferably checked before loading the data for each rectangle. Since the FIFO is **16** entries deep, a further optimization is to wait for all **16** entries to be free after every second rectangle. Further optimizations can be made by moving dXDom, dXSub and dY outside the loop (as they are constant for each rectangle) and doing the FIFO wait after every third rectangle.

The InFIFOSpace FIFO control register contains a count of the number of entries currently free in the FIFO. The chip increments this register for each entry it removes from the FIFO and decrements it every time the host puts an entry in the FIFO.

The DMA Interface

Loading registers directly via the FIFO is often an inefficient way to download data to GLINT. Given that the FIFO can accommodate only a small number of entries, GLINT has to be frequently interrogated to determine how much space is left. Also, consider the situation where a given API function requires a large amount of data to be sent to GLINT. If the FIFO is written directly then a return from this function is not possible until almost all the data has been consumed by GLINT. This may take some time depending on the types of primitives being drawn.

To avoid these problems GLINT provides an on-chip DMA controller which can be used to load data from arbitrary sized (<64K 32-bit words) host buffers into the FIFO. In its simplest form the host software has to prepare a host buffer containing register address tag descriptions and data values. It then writes the base address of this buffer to the DMAAddress register and the count of the number of words to transfer to the DMACount register. Writing to the DMACount register starts the DMA transfer and the host can now perform other work. In general, if the complete set of rendering commands required by a given call to a driver function can be loaded into a single DMA buffer then the driver function can return. Meanwhile, in parallel, GLINT is reading data from the host buffer and loading it into its FIFO. FIFO overflow never occurs since the DMA controller automatically waits until there is room in the FIFO before doing any transfers.

The only restriction on the use of DMA control registers is that before attempting to reload the DMACount register the host software must wait until previous DMA has completed. It is valid to load the DMAAddress register while the previous DMA is in progress since the address is latched internally at the start of the DMA transfer.

Using DMA leaves the host free to return to the application, while in parallel, GLINT is performing the DMA and drawing. This can increase performance significantly over loading a FIFO directly. In addition, some algorithms require that data be loaded multiple times (e.g. drawing the same object across multiple clipping rectangles). Since the GLINT DMA only reads the buffer data, it can be downloaded many times simply by restarting the DMA. This can be very beneficial if composing the buffer data is a time consuming task.

The host can use this hardware capability in various ways. For example, a further optional optimization is to use a double buffered mechanism with two DMA buffers. This allows the [illegible] to be filled [illegible]

previous DMA to complete, thus further improving the parallelism between host and GLINT processing. Thus, this optimization is dependent on the allocation of the host memory. If there is only one DMA host buffer then either it is being filled or it is being emptied—it cannot be filled and emptied at the same time, since there is no way for the host and DMA to interact once the DMA transfer has started. The host is at liberty to allocate as many DMA buffers as it wants; two is the minimum to do double buffering, but allocating many small buffers is generally better, as it gives the benefits of double buffering together with low latency time, so GLINT is not idle while large buffer is being filled up. However, use of many small buffers is of course more complicated.

In general the DMA buffer format consists of a 32-bit address tag description word followed by one or more data words. The DMA buffer consists of one or more sets of these formats. The following paragraphs describe the different types of tag description words that can be used.

DMA Tag Description Format

There are 3 different tag addressing modes for DMA: hold, increment and indexed. The different DMA modes are provided to reduce the amount of data which needs to be transferred, hence making better use of the available DMA bandwidth. Each of these is described in the following sections.

Hold Format

In this format the 32-bit tag description contains a tag value and a count specifying the number of data words following in the buffer. The DMA controller writes each of the data words to the same address tag. For example, this is useful for image download where pixel data is continuously written to the Color register. The bottom 9 bits specify the register to which the data should be written; the high-order 16 bits specify the number of data words (minus 1) which follow in the buffer and which should be written to the address tag (note that the 2 -bit mode field for this format is zero so a given tag value can simply be loaded into the low order 16 bits).

A special case of this format is where the top 16 bits are zero indicating that a single data value follows the tag (i.e. the 32-bit tag description is simply the address tag value itself). This allows simple DMA buffers to be constructed which consist of tag/data pairs.

Increment Format

This format is similar to the hold format except that as each data value is loaded the address tag is incremented (the value in the DMA buffer is not changed; GLINT updates an internal copy). Thus, this mode allows contiguous GLINT registers to be loaded by specifying a single 32-bit tag value followed by a data word for each register. The low-order 9 bits specify the address tag of the first register to be loaded. The 2 bit mode field is set to 1 and the high-order 16 bits are set to the count (minus 1) of the number of registers to update. To enable use of this format, the GLINT register file has been organized so that registers which are frequently loaded together have adjacent address tags. For example, the 32 AreaStipplePattern registers can be loaded as follows:

```
AreaStipplePattern0, Count=31, Mode=1
row 0 bits
row 1 bits
...
row 31 bits
```

Indexed Format

GLINT address tags are 9 bit values. For the purposes of Indexed Format they are organized into major groups and within each group there are up to 16 tags. The low-order 4 bits of a tag give its offset within the group. The high-order 5 bits give the major group number.

The following Register Table lists the individual registers with their Major Group and Offset in the presently preferred embodiment:

Register Table

The following table lists registers by group, giving their tag values and indicating their type. The register groups may be used to improve data transfer rates to GLINT when using DMA.

The following types of register are distinguished:

Unit	Register	Major Group (hex)	Offset (hex)	Type
Rasterizer	StartXDom	00	0	Control
	dXDom	00	1	Control
	StartXSub	00	2	Control
	dXSub	00	3	Control
	StartY	00	4	Control
	dY	00	5	Control
	Count	00	6	Control
	Render	00	7	Command
	ContinueNewLine	00	8	Command
	ContinueNewDom	00	9	Command
	ContinueNewSub	00	A	Command
	Continue	00	B	Command
	FlushSpan	00	C	Command
	BitMaskPattern	00	D	Mixed
Rasterizer	PointTable[0-3]	01	0-3	Control
	RasterizerMode	01	4	Control
Scissor Stipple	ScissorMode	03	0	Control
	ScissorMinXY	03	1	Control
	ScissorMaxXY	03	2	Control
	ScreenSize	03	3	Control
	AreaStippleMode	03	4	Control
	LineStippleMode	03	5	Control
	LoadLineStipple Counters	03	6	Control
	UpdateLineStipple Counters	03	7	Command
	SaveLineStipple State	03	8	Command
	WindowOrigin	03	9	Control
Scissor Stipple	AreaStipplePat-tern[0-31]	04 05	0-F 0-F	Control
Texture Color/Fog	Texel0	0C	0	Control
	Texel1	0C	1	Control
	Texel2	0C	2	Control
	Texel3	0C	3	Control
	Texel4	0C	4	Control
	Texel5	0C	5	Control
	Texel6	0C	6	Control
	Texel7	0C	7	Control
	Interp0	0C	8	Control
	Interp1	0C	9	Control
	Interp2	0C	A	Control
	Interp3	0C	B	Control
	Interp4	0C	C	Control
	TextureFilter	0C	D	Control
Texture/Fog Color	TextureColor Mode	0D	0	Control
	TextureEnvColor	0D	1	Control
	FogMode	0D	2	Control
	FogColor	0D	3	Control
	FStart	0D	4	Control
	dFdx	0D	5	Control
	dFdyDom	0D	6	Control
Color DDA	RStart	0F	0	Control
	dRdx	0F	1	Control
	dRdyDom	0F	2	Control
	GStart	0F	3	Control
	dGdx	0F	4	Control
	dGdyDom	0F	5	Control

-continued

Unit	Register	Major Group (hex)	Offset (hex)	Type
	BStart	0F	6	Control
	dBdx	0F	7	Control
	dBdyDom	0F	8	Control
	AStart	0F	9	Control
	dAdx	0F	A	Control
	dAdyDom	0F	B	Control
	ColorDDAMode	0F	C	Control
	ConstantColor	0F	D	Control
	Color	0F	E	Mixed
Alpha Test	AlphaTestMode	10	0	Control
	AntialiasMode	10	1	Control
Alpha Blend	AlphaBlendMode	10	2	Control
Dither	DitherMode		3	Control
Logical Ops	FBSoftwareWrite Mask	10	4	Control
	LogicalOpMode	10	5	Control
	FBWriteData	10	6	Control
LB Read	LBReadMode	11	0	Control
	LBReadFormat	11	1	Control
	LBSourceOffset	11	2	Control
	LBStencil	11	5	Output
	LBDepth	11	6	Output
	LBWindowBase	11	7	Control
LB Write	LBWriteMode	11	8	Control
	LBWriteFormat	11	9	Control
GID/Stencil/ Depth	Window	13	0	Control
	StencilMode	13	1	Control
	StencilData	13	2	Control
	Stencil	13	3	Mixed
	DepthMode	13	4	Control
	Depth	13	5	Mixed
	ZStartU	13	6	Control
	ZStartL	13	7	Control
	dZdxU	13	8	Control
	dZdxL	13	9	Control
	dZdyDomU	13	A	Control
	dZdyDomL	13	B	Control
	FastClearDepth	13	C	Control
FB Read	FBReadMode	15	0	Control
	FBSourceOffset	15	1	Control
	FBPixelOffset	15	2	Control
	FBColor	15	3	Output
	FBWindowBase	15	6	Control
FB Write	FBWriteMode	15	7	Control
	FBHardwareWrite Mask	15	8	Control
	FBBlockColor	15	9	Control
Host Out	FilterMode	18	0	Control
	StatisticMode	18	1	Control
	MinRegion	18	2	Control
	MaxRegion	18	3	Control
	ResetPickResult	18	4	Command
	MinHitRegion	18	5	Command
	MaxHitRegion	18	6	Command
	PickResult	18	7	Command
	Sync	18	8	Command

This format allows up to 16 registers within a group to be loaded while still only specifying a single address tag description word.

If the Mode of the address tag description word is set to indexed mode, then the high-order 16 bits are used as a mask to indicate which registers within the group are to be used. The bottom 4 bits of the address tag description word are unused. The group is specified by bits 4 to 8. Each bit in the mask is used to represent a unique tag within the group. If a bit is set then the corresponding register will be loaded. The number of bits set in the mask determines the number of data words that should be following the tag description word in the DMA buffer. The data is stored in order of increasing bit corresponding address tag.

DMA Buffer Addresses

Host software must generate the correct DMA buffer address for the GLINT DMA controller. Normally, this means that the address passed to GLINT must be the physical address of the DMA buffer in host memory. The buffer must also reside at contiguous physical addresses as accessed by GLINT. On a system which uses virtual memory for the address space of a task, some method of allocating contiguous physical memory, and mapping this into the address space of a task, must be used.

If the virtual memory buffer maps to non-contiguous physical memory, then the buffer must be divided into sets of contiguous physical memory pages and each of these sets transferred separately. In such a situation the whole DMA buffer cannot be transferred in one go; the host software must wait for each set to be transferred. Often the best way to handle these fragmented transfers is via an interrupt handler.

DMA Interrupts

GLINT provides interrupt support, as an alternative means of determining when a DMA transfer is complete. If enabled, the interrupt is generated whenever the DMACount register changes from having a non-zero to having a zero value. Since the DMACount register is decremented every time a data item is transferred from the DMA buffer this happens when the last data item is transferred from the DMA buffer.

To enable the DMA interrupt, the DMAInterruptEnable bit must be set in the IntEnable register. The interrupt handler should check the DMAFlag bit in the IntFlags register to determine that a DMA interrupt has actually occurred. To clear the interrupt a word should be written to the IntFlags register with the DMAFlag bit set to one.

This scheme frees the processor for other work while DMA is being completed. Since the overhead of handling an interrupt is often quite high for the host processor, the scheme should be tuned to allow a period of polling before sleeping on the interrupt.

Output FIFO and Graphics Processor FIFO Interface

To read data back from GLINT an output FIFO is provided. Each entry in this FIFO is 32-bits wide and it can hold tag or data values. Thus its format is unlike the input FIFO whose entries are always tag/data pairs (we can think of each entry in the input FIFO as being 41 bits wide: 9 bits for the tag and 32 bits for the data). The type of data written by GLINT to the output FIFO is controlled by the FilterMode register. This register allows filtering of output data in various categories including the following:

- Depth: output in this category results from an image upload of the Depth buffer.
- Stencil: output in this category results from an image upload of the Stencil buffer.
- Color: output in this category results from an image upload of the framebuffer.
- Synchronization: synchronization data is sent in response to a Sync command.

The data for the FilterMode register consists of 2 bits per category. If the least significant of these two bits is set (0×1) then output of the register tag for that category is enabled; if the most significant bit is set (0×2) then output of the data for that category is enabled. Both bits set (0×3) enables output of both

enabled at the same time. In this case the tag is written first to the FIFO followed by the data.

For example, to perform an image upload from the framebuffer, the FilterMode register should have data output enabled for the Color category. Then, the rectangular area to be uploaded should be described to the rasterizer. Each pixel that is read from the framebuffer will then be placed into the output FIFO. If the output FIFO becomes full, then GLINT will block internally until space becomes available. It is the programmer's responsibility to read all data from the output FIFO. For example, it is important to know how many pixels should result from an image upload and to read exactly this many from the FIFO.

To read data from the output FIFO the OutputFIFOWords register should first be read to determine the number of entries in the FIFO (reading from the FIFO when it is empty returns undefined data). Then this many 32-bit data items are read from the FIFO. This procedure is repeated until all the expected data or tag items have been read. The address of the output FIFO is described below.

Note that all expected data must be read back. GLINT will block if the FIFO becomes full. Programmers must be careful to avoid the deadlock condition that will result if the host is waiting for space to become free in the input FIFO while GLINT is waiting for the host to read data from the output FIFO.

Graphics Processor FIFO Interface

GLINT has a sequence of 1K×32 bit addresses in the PCI Region 0 address map called the Graphics Processor FIFO Interface. To read from the output FIFO any address in this range can be read (normally a program will choose the first address and use this as the address for the output FIFO). All 32-bit addresses in this region perform the same function: the range of addresses is provided for data transfer schemes which force the use of incrementing addresses.

Writing to a location in this address range provides raw access to the input FIFO. Again, the first address is normally chosen. Thus the same address can be used for both input and output FIFOs. Reading gives access to the output FIFO; writing gives access to the input FIFO.

Writing to the input FIFO by this method is different from writing to the memory mapped register file. Since the register file has a unique address for each register, writing to this unique address allows GLINT to determine the register for which the write is intended. This allows a tag/data pair to be constructed and inserted into the input FIFO. When writing to the raw FIFO address an address tag description must first be written followed by the associated data. In fact, the format of the tag descriptions and the data that follows is identical to that described above for DMA buffers. Instead of using the GLINT DMA it is possible to transfer data to GLINT by constructing a DMA-style buffer of data and then copying each item in this buffer to the raw input FIFO address. Based on the tag descriptions and data written GLINT constructs tag/data pairs to enter as real FIFO entries. The DMA mechanism can be thought of as an automatic way of writing to the raw input FIFO address.

Note, that when writing to the raw FIFO address the FIFO full condition must still be checked by reading the InFIFOSpace register. However, writing tag descriptions does not cause any entries to be entered into the FIFO: such a write simply establishes a set of tags to be paired with the subsequent data. Thus, free space need be ensured only for actual data items that are written (not the tag values). For example, in the simplest case where each tag is followed by a single data item, assuming that the FIFO is empty, then 32 [illegible] before checking again for free space.

Other Interrupts

GLINT also provides interrupt facilities for the following:

Sync: If a Sync command is sent and the Sync interrupt has been enabled then once all rendering has been completed, a data value is entered into the Host Out FIFO, and a Sync interrupt is generated when this value reaches the output end of the FIFO. Synchronization is described further in the next section.

External: this provides the capability for external hardware on a GLINT board (such as an external video timing generator) to generate interrupts to the host processor.

Error: if enabled the error interrupt will occur when GLINT detects certain error conditions , such as an attempt to write to a full FIFO.

Vertical Retrace: if enabled a vertical retrace interrupt is generated at the start of the video blank period.

Each of these are enabled and cleared in a similar way to the DMA interrupt.

Synchronization

There are three main cases where the host must synchronize with GLINT:

before reading back from registers

before directly accessing the framebuffer or the local-buffer via the bypass mechanism

framebuffer management tasks such as double buffering

Synchronizing with GLINT implies waiting for any pending DMA to complete and waiting for the chip to complete any processing currently being performed. The following pseudo-code shows the general scheme:

```
GLINTData data;
// wait for DMA to complete
while (*DMACount != 0) {
  poll or wait for interrupt
}
while (*InFIFOSpace < 2) {
  ; // wait for free space in the FIFO
}
// enable sync output and send the Sync command
data.Word = 0;
data.FilterMode.Synchronization = 0x1;
FilterMode(data.Word);
Sync(0x0);
/* wait for the sync output data */
do {
  while (*OutFIFOWords == 0)
    ; // poll waiting for data in output
FIFO
} while (*OutputFIFO != Sync_tag);
```

Initially, we wait for DMA to complete as normal. We then have to wait for space to become free in the FIFO (since the DMA controller actually loads the FIFO). We need space for 2 registers: one to enable generation of an output sync value, and the Sync command itself. The enable flag can be set at initialization time. The output value will be generated only when a Sync command has actually been sent, and GLINT has then completed all processing.

Rather than polling it is possible to use a Sync interrupt as mentioned in the previous section. As well as enabling the interrupt and setting the filter mode, the data sent in the Sync command must have the most significant bit set in order to generate the interrupt. The interrupt is generated when the tag or data reaches the output end of the Host Out FIFO. Use of the Sync interrupt has to be considered carefully as GLINT will generally empty the FIFO more quickly than it takes to set up and handle the interrupt.

Host Framebuffer Bypass

Normally, the host will access the framebuffer indirectly via commands sent to the GLINT FIFO interface. However,

GLINT does provide the whole framebuffer as part of its address space so that it can be memory mapped by an application. Access to the framebuffer via this memory mapped route is independent of the GLINT FIFO.

Drivers may choose to use direct access to the framebuffer for algorithms which are not supported by GLINT. The framebuffer bypass supports big-endian, little-endian and GIB-endian formats.

A driver making use of the framebuffer bypass mechanism should synchronize framebuffer accesses made through the FIFO with those made directly through the memory map. If data is written to the FIFO and then an access is made to the framebuffer, it is possible that the framebuffer access will occur before the commands in the FIFO have been fully processed. This lack of temporal ordering is generally not desirable.

Framebuffer Dimensions and Depth

At reset time the hardware stores the size of the framebuffer in the FBMemoryControl register. This register can be read by software to determine the amount of VRAM on the display adapter. For a given amount of VRAM, software can configure different screen resolutions and off-screen memory regions.

The framebuffer width must be set up in the FBReadMode register. The first 9 bits of this register define 3 partial products which determine the offset in pixels from one scanline to the next. Typically, these values will be worked out at initialization time and a copy kept in software. When this register needs to be modified the software copy is retrieved and any other bits modified before writing to the register.

Once the offset from one scanline to the next has been established, determining the visible screen width and height becomes a clipping issue. The visible screen width and height are set up in the ScreenSize register and enabled by setting the ScreenScissorEnable bit in the ScissorMode register.

The framebuffer depth (8, 16 or 32-bit) is controlled by the FBModeSel register. This register provides a 2 bit field to control which of the three pixel depths is being used. The pixel depth can be changed at any time but this should not be attempted without first synchronizing with GLINT. The FBModeSel register is not a FIFO register and is updated immediately it is written. If GLINT is busy performing rendering operations, changing the pixel depth will corrupt that rendering.

Normally, the pixel depth is set at initialization time. To optimize certain 2D rendering operations it may be desirable to change it at other times. For example, if the pixel depth is normally 8 (or 16) bits, changing the pixel depth to 32 bits for the duration of a bitblt can quadruple (or double) the blt speed, when the bit source and destination edges are aligned on 32 bit boundaries. Once such a blt sequence has been set up the host software must wait and synchronize with GLINT and then reset the pixel depth before continuing with further rendering. It is not possible to change the pixel depth via the FIFO, thus explicit synchronization must always be used.

Host Localbuffer Bypass

As with the framebuffer, the localbuffer can be mapped in and accessed directly. The host should synchronize with GLINT before making any direct access to the localbuffer.

At reset time the hardware saves the size of the localbuffer in the LBMemoryControl register (localbuffer visible region size). In bypass mode the number of bits per pixel is either 32 or 64. This information is also set in the LBMemoryControl register (localbuffer bypass packing). This pixel [illegible] is the memory offset between one pixel and the next. A further set of 3 bits (localbuffer width) in the LBMemoryControl register defines the number of valid bits per pixel. A typical localbuffer configuration might be 48 bits per pixel but in bypass mode the data for each pixel starts on a 64-bit boundary. In this case valid pixel data will be contained in bits 0 to 47. Software must set the LBReadFormat register to tell GLINT how to interpret these valid bits.

Host software must set the width in pixels of each scanline of the localbuffer in the LBReadMode FIFO register. The first 9 bits of this register define 3 partial products which determine the offset in pixels from one scanline to the next. As with the framebuffer partial products, these values will usually be worked out at initialization time and a copy kept in software. When this register needs to be modified the software copy is retrieved and any other bits modified before writing to the register. If the system is set up so that each pixel in the framebuffer has a corresponding pixel in the localbuffer then this width will be the same as that set for the framebuffer.

The localbuffer is accessible via Regions 1 and 3 of the PCI address map for GLINT. The localbuffer bypass supports big-endian and little-endian formats. These are described in a later section.

Register Read Back

Under some operating environments, multiple tasks will want access to the GLINT chip. Sometimes a server task or driver will want to arbitrate access to GLINT on behalf of multiple applications. In these circumstances, the state of the GLINT chip may need to be saved and restored on each context switch. To facilitate this, the GLINT control registers can be read back. (However, internal and command registers cannot be read back.)

To perform a context switch the host must first synchronize with GLINT. This means waiting for outstanding DMA to complete, sending a Sync command and waiting for the sync output data to appear in the output FIFO. After this the registers can be read back.

To read a GLINT register the host reads the same address which would be used for a write, i.e. the base address of the register file plus the offset value for the register.

Note that since internal registers cannot be read back care must be taken when context switching a task which is making use of continue-draw commands. Continue-draw commands rely on the internal registers maintaining previous state. This state will be destroyed by any rendering work done by a new task. To prevent this, continue-draw commands should be performed via DMA since the context switch code has to wait for outstanding DMA to complete. Alternatively, continue-draw commands can be performed in a non-preemptable code segment.

Normally, reading back individual registers should be avoided. The need to synchronize with the chip can adversely affect performance. It is usually more appropriate to keep a software copy of the register which is updated when the actual register is updated.

Byte Swapping

Internally GLINT operates in little-endian mode. However, GLINT is designed to work with both big- and little-endian host processors. Since the PCIBus specification defines that byte ordering is preserved regardless of the size of the transfer operation, GLINT provides facilities to handle byte swapping. Each of the Configuration Space, Control Space, Framebuffer Bypass and Localbuffer Bypass memory areas have both big and little endian mappings available. The mapping to use typically depends on the endian ordering of the host processor.

The Configuration Space may be set by a resistor in the board design to be either little endian or big endian.

The Control Space in PCI address region 0, is 128K bytes in size, and consists of two 64K sized spaces. The first 64K provides little endian access to the control space registers; the second 64K provides big endian access to the same registers.

The framebuffer bypass consists of two PCI address regions: Region 2 and Region 4. Each is independently configurable to by the Aperture0 and Aperture 1 control registers respectively, to one of three modes: no byte swap, 16-bit swap, full byte swap. Note that the 16 bit mode is needed for the following reason. If the framebuffer is configured for 16-bit pixels and the host is big-endian then simply byte swapping is not enough when a 32-bit access is made (to write two pixels). In this case, the required effect is that the bytes are swapped within each 16-bit word, but the two 16-bit halves of the 32-bit word are not swapped. This preserves the order of the pixels that are written as well as the byte ordering within each pixel. The 16 bit mode is referred to as GIB-endian in the PCI Multimedia Design Guide, version 1.0.

The localbuffer bypass consists of two PCI address regions: Region 1 and Region 3. Each is independently configurable to by the Aperture0 and Aperture 1 control registers respectively, to one of two modes: no byte swap, full byte swap.

To save on the size of the address space required for GLINT, board vendors may choose to turn off access to the big endian regions (3 and 4) by the use of resistors on the board.

There is a bit available in the DMAControl control register to enable byte swapping of DMA data. Thus for big-endian hosts, this control bit would normally be enabled.

Red and Blue Swapping

For a given graphics board the RAMDAC and/or API will usually force a given interpretation for true color pixel values. For example, 32-bit pixels will be interpreted as either ARGB (alpha at byte 3, red at byte 2, green at byte 1 and blue at byte 0) or ABGR (blue at byte 2 and red at byte 0). The byte position for red and blue may be important for software which has been written to expect one byte order or the other, in particular when handling image data stored in a file.

GLINT provides two registers to specify the byte positions of blue and red internally. In the Alpha Blend Unit the AlphaBlendMode register contains a 1-bit field called ColorOrder. If this bit is set to zero then the byte ordering is ABGR; if the bit is set to one then the ordering is ARGB. As well as setting this bit in the Alpha Blend unit, it must also be set in the Color Formatting unit. In this unit the DitherMode register contains a Color Order bit with the same interpretation. The order applies to all of the true color pixel formats, regardless of the pixel depth.

Hardware Data Structures

Some of the hardware data structure implementations used in the presently preferred embodiment will now be described in detail. Of course these examples are provided merely to illustrate the presently preferred embodiment in great detail, and do not necessarily delimit any of the claimed inventions.

Localbuffer

The localbuffer holds the per pixel information corresponding to each displayed pixel and any texture maps. The per pixel information held in the localbuffer are Graphic ID (GID), Depth, Stencil and Frame Count Planes (FCP). The possible formats for each of these fields, and their use are described individually in the following sections.

The maximum width of the localbuffer is 48 bits, but this can be reduced by changing the external memory configuration, albeit at the expense of reducing the functionality or dynamic range of one or more of the fields.

The localbuffer memory can be from 16 bits (assuming a depth buffer is always needed) to 48 bits wide in steps of 4 bits. The four fields supported in the localbuffer, their allowed lengths and positions are shown in the following table:

Field	Lengths	Start bit positions
Depth	16, 24, 32	0
Stencil	0, 4, 8	16, 20, 24, 28, 32
FrameCount	0, 4, 8	16, 20, 24, 28, 32, 36, 40
GID	0, 4	16, 20, 24, 28, 32, 36, 40, 44, 48

The order of the fields is as shown with the depth field at the least significant end and GID field at the most significant end. The GID is at the most significant end so that various combinations of the Stencil and FrameCount field widths can be used on a per window basis without the position of the GID fields moving. If the GID field is in a different positions in different windows then the ownership tests become impossible to do.

The GID, FrameCount, Stencil and Depth fields in the localbuffer are converted into the internal format by right justification if they are less than their internal widths, i.e. the unused bits are the most significant bits and they are set to 0.

The format of the localbuffer is specified in two places: the LBReadFormat register and the LBWriteFormat register.

It is still possible to part populate the localbuffer so other combinations of the field widths are possible (i.e. depth field width of 0), but this may give problems if texture maps are to be stored in the localbuffer as well.

Any non-bypass read or write to the localbuffer always reads or writes all 48 bits simultaneously.

GID field

The 4 bit GID field is used for pixel ownership tests to allow per pixel window clipping. Each window using this facility is assigned one of the GID values, and the visible pixels in the window have their GID field set to this value. If the test is enabled the current GID (set to correspond with the current window) is compared with the GID in the localbuffer for each fragment. If they are equal this pixel belongs to the window so the localbuffer and framebuffer at this coordinate may be updated.

Using the GID field for pixel ownership tests is optional and other methods of achieving the same result are:

clip the primitive to the window's boundary (or rectangular tiles which make up the window's area) and render only the visible parts of the primitive

use the scissor test to define the rectangular tiles which make up the window's visible area and render the primitive once per tile (This may be limited to only those tiles which the primitive intersects).

Depth Field

The depth field holds the depth (Z) value associated with a pixel and can be 16, 24 or 32 bits wide.

Stencil Field

The stencil field holds the stencil value associated with a pixel and can be 0, 4 or 8 bits wide.

The width of the stencil buffer is also stored in the StencilMode register and is needed for clamping and masking during the update methods. The stencil compare mask should be set up to exclude any absent bits from the stencil compare operation.

FrameCount Field

The Frame Count Field holds the frame count value associated with a pixel and can be 0, 4 or 8 bits wide. It is used during animation to support a fast clear mechanism to aid the rapid clearing of the depth and/or stencil fields needed at the start of each frame.

In addition to the fast clear mechanism the extent of all updates to the localbuffer and framebuffer can be recorded (MinRegion and MaxRegion registers) and read back (MinHitRegion and MaxHitRegion commands) to give the bounding box of the smallest area to clear. For some applications this will be significantly smaller than the whole window or screen, and hence faster.

The fast clear mechanism provides a method where the cost of clearing the depth and stencil buffers can be amortized over a number of clear operations issued by the application. This works as follows:

The window is divided up into n regions, where n is the range of the frame counter (16 or 256). Every time the application issues a clear command the reference frame counter is incremented (and allowed to roll over if it exceeds its maximum value) and the n^{th} region is cleared only. The clear updates the depth and/or stencil buffers to the new values and the frame count buffer with the reference value. This region is much smaller than the full window and hence takes less time to clear.

When the localbuffer is subsequently read and the frame count is found to be the same as the reference frame count (held in the Window register) the localbuffer data is used directly. However, if the frame count is found to be different from the reference frame count (held in the Window register) the data which would have been written, if the localbuffer had been cleared properly, is substituted for the stale data returned from the read. Any new writes to the localbuffer will set the frame count to the reference value so the next read on this pixel works normally without the substitution. The depth data to substitute is held in the FastClearDepth register and the stencil data to substitute is held in the StencilData register (along with other stencil information).

The fast clear mechanism does not present a total solution as the user can elect to clear just the stencil planes or just the depth planes, or both. The situation where the stencil planes only are 'cleared' using the fast clear method, then some rendering is done and then the depth planes are 'cleared' using the fast clear will leave ambiguous pixels in the localbuffer. The driver software will need to catch this situation, and fall back to using a per pixel write to do the second clear. Which field(s) the frame count plane refers to is recorded in the Window register.

When clear data is substituted for real memory data (during normal rendering operations) the depth write mask and stencil write masks are ignored to mimic the OpenGL operation when a buffer is cleared.

Localbuffer Coordinates

The coordinates generated by the rasterizer are 16 bit 2's complement numbers, and so have the range +32767 to −32768. The rasterizer will produce values in this range, but any which have a negative coordinate, or exceed the screen width or height (as programmed into the ScreenSize register) are discarded.

Coordinates can be defined window relative or screen relative and this is only relevant when the coordinate gets converted to an actual physical address in the localbuffer. In general it is expected that the windowing system will use absolute coordinates and the graphics system will use relative coordinates (to be independent of where the window is).

GUI systems (such as Windows, Windows NT and X) usually have the origin of the coordinate system at the top left corner of the screen but this is not true for all graphics systems. For instance OpenGL uses the bottom left corner as its origin. The WindowOrigin bit in the LBReadMode register selects the top left (0) or bottom left (1) as the origin.

The actual equations used to calculate the localbuffer address to read and write are:

```
Bottom left origin:
  Destination address = LBWindowBase − Y * W + X
  Source address =
    LBWindowBase − Y*W + X + LBSourceOffset
Top left origin:
  Destination address = LBWindowBase + Y * W + X
  Source address =
    LBWindowBase + Y*W + X + LBSourceOffset
```

where:

X is the pixel's X coordinate.

Y is the pixel's Y coordinate.

LBWindowBase holds the base address in the localbuffer of the current window.

LBSourceOffset is normally zero except during a copy operation where data is read from one address and written to another address. The offset between source and destination is held in the LBSourceOffset register.

W is the screen width. Only a subset of widths are supported and these are encoded into the PP0, PP1 and PP2 fields in the LBReadMode register.

These address calculations translate a 2D address into a linear address.

The Screen width is specified as the sum of selected partial products so a full multiply operation is not needed. The partial products are selected by the fields PP0, PP1 and PP2 in the LBReadMode register.

For arbitrary width screens, for instance bitmaps in 'off screen' memory, the next largest width from the table must be chosen. The difference between the table width and the bitmap width will be an unused strip of pixels down the right hand side of the bitmap.

Note that such bitmaps can be copied to the screen only as a series of scanlines rather than as a rectangular block. However, often windowing systems store offscreen bitmaps in rectangular regions which use the same stride as the screen. In this case normal bitblts can be used.

Texture Memory

The localbuffer is used to hold textures in the GLINT 400TX variant. In the GLINT 300SX variant the texture information is supplied by the host.

Framebuffer

The framebuffer is a region of memory where the information produced during rasterization is written prior to being displayed. This information is not restricted to color but can include window control data for LUT management and double buffering.

The framebuffer region can hold up to 32 MBytes and there are very few restrictions on the format and size of the individual buffers which make up the video stream. Typical buffers include:

True color or color index main planes.

Overlay planes.

Underlay planes.

Window ID planes for LUT and double buffer management.

Cursor planes.

Any combination of these planes can be supported up to a maximum of 32 MBytes, but usually it is the video level processing which is the limiting factor. The following text examines the options and choices available from GLINT for rendering, copying, etc. data to these buffers.

To access alternative buffers either the FBPixelOffset register can be loaded, or the base address of the window held in the FBWindow-Base register can be redefined. This is described in more detail below.

Buffer Organization

Each buffer resides at an address in the framebuffer memory map. For rendering and copying operations the actual buffer addresses can be on any pixel boundary. Display hardware will place some restrictions on this as it will need to access the multiple buffers in parallel to mix the buffers together depending on their relative priority, opacity and double buffer selection. For instance, visible buffers (rather than offscreen bitmaps) will typically need to be on a page boundary.

Consider the following highly configured example with a 1280×1024 double buffered system with 32 bit main planes (RGBA), 8 bit overlay and 4 bits of window control information (WID).

Combining the WID and overlay planes in the same 32 bit pixel has the advantage of reducing the amount of data to copy when a window moves, as only two copies are required—one for the main planes and one for the overlay and WID planes.

Note the position of the overlay and WID planes. This was not an arbitrary choice but one imposed by the (presumed) desire to use the color processing capabilities of GLINT (dither and interpolation) in the overlay planes. The conversion of the internal color format to the external one stored in the framebuffer depends on the size and position of the component. Note that GLINT does not support all possible configurations. For example; if the overlay and WID bits were swapped, then eight bit color index starting at bit 4 would be required to render to the overlay, but this is not supported.

Framebuffer Coordinates

Coordinate generation for the framebuffer is similar to that for the localbuffer, but there are some key differences.

As was mentioned before, the coordinates generated by the rasterizer are 16 bit 2's complement numbers. Coordinates can be defined as window relative or screen relative, though this is only relevant when the coordinate gets converted to an actual physical address in the framebuffer. The WindowOrigin bit in the FBReadMode register selects top left (0) or bottom left (1) as the origin for the framebuffer.

The actual equations used to calculate the framebuffer address to read and write are:

```
Bottom left origin:
  Destination address = FBWindowBase - Y*W + X +
    FBPixelOffset
  Source address = FBWindowBase - Y*W + X +
    FBPixelOffset + FBSourceOffset
Top left Origin:
  Destination address = FBWindowBase + Y*W + X +
    FBPixelOffset
  Source address = FBWindowBase + Y*W + X +
    FBPixelOffset + FBSourceOffset
```

These address calculations translate a 2D address into a linear address, so non power of two framebuffer widths (i.e. 1280) are economical in memory.

The width is specified as the sum of selected partial products so a multiply operation is not needed. The partial products are selected by the fields PP0, PP1 and PP2 in the FBReadMode register. This is the same mechanism as is used to set the width of the localbuffer, but the widths may be set independently.

For arbitrary screen sizes, for instance when rendering to 'off screen' memory such as bitmaps the next largest width from the table must be chosen. The difference between the table width and the bitmap width will be an unused strip of pixels down the right hand side of the bitmap.

Note that such bitmaps can be copied to the screen only as a series of scanlines rather than as a rectangular block. However, often windowing systems store offscreen bitmaps in rectangular regions which use the same stride as the screen. In this case normal bitblts can be used.

Color Formats

The contents of the framebuffer can be regarded in two ways:

As a collection of fields of up to 32 bits with no meaning or assumed format as far as GLINT is concerned. Bit planes may be allocated to control cursor, LUT, multi-buffer visibility or priority functions. In this case GLINT will be used to set and clear bit planes quickly but not perform any color processing such as interpolation or dithering. All the color processing can be disabled so that raw reads and writes are done and the only operations are write masking and logical ops. This allows the control planes to be updated and modified as necessary. Obviously this technique can also be used for overlay buffers, etc. providing color processing is not required.

As a collection of one or more color components. All the processing of color components, except for the final write mask and logical ops are done using the internal color format of 8 bits per red, green, blue and alpha color channels. The final stage before write mask and logical ops processing converts the internal color format to that required by the physical configuration of the framebuffer and video logic. The nomenclature n@m means this component is n bits wide and starts at bit position m in the framebuffer. The least significant bit position is 0 and a dash in a column indicates that this component does not exist for this mode. The ColorOrder is specified by a bit in the DitherMode register.

Some important points to note:

The alpha channel is always associated with the RGB color channels rather than being a separate buffer. This allows it to be moved in parallel and to work correctly in multi-buffer updates and double buffering. If the framebuffer is not configured with an alpha channel (e.g. 24 bit framebuffer width with 8:8:8:8 RGB format) then some of the rendering modes which use the retained alpha buffer cannot be used. In these cases the NoAlphaBuffer bit in the AlphaBlendMode register should be set so that an alpha value of 255 is substituted. For the RGB modes where no alpha channel is present (e.g. 3:3:2) then this substitution is done automatically.

For the Front and Back modes the data value is replicated into both buffers.

All writes to the framebuffer try to update all 32 bits irrespective of the color format. This may not matter if the memory planes don't exist, but if they are being used (as overlay planes, for example) then the write masks (FBSoftwareWriteMask or FBHardwareWriteMask) must be set up to protect the alternative planes.

When reading the framebuffer RGBA components are scaled to their internal width of 8 bits, if needed for alpha blending.

CI values are left justified with the unused bits (if any) set to zero [illegible]

nent. The result is replicated into each of the streams G,B and A giving four copies for CI8 and eight copies for CI4.

The 4:4:4:4 Front and Back formats are designed to support 12 bit double buffering with 4 bit Alpha, in a 32 bit system.

The 3:3:2 Front and Back formats are designed to support 8 bit double buffering in a 16 bit system.

The 1:2:1 Front and Back formats are designed to support 4 bit double buffering in an 8 bit system.

It is possible to have a color index buffer at other positions as long as reduced functionality is acceptable. For example a 4 bit CI buffer at bit position 16 can be achieved using write masking and 4:4:4:4 Front format with color interpolation, but dithering is lost.

The format information needs to be stored in two places: the DitherMode register and the AlphaBlendMode register.

			Internal Color Channel			
	Format	Name	R	G	B	A
Color	0	8:8:8:8	8@0	8@8	8@16	8@24
Order:	1	5:5:5:5	5@0	5@5	5@10	5@15
RGB	2	4:4:4:4	4@0	4@4	4@8	4@12
	3	4:4:4:4	4@0	4@8	4@16	4@24
		Front	4@4	4@12	4@20	4@28
	4	4:4:4:4	4@0	4@8	4@16	4@24
		Back	4@4	4@12	4@20	4@28
	5	3:3:2	3@0	3@3	2@6	----
		Front	3@8	3@11	2@14	
	6	3:3:2	3@0	3@3	2@6	----
		Back	3@8	3@11	2@14	
	7	1:2:1	1@0	2@1	1@3	----
		Front	1@4	2@5	1@7	
	8	1:2:1	1@0	2@1	1@3	----
		Back	1@4	2@5	1@7	
Color	0	8:8:8:8	8@16	8@8	8@0	8@24
Order:	1	5:5:5:5	5@10	5@5	5@0	5@15
BGR	2	4:4:4:4	4@8	4@4	4@0	4@12
	3	4:4:4:4	4@16	4@8	4@0	4@24
		Front	4@20	4@12	4@4	4@28
	4	4:4:4:4	4@16	4@8	4@0	4@24
		Back	4@20	4@12	4@4	4@28
	5	3:3:2	3@5	3@2	2@0	----
		Front	3@13	3@10	2@8	
	6	3:3:2	3@5	3@2	2@0	----
		Back	3@13	3@10	2@8	
	7	1:2:1	1@3	2@1	1@0	----
		Front	1@7	2@5	1@4	
	8	1:2:1	1@3	2@1	1@0	----
		Back	1@7	2@5	1@4	
CI	14	CI8	8@0	0	0	0
	15	CI4	4@0	0	0	0

Overlays and Underlays

In a GUI system there are two possible relationships between the overlay planes (or underlay) and the main planes.

The overlay planes are fixed to the main planes, so that if the window is moved then both the data in the main planes and overlay planes move together.

The overlay planes are not fixed to the main planes but floating, so that moving a window only moves the associated main or overlay planes.

In the fixed case both planes can share the same GID. The pixel offset is used to redirect the reads and writes between the main planes and the overlay (underlay) buffer. The pixel ownership tests using the GID field in the localbuffer work as expected.

In the floating case different GIDs are the best choice, because the same GID planes in the localbuffer can not be used for pixel ownership tests. The alternatives are not to use the GID based pixel ownership tests for one of the buffers but rely on the scissor clipping, or to install a second set of GID planes so each buffer has it's own set. GLINT allows either approach.

If rendering operations to the main and overlay planes both need the depth or stencil buffers, and the windows in each overlap then each buffer will need its own exclusive depth and/or stencil buffers. This is easily achieved with GLINT by assigning different regions in the localbuffer to each of the buffers. Typically this would double the localbuffer memory requirements.

One scenario where the above two considerations do not cause problems, is when the overlay planes are used exclusively by the GUI system, and the main planes are used for the 3D graphics.

VRAM Modes

High performance systems will typically use VRAM for the framebuffer and the extended functionality of VRAM over DRAM can be used to enhance performance for many rendering tasks.

Hardware Write Masks.

These allow write masking in the framebuffer without incurring a performance penalty. If hardware write masks are not available, GLINT must be programmed to read the memory, merge the value with the new value using the write mask, and write it back.

To use hardware write masking, the required write mask is written to the FBHardwareWriteMask register, the FBSoftwareWriteMask register should be set to all 1's, and the number of framebuffer reads is set to 0 (for normal rendering). This is achieved by clearing the ReadSource and ReadDestination enables in the FBReadMode register.

To use software write masking, the required write mask is written to the FBSoftwareWriteMask register and the number of framebuffer reads is set to 1 (for normal rendering). This is achieved by setting the ReadDestination enable in the FBReadMode register.

Block Writes Block writes cause consecutive pixels in the framebuffer to be written simultaneously. This is useful when filling large areas but does have some restrictions:

No pixel level clipping is available;

No depth or stencil testing can be done;

All the pixels must be written with the same value so no color interpolation, blending, dithering or logical ops can be done; and

The area is defined in screen relative coordinates.

Block writes are not restricted to rectangular areas and can be used for any trapezoid. Hardware write masking is available during block writes.

The following registers need to be set up before block fills can be used:

FBBlockColor register with the value to write to each pixel; and

FBWriteMode register with the block width field.

Sending a Render command with the PrimitiveType field set to "trapezoid" and the FastFillEnable and FastFillIncrement fields set up will then cause block filling of the area. Note that during a block fill of a trapezoid any inappropriate state is ignored so even if color interpolation, depth testing and logical ops, for example, are enabled they have no effect.

The block sizes supported are 8, 16 and 32 pixels. GLINT takes care of filling any partial blocks at the end of spans.

Graphics Programming

GLINT provides a rich variety of operations for 2D and 3D graphics [illegible]

The Graphics Pipeline

This section describes each of the units in the graphics Pipeline. FIG. 2C shows a schematic of the pipeline. In this diagram, the localbuffer contains the pixel ownership values (known as Graphic IDs), the FrameCount Planes (FCP), Depth (Z) and Stencil buffer. The framebuffer contains the Red, Green, Blue and Alpha bitplanes. The operations in the Pipeline include:

Rasterizer scan converts the given primitive into a series of fragments for processing by the rest of the pipeline.

Scissor Test clips out fragments that lie outside the bounds of a user defined scissor rectangle and also performs screen clipping to stop illegal access outside the screen memory.

Stipple Test masks out certain fragments according to a specified pattern. Line and area stipples are available.

Color DDA is responsible for generating the color information (True Color RGBA or Color Index(CI)) associated with a fragment.

Texture is concerned with mapping a portion of a specified image (texture) onto a fragment. The process involves filtering to calculate the texture color, and application which applies the texture color to the fragment color.

Fog blends a fog color with a fragment's color according to a given fog factor. Fogging is used for depth cuing images and to simulate atmospheric fogging.

Antialias Application combines the incoming fragment's alpha value with its coverage value when anti aliasing is enabled.

Alpha Test conditionally discards a fragment based on the outcome of a comparison between the fragments alpha value and a reference alpha value.

Pixel Ownership is concerned with ensuring that the location in the framebuffer for the current fragment is owned by the current visual. Comparison occurs between the given fragment and the Graphic ID value in the localbuffer, at the corresponding location, to determine whether the fragment should be discarded.

Stencil Test conditionally discards a fragment based on the outcome of a test between the given fragment and the value in the stencil buffer at the corresponding location. The stencil buffer is updated dependent on the result of the stencil test and the depth test.

Depth Test conditionally discards a fragment based on the outcome of a test between the depth value for the given fragment and the value in the depth buffer at the corresponding location. The result of the depth test can be used to control the updating of the stencil buffer.

Alpha Blending combines the incoming fragment's color with the color in the framebuffer at the corresponding location.

Color Formatting converts the fragment's color into the format in which the color information is stored in the framebuffer.

This may optionally involve dithering.

The Pipeline structure of GLINT is very efficient at processing fragments, for example, texture mapping calculations are not actually performed on fragments that get clipped out by scissor testing. This approach saves substantial computational effort. The pipelined nature does however mean that when programming GLINT one should be aware of what all the pipeline stages are doing at any time. For example, many operations require both a read and/or write to the localbuffer and framebuffer; in this case it is not sufficient to set a logical operation to XOR and enable logical operations, but it is also necessary to enable the [illegible] data from/to the framebuffer.

A Gouraud Shaded Triangle

We may now revisit the "day in the life of a triangle" example given above, and review the actions taken in greater detail. Again, the primitive being rendered will be a Gouraud shaded, depth buffered triangle. For this example assume that the triangle is to be drawn into a window which has its colormap set for RGB as opposed to color index operation. This means that all three color components: red, green and blue, must be handled. Also, assume the coordinate origin is bottom left of the window and drawing will be from top to bottom. GLINT can draw from top to bottom or bottom to top.

Consider a triangle with vertices, v_1, v_2 and v_3 where each vertex comprises X, Y and Z coordinates. Each vertex has a different color made up of red, green and blue (R, G and B) components. The alpha component will be omitted for this example.

Initialization

GLINT requires many of its registers to be initialized in a particular way, regardless of what is to be drawn, for instance, the screen size and appropriate clipping must be set up. Normally this only needs to be done once and for clarity this example assumes that all initialization has already been done.

Other state will change occasionally, though not usually on a per primitive basis, for instance enabling Gouraud shading and depth buffering.

Dominant and Subordinate Sides of a Triangle

As shown in FIG. 4A, the dominant side of a triangle is that with the greatest range of Y values. The choice of dominant side is optional when the triangle is either flat bottomed or flat topped.

GLINT always draws triangles starting from the dominant edge towards the subordinate edges. This simplifies the calculation of set up parameters as will be seen below.

These values allow the color of each fragment in the triangle to be determined by linear interpolation. For example, the red component color value of a fragment at XN, Ym could be calculated by:

adding $dRdy_{13}$, for each scanline between Y_1 and Y_m, to R_1.

then adding dRdx for each fragment along scanline Y_m from the left edge to X_m.

The example chosen has the 'knee,' i.e. vertex 2, on the right hand side, and drawing is from left to right. If the knee were on the left side (or drawing was from right to left), then the Y deltas for both the subordinate sides would be needed to interpolate the start values for each color component (and the depth value) on each scanline. For this reason GLINT always draws triangles starting from the dominant edge and towards the subordinate edges. For the example triangle, this means left to right.

Register Set Up for Color Interpolation

For the example triangle, the GLINT registers must be set as follows, for color interpolation. Note that the format for color values is 24 bit, fixed point 2's complement.

```
// Load the color start and delta values to draw
// a triangle
RStart (R1)
GStart (G1)
BStart (B1)
dRdyDom (dRdy13)        // To walk up the dominant edge
dGdyDom (dGdy13)
dBdyDom (dBdy13)
dRdx (dRdx)             // To walk along the scanline
```

-continued

```
dGdx (dGdx)
dBdx (dBdx)
```

Calculating Depth Gradient Values

To draw from left to right and top to bottom, the depth gradients or deltas) required for interpolation are:

$$dZdy_{13} = \frac{Z_3 - Z_1}{Y_3 - Y_1}$$

And from the plane equation:

$$dZdx = \left\{ (Z_1 - Z_3)\frac{(Y_2 - Y_3)}{c} \right\} - \left\{ (Z_2 - Z_3)\frac{(Y_1 - Y_3)}{c} \right\}$$

where

$$c = \{(X_1 - X_3)(Y_2 - Y_3) - (X_2 - X_3)(Y_1 - Y_3)\}$$

The divisor, shown here as c, is the same as for color gradient values. The two deltas $dZdy1_{13}$ and dZdx allow the Z value of each fragment in the triangle to be determined by linear interpolation, just as for the color interpolation.

Register Set Up for Depth Testing

Internally GLINT uses fixed point arithmetic. Each depth value must be converted into a 2's complement 32.16 bit fixed point number and then loaded into the appropriate pair of 32 bit registers. The 'Upper' or 'U' registers store the integer portion, whilst the 'Lower' or 'L' registers store the 16 fractional bits, left justified and zero filled.

For the example triangle, GLINT would need its registers set up as follows:

```
// Load the depth start and delta values
// to draw a triangle
ZStartU (Z1_MS)
ZStartL (Z1_LS)
dZdyDomU (dZdy13_MS)
dZdyDomL (dZdy13_LS)
dZdxU (dZdx_MS)
dZdxL (dZdx_LS)
```

Calculating the Slopes for each Side

GLINT draws filled shapes such as triangles as a series of spans with one span per scanline. Therefore it needs to know the start and end X coordinate of each span. These are determined by 'edge walking'. This process involves adding one delta value to the previous span's start X coordinate and another delta value to the previous span's end x coordinate to determine the X coordinates of the new span. These delta values are in effect the slopes of the triangle sides. To draw from left to right and top to bottom, the slopes of the three sides are calculated as:

$$dX_{13} = \frac{X_3 - X_1}{Y_3 - Y_1}$$

$$dX_{12} = \frac{X_2 - X_1}{Y_2 - Y_1}$$

$$dX_{23} = \frac{X_3 - X_2}{Y_3 - Y_2}$$

This triangle will be drawn in two parts, top down to the 'knee' (i.e. vertex 2), and then from there to the bottom. The dominant side is the left side so for the top half:

dXDom=dX_{13}

dXSub=dX_{12}

The start X,Y, the number of scanlines, and the above deltas give GLINT enough information to edge walk the top half of the triangle. However, to indicate that this is not a flat topped triangle (GLINT is designed to rasterize screen aligned trapezoids and flat topped triangles), the same start position in terms of X must be given twice as StartXDom and StartXSub.

To edge walk the lower half of the triangle, selected additional information is required. The slope of the dominant edge remains unchanged, but the subordinate edge slope needs to be set to:

dXSub=dX_{23}

Also the number of scanlines to be covered from Y_2 to Y_3 needs to be given. Finally to avoid any rounding errors accumulated in edge walking to X_2 (which can lead to pixel errors), StartXSub must be set to X_2.

Rasterizer Mode

The GLINT rasterizer has a number of modes which have effect from the time they are set until they are modified and can thus affect many primitives. In the case of the Gouraud shaded triangle the default value for these modes are suitable.

Subpixel Correction

GLINT can perform subpixel correction of all interpolated values when rendering aliased trapezoids. This correction ensures that any parameter (color/depth/texture/fog) is correctly sampled at the center of a fragment. Subpixel correction will generally always be enabled when rendering any trapezoid which is smooth shaded, textured, fogged or depth buffered. Control of subpixel correction is in the Render command register described in the next section, and is selectable on a per primitive basis.

Rasterization

GLINT is almost ready to draw the triangle. Setting up the registers as described here and sending the Render command will cause the top half of the example triangle to be drawn.

For drawing the example triangle, all the bit fields within the Render command should be set to 0 except the PrimitiveType which should be set to trapezoid and the SubPixelCorrectionEnable bit which should be set to TRUE.

```
// Draw triangle with knee
// Set deltas
StartXDom (X1<<16)  //  Converted to 16.16 fixed
point
dXDom ((X3 - X1)<<16)/(Y3 - Y1))
StartXSub (X1<<16)
dXSub (((X2 - X1)<<16)/(Y2 - Y1))
StartY (Y1<<16)
dY (1<<16)
Count (Y1 - Y2)
// Set the render command mode
render.PrimitiveType = GLINT_TRAPEZOID_PRIMITIVE
render.SubPixelCorrectionEnable = TRUE
// Draw the top half of the triangle
Render(render)
```

After the Render command has been issued, the registers in GLINT can immediately be altered to draw the lower half of the triangle. Note that only two registers need be loaded and the command ContinueNewSub sent. Once GLINT has received ContinueNewSub, drawing of this sub-triangle will begin.

```
// Setup the delta and start for the new edge
StartXSub (X2<<16)
dXSub (((X3 - X2)<<16)/(Y3 - Y2))
// Draw sub-triangle
ContinueNewSub (Y2 - Y3)   // Draw lower half
```

Rasterizer Unit

The rasterizer decomposes a given primitive into a series of fragments for processing by the rest of the Pipeline.

GLINT can directly rasterize:

aliased screen aligned trapezoids

aliased single pixel wide lines

aliased single pixel points

antialiased screen aligned trapezoids

antialiased circular points

All other primitives are treated as one or more of the above. for example an antialiased line is drawn as a series of antialiased trapezoids.

Trapezoids GLINT's basic area primitives are screen aligned trapezoids. These are characterized by having top and bottom edges parallel to the X axis. The side edges may be vertical (a rectangle), but in general will be diagonal. The top or bottom edges can degenerate into points in which case we are left with either flat topped or flat bottomed triangles. Any polygon can be decomposed into screen aligned trapezoids or triangles. Usually, polygons are decomposed into triangles because the interpolation of values over non-triangular polygons is ill defined. The rasterizer does handle flat topped and flat bottomed 'bow tie' polygons which are a special case of screen aligned trapezoids.

To render a triangle, the approach adopted to determine which fragments are to be drawn is known as 'edge walking'. Suppose the aliased triangle shown in FIG. 4A was to be rendered from top to bottom and the origin was bottom left of the window. Starting at (X1, Y1) then decrementing Y and using the slope equations for edges 1–2 and 1–3, the intersection of each edge on each scanline can be calculated. This results in a span of fragments per scanline for the top trapezoid. The same method can be used for the bottom trapezoid using slopes 2–3 and 1–3.

It is usually required that adjacent triangles or polygons which share an edge or vertex are drawn such that pixels which make up the edge or vertex get drawn exactly once. This may be achieved by omitting the pixels down the left or the right sides and the pixels along the top or lower sides. GLINT has adopted the convention of omitting the pixels down the right hand edge. Control of whether the pixels along the top or lower sides are omitted depends on the start Y value and the number of scanlines to be covered. With the example, if StartY =Y1 and the number of scanlines is set to Y1–Y2, the lower edge of the top half of the triangle will be excluded. This excluded edge will get drawn as part of the lower half of the triangle.

To minimize delta calculations, triangles may be scan converted from left to right or from right to left. The direction depends on the dominant edge, that is the edge which has the maximum range of Y values. Rendering always proceeds from the dominant edge towards the relevant subordinate edge. In the example above, the dominant edge is 1–3 so rendering will be from right to left.

The sequence of actions required to render a triangle (with a 'knee') is:

Load the edge parameters and derivatives for the dominant edge and the first subordinate edges in the first triangle.

Send the Render command. This starts the scan conversion of the first triangle, working from the dominant edge. This means that for triangles where the knee is on the left we are scanning right to left, and vice versa for triangles where the knee is on the right.

Load the edge parameters and derivatives for the remaining subordinate edge in the second triangle.

Send the ContinueNewSub command. This starts the scan conversion of the second triangle.

Pseudocode for the above example is:

```
// Set the rasterizer mode to the default
RasterizerMode (0)
// Setup the start values and the deltas.
// Note that the X and Y coordinates are converted
// to 16.16 format
StartXDom (X1<<16)
dXDom (((X3- X1)<<16)/(Y3 - Y1))
StartXSub (X1<<16)
dXSub (((X2- X1)<<16)/(Y2 - Y1))
StartY (Y1<<16)
dY (-1<16) // Down the screen
Count (Y1 - Y2)
// Set the render mode to aliased primitive with
// subpixel correction.
render.PrimitiveType = GLINT_TRAPEZOID_PRIMITIVE
render.SubpixelCorrectionEnable = GLINT_TRUE
render.AntialiasEnable = GLINT_DISABLE
// Draw top half of the triangle
Render(render)
// Set the start and delta for the second half of
// the triangle.
StartXSub (X2<<16)
dXSub (((X3- X2)<<16)/(Y3 - Y2))
// Draw lower half of triangle
ContinueNewSub (aba(Y2 - Y3))
```

After the Render command has been sent, the registers in GLINT can immediately be altered to draw the second half of the triangle. For this, note that only two registers need be loaded and the command ContinueNewSub be sent. Once drawing of the first triangle is complete and GLINT has received the ContinueNewSub command, drawing of this sub-triangle will start. The ContinueNewSub command register is loaded with the remaining number of scanlines to be rendered.

Lines

Single pixel wide aliased lines are drawn using a DDA algorithm, so all GLINT needs by way of input data is StartX, StartY, dX, dY and length.

For polylines, a ContinueNewLine command (analogous to the Continue command used at the knee of a triangle) is used at vertices.

When a Continue command is issued some error will be propagated along the line. To minimize this, a choice of actions are available as to how the DDA units are restarted on the receipt of a Continue command. It is recommended that for OpenGL rendering the ContinueNewLine command is not used and individual segments are rendered.

Antialiased lines, of any width, are rendered as antialiased screen-aligned trapezoids.

Points

GLINT supports a single pixel aliased point primitive. For points larger than one pixel trapezoids should be used. In this case the PrimitiveType field in the Render command should be set to equal GLINT_POINT_PRIMITIVE.

Anti aliasing

GLINT uses a subpixel point sampling algorithm to antialias primitives. GLINT can directly rasterize antialiased trapezoids and points. Other primitives are composed from these basic primitives.

The rasterizer associates a coverage value with each fragment produced when antialiasing. This value represents the percentage coverage of the pixel by the fragment. GLINT supports two levels of antialiasing quality:

normal, which represents 4×4 pixel subsampling

high, which represents 8×8 pixel subsampling.

Selection between these two is made by the AntialiasingQuality bit within the Render command register.

When rendering antialiased primitives with GLINT the FlushSpan command is used to terminate rendering of a primitive. This is due to the nature of GLINT antialiasing. When a primitive is rendered which does not happen to complete on a scanline boundary, GLINT retains antialiasing information about the last sub-scanline(s) it has processed, but does not generate fragments for them unless a FlushSpan command is received. The commands ContinueNewSub, ContinueNewDom or Continue can then be used, as appropriate, to maintain continuity between adjacent trapezoids. This allows complex antialiased primitives to be built up from simple trapezoids or points.

To illustrate this consider using screen aligned trapezoids to render an antialiased line. The line will in general consist of three screen aligned trapezoids as shown in FIG. 4B. This FIG. illustrates the sequence of rendering an Antialiased Line primitive. Note that the line has finite width.

The procedure to render the line is as follows:

```
// Setup the blend and coverage application units
// as appropriate - not shown
// In this example only the edge deltas are shown
// loaded into registers for clarity. In reality
// start X and Y values are required
// Render Trapezoid A
dY(1<<16)
dXDom(dXDom1<<16)
dXSub(dXSub1<<16)
Count(count1)
render.PrimitiveType = GLINT__TRAPEZOID
render.AntialiasEnable = GLINT__TRUE
render.AntialiasQuality = GLINT__MIN__ANTIALIAS
render.CoverageEnable = GLINT__TRUE
Render(render)
// Render Trapezoid B
dXSub(dXSub2<<16)
ContinueNewSub(count2)
// Render Trapezoid C
dXDom(dXDom2<<16)
ContinueNewDom(count3)
// Now we have finished the primitive flush out
// the last scanline
FlushSpan( )
```

Note that when rendering antialiased primitives, any count values should be given in subscanlines, for example if the quality is 4×4 then any scanline count must be multiplied by 4 to convert it into a subscanline count. Similarly, any delta value must be divided by 4.

When rendering, AntialiasEnable must be set in the Antialias-Mode register to scale the fragments color by the coverage value. An appropriate blending function should also be enabled.

Note, when rendering antialiased bow-ties, the coverage value on the cross-over scanline may be incorrect.

GLINT can render small antialiased points. Antialiased points are treated as circles, with the coverage of the boundary fragments ranging from 0% to 100%. GLINT supports:

point radii of 0.5 to 16.0 in steps of 0.25 for 4×4 antialiasing

point radii of 0.25 to 8.0 in steps of 0.125 for 8×8 antialiasing

To scan convert an antialiased point as a circle, GLINT traverses the boundary in sub scanline steps to calculate the coverage value. For this, the sub-scanline intersections are calculated incrementally using a small table. The table holds the change in X for a step in Y. Symmetry is used so the table only holds the delta values for one quadrant.

StartXDom, StartXSub and StartY are set to the top or bottom of the circle and dY set to the subscanline step. In the case of an even diameter, the last of the required entries in the table is set to zero.

Since the table is configurable, point shapes other than circles can be rendered. Also if the StartXDom and StartXSub values are not coincident then horizontal thick lines with rounded ends, can be rendered.

Block Write Operation

GLINT supports VRAM block writes with block sizes of 8, 16 and 32 pixels. The block write method does have some restrictions: None of the per pixel clipping, stipple, or fragment operations are available with the exception of write masks. One subtle restriction is that the block coordinates will be interpreted as screen relative and not window relative when the pixel mask is calculated in the Framebuffer Units.

Any screen aligned trapezoid can be filled using block writes, not just rectangles.

The use of block writes is enabled by setting the FastFillEnable and FastFillIncrement fields in the Render command register. The framebuffer write unit must also be configured.

Note only the Rasterizer, Framebuffer Read and Framebuffer Write units are involved in block filling. The other units will ignore block write fragments, so it is not necessary to disable them.

Sub Pixel Precision and Correction

As the rasterizer has 16 bits of fraction precision, and the screen width used is typically less than 2^{16} wide a number of bits called subpixel precision bits, are available. Consider a screen width of 4096 pixels. This figure gives a subpixel precision of 4 bits ($4096=2^{12}$). The extra bits are required for a number of reasons:

antialiasing (where vertex start positions can be supplied to subpixel precision)

when using an accumulation buffer (where scans are rendered multiple times with jittered input vertices)

for correct interpolation of parameters to give high quality shading as described below

GLINT supports subpixel correction of interpolated values when rendering aliased trapezoids. Subpixel correction ensures that all interpolated parameters associated with a fragment (color, depth, fog, texture) are correctly sampled at the fragment's center. This correction is required to ensure consistent shading of objects made from many primitives. It should generally be enabled for all aliased rendering which uses interpolated parameters.

Subpixel correction is not applied to antialiased primitives.

Bitmaps

A Bitmap primitive is a trapezoid or line of ones and zeros which control which fragments are generated by the rasterizer. Only fragments where the corresponding Bitmap bit is set are submitted for drawing. The normal use for this is in drawing characters, although the mechanism is available for all primitives. The Bitmap data is packed contiguously into 32 bit words so that rows are packed adjacent to each other. Bits in the mask word are by default used from the least significant end towards the most significant end and are applied to [illegible]

The rasterizer scans through the bits in each word of the Bitmap data and increments the X,Y coordinates to trace out the rectangle of the given width and height. By default, any set bits (1) in the Bitmap cause a fragment to be generated, any reset bits (0) cause the fragment to be rejected.

The selection of bits from the BitMaskPattern register can be mirrored, that is, the pattern is traversed from MSB to LSB rather than LSB to MSB. Also, the sense of the test can be reversed such that a set bit causes a fragment to be rejected and vice versa. This control is found in the RasterizerMode register.

When one Bitmap word has been exhausted and pixels in the rectangle still remain then rasterization is suspended until the next write to the BitMaskPattern register. Any unused bits in the last Bitmap word are discarded.

Image Copy/Upload/Download

GLINT supports three "pixel rectangle" operations: copy, upload and download. These can apply to the Depth or Stencil Buffers (held within the localbuffer) or the framebuffer.

It should be emphasized that the GLINT copy operation moves RAW blocks of data around buffers. To zoom or re-format data, in the presently preferred embodiment, external software must upload the data, process it and then download it again.

To copy a rectangular area, the rasterizer would be configured to render the destination rectangle, thus generating fragments for the area to be copied. GLINT copy works by adding a linear offset to the destination fragment's address to find the source fragment's address.

Note that the offset is independent of the origin of the buffer or window, as it is added to the destination address. Care must be taken when the source and destination overlap to choose the source scanning direction so that the overlapping area is not overwritten before it has been moved. This may be done by swapping the values written to the StartXDom and StartXSub, or by changing the sign of dY and setting StartY to be the opposite side of the rectangle.

Localbuffer copy operations are correctly tested for pixel ownership. Note that this implies two reads of the localbuffer, one to collect the source data, and one to get the destination GID for the pixel ownership test.

GLINT buffer upload/downloads are very similar to copies in that the region of interest is generated in the rasterizer. However, the localbuffer and framebuffer are generally configured to read or to write only, rather than both read and write. The exception is that an image load may use pixel ownership tests, in which case the localbuffer destination read must be enabled.

Units which can generate fragment values, the color DDA unit for example, should generally be disabled for any copy/upload/download operations.

Warning: During image upload, all the returned fragments must be read from the Host Out FIFO, otherwise the GLINT pipeline will stall. In addition it is strongly recommended that any units which can discard fragments (for instance the following tests: bitmask, alpha, user scissor, screen scissor, stipple, pixel ownership, depth, stencil), are disabled otherwise a shortfall in pixels returned may occur, also leading to deadlock.

Note that because the area of interest in copy/upload/download operations is defined by the rasterizer, it is not limited to rectangular regions.

Color formatting can be used when performing image copies, uploads and downloads. This allows data to be formatted from, or to, any of the supported GLINT color formats.

Rasterizer Mode

A number of long-term modes can be set using the Rasterizer-Mode register, these are:

Mirror BitMask: This is a single bit flag which specifies the direction bits are checked in the BitMask register. If the bit is reset, the direction is from least significant to most significant (bit **0** to bit **31**), if the bit is set, it is from most significant to least significant (from bit **31** to bit **0**).

Invert BitMask: This is a single bit which controls the sense of the accept/reject test when using a Bitmask. If the bit is reset then when the BitMask bit is set the fragment is accepted and when it is reset the fragment is rejected. When the bit is set the sense of the test is reversed.

Fraction Adjust: These 2 bits control the action taken by the rasterizer on receiving a ContinueNewLine command. As GLINT uses a DDA algorithm to render lines, an error accumulates in the DDA value. GLINT provides for greater control of the error by doing one of the following:

leaving the DDA running, which means errors will be propagated along a line.

or setting the fraction bits to either zero, a half or almost a half (0×7FFF).

Bias Coordinates: Only the integer portion of the values in the DDAs are used to generate fragment addresses. Often the actual action required is a rounding of values, this can be achieved by setting the bias coordinate bit to true which will automatically add almost a half (0×7FFF) to all input coordinates.

Rasterizer Unit Registers

Real coordinates with fractional parts are provided to the rasterizer in 2's complement 16 bit integer, 16 bit fraction format. The following Table lists the command registers which control the rasterizer unit:

Register Name	Description
Render	Starts the rasterization process
ContinueNewDom	Allows the rasterization to continue with a new dominant edge. The dominant edge DDA is reloaded with the new parameters. The subordinate edge is carried on from the previous trapezoid. This allows any convex polygon to be broken down into a collection of trapezoids, with continuity maintained across boundaries. The data field holds the number of scanlines (or sub scanlines) to fill. Note this count does not get loaded into the Count register.
ContinueNewSub	Allows the rasterization to continue with a new subordinate edge. The subordinate DDA is reloaded with the new

-continued

Register Name	Description
	parameters. The dominant edge is carried on from the previous trapezoid. This is useful when scan converting triangles with a 'knee' (i.e. two subordinate edges). The data field holds the number of scanlines (or sub scanlines) to fill. Note this count does not get loaded into the Count register.
Continue	Allows the rasterization to continue after new delta value(s) have been loaded, but does not cause either of the trapezoid's edge DDAs to be reloaded. The data field holds the number of scanlines (or sub scanlines) to fill. Note this count does not get loaded into the Count register.
ContinueNewLine	Allows the rasterization to continue for the next segment in a polyline. The XY position is carried on from the previous line, but the fraction bits in the DDAs can be: kept, set to zero, half, or nearly one half, under control of the RasterizerMode. The data field holds the number of scanlines to fill. Note this count does not get loaded into the Count register. The use of ContinueNewLine is not recommended for OpenGL because the DDA units will start with a slight error as compared with the value they would have been loaded with for the second and subsequent segments.
FlushSpan	Used when antialiasing to force the last span out when not all sub spans may be defined.

The following Table shows the control registers of the rasterizer, in the presently preferred embodiment:

RasterizerMode	Defines the long term mode of operation of the rasterizer.
StartXDom	Initial X value for the dominant edge in trapezoid filling, or initial X value in line drawing.
dXDom	Value added when moving from one scanline (or sub scanline) to the next for the dominant edge in trapezoid filling. Also holds the change in X when plotting lines so for Y major lines this will be some fraction (dx/dy), otherwise it is normally ± 1.0, depending on the required scanning direction.
StartXSub	Initial X value for the subordinate edge.
dXSub	Value added when moving from one scanline (or sub scanline) to the next for the subordinate edge in trapezoid filling.
StartY	Initial scanline (or sub scanline) in trapezoid filling, or initial Y position for line drawing.
dY	Value added to Y to move from one scanline to the next. For X major lines this will be some fraction (dy/dx), otherwise it is normally ± 1.0, depending on the required scanning direction.
Count	Number of pixels in a line. Number of scanlines in a trapezoid. Number of sub scanlines in an antialiased trapezoid. Diameter of a point in sub scanlines.
BitMaskPattern	Value used to control the BitMask stipple operation (if enabled).
PointTable0 PointTable1 PointTable2 PointTable3	Antialias point data table. There are 4 words in the table and the register tag is decoded to select a word.

For efficiency, the Render command register has a number of bit fields that can be set or cleared per render operation, and which qualify other state information within GLINT. These bits are AreaStippleEnable, LineStippleEnable, ResetLineStipple, TextureEnable FogEnable, CoverageEnable and SubpixelCorrection.

One use of this feature can occur when a window is cleared to a background color. For normal 3D primitives, stippling and fog operations may have been enabled, but these are to be ignored for window clears. Initially the FogMode, AreaStippleMode and LineStippleMode registers are enabled through the UnitEnable bits. Now bits need only be set or cleared within the Render command to achieve the required result, removing the need for the FogMode, AreaStippleMode and LineStippleMode registers to be loaded for every render operation.

The bitfields of the Render command register, in the presently preferred embodiment, are described in the

Bit	Name	Description
0	Area-Stipple-Enable	This bit, when set, enables area stippling of the fragments produced during rasterization. Note that area stipple in the Stipple Unit must be enabled as well for stippling to occur. When this bit is reset no area stippling occurs irrespective of the setting of the area stipple enable bit in the Stipple Unit. This bit is useful to temporarily force no area stippling for this primitive.
1	Line-Stipple-Enable	This bit, when set, enables line stippling of the fragments produced during rasterization in the Stipple Unit. Note that line stipple in the Stipple Unit must be enabled as well for stippling to occur. When this bit is reset no line stippling occurs irrespective of the setting if the line stipple enable bit in the Stipple Unit. This bit is useful to temporarily force no line stippling for this primitive.
2	Reset-Line-Stipple	This bit, when set, causes the line stipple counters in the Stipple Unit to be reset to zero, and would typically be used for the first segment in a polyline. This action is also qualified by the LineStippleEnable bit and also the stipple enable bits in the Stipple Unit. When this bit is reset the stipple counters carry on from where they left off (if line stippling is enabled)
3	FastFillEnable	This bit, when set, causes fast block filling of primitives. When this bit is reset the normal rasterization process occurs.
4, 5	Fast-Fill-Increment	This two bit field selects the block size the framebuffer supports. The sizes supported and the corresponding codes are: 0 = 8 pixels 1 = 16 pixels 2 = 32 pixels
6, 7	Primitive-Type	This two bit field selects the primitive type to rasterize. The primitives are: 0 = Line 1 = Trapezoid 2 = Point
8	Antialias-Enable	This bit, when set, causes the generation of sub scanline data and the coverage value to be calculated for each fragment. The number of sub pixel samples to use is controlled by the AntialiasingQuality bit. When this bit is reset normal rasterization occurs.
9	Antialiasing-Quality	This bit, when set, sets the sub pixel resolution to be 8×8 When this bit is reset the sub pixel resolution is 4×4.
10	UsePoint-Table	When this bit and the AntialiasingEnable are set, the dx values used to remove from one scanline to the next are derived from the Point Table.
11	SyncOn-BitMask	This bit, when set, causes a number of actions: - The least significant bit or most significant bit (depending on the MirrorBitMask bit) in the Bit Mask register is extracted and optionally inverted (controlled by the InvertMask bit). If this bit is 0 then the corresponding fragment is culled from being drawn. After every fragment the Bit Mask register is rotated by one bit. If all the bits in the Bit Mask register have been used then rasterization is suspended until a new BitMaskPattern is received. If any other register is written while the rasterization is suspended then the rasterization is aborted. The register write which caused the abort is then processed as normal. Note the behavior is slightly different when the SyncOnHostData bit is set to prevent a deadlock from occurring. In this case the rasterization doesn't suspend when all the bits have been used and if new BitMaskPattern data words are not received in a timely manner then the subsequent fragments will just reuse the bitmask.
12	SyncOn HostData	When this bit is set a fragment is produced only when one of the following registers has been written by the host: Depth, FBColor, Stencil or Color. If SyncOnBitMask is reset, then if any register other than one of these four is written to, the rasterization is aborted. If SyncOnBitMask is set, then if any register other than one of these four, or BitMaskPattern, is written to, the rasterization is aborted. The register write which caused the abort is then processed as normal. Writing to the BitMaskPattern register doesn't cause any fragments to be generated, but just updates the BitMask register.
13	TextureEnable	This bit, when set, enables texturing of the fragments produced during rasterization. Note that the Texture Units must be suitably enabled as well for any texturing to occur.

-continued

Bit	Name	Description
		When this bit is reset no texturing occurs irrespective of the setting of the Texture Unit controls. This bit is useful to temporarily force no texturing for this primitive.
14	Fog-Enable	This bit, When set, enables fogging of the fragments produced during rasterization. Note that the Fog Unit must be suitably enabled as well for any fogging to occur. When this bit is reset no fogging occurs irrespective of the setting of the Fog Unit controls. This bit is useful to temporarily force no fogging for this primitive.
15	Coverage-Enable	This bit, when set, enables the coverage value produced as part of the antialiasing to weight the alpha value in the alpha test unit. Note that this unit must be suitably enabled as well. When this bit is reset no coverage application occurs irrespective of the setting of the AntialiasMode in the Alpha Test unit.
16	SubPixel-Correction Enable	This bit, when set enables the sub pixel correction of the color, depth, fog and texture values at the start of a scanline. When this bit is reset no correction is done at the start of a scanline. Sub pixel corrections are only applied to aliased trapezoids.

A number of long-term rasterizer modes are stored in the RasterizerMode register as shown below:

Bit	Name	Description
0	Mirror-BitMask	When this bit is set the bitmask bits are consumed from the most significant end towards the least significant end. When this bit is reset the bitmask bits are consumed from the least significant end towards the most significant end.
1	InvertBit-Mask	When this bit is set the bitmask is inverted first before being tested.
2,3	Fraction-Adjust	These bits control the action of a ContinueNewLine command and specify how the fraction bits in the Y and XDom DDAs are adjusted 0: No adjustment is done 1: Set the fraction bits to zero 2: Set the fraction bits to half 3: Set the fraction to nearly half, i.e. 0x7fff
4,5	BiasCoor-dinates	These bits control how much is added onto the StartXDom, StartXSub and StartY values, when they are loaded into the DDA units. The original registers are not affected: 0: Zero is added 1: Half is added 2: Nearly half, i.e. 0x7fff is added

Scissor Unit

Two scissor tests are provided in GLINT, the User Scissor test and the Screen Scissor test. The user scissor checks each fragment against a user supplied scissor region; the screen scissor checks that the fragment lies within the screen.

This test may reject fragments if some part of a window has been moved off the screen. It will not reject fragments if part of a window is simply overlapped by another window (GID testing can be used to detect this).

Stipple Unit

Stippling is a process whereby each fragment is checked against a bit in a defined pattern, and is rejected or accepted depending on the result of the stipple test. If it is rejected it undergoes no further processing; otherwise it proceeds down the pipeline. GLINT supports two types of stippling, line and area.

Area Stippling

A 32×32 bit area stipple pattern can be applied to fragments. The least significant n bits of the fragment's (X,Y) coordinates, index into a 2D stipple pattern. If the selected bit in the pattern is set, then the fragment passes the test, otherwise it is rejected. The number of address bits used, allow 8×8 or 16×16, 8,16 and 32 pixels to be stippled. The address selection can be controlled independently in the X and Y directions. In addition the bit pattern can be inverted or mirrored. Inverting the bit pattern has the effect of changing the sense of the accept/reject test. If the mirror bit is set the most significant bit of the pattern is towards the left of the window, the default is the converse.

In some situations window relative stippling is required but coordinates are only available screen relative. To allow window relative stippling, an offset is available which is added to the coordinates before indexing the stipple table. X and Y offsets can be controlled independently.

Line Stippling

In this test, fragments are conditionally rejected on the outcome of testing a linear stipple mask. If the bit is zero then the test fails, otherwise it passes. The line stipple pattern is 16 bits in length and is scaled by a repeat factor r (in the range 1 to 512). The stipple mask bit b which controls the acceptance or rejection of a fragment is determined using:

$$b=(\text{floor}(s/r)) \bmod 16$$

where s is the stipple counter which is incremented for every fragment (normally along the line). This counter may be reset at the start of a polyline, but between segments it continues as if there were no break.

The stipple pattern can be optionally mirrored, that is the bit pattern is traversed from most significant to least significant bits, rather than the default, from least significant to most significant.

Color DDA Unit

The color DDA unit is used to associate a color with a fragment produced by the rasterizer. This unit should be enabled for rendering operations and disabled for pixel rectangle operations (i.e. copies, uploads and downloads). Two color modes are supported by GLINT, true color RGBA and color index (CI).

Gouraud Shading

When in Gouraud shading mode, the color DDA unit performs linear interpolation given a set of start and increment values. Clamping is used to ensure that the interpolated value does not underflow or overflow the permitted color range.

For a Gouraud shaded trapezoid, GLINT interpolates from the dominant edge of a trapezoid to the subordinate edges. This means that two increment values are required per color component, one to move along the dominant edge and one to move across the trapezoid between the edges.

Note that if one is rendering to multiple buffers and has initialized the start and increment values in the color DDA unit, then any subsequent Render command will cause the start values to be reloaded.

If subpixel correction has been enabled for a primitive, then any correction required will be applied to the color components.

Flat Shading

In flat shading mode, a constant color is associated with each fragment. This color is loaded into the ConstantColor register.

Texture Unit

The texture unit combines the incoming fragment's color (generated in the color DDA unit) with a value derived from interpolating texture map values (texels).

Texture application consists of two stages; derivation of the texture color from the texels (a filtering process) and then application of the texture color to the fragment's color, which is dependent on the application mode (Decal, Blend or Modulate).

GLINT 300SX compared with the GLINT 400TX

Both the GLINT 300SX and GLINT 300TX support all the filtering and application modes described in this section. However, when using the GLINT 300SX, texel values, interpolants and texture filter selections are supplied by the host. This implies that texture coordinate interpolation and texel extraction are performed by the host using texture maps resident on the host. The recommended technique for performing texture mapping using the GLINT 300SX is to scan convert primitives on the host and render fragments as GLINT point primitives.

The GLINT 400TX automatically generates all data required for texture application as textures are stored in the localbuffer and texture parameter interpolation with full perspective correction takes place within the processor. Thus the GLINT 400TX is the processor of choice when full texture mapping acceleration is desired, the GLINT 300SX is more suitable in applications where the performance of texture mapping is not critical.

Texture Color Generation.

Texture color generation supports all the filter modes of OpenGL, that is:

Minification:
- Nearest
- Linear
- NearestMipMapNearest
- NearestMipMapLinear
- LinearMipMapNearest
- LinearMipMapLinear

Magnification:
- Nearest
- Linear

Minification is the name given to the filtering process used whereby multiple texels map to a fragment, while magnification is the name given to the filtering process whereby only a portion of a single texel maps to a single fragment.

Nearest is the simplest form of texture mapping where the nearest texel to the sample location is selected with no filtering applied.

Linear is a more sophisticated algorithm which is dependent on the type of primitive. For lines (which are 1D), it involves linear interpolation between the two nearest texels, for polygons and points which are considered to have finite area, linear is in fact bi-linear interpolation which interpolates between the nearest 4 texels.

Mip Mapping is a technique to allow the efficient filtering of texture maps when the projected area of the fragment covers more than one texel (ie. minification). A hierarchy of texture maps is held with each one being half the size (or one quarter the area) of the preceding one. A pair of maps are selected, based on the projected area of the texture. In terms of filtering this means that three filter operations are performed: one on the first map, one on the second map and one between the maps. The first filter name (Nearest or Linear) in the MipMap name specifies the filtering to do on the two maps, and the second filter name specifies the filtering to do between maps. So for instance, linear mapping between two maps, with linear interpolation between the results is supported (LinearMipMapLinear), but linear interpolation on one map, nearest on the other map, and linear interpolation between the two is not supported.

The filtering process takes a number of texels and interpolants, and with the current texture filter mode produces a texture color.

Fog Unit

The fog unit is used to blend the incoming fragment's color (generated by the color DDA unit, and potentially modified by the texture unit) with a predefined fog color. Fogging can be used to simulate atmospheric fogging, and also to depth cue images.

Fog application has two stages; derivation of the fog index for a fragment, and application of the fogging effect. The fog index is a value which is interpolated over the primitive using a DDA in the same way color and depth are interpolated. The fogging effect is applied to each fragment using one of the equations described below.

Note that although the fog values are linearly interpolated over a primitive the fog values can be calculated on the host using a linear fog function (typically for simple fog effects and depth cuing) or a more complex function to model atmospheric attenuation. This would typically be an exponential function.

Fog Index Calculation—The Fog DDA

The fog DDA is used to interpolate the fog index (f) across a primitive. The mechanics are similar to those of the other DDA units, and horizontal scanning proceeds from dominant to subordinate edge as discussed above.

The DDA has an internal range of approximately +511 to −512, so in some cases primitives may exceed these bounds. This problem typically occurs for very large polygons which span the whole depth of a scene. The correct solution is to tessellate the polygon until polygons lie within the acceptable range, but the visual effect is frequently negligible and can often be ignored.

The fog DDA calculates a fog index value which is clamped to lie in the range 0.0 to 1.0 before it is used in the appropriate fogging equation. (Fogging is applied differently depending on the color mode.)

Antialias Application Unit

Antialias application controls the combining of the coverage value generated by the rasterizer with the color generated in the color DDA units. The application depends on the color mode, either RGBA or Color Index (CI).

Antialias Application

When antialiasing is enabled this unit is used to combine the coverage value calculated for each fragment with the fragment's alpha value. In RGBA mode the alpha value is multiplied by the coverage value calculated in the rasterizer (its range is 0% to 100%). The RGB values remain unchanged and these are modified later in the Alpha Blend unit which must be set up appropriately. In CI mode the coverage value is used to select between two color indices

The Color Look Up Table is assumed to be set up such that each color has 16 intensities associated with it, one per coverage entry.

Polygon Antialiasing

When using GLINT to render antialiased polygons, depth buffering cannot be used. This is because the order the fragments are combined in is critical in producing the correct final color. Polygons should therefore be depth sorted, and rendered front to back, using the alpha blend modes: SourceAlphaSaturate for the source blend function and One for the destination blend function. In this way the alpha component of a fragment represents the percentage pixel coverage, and the blend function accumulates coverage until the value in the alpha buffer equals one, at which point no further contributions can made to a pixel.

For the antialiasing of general scenes, with no restrictions on rendering order, the accumulation buffer is the preferred choice. This is indirectly supported by GLINT via image uploading and downloading, with the accumulation buffer residing on the host.

When antialiasing, interpolated parameters which are sampled within a fragment (color, fog and texture), will sometimes be unrepresentative of a continuous sampling of a surface, and care should be taken when rendering smooth shaded antialiased primitives. This problem does not occur in aliased rendering, as the sample point is consistently at the center of a pixel.

Alpha Test Unit

The alpha test compares a fragment's alpha value with a reference value. Alpha testing is not available in color index (CI) mode. The alpha test conditionally rejects a fragment based on the comparison between a reference alpha value and one associated with the fragment.

Localbuffer Read/Write Unit

The localbuffer holds the Graphic ID, FrameCount, Stencil and Depth data associated with a fragment. The localbuffer read/write unit controls the operation of GID testing, depth testing and stencil testing.

Localbuffer Read

The LBReadMode register can be configured to make 0, 1 or 2 reads of the localbuffer. The following are the most common modes of access to the localbuffer:

- Normal rendering without depth, stencil or GID testing. This requires no localbuffer reads or writes.
- Normal rendering without depth or stencil testing and with GID testing. This requires a localbuffer read to get the GID from the localbuffer.
- Normal rendering with depth and/or stencil testing required which conditionally requires the localbuffer to be updated. This requires localbuffer reads and writes to be enabled.
- Copy operations. Operations which copy all or part of the localbuffer with or without GID testing. This requires reads and writes enabled.
- Image upload/download operations. Operations which download depth or stencil information to the local buffer or read depth, stencil fast clear or GID from the localbuffer.

Localbuffer Write

Writes to the localbuffer must be enabled to allow any update of the localbuffer to take place. The LBWriteMode register is a single bit flag which controls updating of the buffer.

Pixel Ownership (GID) Test Unit

Any fragment generated by the rasterizer may undergo a pixel ownership test. This test establishes the current fragment's write permission to the localbuffer and framebuffer.

Pixel Ownership Test

The ownership of a pixel is established by testing the GID of the current window against the GID of a fragment's destination in the GID buffer. If the test passes, then a write can take place, otherwise the write is discarded. The sense of the test can be set to one of: always pass, always fail, pass if equal, or pass if not equal. Pass if equal is the normal mode. In GLINT the GID planes, if present, are 4 bits deep allowing 16 possible Graphic ID's. The current GID is established by setting the Window register.

If the unit is disabled fragments pass through undisturbed.

Stencil Test Unit

The stencil test conditionally rejects fragments based on the outcome of a comparison between the value in the stencil buffer and a reference value. The stencil buffer is updated according to the current stencil update mode which depends on the result of the stencil test and the depth test.

Stencil Test

This test only occurs if all the preceding tests (bitmask, scissor, stipple, alpha, pixel ownership) have passed. The stencil test is controlled by the stencil function and the stencil operation. The stencil function controls the test between the reference stencil value and the value held in the stencil buffer. The stencil operation controls the updating of the stencil buffer, and is dependent on the result of the stencil and depth tests.

If the stencil test is enabled then the stencil buffer will be updated depending on the outcome of both the stencil and the depth tests (if the depth test is not enabled the depth result is set to pass).

In addition a comparison bit mask is supplied in the StencilData register. This is used to establish which bits of the source and reference value are used in the stencil function test. In addition it should normally be set to exclude the top four bits when the stencil width has been set to 4 bits in the StencilMode register.

The source stencil value can be from a number of places as controlled by a field in the StencilMode register:

LBWriteData Stencil	Use
Test logic	This is the normal mode.
Stencil register	This is used, for instance, in the OpenGL draw pixels function where the host supplies the stencil values in the Stencil register. This is used when a constant stencil values is needed, for example, when clearing the stencil buffer when fast clear planes are not available.
LBSourceData: (stencil value read from the localbuffer)	This is used, for instance, in the OpenGL copy pixels function when the stencil planes are to be copied to the destination. The source is offset from the destination by the value in LBSourceOffset register.
Source stencil value read from the localbuffer	This is used, for instance, in the OpenGL copy pixels function when the stencil planes in the destination are not to be updated. The stencil data will come either from the localbuffer date, or the FCStencil register, depending on whether fast clear operations are enabled.

Depth Test Unit

The depth (Z) test, if enabled, compares a fragment's depth against the corresponding depth in the depth buffer. The result of the depth test can effect the updating of the stencil buffer if stencil testing is enabled. This test is only performed if all the preceding tests (bitmask, scissor, stipple, alpha, pixel ownership, stencil) have passed. The source value can be obtained from a number of places as controlled by a field in the DepthMode register.

Source	Use
DDA (see below)	This is used for normal Depth buffered 3D rendering.
Depth register	This is used, for instance, in the OpenGL draw pixels function where the host supplies the depth values through the Depth register. Alternatively this is used when a constant depth value is needed, for example, when clearing the depth buffer (when fast clear planes are not available) or 2D rendering where the depth is held constant.
LBSourceData: Source depth value from the localbuffer	This is used, for instance, in the OpenGL copy pixels function when the depth planes are to be copied to the destination.
Source Depth	This is used, for instance, in the OpenGL copy pixels function when the depth planes in the destination are not updated. The depth data will come either from the localbuffer or the FCDepth register depending the state of the Fast Clear modes in operation.

When using the depth DDA for normal depth buffered rendering operations the depth values required are similar to those required for the color values in the color DDA unit:
ZStart=Start Z Value
dZdYDom=Increment along dominant edge.
dZdX=Increment along the scan line.

The dZdX value is not required for Z-buffered lines.

The depth unit must be enabled to update the depth buffer. If it is disabled then the depth buffer will only be updated if ForceLBUpdate is set in the Window register.

Framebuffer Read/Write Unit

Before rendering can take place GLINT must be configured to perform the correct framebuffer read and write operations. Framebuffer read and write modes effect the operation of alpha blending, logic ops, write masks, image upload/download operations and the updating of pixels in the framebuffer.

Framebuffer Read

The FBReadMode register allows GLINT to be configured to make 0, 1 or 2 reads of the framebuffer. The following are the most common modes of access to the framebuffer: Note that avoiding unnecessary additional reads will enhance performance.

Rendering operations with no logical operations, software write-masking or alpha blending. In this case no read of the framebuffer is required and framebuffer writes should be enabled.

Rendering operations which use logical ops, software write masks or alpha blending. In these cases the destination pixel must be read from the framebuffer and framebuffer writes must be enabled.

Image copy operations. Here setup varies depending on whether hardware or software write masks are used. For software write masks, the framebuffer needs two reads, one for the source and one for the destination. When hardware write masks are used (or when the software write mask allows updating of all bits in a pixel) then only one read is required.

Image upload. This requires reading of the destination framebuffer reads to be enabled and framebuffer writes to be disabled.

Image download. In this case no framebuffer read is required (as long as software writemasking and logic ops are disabled) and the write must be enabled.

For both the read and the write operations, an offset is added to the calculated address. The source offset (FBSourceOffset) is used for copy operations. The pixel offset (FBPixelOffset) can be used to allow multi-buffer updates. The offsets should be set to zero for normal rendering.

The data read from the framebuffer may be tagged either FBDefault (data which may be written back into the framebuffer or used in some manner to modify the fragment color) or FBColor (data which will be uploaded to the host). The table below summarizes the framebuffer read/write control for common rendering operations:

Read-Source	ReadDestination	Writes	Read Data Type	Rendering Operation
Disabled	Disabled	Enabled	----	Rendering with no logical operations, software write masks or blending.
Disabled	Disabled	Enabled	----	Image download.
Disabled	Enabled	Disabled	FBColor	Image upload.
Enabled	Disabled	Enabled	FBDefault	Image copy with hardware write masks.
Disabled	Enabled	Enabled	FBDefault	Rendering using logical operations, software write masks or blending.
Enabled	Enabled	Enabled	FBDefault	Image copy with software writemasks.

Framebuffer Write

Framebuffer writes must be enabled to allow the framebuffer to be updated. A single 1 bit flag controls this operation.

The framebuffer write unit is also used to control the operation of fast block fills, if supported by the framebuffer. Fast fill rendering is enabled via the FastFillEnable bit in the Render command register, the framebuffer fast block size must be configured to the same value as the FastFillIncrement in the Render command register. The FBBlockColor register holds the data written to the framebuffer during a block fill operation and should be formatted to the 'raw' framebuffer format. When using the framebuffer in 8 bit packed mode the data should be replicated into each byte. When using the framebuffer in packed 16 bit mode the data should be replicated into the top 16 bits.

When uploading images the UpLoadData bit can be set to allow color formatting (which takes place in the Alpha Blend unit).

It should be noted that the block write capability provided by the chip of the presently preferred embodiment is itself believed to be novel. According to this new approach, a graphics system can do masked block writes of variable length (e.g. 8, 16, or 32 pixels, in the presently preferred embodiment). The rasterizer defines the limits of the block to be written, and hardware masking logic in the framebuffer interface permits the block to be filled in, with a specified primitive, only up to the limits of the object being rendered. Thus the rasterizer can step by the Block Fill increment. This permits the block-write capabilities of the VRAM chips to be used optimally, to minimize the length which must be written by separate writes per pixel.

Alpha Blend Unit

Alpha blending combines a fragment's color with those of the corresponding pixel in the framebuffer. Blending is supported in RGBA mode only.

Alpha Blending

The alpha blend unit combines the fragment's color value with that stored in the framebuffer, using the blend equation:

$$C_o = C_s S + C_d D$$

where: C_o is the output color; C_s is the source color (calculated [illegible])

the framebuffer; S is the source blending weight; and D is the destination blending weight. S and D are not limited to linear combinations; lookup functions can be used to implement other combining relations.

If the blend operations require any destination color components then the framebuffer read mode must be set appropriately.

Image Formatting

The alpha blend and color formatting units can be used to format image data into any of the supported GLINT framebuffer formats.

Consider the case where the framebuffer is in RGBA 4:4:4:4 mode, and an area of the screen is to be uploaded and stored in an 8 bit RGB 3:3:2 format. The sequence of operations is:

Set the rasterizer as appropriate

Enable framebuffer reads

Disable framebuffer writes and set the UpLoadData bit in the FBWriteMode register

Enable the alpha blend unit with a blend function which passes the destination value and ignores the source value (source blend Zero, destination blend One) and set the color mode to RGBA 4:4:4:4

Set the color formatting unit to format the color of incoming fragments to an 8 bit RGB 3:3:2 framebuffer format.

The upload now proceeds as normal. This technique can be used to upload data in any supported format.

The same technique can be used to download data which is in any supported framebuffer format, in this case the rasterizer is set to sync with FBColor, rather than Color. In this case framebuffer writes are enabled, and the UpLoadData bit cleared.

Color Formatting Unit

The color formatting unit converts from GLINT's internal color representation to a format suitable to be written into the framebuffer. This process may optionally include dithering of the color values for framebuffers with less than 8 bits width per color component. If the unit is disabled then the color is not modified in any way.

As noted above, the framebuffer may be configured to be RGBA or Color Index (CI).

Color Dithering

GLINT uses an ordered dither algorithm to implement color dithering. Several types of dithering can be selected.

If the color formatting unit is disabled, the color components RGBA are not modified and will be truncated when placed in the framebuffer. In CI mode the value is rounded to the nearest integer. In both cases the result is clamped to a maximum value to prevent overflow.

In some situations only screen coordinates are available, but window relative dithering is required. This can be implemented by adding an optional offset to the coordinates before indexing the dither tables. The offset is a two bit number which is supplied for each coordinate, X and Y. The XOffset, YOffset fields in the DitherMode register control this operation, if window relative coordinates are used they should be set to zero.

Logical Op Unit

The logical op unit performs two functions; logic operations between the fragment color (source color) and a value from the framebuffer (destination color); and, optionally, control of a special GLINT mode which allows high performance flat shaded rendering.

High Speed Flat Shaded Rendering

A special GLINT rendering mode is available which allows high speed rendering of unshaded images. To use the mode the following constraints must be satisfied:

Flat shaded aliased primitive

No dithering required

No logical ops

No stencil, depth or GID testing required

No alpha blending The following are available:

Bit masking in the rasterizer

Area and line stippling

User and Screen Scissor test

If all the conditions are met then high speed rendering can be achieved by setting the FBWriteData register to hold the framebuffer data (formatted appropriately for the framebuffer in use) and setting the UseConstantFBWriteData bit in the LogicalOpMode register. All unused units should be disabled.

This mode is most useful for 2D applications or for clearing the framebuffer when the memory does not support block writes. Note that FBWriteData register should be considered volatile when context switching.

Logical Operations

The logical operations supported by GLINT are:

Mode	Name	Operation	Mode	Name	Operation
0	Clear	0	8	Nor	~(S | D)
1	And	S & D	9	Equivalent	~(S ^ D)
2	And Reverse	S & ~D	10	Invert	~D
3	Copy	S	11	Or Reverse	S | ~D
4	And Inverted	~S & D	12	Copy Invert	~S
5	Noop	D	13	Or Invert	~S | D
6	Xor	S ^ D	14	Nand	~(S & D)
7	Or	S | D	15	Set	1

Where:

S=Source (fragment) Color. D=Destination (framebuffer) Color.

For correct operation of this unit in a mode which takes the destination color, GLINT must be configured to allow reads from the framebuffer using the FBReadMode register.

GLINT makes no distinction between RGBA and CI modes when performing logical operations. However, logical operations are generally only used in CI mode.

Framebuffer Write Masks

Two types of framebuffer write masking are supported by GLINT, software and hardware. Software write masking requires a read from the framebuffer to combine the fragment color with the framebuffer color, before checking the bits in the mask to see which planes are writeable. Hardware write masking is implemented using VRAM write masks and no framebuffer read is required.

Software Write Masks

Software write masking is controlled by the FBSoftwareWriteMask register. The data field has one bit per framebuffer bit which when set, allows the corresponding framebuffer bit to be updated. When reset it disables writing to that bit. Software write masking is applied to all fragments and is not controlled by an enable/disable bit. However it may effectively be disabled by setting the mask to all 1's. Note that the ReadDestination bit must be enabled in the FBReadMode register when using software write masks, in which some of the bits are zero.

Hardware Write Masks

Hardware write masks, if available, are controlled using the FBHardwareWriteMask register. If the framebuffer supports hardware write masks, and they are to be used, then software write masking is [illegible]

bits in the FBSoftwareWriteMask register). This will result in fewer framebuffer reads when no logical operations or alpha blending is needed.

If the framebuffer is used in 8 bit packed mode, then an 8 bit hardware write mask must be replicated to all 4 bytes of the FBHardwareWriteMask register. If the framebuffer is in 16 bit packed mode then the 16 bit hardware write mask must be replicated to both halves of the FBHardwareWriteMask register.

Host Out Unit

Host Out Unit controls which registers are available at the output FIFO, gathering statistics about the rendering operations (picking and extent testing) and the synchronization of GLINT via the Sync register. These three functions are as follows:

Message filtering. This unit is the last unit in the core so any message not consumed by a preceding unit will end up here. These messages will fall in to three classifications: Rasterizer messages which are never consumed by the earlier units, messages associated with image uploads, and finally programmer mistakes where an invalid message was written to the input FIFO. Synchronization messages are a special category and are dealt with later. Any messages not filtered out are passed on the output FIFO.

Statistic Collection. Here the active step messages are used to record the extent of the rectangular region where rasterization has been occurring, or if rasterization has occurred inside a specific rectangular region. These facilities are useful for picking and debug activities.

Synchronization. It is often useful for the controlling software to find out when some rendering activity has finished, to allow the timely swapping or sharing of buffers, reading back of state, etc. To achieve this the software would send a Sync message and when this reached this unit any preceding messages or their actions are guaranteed to have finished. On receiving the Sync message it is entered into the FIFO and optionally generates an interrupt.

Sample Board-Level Embodiment

A sample board incorporating the GLINT chip may include simply:

the GLINT chip itself, which incorporates a PCI interface;

Video RAM (VRAM), to which the chip has read-write access through its frame buffer (FB) port;

DRAM, which provides a local buffer then made for such purposes as Z buffering; and

a RAMDAC, which provides analog color values in accordance with the color values read out from the VRAM.

Thus one of the advantages of the chip of the presently preferred embodiment is that a minimal board implementation is a trivial task.

FIG. 3A shows a sample graphics board which incorporates the chip of FIG. 2B.

FIG. 3B shows another sample graphics board implementation, which differs from the board of FIG. 3A in that more memory and an additional component is used to achieve higher performance.

FIG. 3C shows another graphics board, in which the chip of FIG. 2B shares access to a common frame store with GUI accelerator chip.

FIG. 3D shows another graphics board, in which the chip of FIG. 2B shares access to a common frame store with a video coprocessor (which may be used for video capture and playback functions (e.g. frame grabbing).

Alternative Board Embodiment with Additional Video Processor

In the presently preferred embodiment, the frame buffer interface of the GLINT chip contains additional simple interface logic, so that two chips can both access the same frame buffer memory. This permits the GLINT chip to be combined with an additional chip for management to the graphics produced by the graphical user interface. This provides a migration path for users and applications who need to take advantage of the existing software investment and device drivers for various other graphics chips.

FIG. 3C shows another graphics board, in which the chip of FIG. 2B shares access to a common frame store with a GUI accelerator chip (such as an S3 chip). This provides a path for software migration, and also provides a way to separate 3D rendering tasks from 2D rendering.

In this embodiment, a shared framebuffer is used to enable multiple devices to read or write data to the same physical framebuffer memory. Example applications using the GLINT 300SX:

Using a video device as a coprocessor to GLINT, to grab live video into the framebuffer, for displaying video in a window or acquiring a video sequence;

Using GLINT as a 3D coprocessor to a 2D GUI accelerator, preserving an existing investment in 2D driver software.

In a coprocessor system, the framebuffer is a shared resource, and so access to the resource needs to be arbitrated. There are also other aspects of sharing a framebuffer that need to be considered:

Memory refreshing;

Transfer of data from the memory cells into the shift registers of the VRAM;

Control of writemasks and color registers.

GLINT uses the S3 Shared Frame Buffer Interface (SFBI) to share a framebuffer. This interface is able to handle all of the above aspects for two devices sharing a frame buffer, with the GLINT acting as an arbitration master or slave.

Timing Considerations in Shared Frame-Buffer Interface

The Control Signals used in the Shared Framebuffer interface, in the presently preferred embodiment, are as follows:

GLINT as Primary Controller

FBReqN is internally re-synchronized to System Clock.

FBSelOEN remains negated.

FBGntN is asserted an unspecified amount of time after FBReqN is asserted.—Framebuffer Address, Data and Control lines are tri-stated by GLINT (the control lines should be held high by external pull-up resistors). The secondary controller is now free to drive the Framebuffer lines and access the memory.

FBGntN remains asserted until GLINT requires a framebuffer access, or a refresh or transfer cycle.

FBReqN must remain asserted while FBGntN is asserted.

When FBGntN is removed, the secondary controller must relinquish the address, data and control bus in a graceful manner i.e. RAS, CAS, WE and OE must all be driven high before being tri-stated.

The secondary controller must relinquish the bus and negate FBReqN within 500 ns of FBGntN being negated.

Once FBReqN has been negated, it must remain inactive for at least 2 system clocks (40 ns at 50 MHz).

GLINT as Secondary Controller

Framebuffer Refresh and VRAM transfer cycles by GLINT are turned off when GLINT is a secondary framebuffer controller.

GLINT asserts FBReqN whenever is requires a framebuffer access.

FBGntN is internally re-synchronized to system clock.

When FBGntN is asserted, GLINT drives FBselOEN to enable any external buffers used to drive the control signals, and then drives the framebuffer address, data and control lines to perform the memory access. FBReqN remains asserted while FBGntN is asserted.

When FBGntN is negated, GLINT finishes any outstanding memory cycles, drives the control lines inactive, negates FBselOEN and then tri-states the address, data and control lines, then releases FBReqN. GLINT guarantees to release FBReqN within 500 ns of FBGntN being negated.

GLINT will not reassert FBReqN within 4 system clock cycles (80 ns@ 50 MHz).

Considerations for Board-Level Implementations

The following are some points to be noted when implementing a shared framebuffer design with a GLINT 300SX:

Some 2D GUI Accelerators such as the S3 Vision964, and GLINT use configuration resistors on the framebuffer databus at reset. In this case care should be taken with the configuration setup where it effects read only registers inside either device. If conflicts exist that can not be resolved by the board initialization software, then the conflicts should be resolved by isolating the two devices from each other at reset so they can read the correct configuration information. This isolation need only be done for the framebuffer databus lines that cause problems;

GLINT should be configured as the secondary controller when used with an S3 GUI accelerator, as the S3 devices can only be primary controllers;

GLINT cannot be used on the daughter card interface as described in the S3 documentation, because this gives no access to the PCI bus. A suitable PCI bridge should be used in a design with a PCI 2D GUI accelerator and GLINT so they can both have access to the PCI bus;

The use of ribbon cable to carry the framebuffer signals between two PCI boards is not recommended, because of noise problems and the extra buffering required would impact performance;

The GLINT 300SX does not provide a way of sharing its localbuffer.

The 400TX also allows grabbing of live video into the localbuffer and real-time texture mapping of that video into the framebuffer for video manipulation effects.

Alternative Board Embodiments with Multiple Rendering Accelerator Chips

This technical note describes some system design issues on how multiple GLINT devices can be used in parallel to achieve higher performance. The main driving force for higher performance is the simulation market which, at the low end, demands somewhere between 25–30M texture mapped pixels per second.

There are some key points before we look at different parallel organizations:

To gain any benefit from running multiple GLINTs in parallel, the overall system must be rendering bound. If the system is host bound or geometry bound, then adding in more GLINTs will not improve the systems performance.

The memory systems (i.e. local buffer and framebuffer) are duplicated for each GLINT. Recall that the texture maps are stored in the local buffer. A single GLINT places very high demands on the memory systems, and it would be very difficult to share them between multiple GLINTs. In the presently preferred embodiment there are no provisions for sharing the local buffer, so if this is necessary it would have to be done behind GLINT's back and transparently. The framebuffer can be shared (since GLINT has a SFB interface), but this is likely to be a bottle neck if shared between GLINTs.

Broadcast. In some parallel systems each GLINT will get the same (or mostly the same) primitive data and just render those pixels assigned to it. It is very desirable that this data is written by the host only once, or fetched from the host address space once if DMA is being used. This presents two issues: Firstly the PCI bus does not have any concept of broadcasting to multiple devices, and secondly GLINT does not have a dedicated FIFO status signal pin an external controller can use. Neither of these issues are insurmountable, but will require hardware to solve. However, if the application only uses a 'few' large texture mapped primitives so repeatedly sending or fetching the parameters for each GLINT will not be a problem.

To avoid problems with Antialiasing, Bitmasks for characters, or Line stipple, the area stipple table can be used to reserve scanlines to a processor.

Parallel Configurations

This section looks at some of the common ways of applying parallelism to the rendering operation. The list is not exhaustive and an interested reader is directed to the book by Whitman cited above. No one paradigm is best and the choice is very application or market dependent.

Frame Interleaving

Frame Interleaving is where a GLINT works on frame n, the next GLINT works on frame n+1, etc. Each GLINT does everything for its own frame and the video is sourced from each GLINT's framebuffer in turn. This paradigm is perhaps the simplest one with very little hardware overhead and none of the above complications regarding antialiasing, block copies, bitmasks and line stipples.

This scheme only works when the image is double buffered (normal for simulation systems) and where the increase in transport delay is acceptable. Transport delay is the time it takes for a user to see a visual change after new input stimulus to the system has occurred. With 4 GLINTs this will be 4 frame times attributable to the rendering system, plus whatever else the whole system adds.

The cost of this method is also one of the highest, as ALL the memory has to be duplicated. By contrast, the schemes where the screen is divided up can save depth and color buffer memory (but not texture memory).

Sequential frames will usually have very similar amounts of rendering, unless there is a discontinuity in the viewing position and/or orientation, so load balancing is generally good.

Frame Merging or Primitive Parallelism

Frame merging is a similar technique to frame interleaving where each GLINT has a full local buffer and framebuffer. In this case the primitives are distributed amongst the GLINTs and the resultant partial images composited using the depth information to control which fragment from the multiple buffers is displayed in each pixel position.

GLINT has not been designed to share the local buffer (where the depth information is held) so the compositing is not readily supported. Also the composition frequently needs to be done at video rate so requires some fast hardware.

Alpha blending and Antialiasing presents some problems but the bitmask, block copies and line stipple are easily accommodated. Good load balancing depends on even distribution of primitives. Not all primitives will take the same amount of time to process so a round robin distribution scheme, or a heuristic one with takes into account the expected processing time for each primitive will be needed.

Screen Subdivision—Blocks

Here the screen is divided up into large contiguous regions and a GLINT looks after each region. Primitives which overlap between regions are sent to both regions and scissor clipping used. Primitives contained wholly in one region are ideally just sent to the one GLINT.

The number of regions and the horizontal and/or vertical division of the screen can be chosen as appropriate, but horizontal bands are usually easier for the video hardware to cope with. Each GLINT only needs enough local buffer and frame buffer to cover the pixels in its own region, but texture maps are duplicated in full. Block copies are a problem when the block, or part block is moved between regions. Bit masking and line stipples can be solved with some careful clipping.

Load balancing is very poor in this paradigm, since most of the scene complexity can be concentrated into one region. Dynamically changing the size of the regions based on expected scene complexity (maybe measured from the previous frame) can alleviate the poor load balancing to some extent.

Screen Subdivision—Interleaved Scanlines

The interleave factor is every other n^{th} scanline where n is the number of GLINTs. Vertical interleaves are possible, but not supported by the GLINT rasterizer. Nearly all primitives will overlap multiple scanlines so are ideally broadcast to all GLINTs. Each GLINT will have different start values for the rasterization and interpolation parameters.

Each GLINT only needs enough local buffer and frame buffer to cover the pixels in its own region, but texture maps are duplicated in full.

Some block copies are a problem when the block is moved between non nth scanlines, but horizontal moves are available with any alignment. Bit masking can be solved with some careful clipping, but line stipples have no easy solution. Antialiasing is not normally a problem but with GLINT 300SX there is no provision for sub scanline steps as well as nth scanline steps. Load balancing is excellent in this paradigm which is the main reason it features prominently in the literature.

Thus the simplest and lowest risk method of using multiple GLINTs is Frame Interleaving, but if this is not an option, e.g. because of the transport delay or the amount of memory needed, then the next best choice is the Interleaved Scanline.

Linkage

FIG. 2B shows how the units are connected together. Some general points are:

The order of the units can be configured in two ways. The most general order (Router, Colour DDA, Texture Units, Fog Unit, Alpha Test, LB Rd, GID/Z/Stencil, LB Wr, Multiplexer) and will work in all modes of OpenGL. However, when the alpha test is disabled it is much better to do the Graphics ID, depth and stencil tests before the texture operations rather than after. This is because the texture operations have a high processing cost and this should not be spent on fragments which are later rejected because of depth or stencil tests.

Router Unit Description

The Router Unit allows the order of some of the units to be changed so that texturing can be done before or after the depth test. Any texture operations will cause a loss in performance over the same non-textured rendering, so it is a good idea only to texture those pixels which pass all the depth, stencil and GID tests. OpenGL defines the order in which operations are to be performed on fragments as texture, alpha test, stencil and then depth. It is very likely that in a typical scene many textured fragments will get rejected by the depth test, say, which isn't the most effective use of the texturing capacity. If the alpha test is disabled (or cannot reject fragments) then OpenGL compatible semantics are still maintained if the order is rearranged to be stencil, depth, texture and then alpha test.

The message stream can be re-configured into either of the two orders using the RouterMode message. The reset order is texture, then depth so a to be compatible with OpenGL. Changing the pipeline order is self synchronising so the user doesn't need to wait for the message stream to empty first.

Implementation

This unit is divided into two sub-units: a switcher and a multiplexer. FIG. 5A shows how these are connected together. The basic operation is as follows:

When the Switcher sub-unit receives a RouterMode message it makes a note of the new order, forwards the RouterMode message on and blocks all further messages until it receives a resume signal from the Multiplexer sub unit. When the resume signal is asserted the Switcher re-configures the message paths according to the new order and un-blocks the message stream so it starts to flow again.

When the Multiplexer sub-unit receives the RouterMode message it re-configures the message paths according to the new order and asserts the resume signal to the Switcher. The RouterMode message is consumed. The unit order is controlled using the RouterMode message. It uses the 0-bit of the passed message to indicate if the processing order is:

Bit 0=0	TextureDepth
Bit 0=1	DepthTexture

When the order is TextureDepth (the default after reset) the message routing is done according to FIG. 5B. When the order is DepthTexture the message routing is done according to FIG. 5C.

Disclosed Embodiments

Among the disclosed classes of preferred embodiments, there is provided: A method for processing graphics data through a data path comprising the steps of: (a) receiving a routing command from a data bus input; (b) stalling further input from said data bus input until previous data has exited said data path; (c) resuming said input from said data bus input; (d) if said routing command has a first value, then performing a first set of graphics processes on said data, and then performing a second set of graphics processes on said data; (e) if said routing command has a second value, thenperforming said second set of graphics processes on said data, and thenperforming said first set of graphics processes on said data, wherein some portion of said data may be eliminated by said first or second sets of graphics process according to the results of said processes; wherein steps (d) and (e) are repeated until a new routing command is received; wherein said first set of graphics processes requires a longer processing time than said second set of graphics processes;

Among the disclosed classes of preferred embodiments, there is also provided: A method for processing graphics data through a data path comprising the steps of: (a) receiving a routing command from a data bus input; (b) stalling further input from said data bus input until previous data has exited said data path; (c) resuming said input from said data bus input; (d) if said routing command has a first value, thenperforming a set of texturing processes on said data, and thenperforming a set of pixel elimination processes on said data; (e) if said routing command has a second value, thenperforming said set of pixel elimination processes on said data, and thenperforming said set of texturing processes on said data, wherein some portion of said data may be eliminated by said set of pixel elimination processes according to the results of said processes; wherein steps (d) and (e) are repeated until a new routing command is received; wherein said first set of graphics processes requires a longer processing time than said second set of graphics processes.

Among the disclosed classes of preferred embodiments, there is also provided: A method for rendering graphics data comprising the steps of: (a) receiving a routing command from a data bus input; (b) stalling further input from said data bus input until previous data has exited said data path; (c) resuming said input from said data bus input; (d) if said routing command has a first value, thenperforming a set of texturing processes on said data, and thenperforming a set of pixel elimination processes on said data; (e) if said routing command has a second value, thenperforming said set of pixel elimination processes on said data, and thenperforming said set of texturing processes on said data, wherein some portion of said data may be eliminated by said set of pixel elimination processes according to the results of said processes; (f) rendering said data and writing the results to a memory; (g) displaying the contents of said memory; wherein steps (d) and (e) are repeated until a new routing command is received;wherein said set of texturing processes requires a longer processing time than said set of pixel elimination processes.

Among the disclosed classes of preferred embodiments, there is also provided: A method for processing graphics data through a data path comprising the steps of: (a) receiving a routing command from a data bus input; (b) stalling further input from said data bus input until previous data has exited said data path; (c) resuming said input from said data bus input; (d) if said routing command has a first value, thenreading said graphics data from said data bus input; performing a color DDA process on said data;performing a texturing process on said data;performing an alpha test on said data; if the data has passed the previous test, then performing a graphics ID test on said data; if the data has passed the previous tests, then performing a stencil test on said data;if the data has passed the previous tests, then performing a depth test on said data; and if the data has passed the previous tests, then writing said data to a local bus; (e) if said routing command has a second value, thenreading said graphics data from said data bus input; performing a graphics ID test on said data;if the data has passed the previous test, then performing a stencil test on said data; if the data has passed the previous tests, then performing a depth test on said data; if the data has passed the previous tests, then performing a color DDA process on said data; if the data has passed the previous tests, then performing a texturing process on said data; if the data has passed the previous tests, then performing an alpha test on said data; if the data has passed the previous tests, then writing said data to a local bus; wherein steps (d) and (e) are repeated until a new routing command is received.

Among the disclosed classes of preferred embodiments, there is also provided: A pipelined graphics processing device, comprising;a switching device connected to a data bus input and configured to route graphics data received on said data bus according to instruction data received on said data bus; a multiplexing device connected to said switching device and to a data bus output; a first processing block connected and configured to receive said graphics data from said switching device and pass processed graphics data to said multiplexing device; anda second processing block connected and configured to receive said graphics data from said switching device and pass processed graphics data to said multiplexing device; wherein said switching device routes said graphics data according to a first data path, wherein said graphics data is processed by said first processing block and then by said second processing block, or a second data path, wherein said graphics data is processed by said second processing block before said first processing block, according to said instruction data.

Among the disclosed classes of preferred embodiments, there is also provided: A pipelined graphics processing device, comprising: a routing device connected to a data bus input and data bus output and configured to route graphics data received on said data bus according to instruction data received on said data bus; a first processing block connected and configured to receive said graphics data from said routing device and pass processed graphics data back to said routing device; anda second processing block connected and configured to receive said graphics data from said routing device and pass processed graphics data back to said routing device; wherein said routing device routes data according to a first data path, wherein said graphics data is processed by said first processing block and then by said second processing block, or a second data path, wherein said graphics data is processed by said second processing block before said first processing block, according to said instruction data.

Among the disclosed classes of preferred embodiments, there is also provided: A graphics processing subsystem, comprising: at least four functionally distinct processing units, each including hardware elements which are customized to perform a rendering operation which is not performed by at least some others of said processing units; at least some ones of said processing units being connected to operate asynchronously to one another; a frame buffer, connected to be accessed by at least one of said processing units;said processing units being mutually interconnected in a pipeline relationship, with at least some successive ones of said processing units being interconnected through a FIFO buffer; and wherein at least one said processing unit is connected to look downstream, in said pipeline relationship, past the immediately succeeding one of said processors; and wherein at least two of said processing units may be dynamically reordered in said pipeline relationship; whereby the duty cycle of said processors is increased while permitting use of a reduced depth for said FIFO.

Modifications and Variations

As will be recognized by those skilled in the art, the innovative concepts described in the present application can be modified and varied over a tremendous range of applications, and accordingly the scope of patented subject matter is not limited by any of the specific exemplary teachings given.

The foregoing text has indicated a large number of alternative implementations, particularly at the higher levels, but these are merely a few examples of the huge range of [illegible]

For example, the preferred chip context can be combined with other functions, or distributed among other chips, as will be apparent to those of ordinary skill in the art.

For another example, the described graphics systems and subsystems can be used, in various adaptations, not only in high-end PC's, but also in workstations, arcade games, and high-end simulators.

For another example, the described graphics systems and subsystems are not necessarily limited to color displays, but can be used with monochrome systems.

For another example, the described graphics systems and subsystems are not necessarily limited to displays, but also can be used in printer drivers.

What is claimed is:

1. A method for processing graphics data through a data path comprising the steps of:

(a) receiving a routing command from a data bus input;

(b) stalling further input from said data bus input until previous data has exited said data path;

(c) resuming said input from said data bus input;

(d) if said routing command has a first value, then
performing a first set of graphics processes on said data, and then
performing a second set of graphics processes on said data;

(e) if said routing command has a second value, then
performing said second set of graphics processes on said data, and then
performing said first set of graphics processes on said data, wherein some portion of said data is selectively eliminated by said first or second sets of graphics process according to the results of said processes;

wherein steps (d) and (e) are repeated until a new routing command is received;

wherein said first set of graphics processes requires a longer processing time than said second set of graphics processes.

2. The method of claim 1, wherein said first set of graphics processes comprises the steps of:

reading said graphics data from said data bus input;

performing a color DDA process on said data;

performing a texturing process on said data; and

performing an alpha test on said data.

3. The method of claim 1, wherein said second set of graphics processes comprises the step of if the data has passed all previous tests, then performing a graphics ID test on said data.

4. The method of claim 1, wherein said second set of graphics processes comprises the step of if the data has passed the previous tests, then performing a stencil test on said data.

5. The method of claim 1, wherein said second set of graphics processes comprises the steps of if the data has passed the previous tests, then performing a depth test on said data.

6. The method of claim 1, wherein step (d) comprises steps according to the OpenGL standard.

7. The method of claim 1, wherein step (b) is performed by a switcher connected at said data bus input.

8. The method of claim 1, wherein a multiplexer at an output of said data path indicates when said data path is clear [illegible]

9. A method for processing graphics data through a data path comprising the steps of:

(a) receiving a routing command from a data bus input;

(b) stalling further input from said data bus input until previous data has exited said data path;

(c) resuming said input from said data bus input;

(d) if said routing command has a first value, then
performing a set of texturing processes on said data, and then
performing a set of pixel elimination processes on said data;

(e) if said routing command has a second value, then
performing said set of pixel elimination processes on said data, and then
performing said set of texturing processes on said data, wherein some portion of said data is selectively eliminated by said set of pixel elimination processes according to the results of said processes;

wherein steps (d) and (e) are repeated until a new routing command is received;

wherein said first set of graphics processes requires a longer processing time than said second set of graphics processes.

10. A method for rendering graphics data comprising the steps of:

(a) receiving a routing command from a data bus input;

(b) stalling further input from said data bus input until previous data has exited said data path;

(c) resuming said input from said data bus input;

(d) if said routing command has a first value, then
performing a set of texturing processes on said data, and then
performing a set of pixel elimination processes on said data;

(e) if said routing command has a second value, then
performing said set of pixel elimination processes on said data, and then
performing said set of texturing processes on said data, wherein some portion of said data is selectively eliminated by said set of pixel elimination processes according to the results of said processes;

(f) rendering said data and writing the results to a memory;

(g) displaying the contents of said memory;

wherein steps (d) and (e) are repeated until a new routing command is received;

wherein said set of texturing processes requires a longer processing time than said set of pixel elimination processes.

11. A method for processing graphics data through a data path comprising the steps of:

(a) receiving a routing command from a data bus input;

(b) stalling further input from said data bus input until previous data has exited said data path;

(c) resuming said input from said data bus input;

(d) if said routing command has a first value, then
reading said graphics data from said data bus input;
performing a color DDA process on said data;
performing a texturing process on said data;
performing an alpha test on said data;
if the data has passed the previous test, then performing a graphics ID test on said data;
if the data has passed the previous tests, then perform-

if the data has passed the previous tests, then performing a depth test on said data; and

if the data has passed the previous tests, then writing said data to a local bus;

(e) if said routing command has a second value, then reading said graphics data from said data bus input;

performing a graphics ID test on said data;

if the data has passed the previous test, then performing a stencil test on said data;

if the data has passed the previous tests, then performing a depth test on said data;

if the data has passed the previous tests, then performing a color DDA process on said data;

if the data has passed the previous tests, then performing a texturing process on said data;

if the data has passed the previous tests, then performing an alpha test on said data;

if the data has passed the previous tests, then writing said data to a local bus;

wherein steps (d) and (e) are repeated until a new routing command is received.

12. The method of claim **11**, wherein step (d) comprises steps according to the OpenGL standard.

13. The method of claim **11**, wherein step (b) is performed by a switcher connected at said data bus input.

14. The method of claim **11**, wherein a multiplexer at said local bus indicates when said data path is clear and step (c) can begin.

15. A pipelined graphics processing device, comprising:

a switching device connected to a data bus input and configured to route graphics data received on said data bus according to instruction data received on said data bus;

a multiplexing device connected to said switching device and to a data bus output;

a first processing block connected and configured to receive said graphics data from said switching device and pass processed graphics data to said multiplexing device; and

a second processing block connected and configured to receive said graphics data from said switching device and pass processed graphics data to said multiplexing device;

wherein said switching device routes said graphics data according to a first data path, wherein said graphics data is processed by said first processing block and then by said second processing block, or a second data path, wherein said graphics data is processed by said second processing block before said first processing block, according to said instruction data.

16. The device of claim **15**, wherein said first data path processes said graphics data according to the OpenGL standard.

17. The device of claim **15**, wherein said switching device halts all input data until the current data path is clear before switching data paths.

18. The device of claim **15**, wherein said multiplexing device is configured to determine when the current data path is clear and to allow said switching device to switch data paths.

19. A pipelined graphics processing device, comprising:

a routing device connected to a data bus input and data bus output and configured to route graphics data received on said data bus according to instruction data received on said data bus;

a first processing block connected and configured to receive said graphics data from said routing device and pass processed graphics data back to said routing device; and

a second processing block connected and configured to receive said graphics data from said routing device and pass processed graphics data back to said routing device;

wherein said routing device routes data according to a first data path, wherein said graphics data is processed by said first processing block and then by said second processing block, or a second data path, wherein said graphics data is processed by said second processing block before said first processing block, according to said instruction data.

20. A graphics processing subsystem, comprising:

at least four functionally distinct processing units, each including hardware elements which are customized to perform a rendering operation which is not performed by at least some others of said processing units; at least some ones of said processing units being connected to operate asynchronously to one another;

a frame buffer, connected to be accessed by at least one of said processing units;

said processing units being mutually interconnected in a pipeline relationship, with at least some successive ones of said processing units being interconnected through a FIFO buffer;

and wherein at least one said processing unit is connected to look downstream, in said pipeline relationship, past the immediately succeeding one of said processors;

and wherein at least two of said processing units are selectively dynamically reordered in said pipeline relationship;

whereby the duty cycle of said processors is increased while permitting use of a reduced depth for said FIFO.

21. The graphics processing subsystem of claim **20**, wherein said processing units include a texturing unit.

22. The graphics processing subsystem of claim **20**, wherein said processing units include a scissoring unit.

23. The graphics processing subsystem of claim **20**, wherein said processing units include a memory access unit which reads and writes a local buffer memory.

24. The graphics processing subsystem of claim **20**, wherein at least some ones of said processing units include internally paralleled data paths.

25. The graphics processing subsystem of claim **20**, wherein all of said processing units are integrated into a single integrated circuit.

26. The graphics processing subsystem of claim **20**, wherein all of said processing units, but not said frame buffer, are integrated into a single integrated circuit.

* * * * *

US005987256A

United States Patent [19]

Wu et al.

[11] Patent Number: 5,987,256

[45] Date of Patent: Nov. 16, 1999

[54] **SYSTEM AND PROCESS FOR OBJECT RENDERING ON THIN CLIENT PLATFORMS**

[75] Inventors: **Bo Wu**, **Ling Lu**, both of San Jose, Calif.

[73] Assignee: **Entrust Technology, Inc.**, San Jose, Calif.

[21] Appl. No.: **08/922,898**

[22] Filed: **Sep. 3, 1997**

[51] **Int. Cl.**[6] **G06F 9/45**

[52] **U.S. Cl.** **395/707**

[58] **Field of Search** 395/707

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,504,917	4/1996	Austin et al.	345/522
5,513,127	4/1996	Gard et al.	395/200.53
5,551,015	8/1996	Goettelmann et al.	395/707
5,586,020	12/1996	Isozaki	395/707
5,826,089	10/1998	Ireton	395/707

OTHER PUBLICATIONS

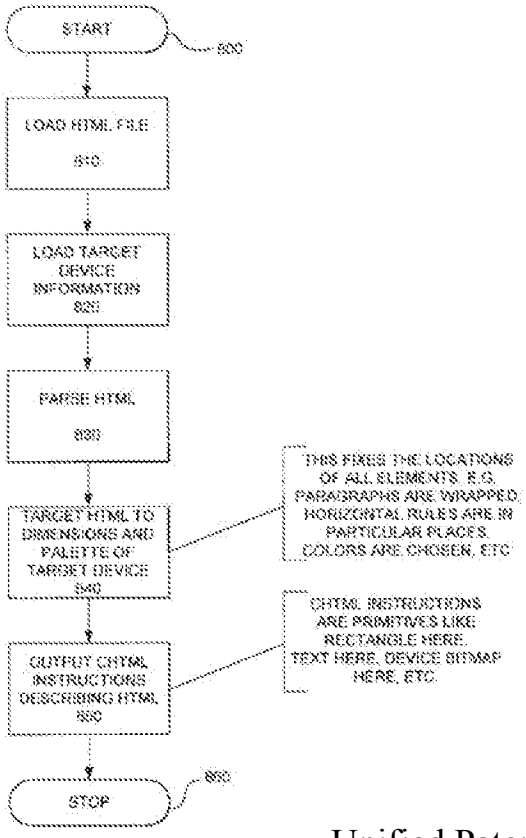
Blickstein et al. The GEM optimizing compiler system; Digital Technical Journal vol. 4 No. 4 Special Issue 1992 pp. 121–136.

Primary Examiner—Robert W. Downs
Assistant Examiner—Wei Zhen
Attorney, Agent, or Firm—Wilson Sonsini Goodrich & Rosati

[57] **ABSTRACT**

A system for processing an object specified by an object specifying language such as HTML, JAVA or other languages relying on relative positioning, that require a rendering program utilizing a minimum set of resources, translates the code for use in a target device that has limited processing resources unsuited for storage and execution of the HTML rendering program, JAVA virtual machine, or other rendering engine for the standard. Data concerning such an object is generated by a process that includes first receiving a data set specifying the object according to the object specifying language, translating the first data set into a second data set in an intermediate object language adapted for a second rendering program suitable for rendering by the target device that utilizes actual target display coordinates. The second data set is stored in a machine readable storage device, for later retrieval and execution by the thin client platform.

14 Claims, 11 Drawing Sheets



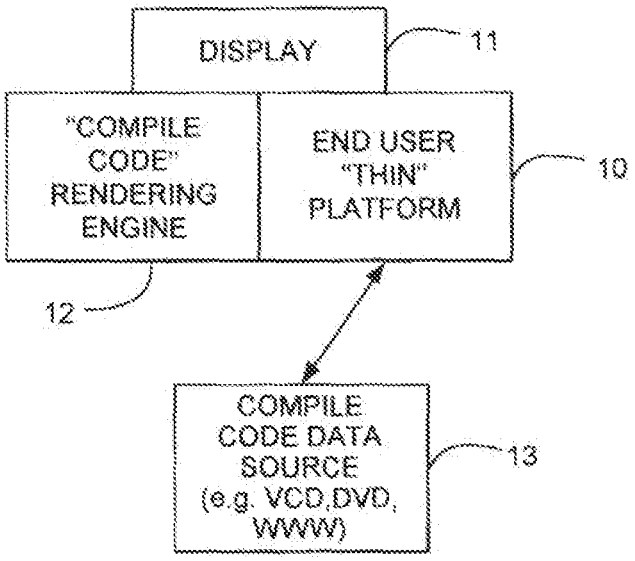

DISPLAY
11
"COMPILE CODE" RENDERING ENGINE
END USER "THIN" PLATFORM
10
12
COMPILE CODE DATA SOURCE (e.g. VCD,DVD, WWW)
13

FIG. 1

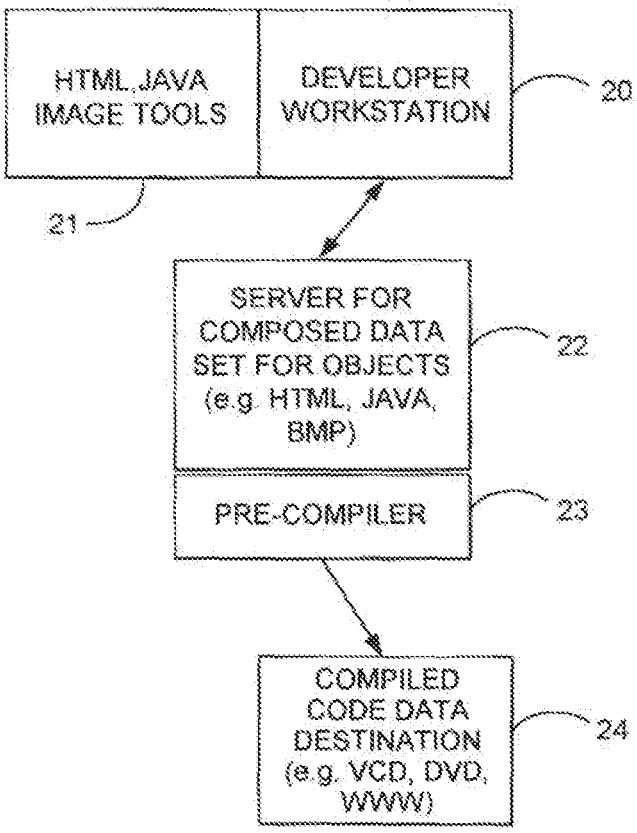

HTML,JAVA IMAGE TOOLS
DEVELOPER WORKSTATION
20
21
SERVER FOR COMPOSED DATA SET FOR OBJECTS (e.g. HTML, JAVA, BMP)
22
PRE-COMPILER
23
COMPILED CODE DATA DESTINATION (e.g. VCD, DVD, WWW)
24

FIG.2

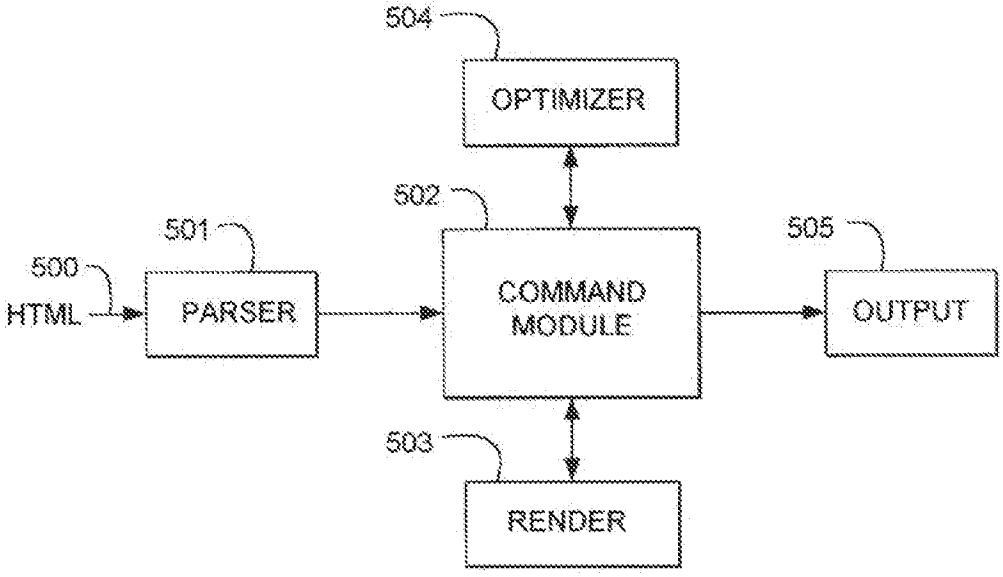

504
OPTIMIZER
502
501
505
500
HTML
PARSER
COMMAND MODULE
OUTPUT
503
RENDER

FIG. 3

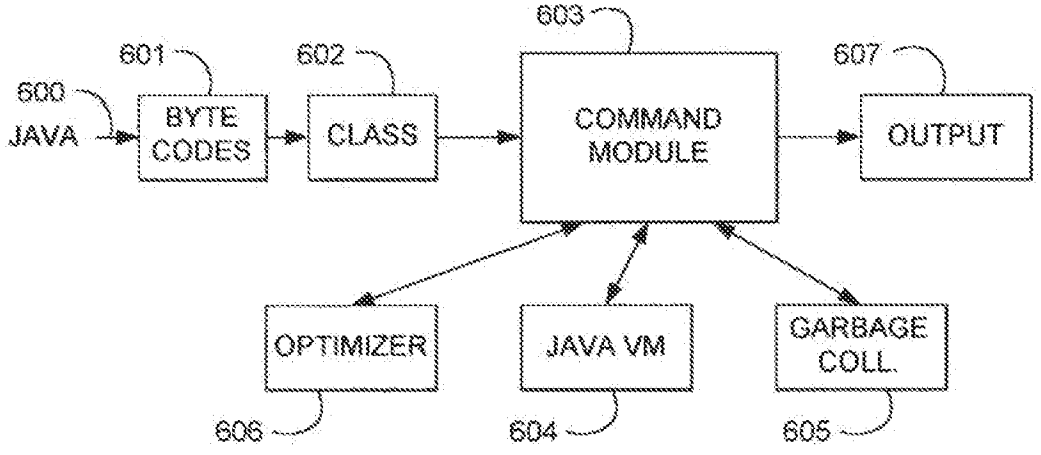

603
601
602
607
600
JAVA
BYTE CODES
CLASS
COMMAND MODULE
OUTPUT
OPTIMIZER
JAVA VM
GARBAGE COLL.
606
604
605

FIG.4

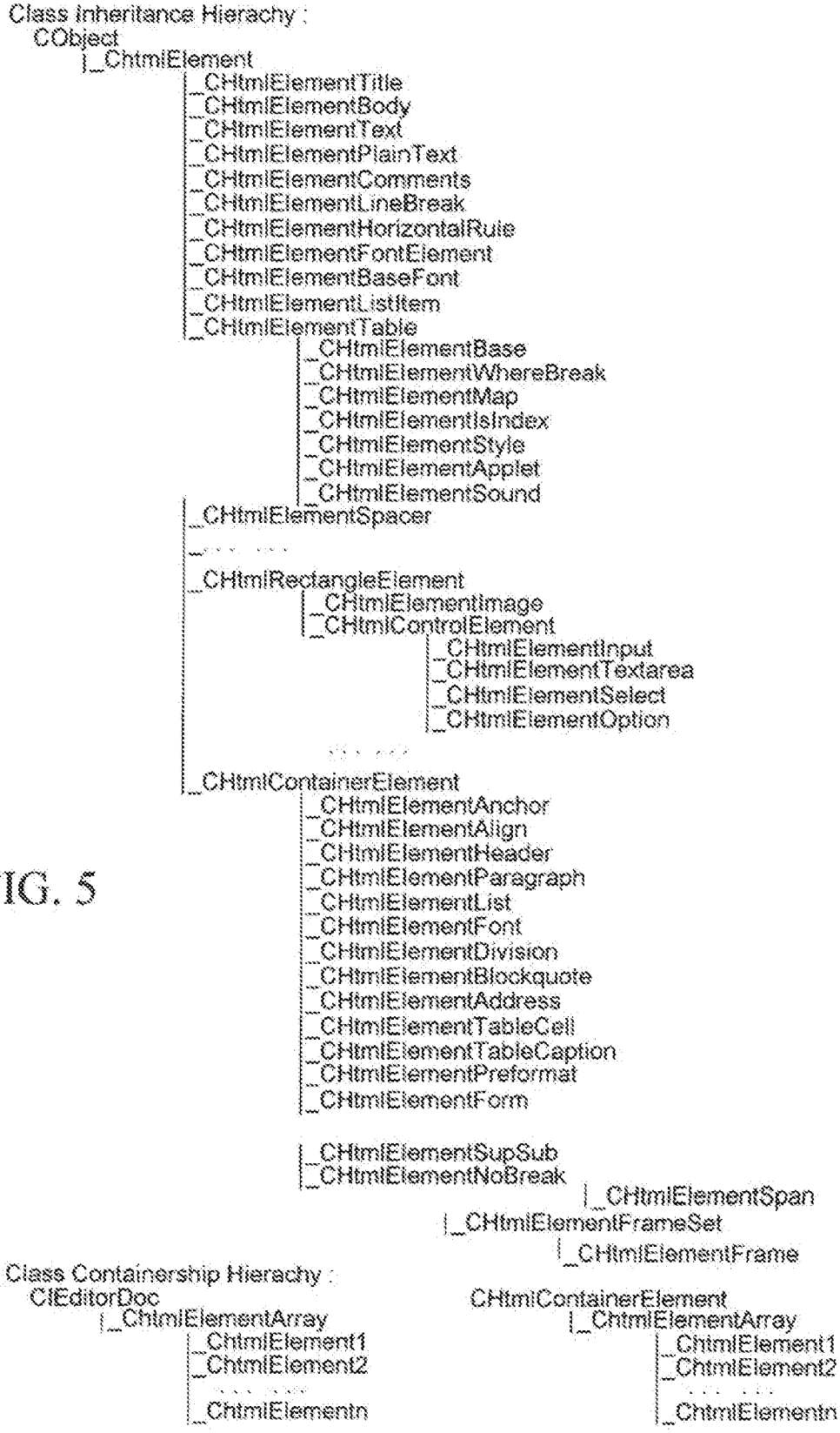

Class Inheritance Hierachy :
CObject
_ChtmlElement
_CHtmlElementTitle
_CHtmlElementBody
_CHtmlElementText
_CHtmlElementPlainText
_CHtmlElementComments
_CHtmlElementLineBreak
_CHtmlElementHorizontalRule
_CHtmlElementFontElement
_CHtmlElementBaseFont
_CHtmlElementListItem
_CHtmlElementTable
_CHtmlElementBase
_CHtmlElementWhereBreak
_CHtmlElementMap
_CHtmlElementIsIndex
_CHtmlElementStyle
_CHtmlElementApplet
_CHtmlElementSound
_CHtmlElementSpacer
_CHtmlRectangleElement
_CHtmlElementImage
_CHtmlControlElement
_CHtmlElementInput
_CHtmlElementTextarea
_CHtmlElementSelect
_CHtmlElementOption
_CHtmlContainerElement
_CHtmlElementAnchor
_CHtmlElementAlign
_CHtmlElementHeader
_CHtmlElementParagraph
_CHtmlElementList
_CHtmlElementFont
_CHtmlElementDivision
_CHtmlElementBlockquote
_CHtmlElementAddress
_CHtmlElementTableCell
_CHtmlElementTableCaption
_CHtmlElementPreformat
_CHtmlElementForm
_CHtmlElementSupSub
_CHtmlElementNoBreak
_CHtmlElementSpan
_CHtmlElementFrameSet
_CHtmlElementFrame
Class Containership Hierachy :
ClEditorDoc
_ChtmlElementArray
_ChtmlElement1
_ChtmlElement2
_ChtmlElementn
CHtmlContainerElement
_ChtmlElementArray
_ChtmlElement1
_ChtmlElement2
_ChtmlElementn

FIG. 5

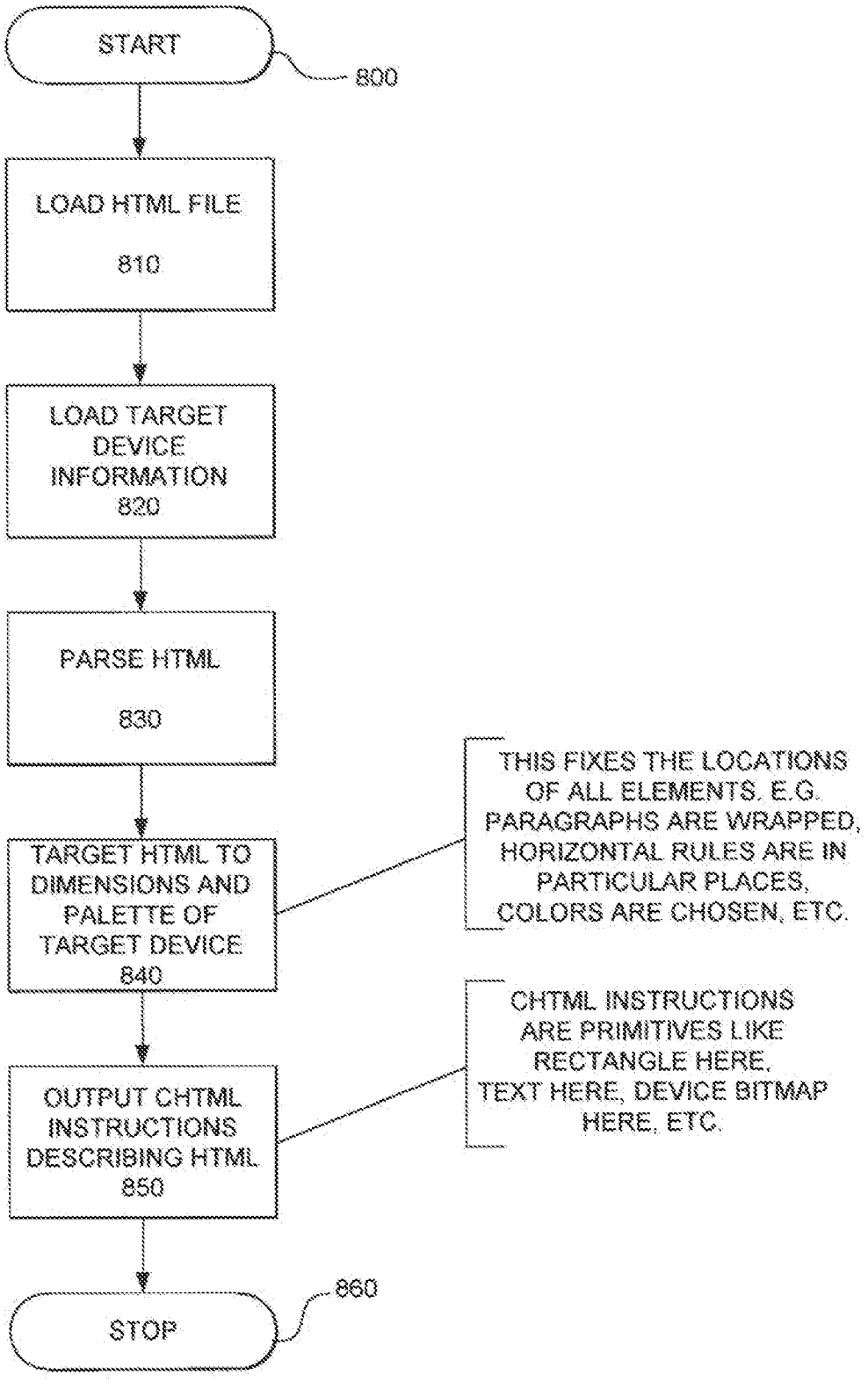

START
800
LOAD HTML FILE
810
LOAD TARGET DEVICE INFORMATION
820
PARSE HTML
830
TARGET HTML TO DIMENSIONS AND PALETTE OF TARGET DEVICE
840
THIS FIXES THE LOCATIONS OF ALL ELEMENTS. E.G. PARAGRAPHS ARE WRAPPED, HORIZONTAL RULES ARE IN PARTICULAR PLACES, COLORS ARE CHOSEN, ETC.
OUTPUT CHTML INSTRUCTIONS DESCRIBING HTML
850
CHTML INSTRUCTIONS ARE PRIMITIVES LIKE RECTANGLE HERE, TEXT HERE, DEVICE BITMAP HERE, ETC.
STOP
860

FIG. 6

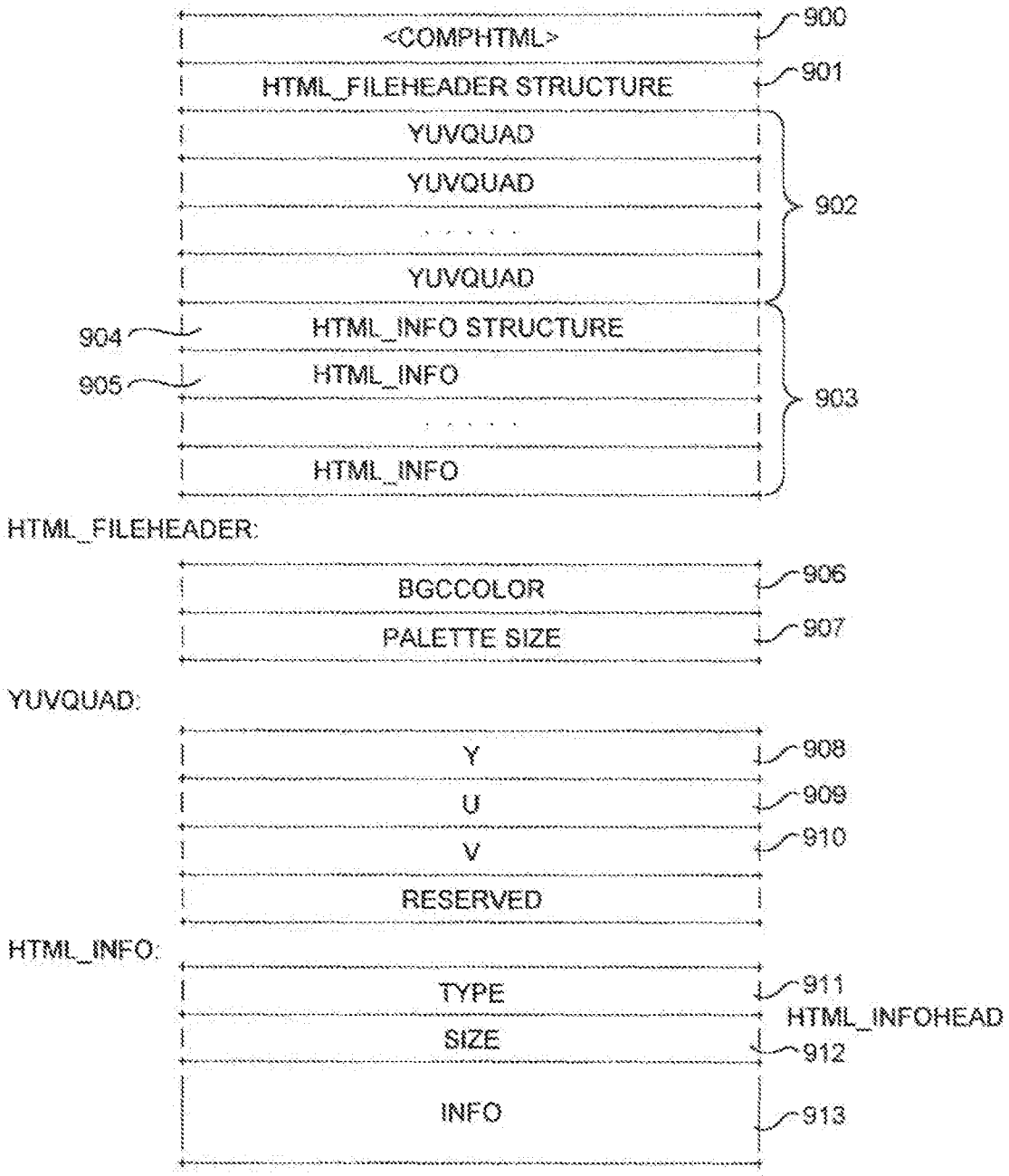

<COMPHTML>
900
HTML_FILEHEADER STRUCTURE
901
YUVQUAD
YUVQUAD
. . . . .
YUVQUAD
902
904
HTML_INFO STRUCTURE
905
HTML_INFO
. . . . .
HTML_INFO
903
HTML_FILEHEADER:
BGCCOLOR
906
PALETTE SIZE
907
YUVQUAD:
Y
908
U
909
V
910
RESERVED
HTML_INFO:
TYPE
911
HTML_INFOHEAD
SIZE
912
INFO
913

FIG. 7

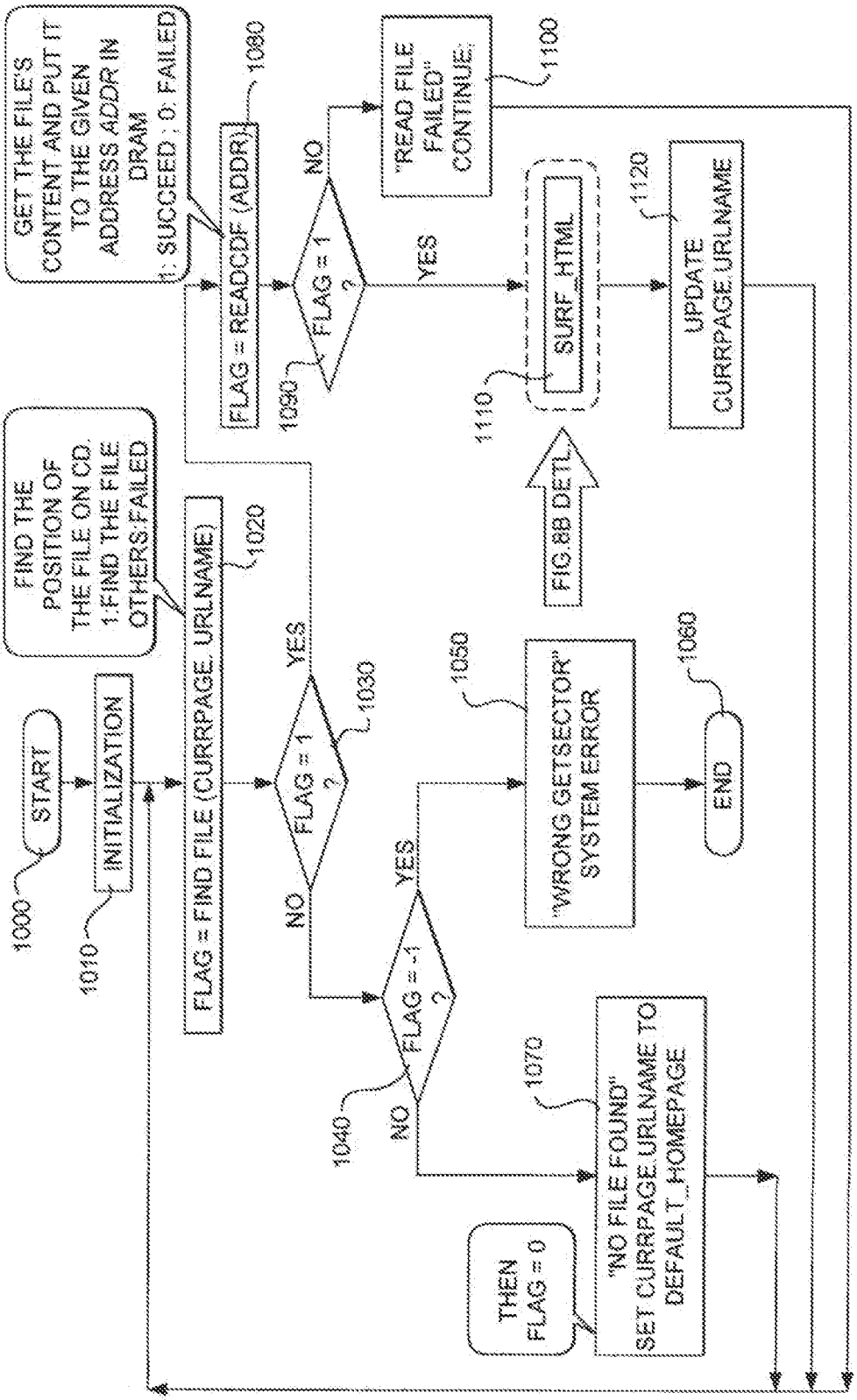

START
1000
INITIALIZATION
1010
FIND THE POSITION OF THE FILE ON CD. 1:FIND THE FILE OTHERS:FAILED
FLAG = FIND FILE (CURRPAGE.URLNAME)
1020
FLAG = 1 ?
1030
YES
NO
FLAG = -1 ?
1040
YES
NO
"WRONG GETSECTOR" SYSTEM ERROR
1050
END
1060
THEN FLAG = 0
"NO FILE FOUND" SET CURRPAGE.URLNAME TO DEFAULT_HOMEPAGE
1070
GET THE FILE'S CONTENT AND PUT IT TO THE GIVEN ADDRESS ADDR IN DRAM
1: SUCCEED ; 0: FAILED
FLAG = READCDF (ADDR)
1080
FLAG = 1 ?
1090
YES
NO
"READ FILE FAILED" CONTINUE;
1100
FIG.8B DETL
SURF_HTML
1110
UPDATE CURRPAGE.URLNAME
1120

FIG. 8A

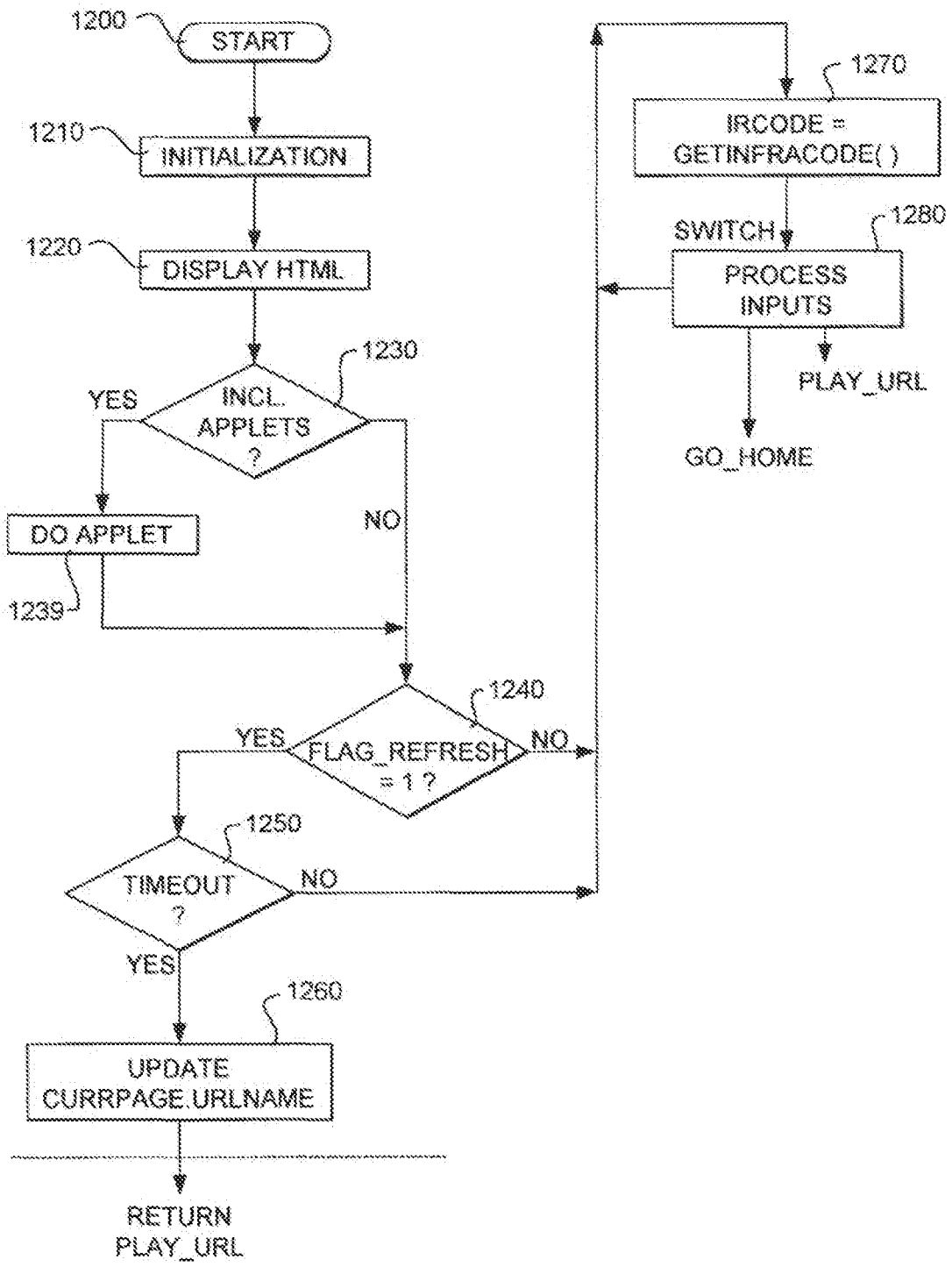

1200
START
1210
INITIALIZATION
1220
DISPLAY HTML
1230
INCL. APPLETS ?
YES
NO
DO APPLET
1239
1240
FLAG_REFRESH = 1 ?
YES
NO
1250
TIMEOUT ?
NO
YES
1260
UPDATE CURRPAGE.URLNAME
RETURN PLAY_URL
1270
IRCODE = GETINFRACODE( )
SWITCH
1280
PROCESS INPUTS
PLAY_URL
GO_HOME

FIG.8B

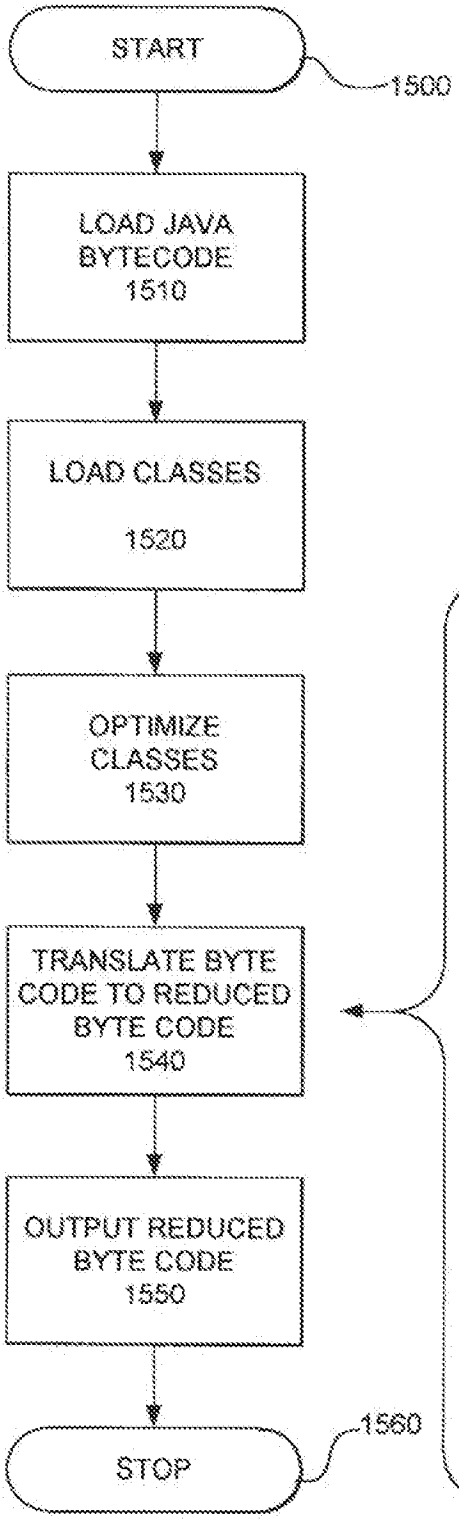

START
1500
LOAD JAVA BYTECODE 1510
LOAD CLASSES 1520
OPTIMIZE CLASSES 1530
TRANSLATE BYTE CODE TO REDUCED BYTE CODE 1540
OUTPUT REDUCED BYTE CODE 1550
STOP
1560

FIG. 9

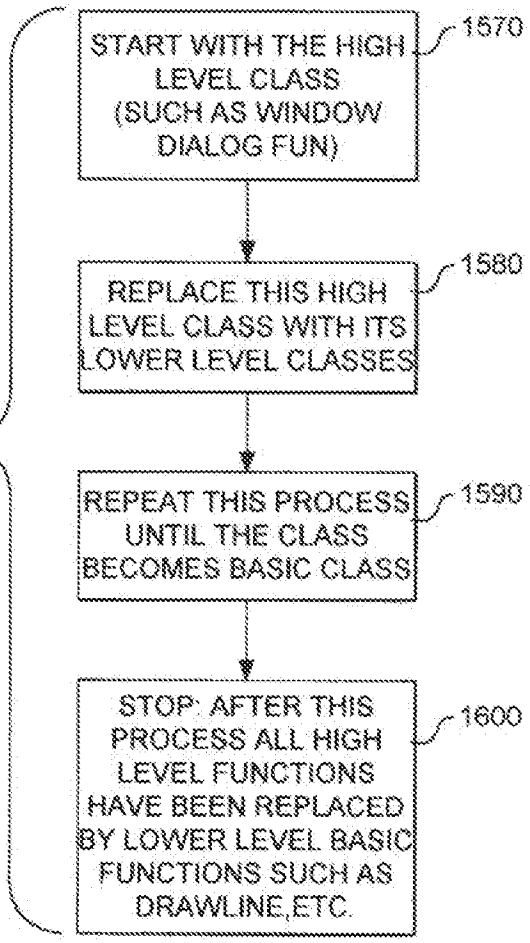

START WITH THE HIGH LEVEL CLASS (SUCH AS WINDOW DIALOG FUN)
1570
REPLACE THIS HIGH LEVEL CLASS WITH ITS LOWER LEVEL CLASSES
1580
REPEAT THIS PROCESS UNTIL THE CLASS BECOMES BASIC CLASS
1590
STOP: AFTER THIS PROCESS ALL HIGH LEVEL FUNCTIONS HAVE BEEN REPLACED BY LOWER LEVEL BASIC FUNCTIONS SUCH AS DRAWLINE,ETC.
1600

FIG. 9A

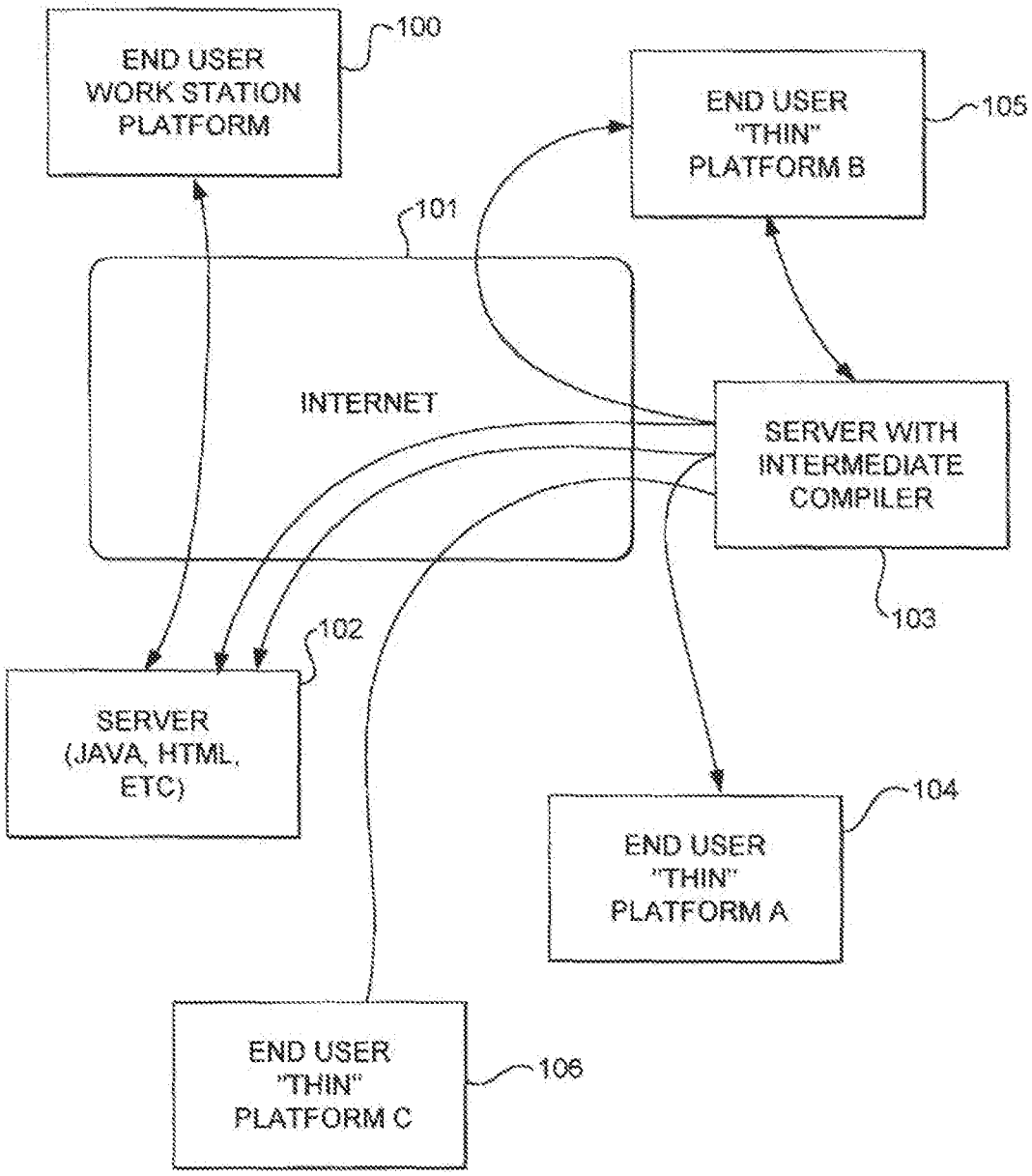

END USER WORK STATION PLATFORM
100
END USER "THIN" PLATFORM B
105
101
INTERNET
SERVER WITH INTERMEDIATE COMPILER
103
102
SERVER (JAVA, HTML, ETC)
104
END USER "THIN" PLATFORM A
END USER "THIN" PLATFORM C
106

FIG. 10

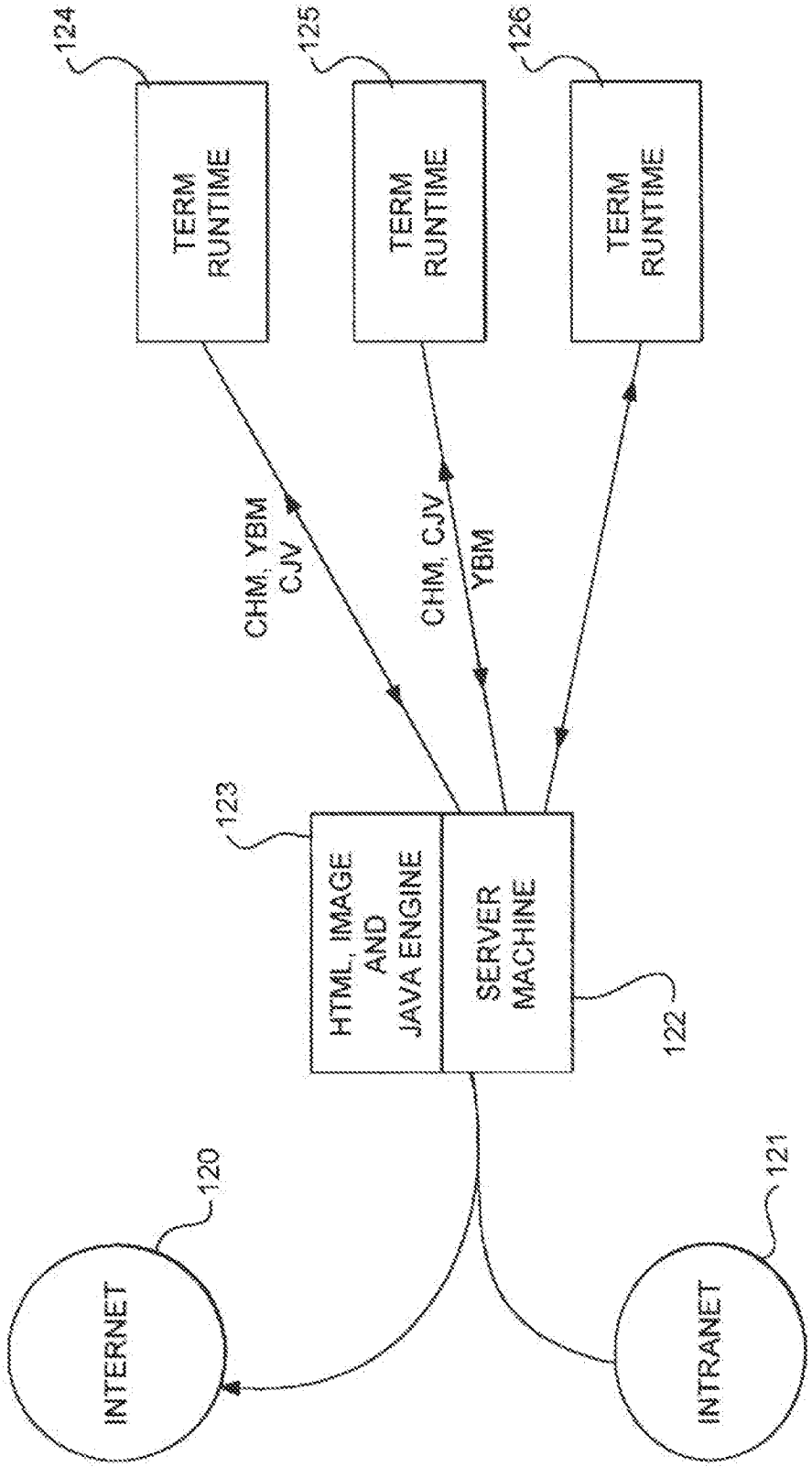

124
125
126
TERM RUNTIME
TERM RUNTIME
TERM RUNTIME
CHM, YBM CJV
CHM, CJV YBM
123
HTML, IMAGE AND JAVA ENGINE
SERVER MACHINE
122
120
121
INTERNET
INTRANET

FIG. 11

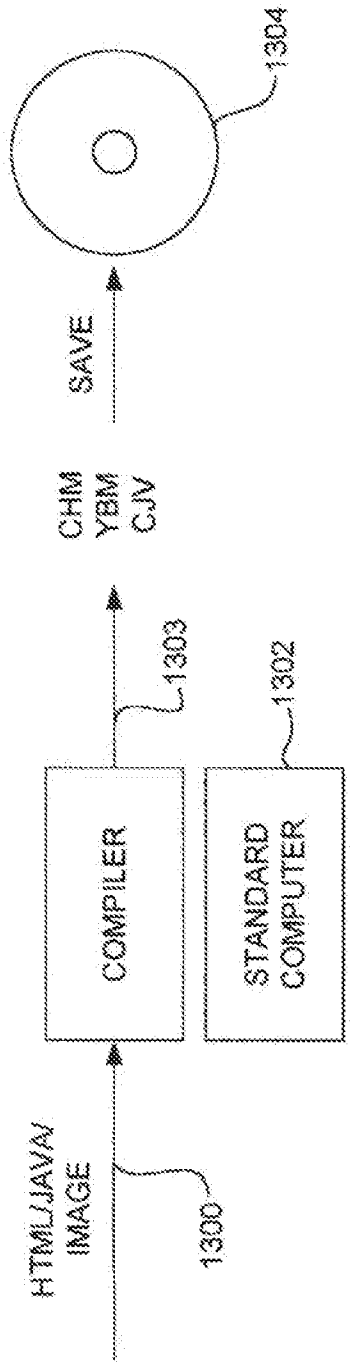


FIG. 12A

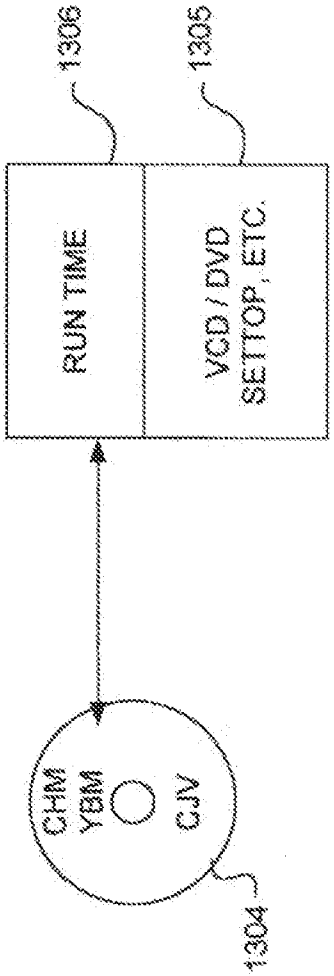


FIG. 12B

SYSTEM AND PROCESS FOR OBJECT RENDERING ON THIN CLIENT PLATFORMS

COPYRIGHT DISCLAIMER

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention generally relates to a method of providing full feature program processing according to a variety of standard language codes such as HTML, JAVA and other standard languages, for execution on a thin client platform. More particularly the invention relates to methods for compiling and rendering full feature standard HTML and JAVA programs into a format which is efficient for a limited processing resource platforms.

2. Description of Related Art

Standard HTML and JAVA programs, and other hypertext languages, are designed for computers having a significant amount of data processing resources, such as CPU speed and memory bandwidth, to run well. One feature of these object specifying languages is the ability to specify a graphic object for display using relative positioning. Relative positioning enables the display of the graphic object on displays having a wide range of dimensions, resolutions, and other display characteristics. However, relative positioning of graphic objects requires that the target device have computational resources to place the graphic object on the display at specific coordinates. Thus, there are a number of environments, such as TV set top boxes, hand held devices, digital video disk DVD players, compact video disk VCD players or thin network computer environments in which these standard object specifying languages are inefficient or impractical. The original HTML and JAVA programs run very slowly, or not at all, in these types of thin client environments. To solve these problems, simpler versions of HTML and JAVA have been proposed, which have resulted in scripting out some of the features. This trades off some of the nice functionality of HTML and JAVA, which have contributed to their wide acceptance. Furthermore, use in thin client environments of the huge number of files that are already specified according to these standards, is substantially limited.

SUMMARY OF THE INVENTION

The present invention provides a system and method for processing an Display object specified by an object specifying language such as HTML, JAVA or other languages relying on relative positioning, that require a rendering program utilizing a minimum set of resources, for use in a target device that has limited processing resources unsuited for storage and execution of the HTML rendering program, JAVA virtual machine, or other rendering engine for the standard. Thus, the invention can be characterized as a method for storing data concerning such an object that includes first receiving a data set specifying the object according to the object specifying language, translating the first data set to a second data set in an intermediate object language adapted for a second rendering program suitable for rendering by the target device that utilizes actual target display coordinates. The second data set is stored in a machine readable storage device, for later retrieval and execution by the thin client platform.

The object specifying language according to alternative embodiments comprises a HTML standard language or other hypertext mark up language, a JAVA standard language or other object oriented language that includes object specifying tools.

The invention also can be characterized as a method for sending data concerning such an object to a target device having limited processing resources. This method includes receiving the first data set specifying the object according to the first object specifying language, translating the first data set to a second data set in an intermediate object language, and then sending the second data set to the target device. The target device then renders the object by a rendering engine adapted for the intermediate object language. The step of sending the second data set includes sending the second data set across a packet switched network such as the Internet or the World Wide Web to the target device. Also, the step of translating according to one aspect of the invention includes sending the first data set across a packet switched network to a translation device, and executing a translation process on the translation device to generate the second data set. The second data set is then transferred from the translation device, to the target device, or alternatively from the translation device back to the source of the data, from which it is then forwarded to the target device.

According to other aspects of the invention, the step of translating the first data set includes first identifying the object specifying language of the first data set from among a set of object specifying languages, such as HTML and JAVA. Then, a translation process is selected according to the identified object specifying language.

According to yet another aspect of the invention, before the step of translating the steps of identifying the target device from among a set of target devices, and selecting a translation process according to the identified target device, are executed.

In yet another alternative of the present invention, a method for providing data to a target device is provided. This method includes requesting for the target device a first data set from a source of data, the first data set specifying the object according to the object specifying language; translating the first data set to a second data set in an intermediate language adapted for execution according to a second rendering program by the target device. The second data set is then sent to this target device. This allows a thin platform target device to request objects specified by full function HTML, JAVA and other object specifying languages, and have them automatically translated to a format suitable for rendering in the thin environment.

Thus, the present invention provides a method which uses a computer to automatically compile standard HTML, JAVA and other programs so that such programs can run both CPU and memory efficiently on a thin client platform such as a TV set top box, a VCD/DVD player, a hand held device, a network computer or an embedded computer. The automatic compilation maintains all the benefits of full feature HTML and JAVA or other language.

The significance of the invention is evident when it is considered that in the prior art, standard HTML and JAVA were reduced in features or special standards are created for the thin client environment. Thus according to the prior art

approaches, the standard programs and image files on the Internet need to be specially modified to meet the needs of special thin client devices. This is almost impossible considering the amount of HTML and JAVA formatted files on the Web. According to the invention each HTML file, compiled JAVA class file or other object specifying language data set is processed by a standard full feature HTML browser JAVA virtual machine, or other complementary rendering engine, optimized for a target platform on the fly, and then output into a set of display oriented language codes which can be easily executed and displayed on a thin client platform. Furthermore, the technique can use in general to speed up the HTML and JAVA computing in standard platforms.

Other aspects and advantages of the present invention can be seen upon review of the figures, the detailed description and the claims which follow.

BRIEF DESCRIPTION OF THE FIGURES

FIG. **1** is a simplified diagram of a end user thin platform for execution of a compiled code data source according to the present invention.

FIG. **2** is a simplified diagram of a user workstation and server for precompiling a composed data set according to the present invention.

FIG. **3** is a simplified diagram of a precompiler for a HTML formatted file.

FIG. **4** is a simplified diagram of a precompiler for a JAVA coded program.

FIG. **5** is a class inheritance hierarchy for a precompiler for HTML.

FIG. **6** is a flow chart for the HTML precompiler process.

FIG. **7** illustrates the compiled HTML structure according to one embodiment of the present invention.

FIGS. **8A-8B** illustrate a compiled HTML run time engine for execution on the thin platform according to the present invention.

FIG. **9** is a flow chart of the process for precompiling a JAVA program according to the present invention.

FIG. **9A** is a flow chart of one example process for translating the byte codes into a reduced byte code in the sequence of FIG. **9**.

FIG. **10** is a schematic diagram illustrating use of the present invention in the Internet environment.

FIG. **11** is a schematic diagram illustrating use of the present invention in a "network computer" environment.

FIG. **12A** is a schematic diagram illustrating use of the present invention in an off-line environment for producing a compiled format of the present invention and saving it to a storage medium.

FIG. **12B** illustrates the off-line environment in which the stored data is executed by thin platform.

DETAILED DESCRIPTION

A detailed description of preferred embodiments of the present invention is provided with respect to FIGS. **1–12A** and **12B**. FIGS. **1–2** illustrated simplified implementation of the present invention. FIGS. **3–9** and **9A** illustrate processes executed according to the present invention. FIGS. **10–12A** and **12B** illustrate the use of the present invention in the Internet environment or other packet switched network environment.

FIG. **1** illustrates a "thin" platform which includes a limited set of data processing resources represented by box **10**, a display **11**, and a "compiled code" rendering engine **12** for a display oriented language which relies on the data processing resources **10**. The end user platform **10** is coupled to a compiled code data source **13**. A compiled code data sources comprises, for example a VCD, a DVD, or other computer readable data storage device. Alternatively, the compiled code data source **13** consists of a connection to the World Wide Web or other packet switched or point-to-point network environment from which compiled code data is retrieved.

The limited data processing resources of the thin platform **10** include for example a microcontroller and limited memory. For example, 512k of RAM associated with a 8051 microcontroller, or a 66 MHz MIPS RISC CPU and 512k of dynamic RAM may be used in a representative thin platform. Other thin user platforms use low cost microprocessors with limited memory. In addition, other thin platforms may comprise high performance processors which have little resources available for use in rendering the compiled code data source. Coupled with the thin platform is a compiled code rendering engine **12**. This rendering engine **12** is a relatively compact program which runs efficiently on the thin platform data processing resources. The rendering engine translates the compiled code data source data set into a stream of data suitable for the display **11**. In this environment, the present invention is utilized by having the standard HTML or JAVA code preprocessed and compiled into a compiled HTML/JAVA format according to the present invention using the compiler engine described in more detail below on a more powerful computer. The compiled HTML/JAVA codes are saved on the storage media. A small compiled HTML/JAVA run time engine **12** is embedded or loaded into the thin client device. The run time engine **12** is used to play the compiled HTML/JAVA files on the thin platform **10**. This enables the use of a very small client to run full feature HTML or JAVA programs. The machine can be used both online, offline or in a hybrid mode.

FIG. **2** illustrates the environment in which the compiled code data is generated according to the present invention. Thus for example, a developer workstation **20** is coupled with image rendering tools such as HTML, JAVA, or other image tools **21**. The workstation **20** is coupled to a server for the composed data **22**. The server includes a precompiler **23** which takes the composed data and translates it into the compiled code data. Compiled code data is then sent to a destination **24** where it is stored or rendered as suits the needs of a particular environment. Thus for example, the destination may be a VCD, DVD or the World Wide Web.

According to the environment of FIG. **2** compiled HTML and JAVA "middleware" is implemented on an Internet server. Thus the thin set top box or other compiled code data destination **24** is coupled to the Internet/Intranet through the compiled HTML/JAVA middleware **22**, **23**. A small compiled HTML/JAVA run time engine is embedded in the thin destination device. All the HTML/JAVA files created in the workstation **20** go through the middleware server **22** to reach the thin client devices. The HTML/JAVA files are converted to the compiled format on the fly by the precompiler **23** on the middleware server **22**. The server **22** passes the compiled code onto the destination device. This allows for most software updates of precompiler techniques to be made in the server environment without the need to update the destination devices. Also, any changes in the run time engine that need to be executed in the destination device **24** can be provided through the link to the server **22**.

FIGS. **3** and **4** illustrate simplified diagrams of the precompilers for HTML and JAVA respectively. In FIG. **3**

standard HTML files are received at input **500** and applied to a HTML parser **501**. The output of the parser is applied to a command module **502** which includes a HTML rendering engine **503**, and memory resident HTML objects optimizing engine **504**. The output consists of the compiled HTML output engine **505** generates the output with simplified graphics primitives.

The basic class inheritance hierarchy for the HTML precompiling is shown in FIG. **5**. The process of translating a HTML file to the compiled HTML structure of the present invention is illustrated in FIG. **6**. The process begins at point **800** in FIG. **6**. The first step involves loading the HTML file into the rendering device. Next information concerning the target device is loaded (step **820**). The HTML file is then parsed by searching for HTML tags, and based on such tags creating the class structure of FIG. **5** (step **830**).

Using the parameters of the target device, and the parsing class structure set up after the parsing process, the algorithm does HTML rendering based on a class hierarchy adapted to the dimensions and palette of the target device (step **840**). This fixes the coordinates of all the graphic objects specified by the HTML code on the screen of the target device. For example, the paragraphs are word wrapped, horizontal rules are placed in particular places, the colors are chosen, and other device specific processes are executed.

After the rendering, all the display information is saved back into the class structure of FIG. **5**. Finally the process goes through the class hierarchy and outputs the rendering information in compiled HTML format (step **850**). The compiled HTML instructions are primitives that define rectangles, text, bitmaps and the like and their respective locations. After outputting the compiled instructions, the process is finished (step **860**).

A simplified pseudo code for the HTML compilation process is provided in Table 1.

TABLE 1

Copyright ExReach 1997

```
function convert_html (input : pointer) : chtmlfile;
    // this takes a pointer to an HTML file and translates it into a CHTML binary file.
begin
    deviceInfo := LoadDeviceInfo( );    // Loads size and colors of target device
    Parse HTML file                     // use a parser to break the HTML file up into
                                        // tags represented in a fashion suitable for display
    For each HTML tag (<IMG . . . > = 1 tag, <P> a paragraph </P> = 2 tag),
select a sequence of CHTML instructions to render the tag on the output device.
As instructions are selected, colors and positioning are optimized based on the
device size and palette.
        CHTML instructions include:
            TITLE       string
            TEXT        formatted text at a specific position,
                        complex formatting will
                        require multiple CHTML TEXT instructions
            IMAGE       image information including image-map,
                        animation info, image data
            ANCHOR      HTML reference
        Basic geometric instructions such as SQUARE, FILLEDSQUARE, CIRCLE,
        FILLEDCIRCLE, and LINE, permit the complex rendering required by some
        HTML instructions to be decomposed into basic drawing instructions. For
        example, the bullets in front of lists can be described in CHTML instructions
        as squares and circles at specific locations.
        CHTML instructions including TEXT and IMAGE instructions can be
        contained within anchors. The CHTML compiler must properly code all
        instructions to indicate if an instruction is contained in an anchor.
    The CHTML instructions can then be written to the output file along with some header
    information.
end;
```

Table 2 sets forth the data structure for the precompiling process.

TABLE 2

Copyright ExReach 1997

```
/* HTML font structure */
typedef struct tagHTMLFont
{
    char name[64];
    int size;
    int bold;
    int italic;
    int underline;
    int strikeout;
} HTMLFont;
/* FG point structure */
typedef struct tagFGPoint
```

TABLE 2-continued


```
{
    int fX;
    int fY;
} FGPoint;
/* FG rectangle structure */
typedef struct tagFGRect
{
    int fLeft;
    int fTop;
    int fRight;
    int fBottom;
} FGRect;
/* html node types, used by fType attribute in HTML_InfoHead structure */
#define HTML_TYPE_TITLE    0          /* title of the html page */
#define HTML_TYPE_TEXT     1          /* text node */
#define HTML_TYPE_CHINESE  2          /* chinese text node */
#define HTML_TYPE_IMAGE    3          /* image node */
#define HTML_TYPE_SQUARE   4          /* square frame */
#define HTML_TYPE_FILLEDSQUARE 5      /* filled square */
#define HTML_TYPE_CIRCLE   6          /* circle frame */
#define HTML_TYPE_FILLEDCIRCLE 7      /* filled circle */
#define HTML_TYPE_LINE     8          /* line */
#define HTML_TYPE_ANCHOR   9          /* anchor node */
#define HTML_TYPE_ANIMATION   10      /* animation node */
#define HTML_TYPE_MAPAREA     11      /* client side image map area node */
/* header info of compiled html file */
typedef struct tagHTML_FileHead
{
    unsigned int fBgColor;                /* background color index */
    unsigned int fPaletteSize;            /* size of palette */
} HTML_FileHead;
/* header info of each html node */
typedef struct tagHTML_InfoHead
{
    unsigned int fType;                   /* type of the node */
    unsigned int fSize;                   /* size of htmlInfo */
} HTML_InfoHead;
/* html info structure */
typedef struct tagHTML_Info
{
    HTML_InfoHead htmlHead;               /* header info */
    unsigned char htmlInfo[1];            /* info of the html node */
} HTML_Info;
/* html title structure */
typedef struct tagHTML_Title
{
    unsigned int textLen;                 /* length of text buffer */
    char textBuffer[1];                   /* content of text buffer */
} HTML_Title;
/* html text structure */
typedef struct tagHTML_Text
{
    FGPoint dispPos;                      /* display coordinates */
    int anchorID;                         /* anchor id if it's inside an anchor, -1 if not */
    HTMLFont textFont;                    /* font of the text */
    unsigned int textColor;               /* color index of the text */
    unsigned int textLen;                 /* length of text buffer */
    char textBuffer[1];                   /* content of text buffer */
} HTML_Text;
/* html chinese structure */
typedef struct tagHTML_Chinese
{
    FGPNT dispPos;                        /* display coordinates */
    int anchorID;                         /* anchor id if it's inside an anchor, -1 if not */
    unsigned int textColor;               /* color index of the text */
    unsigned int buffLen;                 /* length of the bitmap buffer (16* 16) */
    char textBuffer[1];                   /* content of text buffer */
} HTML_Chinese;
/* html image structure */
typedef struct tagHTML_Image
{
    FGRect dispPos;                       /* display coordinates */
    int anchorID;                         /* anchor id if it's inside an anchor, -1 if not */
```

TABLE 2-continued


```
    int animationID; /* animation id if it supports animation, -1 if not */
    int animationDelay;                    /* delay time for animation */
    char mapName[64];                /* name of client side image map, empty if no
image map */
    void *data;                              /* used to store image
data */
    unsigned int fnameLen;               /* length of the image file name */
    char fname[1];                         /* image filename */
} HTML_Image;
/* square structure */
typedef struct tagHTML_Square
{
    FGRect dispPos;                        /* display coordinates */
    unsigned int borderColor; /* border color index */
} HTML_Square;
/* filled square structure */
typedef struct tagHTML_FilledSquare
{
    FGRect dispPos;                        /* display coordinates */
    unsigned int brushColor; /* the inside color index */
} HTML_FilledSquare;
/* circle structure */
typedef struct tagHTML_Circle
{
    FGRect dispPos;                        /* display coordinates */
    unsigned int borderColor; /* border color index */
} HTML_Circle;
/* circle structure */
typedef struct tagHTML_FilledCircle
{
    FGRect dispPos;                        /* display coordinates */
    unsigned int brushColor; /* the inside color index */
} HTML_FilledCircle;
/* line structure */
typedef struct tagHTML_Line
{
    FGPoint startPos;                    /* line starting position */
    FGPoint endPos;                      /* line end position */
    int style;                           /* style of the line (solid, dashed, dotted,
etc) */
    unsigned int penColor;               /* pen color index */
} HTML_Line;
/* anchor structure */
typedef struct tagHTML_Anchor
{
    int anchorID;                        /* id of the anchor */
    unsigned int hrefLen;                /* length of href */
    char href[1];                        /* url of the anchor */
} HTML_Anchor;
/* animation structure */
typedef struct tagHTML_Animation
{
    int animationID;                     /* id of the animation */
    unsigned int frameTotal; /* total number of animation frames */
    long runtime;                        /* animation runtime */
} HTML_Animation;
#define SHAPE_RECTANGLE 0
#define SHAPE_CIRCLE 1
#define SHAPE_POLY 2
/* image map area structure */
typedef struct tagHTML_MapArea
{
    char mapName[64];                    /* name of client side image map
*/
    int shape;                           /* shape of the area */
    int numVer;                          /* number of vertix */
    int coord[6][2];                 /* coordinates */
    unsigned int hrefLen;            /* length of href */
    char href[1];                        /* url the area pointed to */
} HTML_MapArea;
```

An example routine for reading this file into the thin platform memory follows in Table 3.

TABLE 3


```
reading this file.
#define BLOCK_SIZE 256
/* returns number of nodes */
long read_chm(const char *filename,          /* input: chm file name */
    HTML_Info ***ppNodeList, /* output: array of HTML_Info */
                                                    including sentences. */
    YUVQUAD **ppPalette,                            /* output: page palette */
    unsigned int *palette_size)                     /* output: palette size */
{
    int fd;
    char head[12];
    long total_nodes = 0;
    long max_nodes = 0;
    HTML_FileHead myFileHead;
    HTML_InfoHead myInfoHead;
    HTML_Info *pNodeInfo;
    void *pNodeData;
    long i;
    HTML_InfoHead *plHead;
    if (!ppNodeList || !ppPalette || !palette_size)
        return 0;
    (*ppNodeList) = NULL;
    (*ppPalette) = NULL;
    (*palette_size) = 0
    /* open file */
    fd = _open(filename, _O_BINARY|_O_RDONLY);
    if(fd < 0)
        return 0;
    /* read header and check for file type */
    if (_read(fd, head, 10) != 10)
    {
        _close(fd);
        return 0;
    }
    if (strncmp(head, "rCOMPHTML1.", 10))
    {
        _close(fd);
        return 0;
    }
    /* read file header */
    if(_read(fd, &myFileHead, sizeof(HTML_FileHead)) !=
sizeof(HTML_FileHead))
    {
        _close(fd);
        return 0;
    }
    (*palette_size) = myFileHead.paletteSize;
    /* read the palette */
    if ((*palette_size) > 0)
    {
        (*ppPalette) = (YUVQUAD *)malloc(sizeof(YUVQUAD)*
(*palette_size));
        if (_read(fd, (*ppPalette), sizeof(YUVQUAD) * (*palette_size))
            != (int) (sizeof(YUVQUAD) * (*palette_size)))
        {
            _close(fd);
            return 0;
        }
    }
    /* read sections along with other html nodes */
    while (1)
    {
        if (_read(fd, &myInfoHead, sizeof(HTML_InfoHead))
            != sizeof(HTML_InfoHead))
        {
            break;
        }
        if (myInfoHead.hSize > 0)
        {
            pNodeInfo = (HTML_Info *) malloc(myInfoHead.hSize +
sizeof(HTML_InfoHead));
            if (!pNodeInfo)
                break;
            memcpy(pNodeInfo, &myInfoHead,
sizeof(HTML_InfoHead));
            if (_read(fd, &pNodeInfo[sizeof(HTML_InfoHead)],
myInfoHead.hSize)
```

TABLE 3-continued


```
                    != (int)sizeof(pHead.hSize)
                {
                    break;
                }
                /* check if we need to do memory allocation */
                if (total_nodes >= max_nodes)
                {
                    if(!max_nodes)
                    {
                        /* no node in the list yet */
                        (*ppNodeList) = (HTML_Info **)
malloc(
                            sizeof(HTML_Info *)*
BLOCK_SIZE);
                    }
                    else
                    {
                        (*ppNodeList) = (HTML_Info **)
realloc(*ppNodeList,
                            max_nodes + sizeof(HTML_Info
*) * BLOCK_SIZE);
                    }
                    if (!(*ppNodeList))
                        break;
                    max_nodes += BLOCK_SIZE;
                }
                (*ppNodeList)[total_nodes] = pNodeInfo;
                total_nodes++;
            }
        }
        _close(fh);
        /* test our data */
        for (i = 0; i < total_nodes; i++)
        {
            pNodeInfo = (*ppNodeList)[i];
            pHead = (HTML_InfoHead *) pNodeInfo;
            pNodeData = pNodeInfo + sizeof(HTML_InfoHead);
            if(pHead->hType == HTML_TYPE_TEXT)
            {
                HTML_Text *pText = (HTML_Text *) pNodeData;
            }
            else if(pHead->hType == HTML_TYPE_IMAGE)
            {
                HTML_Image *pImage = (HTML_Image *) pNodeData;
                if (pImage->fnameLen > 0)
                {
                    /* load the image file */
                    pImage->hData = load_vbm(pImage->fname);
                }
            }
            else if (pHead->hType == HTML_TYPE_ANCHOR)
            {
                HTML_Anchor *pAnchor = (HTML_Anchor *)
pNodeData;
            }
            else if(pHead->hType == HTML_TYPE_ANIMATION)
            {
                HTML_Animation *pAnimation = (HTML_Animation *)
pNodeData;
            }
            else if(pHead->hType == HTML_TYPE_MAPAREA)
            {
                HTML_MapArea *pMapArea = (HTML_MapArea *)
pNodeData;
            }
            else if (pHead->hType == HTML_TYPE_LINE)
            {
                HTML_Line *pLine = (HTML_Line *) pNodeData;
            }
            else if(pHead->hType == HTML_TYPE_SQUARE)
            {
                HTML_Square *pSquare = (HTML_Square *) pNodeData;
            }
            else if(pHead->hType == HTML_TYPE_CIRCLE)
            {
                HTML_Circle *pCircle = (HTML_Circle *) pNodeData;
            }
```

TABLE 3-continued


```
            else if (pHead->nType == HTML_TYPE_FILLEDSQUARE)
            {
                HTML_FilledSquare *pFilledSquare =
(HTML_FilledSquare *) pNodeData;
            }
            else if (pHead->nType == HTML_TYPE_FILLEDCIRCLE)
            {
                HTML_FilledCircle *pFilledCircle = (HTML_FilledCircle
*) pNodeData;
            }
            else if(pHead->nType == HTML_TYPE_TITLE)
            {
                HTML_Title *pTitle = (HTML_Title *) pNodeData;
            }
    }
    return total_nodes;
}
```

The compiled HTML file structure is set forth in FIG. **7** as described in Table 2. The file structure begins with a ten character string COMPHTML **900**. This string is followed by a HTML file header structure **901**. After the file header structure, a YUV color palette is set forth in the structure **902** this consists of an array of YUVQUAD values for the target device. After the palette array, a list **903** of HTML information structures follows. Usually the first HTML information structure **904** consists of a title. Next, a refresh element typically follows at point **905**. This is optional. Next in the line is a background color and background images if they are used in this image. After that, a list of display elements is provided in proper order. The anchor node for the HTML file is always in front of the nodes that it contains. An animation node is always right before the animation image frames start. The image area nodes usually appear at the head of the list.

The HTML file header structure includes a first value BgColor at point **906** followed by palette size parameters for the target device at point **907**. The YUVQUAD values in the color palette consist of a four word structure specifying the Y, U, and V values for the particular pixel at points **908–910**. The HTML information structures in the list **903** consist of a type field **911**, a size field **912**, and the information which supports the type at field **913**. The type structures can be a HTML_Title, HTML_Text, HTML_Chinese, HTML_Xxge, HTML_Square, HTML_FilledSquare, HTML_Circle, HTML_FilledCircle, HTML_Line, HTML_Author, HTML_Animation, . . .

Functions that would enable a thin platform to support viewing of HTML-based content pre-compiled according to the present invention includes the following:

General graphics functions

- int DrawPoint (int x, int y, COLOR color, MODE mode);
- int DrawLine (int x1, int y1, int x2, int y2, COLOR color, MODE mode);
- int DrawRectangle(int x1, int y1, int x2, int y2, COLOR color, MODE mode);
- int FillRectangle(int x1, int y1, int x2, int y2, COLOR color, MODE mode);
- int ClearScreen(COLOR color);

Color palette

- int ChangeYUVColorPalette( );

Bitmap function

- int BitBlt(int dst_x1, int dst_y1, int dst_x2, int dst_y2, unsigned char *bitmap, MODE mode);

String drawing functions

- int GetStringWidth(char *str, int len);
- int GetStringHeight(char *str, int len);
- int DrawStringOnScreen(int x, int y, char *str, int len, COLOR color, MODE mode);

Explanation

- All (x, y) coordinates are based on the screen resolution of the target display device (e.g. 320x240 pixels).
- COLOR is specified as an index to a palette.
- MODE defines how new pixels replace currently displayed pixels (COPY, XOR, OR, AND).
- Minimum support for DrawLine is a horizontal or vertical straight line, although it would be nice to have support for diagonal lines.
- The ChangeYUVColorPalette function is used for every page.
- BitBlt uses (x1, y1) and (x2, y2) for scaling but it is not a requirement to have this scaling functionality.
- String functions are used for English text output only. Bitmaps are used for Chinese characters.

FIGS. **8A** and **8B** set forth the run time engine suitable for execution on a thin client platform for display of the compiled HTML material which includes the function outlined above in the "display" step **1220** of FIG. **8B**.

The process of FIG. **8A** starts at block **1000**. The run time engine is initialized on the client platform by loading the appropriate elements of the run time engine and other processes known in the art (step **1010**). The next step involves identifying the position of the file, such as on the source CD or other location from which the file is to be retrieved and setting a flag (step **1020**). The flag is tested at step **1030**. If the flag is not set, then the algorithm branches to block **1040** at which the flag is tested to determine whether it is −1 or not. If the flag is −1, then the algorithm determines that a system error has occurred (step **1050**) and the process ends at step **1060**. If the flag at step **1040** is not −1, then the file has not been found (step **1070**). Thus after step **1070** the algorithm returns to step **1020** to find the next file or retry.

If at step **1030**, the flag is set to 1 indicating that the file was found, then the content of the file is retrieved using a program like that in Table 3, and it is stored at a specified address. A flag is returned if this process succeeds set equal to 1 otherwise it is set equal to 0 (step **1080**). Next the flag is tested (step **1090**). If the flag is not equal to 1, the reading

of the file failed (step **1100**). The process then returns to step **1020** to find the next file or retry.

If the flag is set to 1, indicating that the file has been successfully loaded into the dynamic RAM of the target device, then the "Surf_HTML" process is executed (step **1110**). The details of this process are illustrated in FIG. **8B**. Next the current page URL name is updated according to the HTML process (step **1120**). After updating the current URL name, the process returns to step **1020** to find the next file.

FIG. **8B** illustrates the "Surf_HTML" process of step **1110** in FIG. **8A**. This process starts at point **1200**. The first part is initialization step **1210**. A display routine is executed at step **1220** having the fixed coordinate functions of the precompiled HTML data set. First, the process determines whether applets are included in the file (step **1230**). If they are included, then the applet is executed (step **1240**). If no applets are included or after execution of the applet, then a refresh flag is tested (step **1240**). If the flag is equal to 1, then it is tested whether a timeout has occurred (step **1250**). If a timeout has occurred, then the current page is updated (step **1260**) and the process returns set **1210** of FIG. **8B**, for example.

If at block **1240** the refresh flag was not equal to 1, or at block **1250** the timeout had not expired, then the process proceeds to step **1270** to get a user supplied input code such as an infrared input signal provided by a remote control at the target device code. In response to the code, a variety of process are executed as suits a particular target platform to handle the user inputs (step **1280**). The process returns a GO_HOME, or a PLAY_URL command, for example, which result in returning the user to a home web page or to a current URL, respectively. Alternatively the process loops to step **1270** for a next input code.

As mentioned above, FIG. **4** illustrates the JAVA precompiler according to the present invention. The JAVA precompiler receives standard full feature JAVA byte codes as input on line **600**. Byte codes are parsed at block **601**. A JAVA class loader is then executed at block **602**. The classes are loaded into a command module **603** which coordinates operations of a JAVA virtual machine **604**, a JAVA garbage collection module **605**, and a JAVA objects memory mapping optimizing engine **606**. The output is applied by block **607** which consists of a compiled JAVA bytecode format according to the present invention.

The process is illustrated in FIG. **9** beginning at block **1500**. First the JAVA bytecode file is loaded (block **1510**). Next, the JAVA classes are loaded based on the interpretation of the bytecode (step **1520**). Next the classes are optimized at step **1530**. After optimizing the classes, the byte codes are translated to a reduced bytecode (step **1540**). Finally the reduced bytecode is supplied (step **1550**) and the algorithm stops at step **1560**. Basically the process receives a JAVA source code file which usually has the format of a text file with the extension JAVA. The JAVA compiler includes a JAVA virtual machine plus compiler classes such as SUN-TOOLS.JAVAC which are commercially available from Sun Micro Systems. The JAVA class file is parsed which typically consists of byte codes with the extension .CLASS. A class loader consists of a parser and bytecode verifier and processes other class files. The class structures are processed according to the JAVA virtual machine specification, such as the constant pool, the method tables, and the like. An interpreter and compiler are then executed. The JAVA virtual machine executes byte codes in methods and outputs compiled JAVA class files starting with "Main". The process of loading and verifying classes involves first finding a class. If the class is already loaded a read pointer to the class is returned, if not, the class is found from the user specified class path or directory, in this case a flash memory chunk. After finding the class, the next step is executed. This involves loading the bytes from the class file. Next, class file bytes are put into a class structure suitable for run time use, as defined by the JAVA virtual machine specification. The process recursively loads and links the class to its super classes. Various checks and initializations are executed to verify and prepare the routine for execution. Next, initialization is executed for the method of the class. First the process ensures that all the super classes are initialized, and then cause the initialization method for the class. Finally, the class is resolved by resolving a constant pool entry the first time it is encountered. A method is executed with the interpreter and compiler by finding the method. The method may be in the current class, its super class or other classes as specified. A frame is created for the method, including a stack, local variables and a program counter. The process starts executing the bytecode instructions. The instructions can be stack operations, branch statements, loading/storing values, from/to the local variables or constant pool items, or invoking other methods. When an invoked method is a native function, the implemented platform dependent function is executed.

In FIG. **9A**, the process of translating JAVA byte codes into compiled byte codes (step **1504** of FIG. **9**) is illustrated. According to the process FIG. **9A**, the high level class byte codes are parsed from the sequence. For example, Windows dialog functions are found (**1570**). The high level class is replaced with its lower level classes (**1580**). This process is repeated until all the classes in the file become basic classes (**1590**). After this process, all the high level functions have been replaced by lower level level basic functions, such as draw a line, etc. (**1600**).

JAVA byte codes in classes include a number of high level object specifying functions such as a window drawing function and other tool sets. According to the present invention, these classes are rendered by the precompiler into a set of specific coordinate functions such as those outlined above in connection with the HTML precompiler. By precompiling the object specifying functions of the JAVA byte code data set, significant processing resources are freed up on the thin client platform for executing the other programs carried in a JAVA byte code file. Furthermore, the amount of memory required to store the run time engine and JAVA class file for the thin client platform according to the present invention which is suitable for running a JAVA byte code file is substantially reduced.

FIG. **10** illustrates one environment in which use of the present invention is advantageous. In particular, in the Internet environment a wide variety of platforms are implemented. For example, an end user workstation platform **100** is coupled to the Internet **101**. An Internet server platform **102** is also coupled to the Internet **101** and includes storage for JAVA data sets, HTML data sets, and other image files. A server **103** with an intermediate compiler according to the present invention for one or more of the data sets available in the Internet is coupled to the Internet **101** as well. A variety of "thin" platforms are also coupled to the Internet and/or the server **103**. For example, an end user thin platform A **104** is coupled to the server **103**. End user thin platform B **105** is coupled to the server **103** and to the Internet **101**. End user thin platform C **106** is coupled to the Internet **101** and via the Internet all the other platforms in the network. A variety of scenarios are thus instituted. The source of data sets for end user platform C **106** consists of the World Wide Web. When it requests a file from server

102, the file is first transferred to the intermediate compiler at server 103, and from server 103 to the end user platform 106. End user platform A 104 is coupled directly to the server 103. When it makes a request for a file, the request is transmitted to the server 103, which retrieves the file from its source at server 102, translates it to the compiled version and sends it to platform A 104. End user platform B is coupled to both the server 103 and to the Internet 101. Thus, it is capable of requesting files directly from server 102. The server 102 transmits the file to server 103 from which the translated compiled version is sent to platform B 105. Alternatively, platform B may request a file directly from server 103 which performs all retrieval and processing functions on behalf of platform B.

FIG. 11 illustrates an alternative environment for the present invention. For example, the Internet 120 and an Intranet 121 are connected together. A server 122 is coupled to the Intranet 121 and the Internet 120. The server 122 includes the HTML and JAVA intermediate compiling engines according to the present invention as represented by block 123. The server 122 acts as a source of precompiled data sets for thin client platforms 124, 125 and 126 each of which has a simplified run time engine suitable for the compiled data sets. Thus the powerful HTML/JAVA engine resides on the network server 122. The thin network computers 124, 125, 126 are connected to the server have only the simplified run time engine for the compiled image set. Thus, very small computing power is required for executing the display. Thus computing tasks are done using the network server, but displayed on a thin network computer terminals 124-126.

FIGS. 12A and 12B illustrate the off-line environment for use of the present invention. In FIG. 12A, the production of the compiled files is illustrated. Thus, a standard object file, such as an HTML or JAVA image, is input online 1300 to a compiler 1301 which runs on a standard computer 1302. The output of the compiler on line 1303 is the compiled bitmap, compiled HTML or compiled JAVA formatted file. This file is then saved on a non-volatile storage medium such as a compact disk, video compact disk or other storage medium represented by the disk 1304.

FIG. 12B illustrates the reading of the data from the disk 1304 and a thin client such as a VCD box, a DVD box or a set top box 1305. The run time engine 1306 for the compiled data is provided on the thin platform 1305.

Thus, off-line fill feature HTML and JAVA processing is provided for a run time environment on a very thin client such as a VCD/DVD player. The standard HTML/JAVA objects are pre-processed and compiled into the compiled format using the compiler engine 1301 on a more powerful computer 1302. The compiled files are saved on a storage medium such as a floppy disk, hard drive, a CD-ROM, a VCD, or a DVD disk. A small compiled run time engine is embedded or loaded into the thin client device. The run time engine is used to play the compiled files. This enables use of a very small client for running full feature HTML and JAVA programs. Thus, the machine can be used in both online, and off-line modes, or in a hybrid mode.

The foregoing description of a preferred embodiment of the invention has been presented for purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Obviously, many modifications and variations will be apparent to practitioners skilled in this art. It is intended that the scope of the invention be defined by the following claims and their equivalents.

What is claimed is:

1. A method of translating a document on a first device for use on a second device, the document being in a standard HTML language, the method comprising:

reading the document;

reading a profile describing characteristics of the second device, the profile including a display resolution and a supported image format; and

translating the document on the first device according to the profile, the translating including

retrieving a plurality of images referenced by the document,

generating a color palette for the second platform using the plurality of images and the document,

executing the document according to the standard HTML language using the profile to generate a plurality of drawing instructions for displaying the document on the second device,

translating the plurality of images from respective formats to the supported image format, and

outputting a translated document, the translated document including at least a reference to the color palette, the plurality of images in the supported image format, and the plurality of drawing instructions.

2. The method of claim 1 wherein the reading the document further comprises retrieving the document from a world wide web (WWW) site based on a uniform resource locator (URL).

3. The method of claim 1 wherein the profile includes a maximum number of colors for the color palette.

4. The method of claim 3 wherein the generating the color palette using the plurality of images and the document comprises:

creating a set of colors comprised of all colors used in the plurality of images and all colors used in the document;

reducing the set of colors to contain no more than the maximum number of colors for the color palette.

5. The method of claim 1 wherein the document includes a plurality of references to a plurality of images, each of the plurality of references comprising a URL, and the retrieving a plurality of images referenced by the document further comprises retrieving respective images using the plurality of references.

6. The method of claim 1 wherein the executing the document according to the standard HTML language using the profile to generate a plurality of drawing instructions for displaying the document on the second device further comprises:

executing the document for display on the second device according to the display resolution;

positioning HTML elements in the document according to the display resolution;

word wrapping HTML text elements in the document according to the display resolution; and

generating a plurality of text drawing elements.

7. The method of claim 6 wherein the generating a plurality of text drawing elements further comprises generating a text element for each text segment, a text segment comprised of one or more characters from the document, the one or more characters sharing a font, a size, a style, and a color, and the text segment occupying not more than one line in the font at the size in the style, each text element including an absolute position at which the text segment should be displayed on the second device.

8. The method of claim 6 further comprising generating a plurality of graphics drawing elements including

generating a plurality of line elements;

generating a plurality of rectangle elements; and

generating a plurality of circle elements.

9. The method of claim 6 further comprising generating a plurality of link elements, each link element including a URL of a corresponding linked item.

10. The method of claim 1 wherein the supported image format includes a color palette indexed bitmap format and the translating the plurality of images from respective formats to the supported image format comprises:

decoding each of the plurality of images into a red-green-blue bitmap format;

selecting a color in the color palette for pixels in each of the plurality of images; and

outputting a color palette indexed bitmap format for each of the plurality of images.

11. The method of claim 10 wherein the document comprises a plurality of Java classes, and wherein the executing the document according to the standard HTML language using the profile to generate a plurality of drawing instructions for displaying the document on the second device comprises:

loading and verifying the plurality of Java classes;

initializing methods associated with the plurality of Java classes; and

replacing calls to complex drawing operations with a plurality of graphics drawing elements and a plurality of text drawing elements.

12. The method of claim 1 wherein the translated document includes a plurality of text elements and a plurality of graphics drawing elements.

13. The method of claim 1 wherein the standard HTML language comprises a Java language program.

14. The method of claim 1 wherein the translating the document on the first device for use on the second device further comprises:

receiving a request at the first device over a packet switched network from the second device, the request including a URL;

retrieving the document using the URL responsive to the request; and

providing the translated document to the second device over the packet switched network.

* * * * *

US005760783A

United States Patent [19]

Migdal et al.

[11] Patent Number: 5,760,783

[45] Date of Patent: Jun. 2, 1998

[54] **METHOD AND SYSTEM FOR PROVIDING TEXTURE USING A SELECTED PORTION OF A TEXTURE MAP**

[75] Inventors: **Christopher Joseph Migdal**, Mt. View; **James L. Foran**, Milpitas; **Michael Timothy Jones**, Los Altos; **Christopher Clark Tanner**, San Jose, all of Calif.

[73] Assignee: **Silicon Graphics, Inc.**, Mountain View, Calif.

[21] Appl. No.: **554,047**

[22] Filed: **Nov. 6, 1995**

[51] **Int. Cl.**[6] **G06T 11/00**

[52] **U.S. Cl.** **345/430**

[58] **Field of Search** 395/128–132, 395/125–127; 345/430

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,727,365	2/1988	Bunker et al.	340/728
4,974,176	11/1990	Buchner et al.	364/522
5,097,427	3/1992	Lathrop et al.	395/130
5,490,240	2/1996	Foran et al.	395/130

FOREIGN PATENT DOCUMENTS

0 447 227 A2	9/1991	European Pat. Off. .
0 513 474 A1	11/1992	European Pat. Off. .

OTHER PUBLICATIONS

Blinn, Jim, "Jim Blinn's Corner: The Truth About Texture Mapping," *IEEE Computer Graphics & Applications*, Mar., 1990, pp. 78–83.

Foley et al., "17.4.3 Other Pattern Mapping Techniques," *Computer Graphics: Principles and Practice*, 1990, pp. 826–828.

Cosman, M., "Global Terrain Texture: Lowering the Cost," *Proceedings of the 1994 Image VII Conference*, Tempe, Arizona: The Image Society, pp. 53–64.

Dungan, W. et al., "Texture Tile Considerations for Raster Graphics," *Siggraph '78 Proceedings* (1978) pp. 130–134.

Economy, R. et al., "The Application of Aerial Photography and Satellite Imagery to Flight Simulation," pp. 280–287.

Foley et al., *Computer Graphics Principles and Practice*, Second Edition, Addison–Wesley Publishing Company, Reading, Massachusetts (1990), pp. 742–743 and 826–828.

Watt, A., *Fundamentals of Three–Dimensional Computer Graphics*, Addison–Wesley Publishing Company, USA (1989), pp. 227–250.

Williams, L., "Pyramidal Parametrics," *Computer Graphics*, vol. 17, No. 3, Jul. 1983, pp. 1–15.

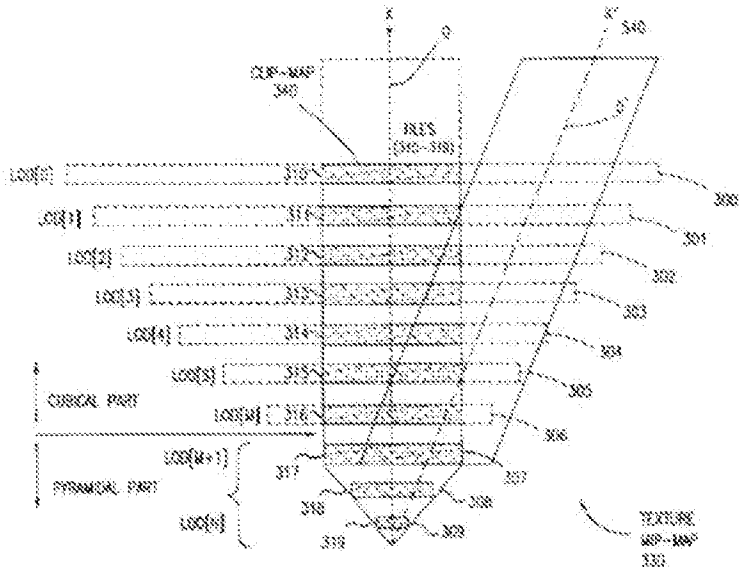
Primary Examiner—Almis R. Jankus

Attorney, Agent, or Firm—Sterne, Kessler, Goldstein & Fox P.L.L.C.

[57] **ABSTRACT**

An apparatus and method for quickly and efficiently providing texel data relevant for displaying a textured image. A large amount of texture source data, such as photographic terrain texture, is stored as a two-dimensional or three-dimensional texture MIP-map on one or more mass storage devices. Only a relatively small clip-map representing selected portions of the complete texture MIP-map is loaded into faster, more expensive memory. These selected texture MIP-map portions forming the clip-map consist of tiles which contain those texel values at each respective level of detail that are most likely to be mapped to pixels being rendered for display based upon the viewer's eyepoint and field of view. To efficiently update the clip-map in real-time, texel data is loaded and discarded from the edges of tiles. Attempts to access a texel lying outside of a particular clip-map tile are accommodated by utilizing a substitute texel value obtained from the next coarser resolution clip-map tile which encompasses the sought texel.

28 Claims, 14 Drawing Sheets



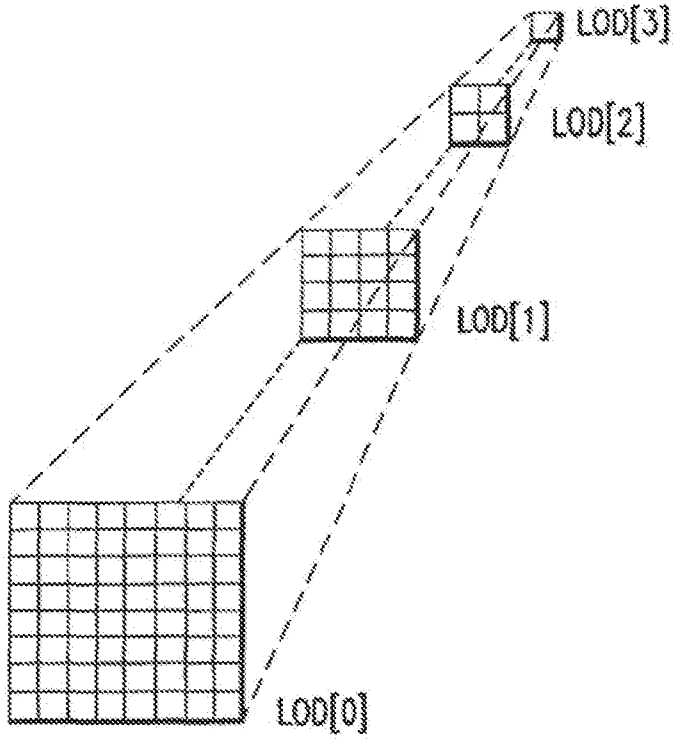


FIG. 1A

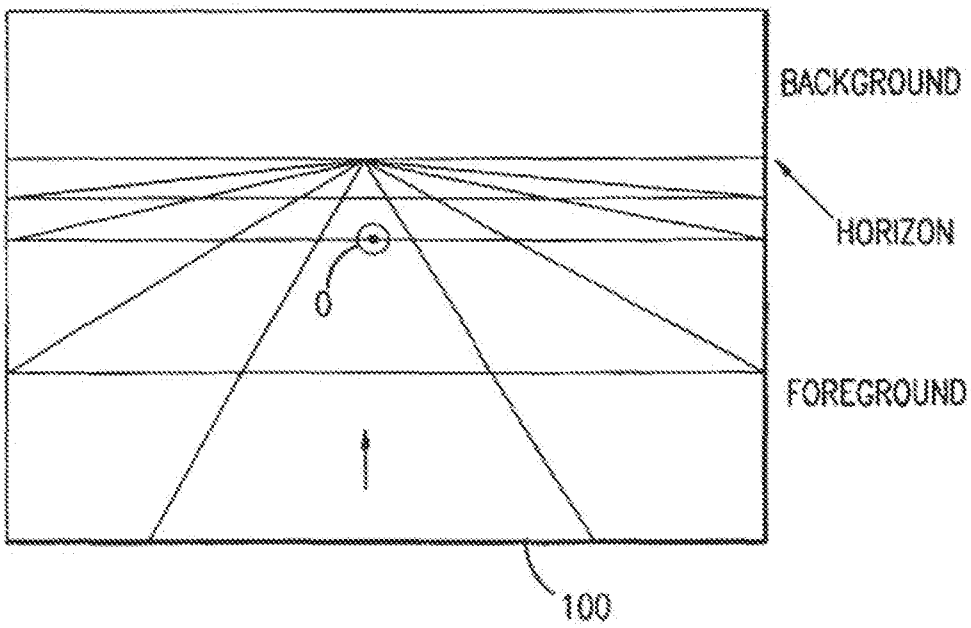


FIG. 1B

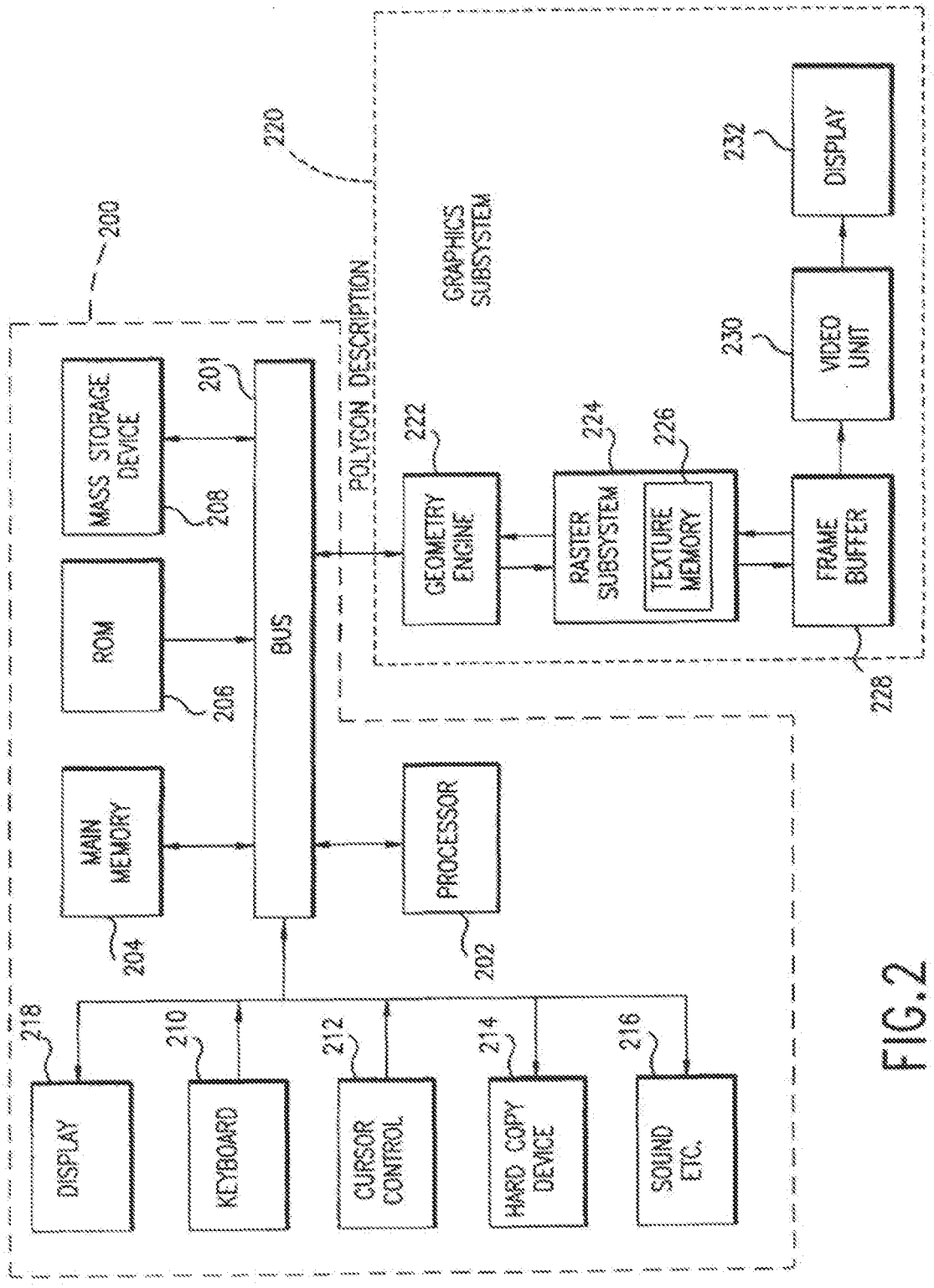

200
220
GRAPHICS SUBSYSTEM
POLYGON DESCRIPTION
MASS STORAGE DEVICE
201
208
ROM
206
BUS
MAIN MEMORY
204
PROCESSOR
202
GEOMETRY ENGINE
222
RASTER SUBSYSTEM
224
TEXTURE MEMORY
226
FRAME BUFFER
228
VIDEO UNIT
230
DISPLAY
232
DISPLAY
218
KEYBOARD
210
CURSOR CONTROL
212
HARD COPY DEVICE
214
SOUND ETC.
216

FIG.2

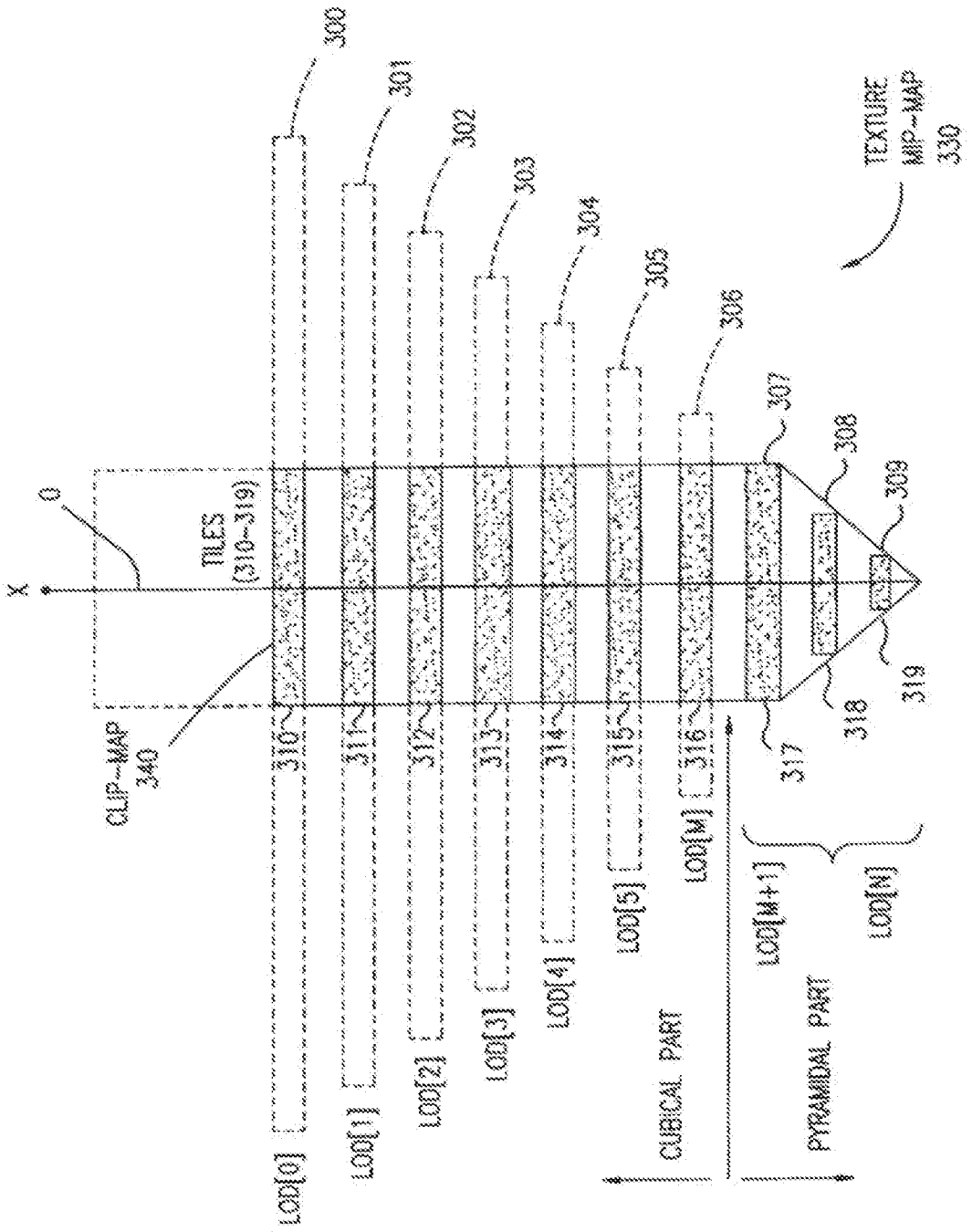

300
301
302
303
304
305
306
307
308
309
TEXTURE MIP-MAP 330
0
X
TILES (310-319)
CLIP-MAP 340
310
311
312
313
314
315
316
317
318
319
LOD[0]
LOD[1]
LOD[2]
LOD[3]
LOD[4]
LOD[5]
LOD[M]
LOD[M+1]
LOD[N]
CUBICAL PART
PYRAMIDAL PART

FIG.3

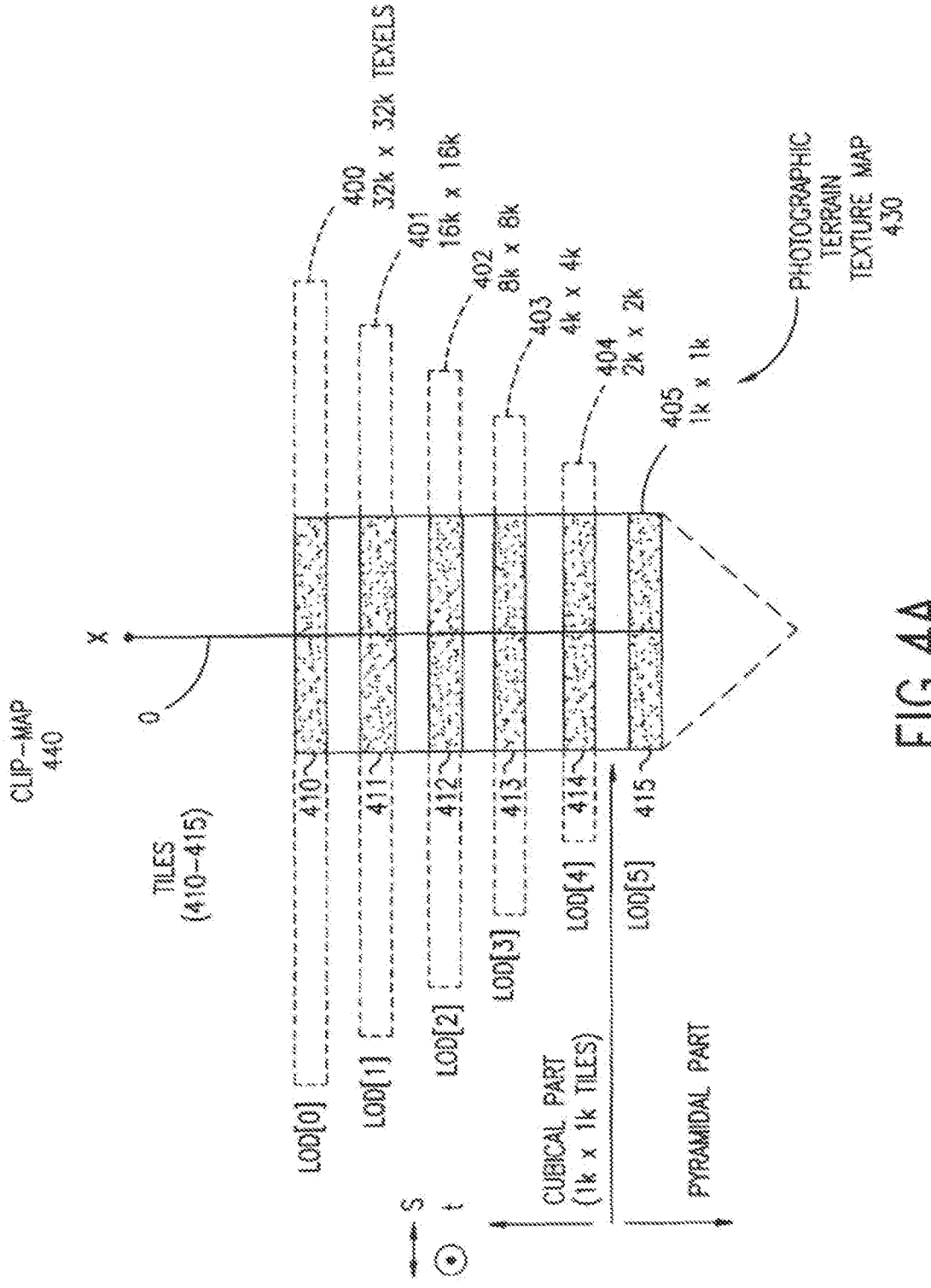

CLIP-MAP
440
TILES
(410–415)
x
0
s
t
LOD[0]
LOD[1]
LOD[2]
LOD[3]
LOD[4]
LOD[5]
410
411
412
413
414
415
400
32k x 32k TEXELS
401
16k x 16k
402
8k x 8k
403
4k x 4k
404
2k x 2k
405
1k x 1k
PHOTOGRAPHIC
TERRAIN
TEXTURE MAP
430
CUBICAL PART
(1k x 1k TILES)
PYRAMIDAL PART

FIG. 4A

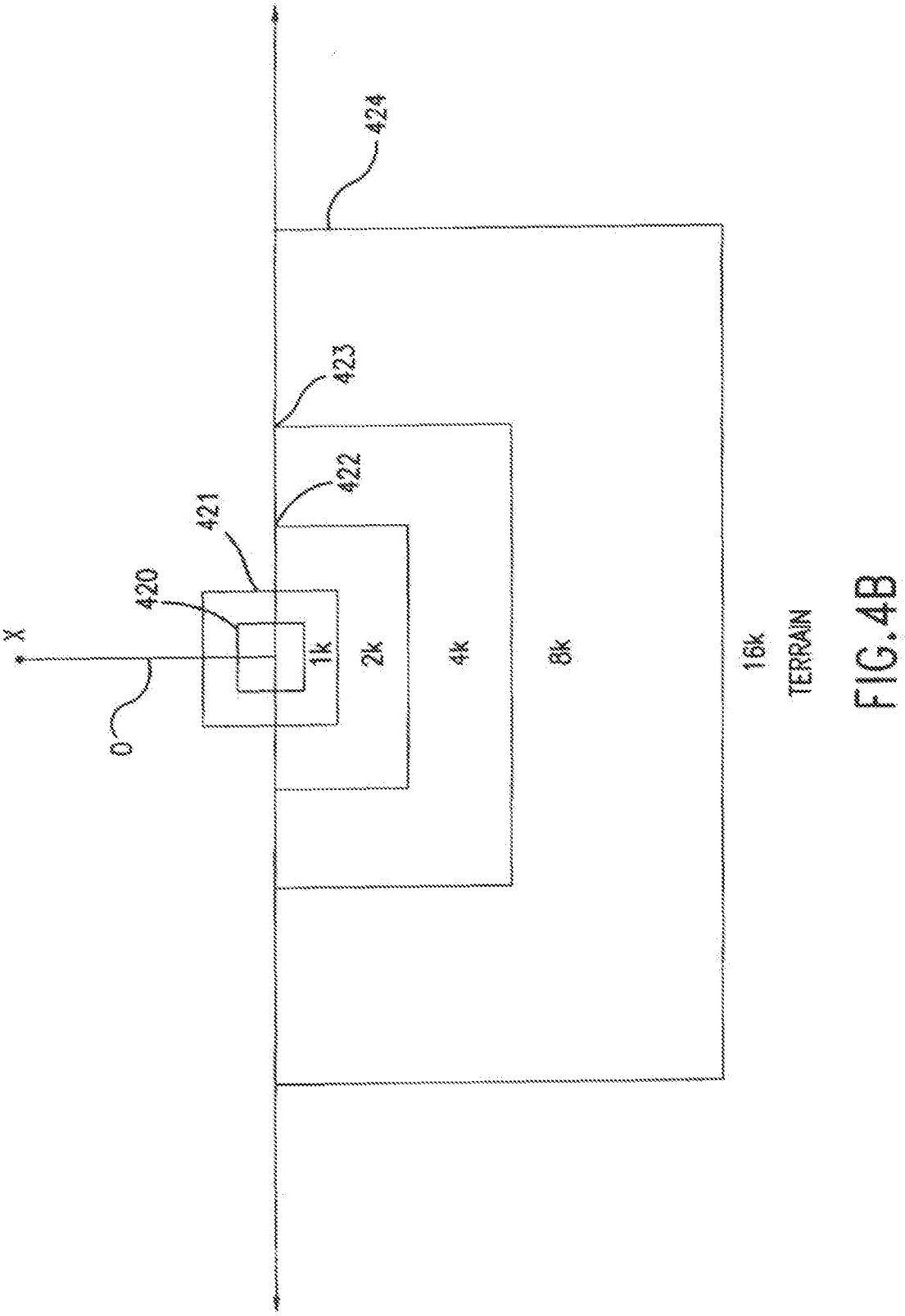


FIG.4B

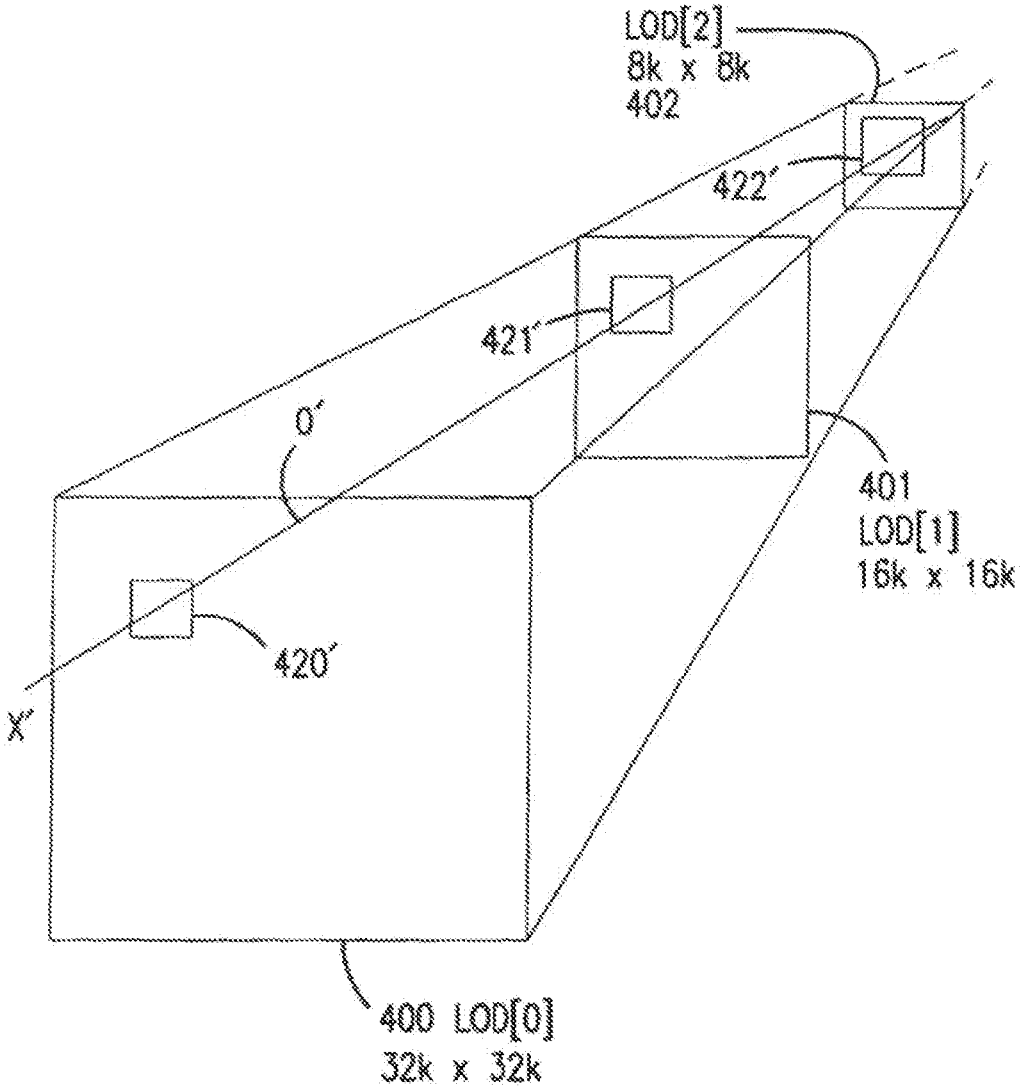

LOD[2]
8k x 8k
402
422'
421'
0'
401
LOD[1]
16k x 16k
420'
X'
400 LOD[0]
32k x 32k

FIG.4C

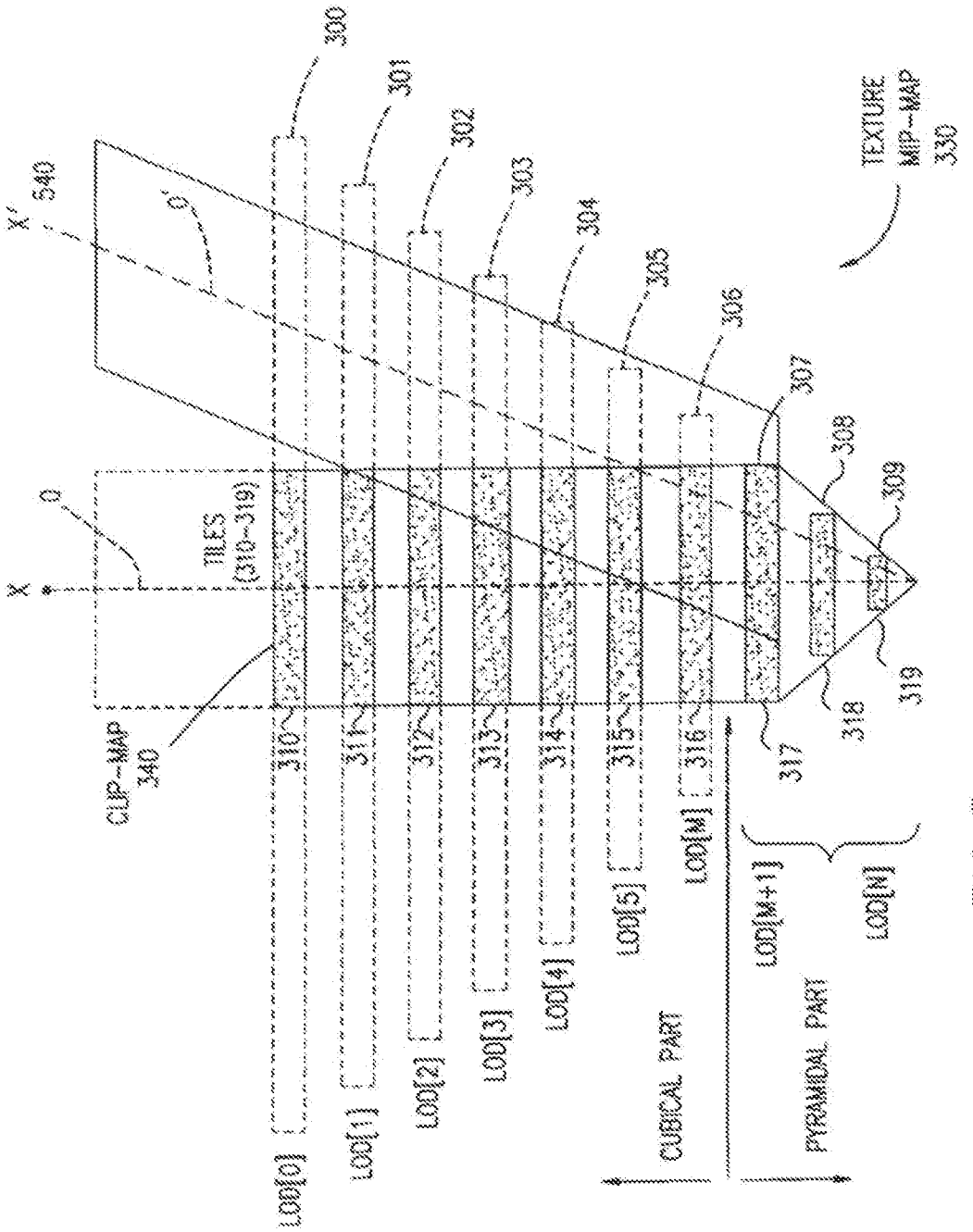

X'
540
0'
X
0
TILES
(310-319)
CUP-MAP
340
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
TEXTURE
MIP-MAP
330
LOD[0]
LOD[1]
LOD[2]
LOD[3]
LOD[4]
LOD[5]
LOD[M]
LOD[M+1]
LOD[N]
CUBICAL PART
PYRAMIDAL PART

FIG.5

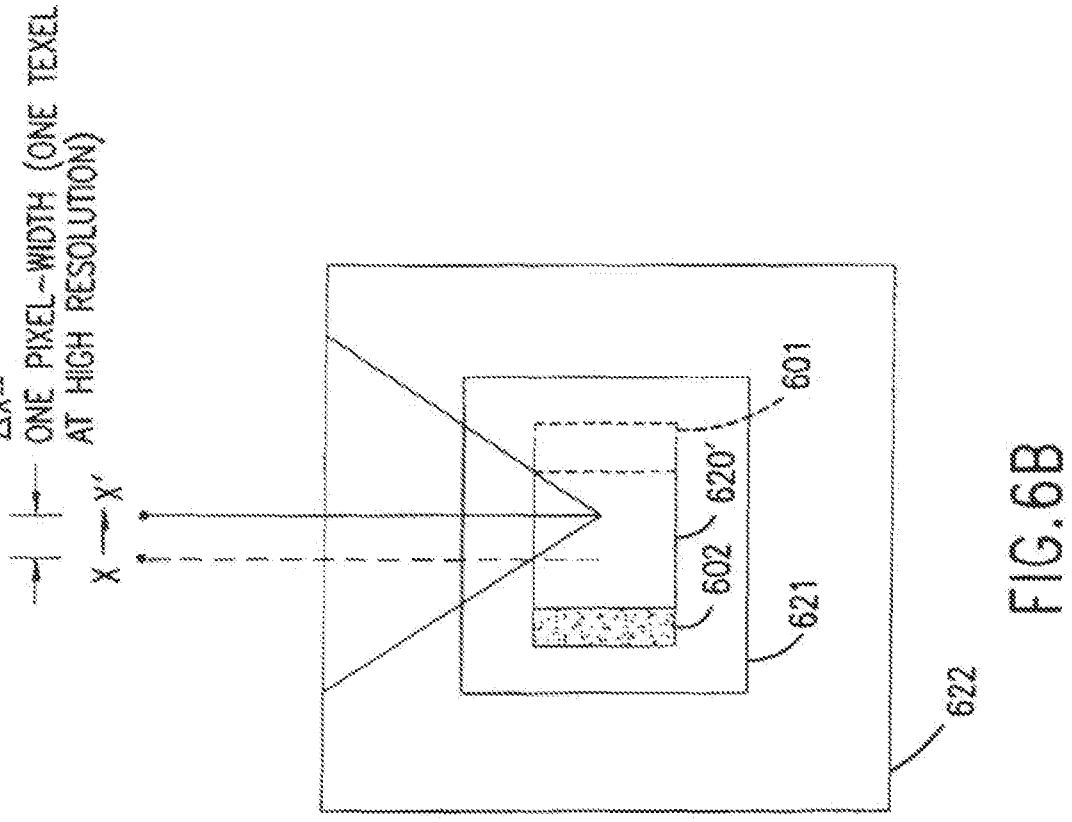

ΔX=
ONE PIXEL-WIDTH (ONE TEXEL AT HIGH RESOLUTION)
X — X'
601
620'
602
621
622

FIG.6B

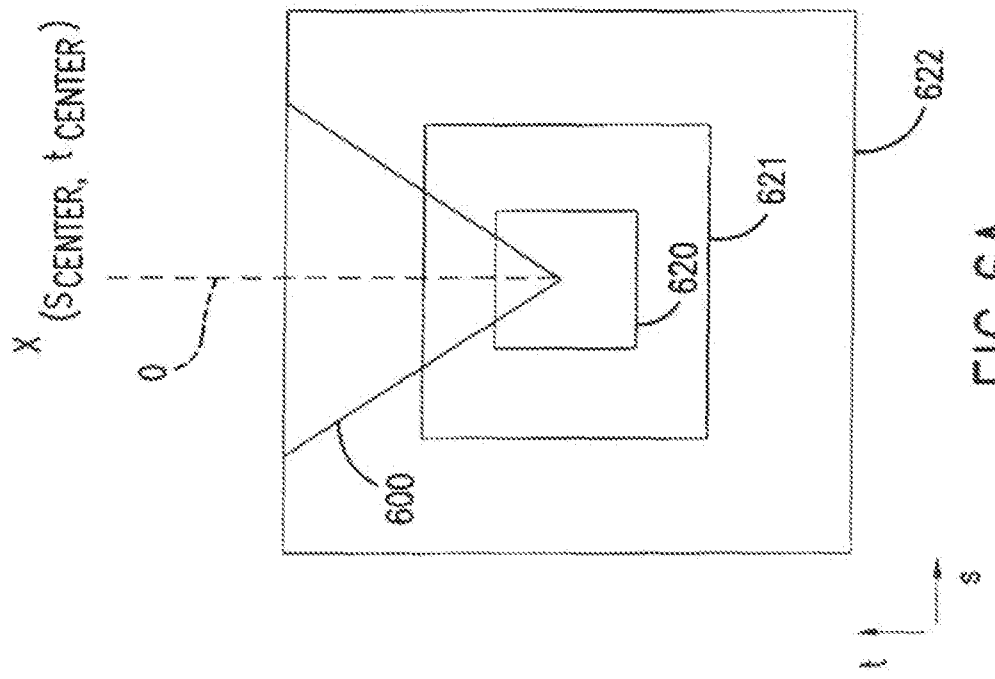

X (SCENTER, t CENTER)
0
600
620
621
622
t
s

FIG.6A

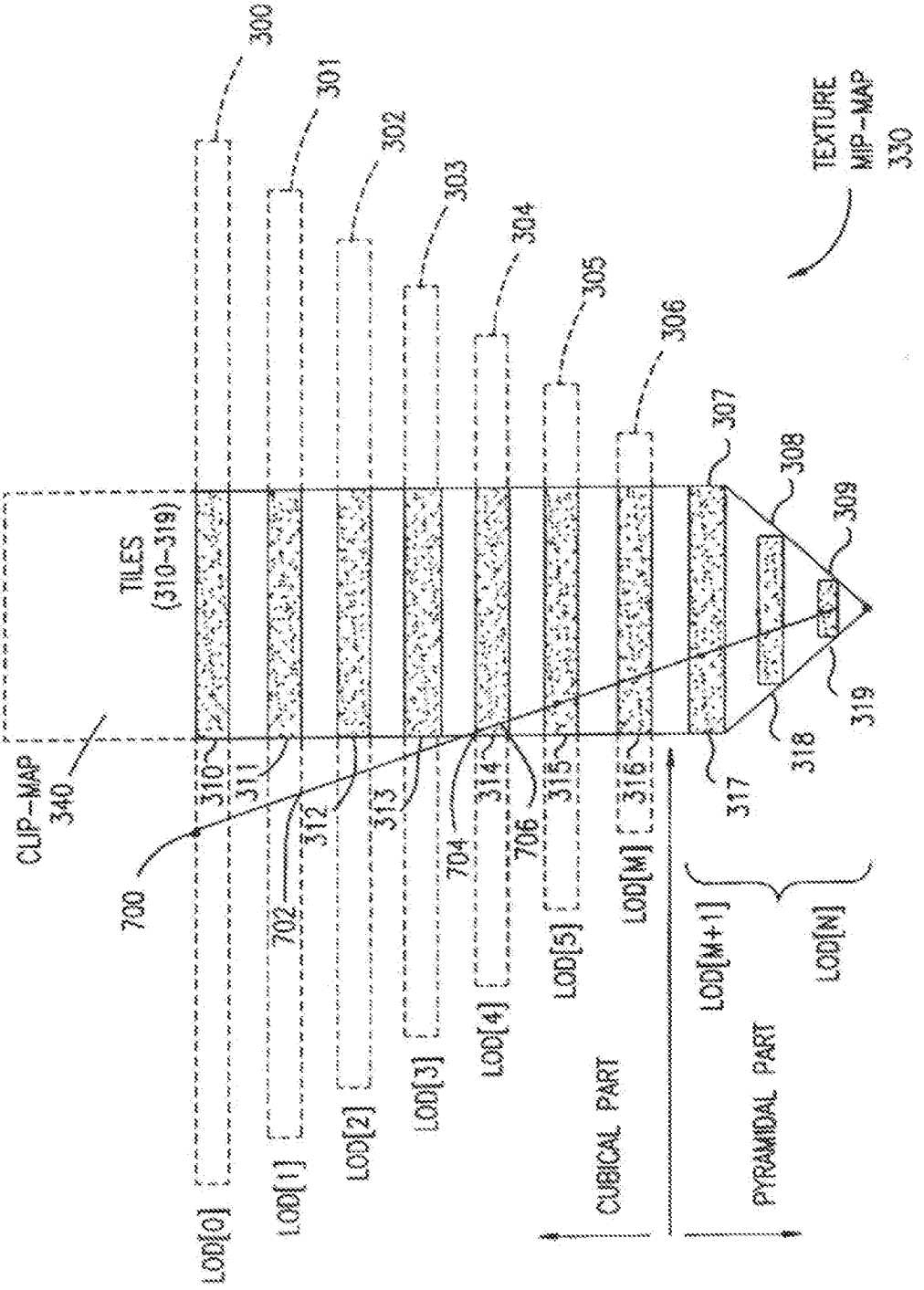

300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
TEXTURE MIP-MAP 330
TILES (310-319)
CLIP-MAP 340
700
702
704
706
LOD[0]
LOD[1]
LOD[2]
LOD[3]
LOD[4]
LOD[5]
LOD[M]
LOD[M+1]
LOD[N]
CUBICAL PART
PYRAMIDAL PART

FIG. 7

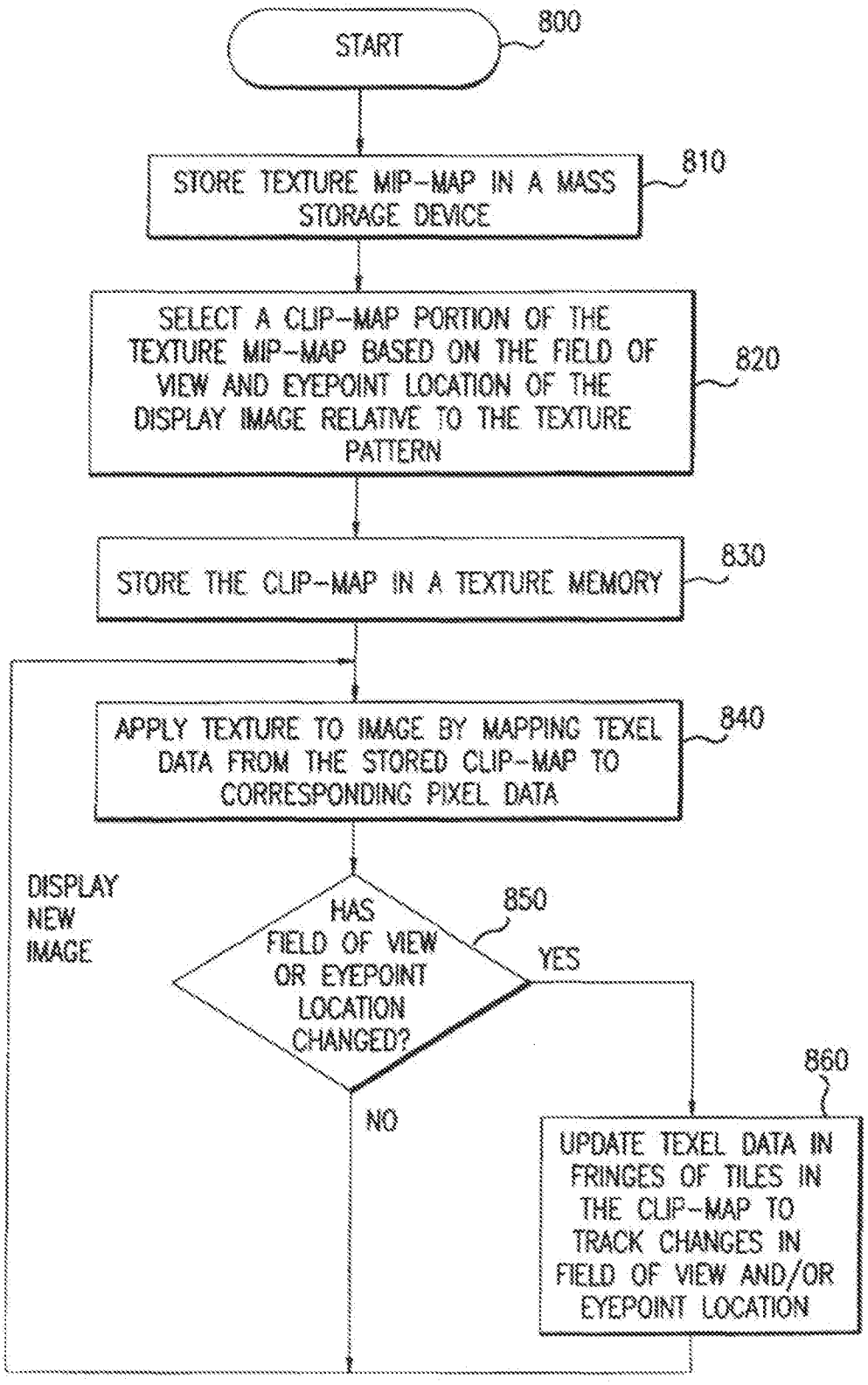

START
800
STORE TEXTURE MIP-MAP IN A MASS STORAGE DEVICE
810
SELECT A CLIP-MAP PORTION OF THE TEXTURE MIP-MAP BASED ON THE FIELD OF VIEW AND EYEPOINT LOCATION OF THE DISPLAY IMAGE RELATIVE TO THE TEXTURE PATTERN
820
STORE THE CLIP-MAP IN A TEXTURE MEMORY
830
APPLY TEXTURE TO IMAGE BY MAPPING TEXEL DATA FROM THE STORED CLIP-MAP TO CORRESPONDING PIXEL DATA
840
DISPLAY NEW IMAGE
HAS FIELD OF VIEW OR EYEPOINT LOCATION CHANGED?
850
YES
NO
860
UPDATE TEXEL DATA IN FRINGES OF TILES IN THE CLIP-MAP TO TRACK CHANGES IN FIELD OF VIEW AND/OR EYEPOINT LOCATION

FIG.8A

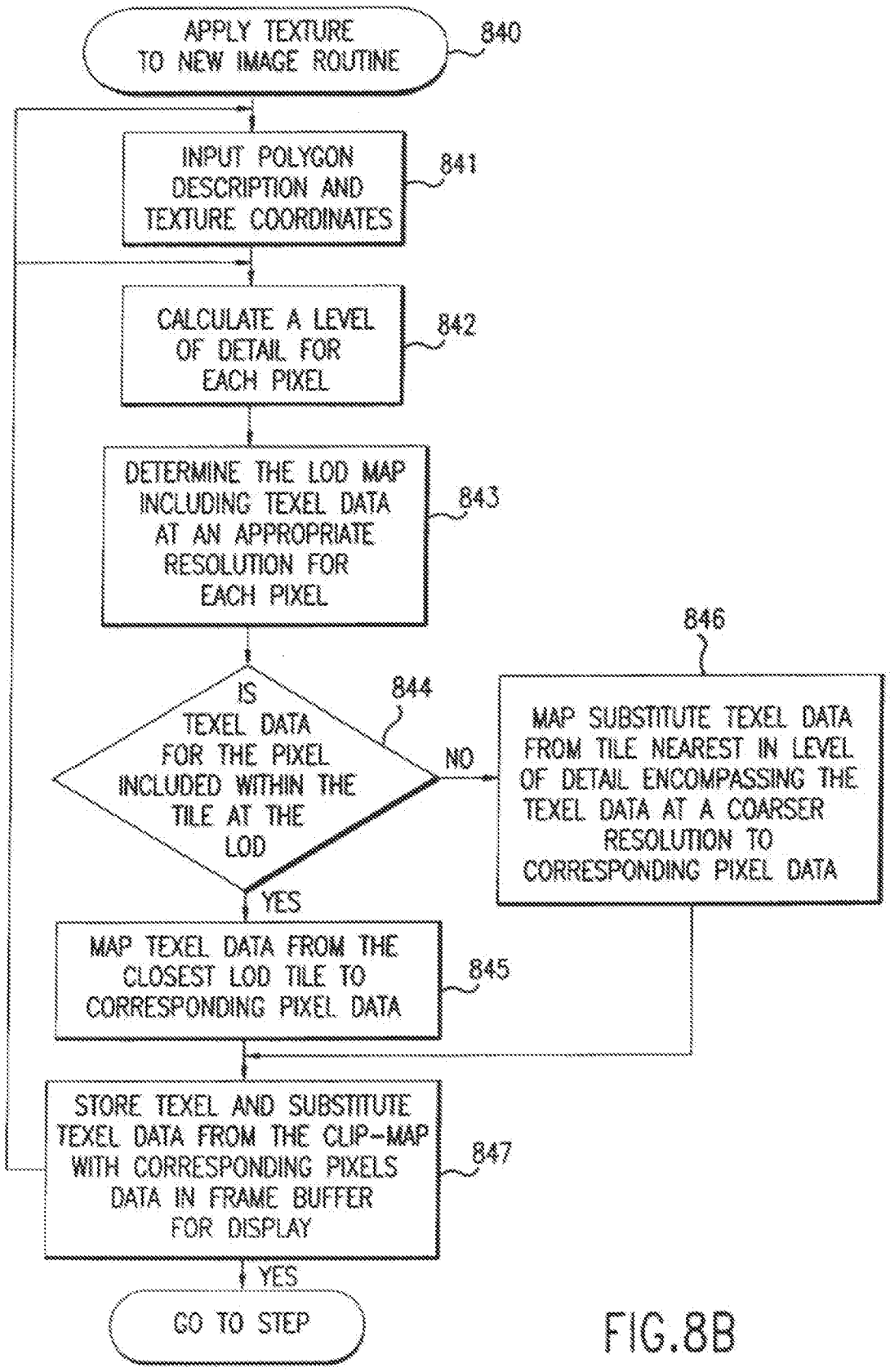

APPLY TEXTURE TO NEW IMAGE ROUTINE
840
INPUT POLYGON DESCRIPTION AND TEXTURE COORDINATES
841
CALCULATE A LEVEL OF DETAIL FOR EACH PIXEL
842
DETERMINE THE LOD MAP INCLUDING TEXEL DATA AT AN APPROPRIATE RESOLUTION FOR EACH PIXEL
843
IS TEXEL DATA FOR THE PIXEL INCLUDED WITHIN THE TILE AT THE LOD
844
NO
YES
MAP SUBSTITUTE TEXEL DATA FROM TILE NEAREST IN LEVEL OF DETAIL ENCOMPASSING THE TEXEL DATA AT A COARSER RESOLUTION TO CORRESPONDING PIXEL DATA
846
MAP TEXEL DATA FROM THE CLOSEST LOD TILE TO CORRESPONDING PIXEL DATA
845
STORE TEXEL AND SUBSTITUTE TEXEL DATA FROM THE CLIP-MAP WITH CORRESPONDING PIXELS DATA IN FRAME BUFFER FOR DISPLAY
847
YES
GO TO STEP

FIG.8B

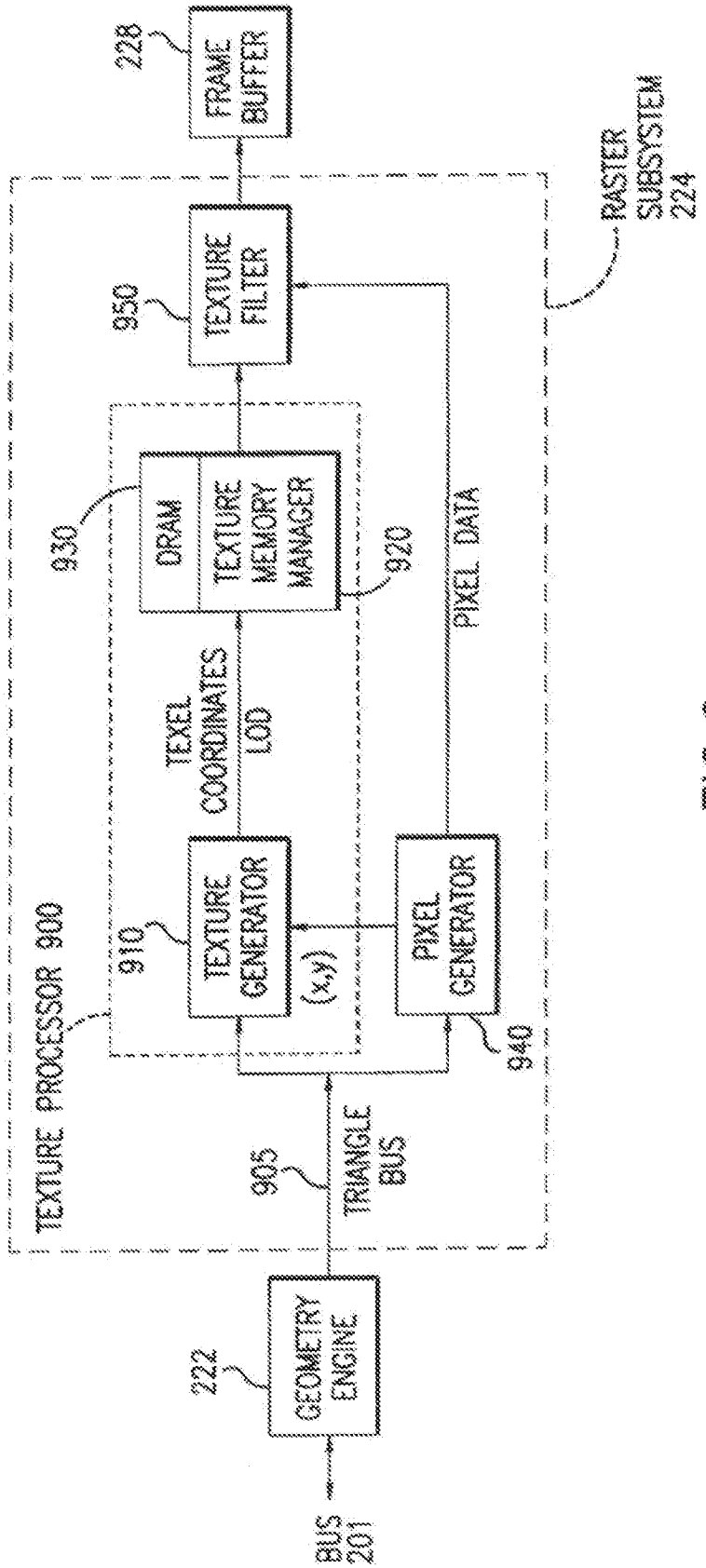

228
FRAME BUFFER
RASTER SUBSYSTEM 224
950
TEXTURE FILTER
930
DRAM
TEXTURE MEMORY MANAGER
920
PIXEL DATA
TEXEL COORDINATES
LOD
910
TEXTURE GENERATOR
(x,y)
PIXEL GENERATOR
940
TEXTURE PROCESSOR 900
905
TRIANGLE BUS
222
GEOMETRY ENGINE
BUS 201

FIG.9

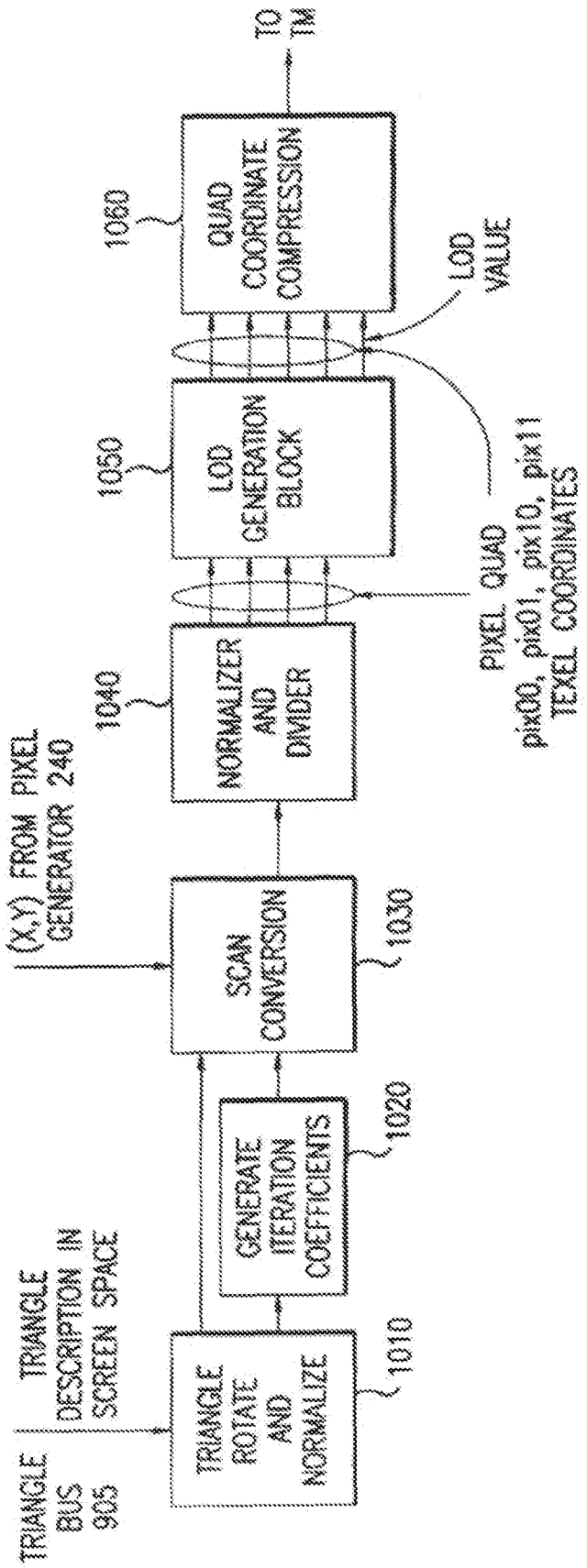

TRIANGLE BUS 905
TRIANGLE DESCRIPTION IN SCREEN SPACE
TRIANGLE ROTATE AND NORMALIZE
1010
GENERATE ITERATION COEFFICIENTS
1020
SCAN CONVERSION
1030
(X,Y) FROM PIXEL GENERATOR 240
NORMALIZER AND DIVIDER
1040
LOD GENERATION BLOCK
1050
PIXEL QUAD pix00, pix01, pix10, pix11 TEXEL COORDINATES
LOD VALUE
QUAD COORDINATE COMPRESSION
1060
TO TM

FIG. 10

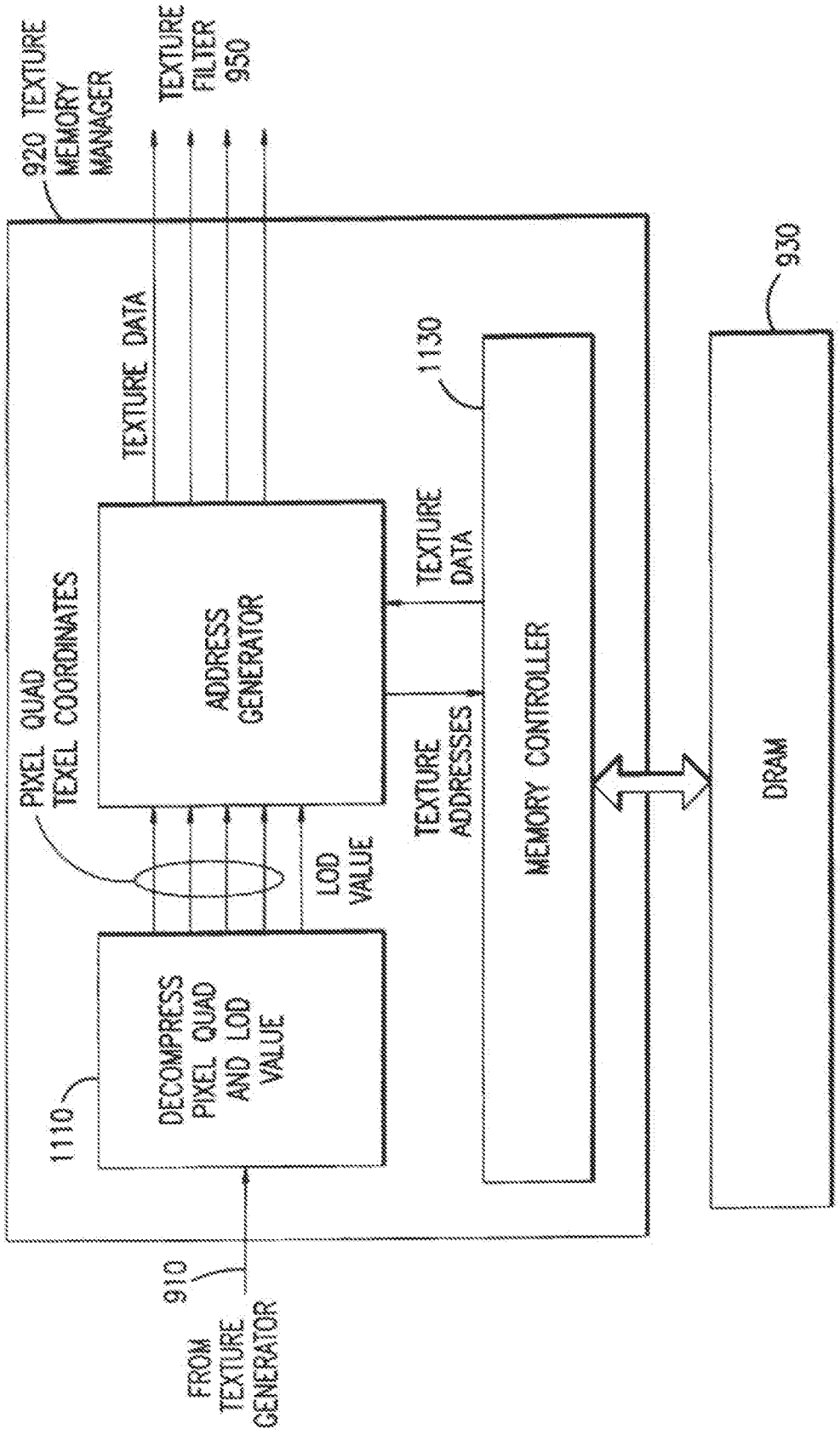

920 TEXTURE MEMORY MANAGER
TEXTURE FILTER 950
930
1130
TEXTURE DATA
TEXTURE DATA
ADDRESS GENERATOR
PIXEL QUAD TEXEL COORDINATES
LOD VALUE
TEXTURE ADDRESSES
MEMORY CONTROLLER
DRAM
1110
DECOMPRESS PIXEL QUAD AND LOD VALUE
910
FROM TEXTURE GENERATOR

FIG. 11

METHOD AND SYSTEM FOR PROVIDING TEXTURE USING A SELECTED PORTION OF A TEXTURE MAP

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention pertains to the field of computer graphics. More particularly, the present invention relates to an apparatus and method for providing texel data from selected portions of a texture MIP-map (referred to herein as a clip-map).

2. Related Art

Computer systems are commonly used for displaying graphical objects on a display screen. These graphical objects include points, lines, polygons, and three dimensional solid objects. By utilizing texture mapping techniques, color and other details can be applied to areas and surfaces of these objects. In texture mapping, a pattern image, also referred to as a "texture map," is combined with an area or surface of an object to produce a modified object with the added texture detail. For example, given the outline of a featureless cube and a texture map defining a wood grain pattern, texture mapping techniques can be used to "map" the wood grain pattern onto the cube. The resulting display is that of a cube that appears to be made of wood. In another example, vegetation and trees can be added by texture mapping to an otherwise barren terrain model. Likewise, labels can be applied onto packages or cans for visually conveying the appearance of an actual product. Textures mapped onto geometric surfaces provide motion and spatial cues that surface shading alone might not provide. For example, a sphere rotating about its center appears static until an irregular texture or pattern is affixed to its surface.

The resolution of a texture varies, depending on the viewpoint of the observer. The texture of a block of wood displayed up close has a different appearance than if that same block of wood were to be displayed far away. Consequently, there needs to be some method for varying the resolution of the texture (e.g., magnification and minification). One approach is to compute the variances of texture in real time, but this filtering is too slow for complex textures and/or requires expensive hardware to implement.

A more practical approach first creates and stores a MIP-map (multum in parvo meaning "many things in a small place"). The MIP-map consists of a texture pattern pre-filtered at progressively lower or coarser resolutions and stored in varying levels of detail (LOD) maps. See, e.g., the explanation of conventional texture MIP-mapping in Foley et al., *Computer Graphics Principles and Practice*, Second Edition, Addison-Wesley Publishing Company, Reading, Mass. (1990), pages 742–43 and 826–828 (incorporated by reference herein).

FIG. 1A shows a conventional set of texture LOD maps having pre-filtered texel data associated with a particular texture. Four different levels of detail (LOD[0]-LOD[3]m) are shown. Each successive coarser texture LOD has a resolution half that of the preceding LOD until a unitary LOD is reached representing an average of the entire high resolution base texture map LOD[0]. Thus, in FIG. 1A, LOD[0] is an 8×8 texel array; LOD[1] is a 4×4 texel array; LOD[2] is a 2×2 texel array; and LOD [3] is a single 1×1 texel array. Of course, in practice each LOD can contain many more texels, for instance, LOD[0] can be 8k×8k, LOD[1] 4k×4k, and so forth depending upon particular hardware or processing limits.

The benefit of MIP-mapping is that filtering is only performed once on texel data when the MIP-map is initially created and stored in LOD maps. Thereafter, texels having a dimension commensurate with pixel size are obtained by selecting the closest LOD map having an appropriate resolution. By obtaining texels from the pre-filtered LOD maps, filtering does not have to be performed during run-time. More sophisticated filtering operations can be executed beforehand during modeling without delaying real-time operation speed.

To render a display at the appropriate image resolution, a texture LOD is selected based on the relationship between the smallest texel dimension and the display pixel size. For a perspective view of a landscape **100**, as shown in FIG. 1B, the displayed polygonal image is "magnified" in a foreground region relative to polygonal regions located closer to the center horizon and background along the direction indicated by the arrow. To provide texture for pixels in the closest foreground region, then, texels are mapped from the finest resolution map LOD[0]. Appropriate coarser LODs are used to map texel data covering pixels located further away from the viewer's eyepoint. Such multi-resolution texture MIP-mapping ensures that texels of the appropriate texture LOD gets selected during pixel sampling. To avoid discontinuities between images at varying resolutions, well-known techniques such as linear interpolation are used to blend the texel values of two LODs nearest a particular image pixel.

One significant drawback to conventional MIP-mapping, however, is the amount of memory consumed by the various texture LOD maps. Main memory in the form of a dynamic random access memory (DRAM) or a static random access memory (SRAM) is an expensive and inefficient site for a large texture MIP-map. Each additional level of detail map at a higher level of detail requires four times more memory. For example, a 16×16 texture array having 256 texture picture elements (texels), is four times bigger than an 8×8 texture array which has 64 texels. To put this increase in perspective, a texture MIP-map having six levels of detail requires over 4,096 times more memory than the texture map at the finest resolution. Implementing large texture MIP-maps quickly becomes an expensive luxury. In addition, for large texture MIP-maps, many portions of the stored MIP-map are not used in a display image.

Memory costs become especially prohibitive in photographic texture applications where the source texture, such as, satellite data or aerial photographs, occupy a large storage area. Creating a pre-filtered MIP-map representation of such source texture data further increases memory consumption.

This problem is further exacerbated by the fact that in order to increase the speed at which images are rendered for display, many of the high-performance computer systems contain multiple processors. A parallel, multiple processor architecture typically stores individual copies of the entire MIP-map in each processor memory.

Thus, there is a need to efficiently implement large texture maps for display purposes so as to minimize attendant memory and data retrieval costs. Visual quality must not be sacrificed for memory savings. Final images in an improved texture mapping system need to be virtually indistinguishable from that of images generated by a traditional MIP-map approach.

There is also a need to maintain real-time display speeds even when navigating through displays drawn from large texture maps. For example, flight simulations must still be

performed in real-time even when complex and voluminous source data such as satellite images of the earth or moon, are used to form large texture motifs.

SUMMARY OF THE INVENTION

The present invention pertains to an apparatus and method for providing texture by using selected portions of a texture MIP-map. The selected portions are referred to herein as a clip-map. Texel data relevant to a display image is stored, accessed, and updated efficiently in a clip-map in texture memory.

Entire texture MIP-maps are stored onto one or more mass storage devices, such as hard disk drives, optical disk drives, tape drives, CD drives, etc. According to the present invention, however, only a clip-map needs to be loaded into a more expensive but quicker texture memory (e.g., DRAM). Two dimensional or three dimensional texture data can be used. The clip-map is identified and selected from within a texture MIP-map based upon the display viewer's current eyepoint and field of view. The clip-map is composed of a set of selected tiles. Each tile corresponds to the respective portion of a texture level of detail map at or near the current field of view being rendered for display.

Virtually unlimited, large amounts of texture source data can be accommodated as texture MIP-maps in cheap, mass storage devices while the actual textured image displayed at any given time is readily drawn from selected tiles of corresponding clip-maps stored in one or more texture memories. In one example, the clip-map consists of only 6 million texels out of a total of 1.365 billion texels in a complete texture MIP-map—a savings of 1.36 billion texels! Where texture information is represented as a 8-bit color value, a texture memory savings of 10.9 gigabits (99.6%) is obtained.

According to another feature of the present invention, real-time flight over a large texture map is obtained through efficient updating of the selected clip-maps. When the eyepoint of a viewer shifts, the edges of appropriate clip-map tiles stored in the texture memory are updated along the direction of the eyepoint movement. New texel data for each clip-map tile is read from the mass storage device and loaded into the texture memory to keep the selected clip-map tiles in line with the shifting eyepoint and field of view. In one particularly efficient embodiment, when the eyepoint moves a distance equal to one texel for a particular LOD, one texel row of new texture LOD data is added to the respective clip-map tile to keep pace with the direction of the eyepoint movement. The texel row in the clip-map tile which encompasses texel data furthest from the moving eyepoint is discarded.

In a further feature of the present invention, a substitute texel value is used when an attempt is made to access a texel lying outside of a particular clip-map tile at the most appropriate resolution. The substitute texel value is obtained from the next coarser resolution clip-map tile which encompasses the texel being sought. The substitution texel that is chosen is the one closest to the location of the texel being accessed. Thus, this approach returns usable texel data from a clip-map even when mapping wayward pixels lying outside of a particular clip-map tile. Of course, for a given screen size, the tile size and tile center position can be calculated to guarantee that there would be no wayward pixels.

Finally, in one specific implementation of the present invention, texture processing is divided between a texture generator and a texture memory manager in a computer graphics raster subsystem. Equal-sized square tiles simplify texel addressing. The texture generator includes a LOD generation block for generating an LOD value identifying a clip-map tile for each pixel quad. A texture memory manager readily accesses the texel data from the clip-map using tile offset and update offset information.

Further embodiments, features, and advantages of the present inventions, as well as the structure and operation of the various embodiments of the present invention, are described in detail below with reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE FIGURES

The accompanying drawings, which are incorporated herein and form a part of the specification, illustrate the present invention and, together with the description, further serve to explain the principles of the invention and to enable a person skilled in the pertinent art to make and use the invention.

In the drawings:

FIG. 1A shows a conventional multi-resolution MIP-map covering four levels of detail.

FIG. 1B shows a conventional example of a polygon perspective of a landscape to which texture MIP-mapping can be applied.

FIG. 2 shows a block diagram of an example computer graphics system implementing the present invention.

FIG. 3 shows a side view of a ten-level texture MIP-map and the selected tiles that constitute a clip-map according to the present invention.

FIG. 4A shows a side view of the first six levels of a clip-map for photographic terrain texture in one example of the present invention.

FIG. 4B shows the progressively larger areas of a terrain texture covered by coarser tiles in the present invention.

FIG. 4C shows three LOD-maps and associated clip-map tile areas relative to an observer's field of view.

FIG. 5 shows the shifting of selected tiles in a clip-map to track a change in the viewer eyepoint.

FIGS. 6A and 6B illustrate an efficient updating of clip-map tiles according to the present invention to follow eyepoint changes.

FIG. 7 shows obtaining a substitute texel value from the next closest clip-map tile having the highest resolution according to the present invention.

FIGS. 8A and 8B are flowcharts describing steps for obtaining a textured display image using a texture clip-map according to the present invention.

FIGS. 9 to 11 are block diagrams illustrating one example of a computer graphics subsystem implementing the present invention.

FIG. 9 shows a raster subsystem including a texture processor having a texture generator and a texture memory manager according to the present invention.

FIG. 10 shows a block diagram of the texture generator in FIG. 9.

FIG. 11 shows a block diagram of the texture memory manager in FIG. 9.

The present invention will now be described with reference to the accompanying drawings. In the drawings, like reference numbers indicate identical or functionally similar elements. Additionally, the left-most digit(s) of a reference number identifies the drawing in which the reference number first appears.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

I. Overview and Discussion

II. Terminology

I. Overview and Discussion

The present invention provides an apparatus and method for efficiently storing and quickly accessing texel data relevant for displaying a textured image. A large amount of texture source data is stored as a multi-resolution texture MIP-map on one or more mass storage devices. Only a relatively small clip-map representing selected portions of the complete texture MIP-map is loaded into faster, more expensive texture memory. These selected texture MIP-map portions include tiles which contain those texel values at each respective level of detail that are most likely to be mapped to pixels being rendered for display.

When the eyepoint or field of view is changed, the tiles stored in the texture memory are updated accordingly. In one efficient embodiment for updating the clip-map in real-time, new texel data is read from the mass storage device and loaded into the fringes of tiles to track shifts in the eyepoint. To maintain the size of the clip-map, tile texel data corresponding to locations furthest away from a new eyepoint is discarded. Anomalous attempts to access a texel lying outside of a particular clip-map tile are accommodated by utilizing a substitute texel value obtained from the next coarser resolution clip-map tile which encompasses the texel being sought.

II. Terminology

To more clearly delineate the present invention, an effort is made throughout the specification to adhere to the following term definitions as consistently as possible.

The term "texture map" refers to source data representing a particular texture motif at its highest resolution. A "texture MIP-map" and equivalents thereof such as a "MIP-map of texel data" are used to refer to conventional multum in parvo MIP-map representations of a texture map at successive multiple levels of details (LOD), that is, varying degrees of resolution.

On the other hand, "clip-map" is used to refer to portions of a MIP-map selected according to the present invention. Thus, a clip-map is a multi-resolution map made up of a series of tiles wherein at least some of the tiles represent smaller, selected portions of different levels of detail in a MIP-map. When two-dimensional texture data sets are used, these tiles are two-dimensional texel arrays. When three-dimensional texture data sets are used, these tiles are three-dimensional texel arrays, i.e. cubes.

Finally, texel array dimensions are given in convenient 1k×1k, 2k×2k, etc., shorthand notation. In actual implementations using digital processing, 1k equals 1,024, 2k equals 2,048, etc.

III. Example Environment

The present invention is described in terms of a computer graphics display environment for displaying images having textures drawn from multi-resolution texture MIP-maps. Moreover, sophisticated texture motifs covering a large area, such as satellite data and aerial photographs, are preferred to fully exploit the advantages of the clip-map system and method described herein. As would be apparent to a person skilled in the pertinent art, the present invention applies generally to different sizes and types of texture patterns limited only by the imagination and resources of the user.

Although the present invention is described herein with respect to two-dimensional texture mapping, the present invention can be extended to three-dimensional texture mapping when the requisite additional software and/or hardware resources are added. See e.g. the commonly-assigned, U.S. patent application Ser. No. 08/088,716, now U.S. Pat. No. 5,490,240 (Attorney Docket No. 15-4-99.00 (1452.0140000), filed Jul. 9, 1993, entitled "A System and Method of Generating Interactive Computer Graphic Images Incorporating Three Dimensional Textures," by James L. Foran et al. (incorporated herein by reference in its entirety).

IV. Computer Graphics System

Referring to FIG. 2, a block diagram of a computer graphics display system **200** is shown. System **200** drives a graphics subsystem **220** for generating textured display images according to the present invention. In a preferred implementation, the graphics subsystem **220** is utilized as a high-end, interactive computer graphics workstation.

System **200** includes a host processor **202** coupled through a data bus **201** to a main memory **204**, read only memory (ROM) **206**, and mass storage device **208**. Mass storage device **208** is used to store vast amounts of digital data relatively cheaply. For example, the mass storage device **208** can consist of one or more hard disk drives, floppy disk drives, optical disk drives, tape drives, CD ROM drives, or any number of other types of storage devices having media for storing data digitally.

Different types of input and/or output (I/O) devices are also coupled to processor **202** for the benefit of an interactive user. An alphanumeric keyboard **210** and a cursor control device **212** (e.g., a mouse, trackball, joystick, etc.) are used to input commands and information. The output devices include a hard copy device **214** (e.g., a laser printer) for printing data or other information onto a tangible medium. A sound recording or video option **216** and a display screen **218** can be coupled to the system **200** to provide for multimedia capabilities.

Graphics data (i.e. a polygonal description of a display image or scene) is provided from processor **202** through data bus **201** to the graphics subsystem **220**. Alternatively, as would be apparent to one skilled in the art, at least some of the functionality of generating a polygonal description could be transferred to the computer graphics subsystem as desired.

Processor **202** also passes texture data from mass storage device **208** to texture memory **226** to generate and manage a clip-map as described below. Including the software and/or hardware in processor **202** for generating and managing clip-maps is one example for implementing the present invention. Separate modules or processor units for generating and managing a texture clip-map could be provided along data bus **201** or in graphics subsystem **220**, as would be apparent to one skilled in the art considering this description.

The graphics subsystem **220** includes a geometry engine **222**, a raster subsystem **224** coupled to a texture memory **226**, a frame buffer **228**, video board **230**, and display **232**.

Processor **202** provides the geometry engine **222** with a polygonal description (i.e. triangles) of a display image in object space. The geometry engine **222** essentially transforms the polygonal description of the image (and the objects displayed therein) from object space (also known as world or global space) into screen space.

Raster subsystem **224** maps texture data from texture memory **226** to pixel data in the screen space polygonal description received from the geometry engine **222**. Pixel data and texture data are eventually filtered, accumulated, and stored in frame buffer **228**. Depth comparison and other display processing techniques can be performed in either the raster subsystem **224** or the frame buffer **228**. Video unit **230** reads the combined texture and pixel data from the frame buffer **228** and outputs data for a textured image to screen display **232**. Of course, as would be apparent to one skilled in the art, the output image can be displayed on display **218**, in addition to or instead of display **232**. The digital data representing the output textured display image can also be saved, transmitted over a network, or sent to other applications.

The present invention is described in terms of this example high-end computer graphics system environment. Description in these terms is provided for convenience only. It is not intended that the invention be limited to application in this example environment. In fact, after reading the following description, it will become apparent to a person skilled in the relevant art how to implement the invention in alternative environments.

V. Texture MIP-Mapping

In the currently preferred embodiment of the present invention, large texture maps are stored on one or more mass storage devices **208**. A MIP-map representation of the texture maps can either be pre-loaded onto the mass storage device **208** or can be computed by the processor **202** and then stored onto mass storage device **208**. Two-dimensional or three-dimensional texture data sets are accommodated.

As is well-known in computer graphics design, the filtering and MIP-structure development necessary to derive and efficiently store the successive levels of detail for a texture MIP-map can be effectuated off-line prior to run-time operation. In this way, high-quality filtering algorithms can be utilized over a large texture map or database without hindering on-line image display speed and performance. Alternatively, if a less flexible but fast approach is acceptable, hardware can be used to produce the successive coarser levels of detail directly from input texture source data.

Under conventional texture mapping techniques, even if texture data were to be accessed from a remote, large texture MIP-map, the rendering of a textured image for display in real-time would be impractical, if not impossible. The present invention, however, realizes the advantages of accommodating large texture MIP-maps in one or more mass storage devices **208** without reducing texture access time. A relatively small clip-map representing only selected portions of a complete texture MIP-map is stored in a texture memory **226** having a fast rate of data return. In this way, texture memory **226** acts as a cache to provide texture rapidly to the raster subsystem **224**.

This hierarchical texture mapping storage scheme allows huge texture MIP-maps to be stored rather inexpensively on the mass storage device **208**. Based on the viewer eye point and/or field of view, only selected portions of a texture MIP-map corresponding to the texture motif to be rendered for display need to be loaded into the texture memory **226**. In this manner, large 2-D or 3-D texture MIP-maps can be used to provide texture rather inexpensively, yet the textured images can be rendered in real-time.

VI. Selecting Portions of a Texture MIP-Map

The process for determining which portions of a complete texture MIP-map are to be loaded from mass storage devices **208** into texture memory **226** to form a clip-map will now be described in more detail. FIG. **3** shows a side view of a complete texture MIP-map **330** having ten conventional levels of details (not shown to actual geometric scale). The levels of detail **300** to **309** each correspond to successively coarser resolutions of a texture map. The highest level of detail LOD[0] corresponds to the finest resolution texel map **300**. Each subsequent level of detail map **301** to **309** are filtered to have half the resolution of the preceding level of detail. Thus, each coarser level of detail covers an area of the texture map four times greater than the preceding level of detail.

FIG. **3** further illustrates the selected portions of the texture MIP-map **330** constituting a clip-map **340** according to the present invention. Clip-map **340** consists of relatively small tiles **310**–**319** which are regions of the levels of detail maps **300**–**309**. The actual size and shape of these tiles depends, inter alia, on the eye point and/or field of view of the display viewer. Each of these tiles must substantially encompass a potential field of view for a display view. To simplify addressing and other design considerations, equal-sized square tiles, i.e. square texel arrays, are used which can each be addressed relative to a common, fixed central eye point X and a center line O running through the clip map. For 3-D texture, square cubes consisting of a 3-D texel array are used.

Clip-map **340** essentially consists of a set of tiles, including a cubical part (**310**–**316**) and a pyramidal part (**317**–**319**). The cubical part consists of a shaft of tiles (**310**–**316**) of equal size. In the pyramidal part, the tiles consist of the actual level of detail maps (LOD[M+1]-LOD [N]). The pyramidal part begins at the first level of detail map (LOD[M+1]) which is equal to or smaller than a tile in the cubical part and extends down to a 1×1 texel (LOD[N]).

The reduced memory requirements for storing a clip-map instead of a complete texture MIP-map are clear. A complete, conventional 2-D texture MIP-map having dimensions given by "size in s" and "size in t" uses at least the following memory M:

$$M_{\text{complete MIP-map}} = 4/3 * (\text{size in s}) * (\text{size in t}) * \text{texel size (in bytes)}.$$

The smaller, clip-map example having equal-sized, square tiles in the cubical part only uses the following memory M:

$$M_{\text{clip-map}} = [(\text{number of levels in cubical part}) * (\text{tile size})^2] + 4/3 * (\text{size in s of pyramidal part}) * (\text{size in t of pyramidal part})] * \text{texel size (in bytes)}.$$

Ten levels of detail are shown in FIG. **3** to illustrate the principle of the present invention. However, a smaller or greater number of levels of detail can be utilized. In a preferred example of the present invention, **16** levels of detail are supported in a high-end interactive computer graphics display workstation.

VII. Photographic Terrain Texture

Substantial reductions in memory costs and great improvements in real-time display capability are immedi-

ately realized by using a clip-map to render textured images. These advantages are quite pronounced when large texture maps such as a photographic terrain texture are implemented.

For example, source data from satellites covering 32 or more square kilometers of a planet or lunar surface is available. Such terrain can be adequately represented by a photographic texture MIP-map **430** having sixteen level of detail maps. The six highest resolution LOD maps **400**–**405** and tiles **410**–**415** are shown in FIG. **4**A. The highest resolution level LOD[0] consists of a 32k×32k array of texels. Successive level of details LOD[1]-LOD[5] correspond to the following texel array sizes: 16k×16k, 8k×8k, 4k×4k, 2k×2k, and 1k×1k. The remaining pyramidal part not shown consists of texel arrays 512×512, 256×256, . . . 1X1. Thus, a total of 1.365 billion texels must be stored in mass storage device **208**.

The size of the clip-map, however, is a function of the field of view and how close the observer is to the terrain. Generally, a narrow field of view requires a relatively small tile size increasing the memory savings. For example, the higher resolutions in the cubical part of clip-map **440** need only consist of 1k×1k tiles **410**–**414** for most close perspective images. The entire clip-map **440** then contains 6 million texels—a savings of 1.36 billion texels! Where texture information is represented as a 8-bit color value, a memory savings of 10.9 gigabits (99.6%) is obtained.

By storing the smaller clip-map **440** in texture memory **226**, further advantages inherent in a hierarchial memory system can be realized. The complete texture MIP-map **430** of 1.365 billion texels can be stored in cheap mass storage device **208** while the small clip-map **440** is held in a faster texture memory **226**, such as DRAM or SRAM. Sophisticated texel data can then be used to render rich textured images in real-time from the easily-accessed clip-map **440**. For example, a screen update rate of 30 to 60 Hz, i.e. 1/30 to 1/60 sec., is realized. The transport delay or latency of 1 to 3 frames of pixel data is approximately 10 to 50 msec. The above screen update rates and latency are illustrative of a real-time display. Faster or slower rates can be used depending on what is considered real-time in a particular application.

As shown in FIG. **4**B, the texel data stored in clip-map **440** can provide texture over a large display image area. For example, each high resolution texel of LOD[0] in a 32k×32k texel array can cover one square meter of geographic area in the display image. The 1k×1k tile **410** contains texel data capable of providing texture for a display image located within one square kilometer **420**.

Moreover, typical images are displayed at a perspective as described with respect to FIG. **1**B. The highest texel resolution included in tile **410** need only be used for the smaller areas magnified in a foreground region. Because of their progressively coarser resolution, each successive tile **411**–**414** covers the following broader areas 4 square kilometers (**421**), 16 square kilometers (**422**), 64 square kilometers (**423**), and 256 square kilometers (**424**), respectively. The complete 1,024 square kilometer area covered by texel data in tile **415** is not shown due to space limitations.

Even though the tiles may be equal-sized texel arrays, each tile covers a geometrically large area of a texture map because of filtering, albeit at a coarser level of detail. FIG. **4**C shows a perspective view of regions **420**' to **422**' covered by tiles **410** to **412** within the respective first three level of detail maps **400** to **402**. Each of the regions **420**' to **422**' are aligned along a center line O' stemming from an eyesight location X' to the center of the coarsest 1IX1tile (not shown) in the pyramidal part of the clip-map **440**.

Thus, the clip-map **440** contains sufficient texel data to cover larger minified areas in the background of a display where coarser texture detail is appropriate. As a result, high quality textured display images, in perspective or warped, are still obtained for large texture patterns by using texel data from a clip-map.

VIII. Updating the Clip-Map During Real-Time Operation

The discussion thus far has considered only stationary eyepoints (X or X'). Many graphics display applications, such as flight applications over a textured terrain, present a constantly changing display view. As is well-known in graphics design, the display view can simulate flight by constantly moving the eyepoint along a terrain or landscape being viewed. Such flight can be performed automatically as part of a program application, or manually in response to user input such as mouse or joystick movement. Hyperlinks or jumps can be selected by a user to abruptly select a new viewpoint and/or field of view.

Regardless of the type of movement, today's user demands that new views be displayed in real-time. Delays in mapping texture data directly from large texture maps are intolerable. Reloading an entire new texture MIP-map for a new display viewpoint is often impractical.

As shown in FIG. **5**, when the eyepoint X shifts to a new point X' for a new display view, the texel data forming the clip-map **340** must similarly shift to track the new field of view along the axis O'. According to one feature of the present invention, portions of the texture MIP-map **330** forming a new "slanted" clip-map **540** are loaded into the texture memory **226**. The "slanted" clip-map **540** is necessarily drawn in a highly stylized and exaggerated fashion in FIG. **5** in the interest of clarity. Actual changes from one display view to the next are likely less dramatic. The actual tiles in the slanted clip-map would also be more staggered if the level of detail maps were drawn in true geometric proportion.

New tiles can be calculated and stored when the eyepoint and/or field of view changes to ensure that clip-map **540** contains the texel data which is most likely to be rendered for display. Likewise, the size and/or shape of the tiles can be altered to accommodate a new display view.

Texel data can be updated by loading an entire new slanted clip-map **540** from mass storage device **208** into texture memory **226**. Full texture loads are especially helpful for dramatic changes in eyepoint location and/or field of view.

IX. Efficiently Updating the Clip-Map

According to a further feature of the present invention, subtexture loads are performed to efficiently update texel data at the edges of tiles on an on-going basis. For example, as the eyepoint shifts, a new row of texel data is added to a tile in the direction of the eyepoint movement. A row located away from a new eyepoint location is discarded. Coarser tiles need not be updated until the eyepoint has moved sufficiently far to require a new row of texel data. Thus, the relatively small amount of texel data involved in a subtexture loads allows clip-map tiles to be updated in real-time while maintaining alignment with a moving eyepoint X.

For example, FIGS. **6**A and **6**B illustrate, respectively, the areas of texel data covered by clip-map tiles before and after

a subtexture load in the highest resolution tile **410**. Each of the areas **620** to **622** correspond to the regions of a texture map covered by 1k×1k tiles **410**-**412**, as described earlier with respect to the terrain of FIG. **4**B. A field of view **600** along the direction O marks the display area which must be covered by texture detail. Hence, only those texels residing within triangle **600** need to be stored or retained in the texture memory **226**. Texels around the fringes of triangle **600**, of course, can be added to provide additional texture data near the edges of a display image.

As shown in FIG. **6**B, each time the eyepoint advances one pixel-width (the pixel-width is exaggerated relative to the overall tile size to better illustrate the updating operation), a new texel row **601** located forward of the eyepoint is loaded from mass storage device **208** into the highest resolution tile **410** in texture memory **226**. The texel row **602** furthest from the new eyepoint X' is then discarded. In this way, tile **410** contains texel data for an area **620**' covering the new display area **600**. For small changes, then, coarser tiles (**411**, **412**, etc.) do not have to be updated. Because an equal amount of texels are discarded and loaded, the tile size (and amount of texture memory consumed by the clip-map) remains constant.

When the eyepoint moves a greater distance, texture data is updated similarly for the tiles at coarser LODs. Because two texels from an LOD are filtered to one texel in each direction s or t in texture space to form a successive LOD, the minimum resolution length for each LOD[n] is 2^n pixels, where n=0 to N. Accordingly, the tiles for LOD[1], LOD[2] . . . LOD[4] in the cubical part of a clip-map **440** are only updated when the eyepoint has moved two, four, eight, and sixteen pixels respectively. Because each level of detail in the pyramidal part is already fully included in the tile **415**, no updating is necessary in theory. To simplify an updating algorithm, however, when tiles in either the cubical part or the pyramidal part reach the end of a level of detail map, garbage or useless data can be considered to be loaded. Substitute texel data drawn from a coarser tile would be used instead of the garbage data to provide texture detail in those regions.

According to the present invention, then, the amount of texel data which must be loaded at any given time to update clip-map **540** is minimal. Real-time display operation is not sacrificed.

Texel data can be updated automatically and/or in response to a user-provided interrupt. Subtexture loads are further made in advance of when the texel data is actually rendered for display.

Finally, a check can be made to prevent attempts to draw an image using texel data which is being updated. Fringe regions are defined at the edges of tiles in the cubical part of the clip-map. The fringes include at least those texels being updated. To better accommodate digital addressing, it is preferred that the fringes consist of a multiple of eight texels. For example, in a 1k×1k tile having 1,024 texels on a side, eight texels at each edge form the fringe regions leaving 1,008 texels available to provide texture. Any attempt to access a texel in the fringe is halted and a substitute texel from the next coarsest level of detail is used instead. In this way, accesses by the raster subsystem **224** to specific texel data do not conflict with any texel updating operation.

X. Substitute Texel Data

According to a further feature of the present invention, substitute texel data is returned for situations where pixel data lying outside of a clip-map tile at a desired level of detail is to be mapped. When the raster subsystem **224** seeks to map a pixel not included in a clip-map tile corresponding to the desired level of detail, there is a problem in that the texture memory **226** cannot provide texel data at the most appropriate resolution at that particular pixel. The likelihood of such a situation arising can be minimized by brutishly mandating larger tile sizes. Of course, for a given screen size, the tile size and center position can be calculated to guarantee that there would be no wayward pixels.

The inventors, however, have discovered a more elegant solution which does not require an unnecessary expansion of the clip-map to accommodate wayward pixels. As shown in FIG. **7**, substitute texel data is derived for a pixel **700** lying outside of a clip-map **340**. A line **702** is first determined between the out-of-bounds pixel **700** and the apex of the pyramid part (center of the coarsest 1×1 texel tile LOD[N]). At some point **704**, this line **702** intersects the shaft of the clip-map. Substitute texel data **706**, covering pixel **700**, is then drawn from the nearest, coarser tile **314**.

In practice, when the resolution between levels of detail varies by a factor of 2, substitute texel data is easily drawn from the next coarser level of detail by shifting a texel address one bit. Operations for obtaining memory addresses for a texel located in a clip-map tile at the desired level of detail are further described below. Arithmetic and shifting operations to obtain a substitute texel memory address from the tile at the next nearest level of detail which covers the sought pixel is also described below.

By returning substitute texel having the next best level of detail, the overall texture detail remains rich as potential image degradation from pixels lying outside the clip-map is reduced. Moreover, by accommodating wayward pixels, greater latitude is provided in setting tile size, thereby, reducing the storage capacity required of texture memory **226**.

XI. Overall Clip-Map Operation

FIGS. **8**A and **8**B are flowcharts illustrating the operation of the present invention in providing texture data from a clip-map to display images.

First, a texture MIP-map representation of a texture map is stored in an economical memory such as, a mass storage device **208** (step **810**). Processor **202** can perform pre-filtering to calculate a texture MIP-map based on a texture map supplied by the user. Alternatively, the texture MIP-map can be loaded and stored directly into the mass storage device **208**.

Portions of the MIP-maps are then selected based on a particular field of view and/or eyepoint location to form a clip-map (step **820**). The clip-map is stored in a faster texture memory **226** (step **830**). Using texel data stored in the clip-map, texture can be mapped quickly and efficiently to corresponding pixel data to display a new textured image (step **840**).

To track changes in the field of view and/or eyepoint location of a display view, only the fringes of the tiles in the clip-map are updated (steps **850** and **860**). Such subtexture loads can be performed in real-time with minimal processor overhead. Unlike conventional systems, an entire texture load operation need not be performed.

FIG. **8**B shows the operation in step **840** for processing texture for a new image in greater detail. In step **841**, a description of polygonal primitives and texture coordinates is input. Triangle vertices are typically provided in screen space by a geometry engine **222**. A raster subsystem **224** then maps texture coordinates at the vertices to pixels.

Texture coordinates can be calculated for two-dimensional or three-dimensional texture LOD maps.

In step **842**, an appropriate level of detail is calculated for each pixel according to standard LOD calculation techniques based on the pixel dimension and texel dimension. A level of detail map closest to this appropriate level of detail is determined for the pixel (step **843**).

Texture closest to the appropriate level of detail is then obtained from the finest resolution tile in the clip-map which actually encompasses a texel corresponding to the pixel (steps **844** to **847**). First, a check is made to determine whether texel data for the pixel is included within a tile corresponding to the appropriate level of detail map determined in step **843**. Because the tiles are determined based on eyepoint location and/or field of view, texel data for a pixel is likely found within a tile at an appropriate level of detail. In this case, a texel is accessed from the corresponding tile and mapped to a corresponding pixel (step **845**).

As described earlier with respect to FIG. 7, when a texel at the appropriate level of detail is not included within a corresponding tile, a coarser substitute texel is accessed. The substitute texel is chosen from the tile at the nearest level of detail which encompasses the originally-sought texel (step **846**). Texels mapped to pixels in step **845** and substitute texels mapped to corresponding pixels in step **846** are accumulated, filtered, and stored in a frame buffer **228** for subsequent display (step **847**).

Steps **841** to **847** are repeated for each input polygon description until a complete display image has been mapped to texel data and stored in the frame buffer **228**.

As would be apparent to one skilled in computer-generated textured graphics, the "clip-map" process described with respect to FIGS. **8**A and **8**B, can be carried out through firmware, hardware, software executed by a processor, or any combination thereof.

XII. Specific Implementation

FIGS. **9** to **11** illustrate one preferred example of implementing texture processing using a clip-map within computer graphics subsystem **220** according to the present invention. FIG. **9** shows a block diagram of a texture processor **900** within raster subsystem **224**. Texture processor **900** includes a texture generator **910** and a texture memory manager **920**. FIG. **10** shows a block diagram of the texture generator **910**. FIG. **11** shows a block diagram of a texture memory manager **930**. The operation of texture processor **900** in managing a clip-map to provide a texture display image will be made even more clear by the following description.

As shown in FIG. **9**, raster subsystem **224** includes texture processor **900**, pixel generator **940**, and texture filter **950**. The texture processor **900** includes a texture generator **910** coupled to a texture memory manager **920**. Texture memory manager **920** is further coupled to texture memory (DRAM) **930**.

Both the texture generator **910** and the pixel generator **940** are coupled to the geometry engine **222** via a triangle bus **905**. As explained earlier with respect to FIG. 2, polygonal primitives (i.e. triangles) of an image in screen space (x,y), are output from the geometry engine **222**. Texture generator **910** outputs specific texel coordinates for a pixel quad and an appropriate LOD value based on the triangle description received from geometry engine **222** and the (x,y) screen space coordinates of the pixel quad received from pixel generator **940**. The LOD value identifies the clip-map tile in DRAM **930** which includes a texel at the desired level of detail. When a substitute texel must be used as described above, the LOD value identifies the clip-map tile at the closest coarser level of detail which covers the sought pixel data.

Texture memory manager **920** retrieves the texel or substitute texel from the clip-map stored in DRAM **930**. The retrieved texel data is then sent to a texture filter **950**.

Texture filter **950** filters texel data sent by the texture memory according to conventional techniques. For example, bi-linear and higher order interpolations, blending, smoothing, and texture sharpening techniques can be applied to textures to improve the overall quality of the displayed image. Texture filter **950** (or alternatively the frame buffer **228**) further combines and accumulates the texel data output from texture memory manager **930** and the corresponding pixel data output by the pixel generator **940** for storage in frame buffer **228**.

FIG. **10** shows component modules **1010**–**1060** forming texture generator **910**. Blocks **1010** to **1040** represent graphics processing modules for scan converting primitives. For purposes of this example, it is presumed that the primitive description consists of triangles and that pixel data is processed as 2×2 pixel quads. Module **1010** rotates and normalizes the input triangles received across triangle bus **905**. Module **1020** generates iteration coefficients. Scan conversion module **1030** then scan converts the triangles based on the outputs of modules **1010** and **1020** and the (x,y) coordinates output from a stepper (not shown) in pixel generator **940**. Texture coordinates for each pixel quad are ultimately output from scan conversion module **1030**. Normalizer and divider **1040** outputs normalized texture coordinates for the pixel quad. Such scan conversion processing is well-known for both two-dimensional and three-dimensional texture mapping and need not be described in further detail.

LOD generation block **1050** determines an LOD value for texture coordinates associated with a pixel quad. Compared to LOD generation blocks used in conventional texture MIP-mapping, LOD generation block **1050** is tailored to consider the contents of the clip-map and whether a substitute texel is used. The LOD value output from LOD generation block **1050** identifies the clip-map tile which includes texels or substitute texels covering a pixel quad.

LOD generation block **1050** essentially performs two calculations to derive the LOD value, as described previously with respect to steps **842** to **846**. LOD generation block **1050** first calculates an appropriate level of detail for the pixel quad (or pixel) according to standard LOD generation methods based on the individual pixel size and texel dimension. A level of detail map closest to the calculated level of detail is determined.

According to the present invention, then, a check is made to determine whether texel data for the pixel quad is included within a tile corresponding to the appropriate level of detail. When a texel is included within a tile at the appropriate level of detail, a LOD value corresponding to this tile is output. Otherwise, a LOD value is output identifying a tile at a lower level of detail which includes a substitute texel, as described earlier.

The above discussion largely refers to a pixel quad (i.e. a 2×2 pixel array). However, as would be apparent to one skilled in the art, the invention is not limited to use of a pixel quad. Using a pixel quad merely reduces the number of calculations and the amount of data which must be tracked. If desired, a separate LOD value could be calculated by LOD block **1050** for each pixel.

Other modules (not shown) can further use the pixel quad and LOD value output from LOD generation block **1050** to

perform supersampling, clamping, or other graphics display optimizing processes.

The present invention takes further advantage of the use of a pixel quad to reduce the amount of data required to be sent from the texture generator **910** to the texture memory manager **920**. Quad coordinate compression module **1060** compresses the texture coordinates of a pixel quad and the LOD value data sent to one or more texture memory managers **920**. In particular, in a 2×2 pixel quad, texture coordinates for one pixel are needed but the other three pixels can be defined relative to the first pixel. In this way, only the differences (i.e. the offsets) between the centers of the other three pixels relative to the center of the first pixel need to be transmitted.

FIG. 11 shows component modules forming texture memory manager **920**. Module **1110** decompresses texture coordinates of a pixel quad and LOD value information received from texture generator **910**. Texture coordinates for one pixel are received in full. Texture coordinates for the other three pixels can be determined by the offsets to the first pixel texture coordinates. The LOD value is associated with each pixel.

The texture coordinates sent from texture generator **910** are preferably referenced to the global texture MIP-map stored and addressed in mass storage device **208**. Address generator **1120** translates the texture coordinates for the pixel quad from the global texture MIP-map space of mass storage device **208** to texture coordinates specific to the clip-map stored and addressed in DRAM **930**. Alternatively, such translation could be carried out in the texture generator **910** depending upon how processing was desired to be distributed.

Address generator **1120** first identifies a specific tile at the level of detail indicated by the LOD value. To translate texture coordinates from global texture MIP-map space to the specific tile, the address generator **1120** considers both (1) the offset of the specific tile region relative to a complete level of detail map (i.e. tile offset) and (2) the center eyepoint location of a tile (i.e. update offset).

Memory addresses corresponding to the specific texture coordinates are then sent to a memory controller **1130**. Memory controller **1130** reads and returns texture data from DRAM **930**, through address generator **1120**, to texture filter **950**.

As would be apparent to one skilled in the art from the foregoing description, a conventional LOD value can be sent from the LOD generation block **1050** without regard to the selected portions of the texture MIP-map stored in the clip-map. The steps for determining whether a texel is within a tile and for determining a LOD value for a substitute texel would then be carried out at the texture memory manager **920**.

XIII. Square Clip-Map Tiles Example

Selecting, managing and accessing texel data in a clip-map will now be discussed with respect to the specific square clip-map **440**. According to another feature of the present invention, each tile in DRAM **930** is configured as a square centered about a common axis stemming from the eyepoint location. Each tile has a progressively coarser resolution varying by factor of 2. These restrictions simplify texel addressing for each tile in the cubical part of a clip-map considerably. The additional cost in hardware and/or software to address the tiles is minimal both when the tiles are initially selected and after any subtexture loads update the tiles.

By selecting square tiles to form the initial clip-map stored in DRAM **930**, the work of a processor **282** (or a dedicated processor unit in the graphics subsystem **220**) is straightforward. First, the center of the finest level (LOD[0]) is chosen. Preferably the center (s_{center},t_{center}) is defined as integers in global (s,t) texture coordinates. A finest resolution tile **410** is then made up from the surrounding texel data in LOD[0] map **400** within a predetermined distance d from the center point. All the other tiles for the levels of the cubical part (LOD[1]-LOD[M]) are established by shifting the center position down along the eyepoint. Thus, the texture coordinates ($s_{center[n]}$,$t_{center[n]}$) for a tile at level of detail LOD[n] are given by:

$$s_{center[n]} = s_{center} >> n$$

$$t_{center[n]} = t_{center} >> n$$

where >>n denotes n shift operations. Texel data for the other tiles **401**–**404** is likewise swept in from regions a predetermined distance d in the s and t direction surrounding each center point.

Simple subtraction and comparison operations are carried out in texture generator **910** to determine whether a texel for a pixel quad is within a tile at an appropriate LOD. The finest, appropriate level of detail is determined by conventional techniques (see steps **842** and **843**). The additional step of checking whether the desired texel for a pixel quad is actually included within a tile at that LOD (step **844**) can be performed by calculating the maximum distance from four sample points in a pixel quad to the center of the tile. To be conservative, s and t distances for each of four sample points (s_0, t_0) . . . (s_3, t_3) can be calculated within a LOD generation block **1050** as follows:

$$s_0 dist = |s_{center} - s_0|$$

$$t_0 dist = |t_{center} - t_0|$$

. . .

$$s_3 dist = |s_{center} - s_3|$$

$$t_3 dist = |t_{center} - t_3|$$

where the tile center point at a LOD value n is given by (s_{center}, t_{center}).

Because the four samples of a pixel quad are strongly related, performing only two of the above subtractions is generally sufficient. Maximum distances s_{max} and t_{max} for a pixel quad are then determined by comparison. The use of square tiles means only one maximum distance in s or t needs to be calculated.

Based on the maximum distances in s and t, the finest available tile including texel or substitute texel data for the pixel quad is determined in a few arithmetic operations. If the constant size of the tiles is defined by s_{tile} and t_{tile} where the tile size equals ($2^{s_{tile}}$, $2^{t_{tile}}$) the finest available LOD value is given by the number of significant bits (sigbits) as follows:

$$LOD\ s\ finest = sigbits(s_{max}) - s_{tile}$$

$$LOD\ t\ finest = sigbits(t_{max}) - t_{tile}$$

As would be apparent to one skilled in the art, LOD generation block **1050** can perform the above calculations and output the greater of the two numbers as the LOD value identifying the appropriate finest resolution tile containing texel or substitute data.

Finally, in addition to reducing the work of LOD generation block **1050**, the restrictive use of equal-sized square

tiles and power of two changes in resolution between tiles simplifies the work of the texture memory manager **920**. Address generator **1120** can translate global texture coordinates referencing LOD maps to specific tile texture coordinates in the cubical part of the clip-map by merely subtracting a tile offset and an update offset. The tile offset represents the offset from the corner of an LOD map to the corner of a tile. The update offset accounts for any updates in the tile regions which perform subtexture loads to track changes in the eyepoint location and/or field of view.

Thus, an address generator **1120** can obtain specific s and t tile coordinates (s_{fine}, t_{fine}) for a fine tile having a level of detail equal to the LOD value provided by the texture generator **910** as follows:

$$s_{fine}=s_{TG}-\text{s tile offset}_{LOD\ value}-\text{s update offset}_{LOD\ value}$$

$$t_{fine}=t_{TG}-\text{t tile offset}_{LOD\ value}-\text{t update offset}_{LOD\ value}$$

where s_{TG} and t_{TG} represent the global s and t texture coordinates provided by a texture generator **910**, s and t tile offset $_{LOD\ value}$ represent tile offset values in s and t for a tile at the LOD value provided by the texture generator **910**, and s and t update offset$_{LOD\ value}$ represent update offset values in s and t for a tile at the LOD value.

Some texture filters and subsequent processors also use texel data from the next coarser level of detail. In this case, texture memory manager **920** needs to provide texel data from the next coarser tile as well. Once the specific s_{fine} or t_{fine} coordinates are calculated as described above s and t coordinates (s_{coarse}, t_{coarse}) for the next coarser tile are easily calculated.

In particular, the global texture coordinates (s_{TG}, t_{TG}) are shifted one bit and reduced by 0.5 to account for the coarser resolution. The tile offset and update offset values (tile offset $_{LOD\ coarse}$ and update offset $_{LOD\ coarse}$) for the next coarser tile are also subtracted for each s and t coordinate. Thus, address generator **1120** determines specific texture coordinate in the next coarser tile as follows:

$$s_{coarse}=\text{trunc}[(s_{TG}>>1)-0.5]-\text{s tile offset}_{LOD\ coarse}-\text{s update offset}_{LOD\ coarse}\text{ and}$$

$$t_{coarse}=\text{trunc}[(t_{TG}>>1)-0.5]-\text{t tile offset}_{LOD\ coarse}-\text{t update offset}_{LOD\ coarse}$$

XIV. Conclusion

While specific embodiments of the present invention have been described above, it should be understood that they have been presented by way of example only, and not limitation. It will be understood by those skilled in the art that various changes in form and details may be made therein without departing from the spirit and scope of the invention as defined in the appended claims. Thus, the breadth and scope of the present invention should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

What is claimed is:

1. A computer graphics raster subsystem for providing texture from a texture pattern to an image to be rendered for display in real-time comprising:

texture memory for storing a select portion of a texture map representation of said texture pattern, said select texture map portion containing texture data at multiple levels of detail to substantially cover said image in a display view;

texture mapping means for mapping texture data from said select texture map portion stored in said texture memory to corresponding pixel data defining said display image; and

clip-map updating means for updating edges of said select texture map portion to track changes in the location of the eyepoint in real-time.

2. The system of claim **1**, wherein said texture memory stores said texture data in a two-dimensional or a three-dimensional texel array.

3. A computer graphics processing system for providing texture from a texture pattern in a display image, comprising:

first texture memory for storing a texture map, said texture map having multiple level of detail maps for storing texels representing the texture pattern filtered at successively coarser resolutions;

clip-map selecting means for selecting a clip-map within said texture map, said clip-map consisting of a set of tiles corresponding to regions of said level of detail maps substantially covering the display image;

second texture memory for storing said clip-map;

texture processor means for retrieving texels from said clip-map which map to pixel data forming the display image; and

clip-map updating means for updating texels at edges of said tiles stored in said second texture memory to track changes in the location of an eyepoint.

4. The computer graphics system of claim **3**, wherein said first and second texture memory constitute a hierarchial memory arrangement relative to said texture processor means, said texture processor means accesses texels stored in said second texture memory faster than said first texture memory.

5. The computer graphics system of claim **3**, wherein:

said first texture memory comprises a mass storage device, and

said second texture memory comprises at least one of a static random access memory (SRAM) device and a dynamic random access memory (DRAM) device.

6. The computer graphics system of claim **3**, wherein said clip-map selecting means determines the area of the texture pattern covered by each tile based on the location of an eyepoint of the display image.

7. The computer graphics system of claim **3**, wherein said clip-map updating means updates texels at edges of said tiles stored in said second texture memory to track changes in the location of a new eyepoint for a new display image.

8. The computer graphics system of claim **7**, wherein said clip-map updating means discards texels from said second texture memory and loads texels from said first texture memory into said second texture memory; said texels being discarded from at least one tile edge located furthest from said new eyepoint and said texels being loaded into at least one tile edge located closer to said new eyepoint.

9. The computer graphics subsystem of claim **3**, wherein, said texture processor means further retrieves coarser substitute texels from said clip-map.

10. The computer graphics system of claim **3**, wherein textured display images are output for display in real-time.

11. The computer graphics system of claim **3**, wherein said clip-map contains over 99% less texels than said texture map.

12. The computer graphics system of claim **3**, wherein said texture processor means comprises:

a texture generator; and

a texture memory manager coupled between said texture generator and said second texture memory.

wherein said texture generator includes a texture coordinate generator for generating texel coordinates for each pixel and a LOD generator for generating a LOD value for each pixel, said LOD value identifies the tile which covers the pixel and has texel dimensions closest in size to the pixel, said texel coordinates and LOD value being output for each pixel to said texture memory manager; and

wherein said texture memory manager retrieves texels from said clip-map for combining with said pixels to form a textured display image.

13. The system of claim 3, wherein said second texture memory stores texels in said clip-map in a two-dimensional or a three-dimensional texel array.

14. A computer graphics processing system for providing texture from a texture pattern in a display image, comprising:

first texture memory for storing a texture map said texture map having multiple level of detail maps for storing texels representing the texture pattern filtered at successively coarser resolutions;

clip-map selecting means for selecting a clip-map within said texture map, said clip-map consisting of a set of tiles corresponding to regions of said level of detail maps substantially covering the display image;

second texture memory for storing said clip-map; and

texture processor means for retrieving texels from said clip-map which map to pixel data forming the display image, wherein said texture processor means comprises:

texture coordinate generator for generating at least one texture coordinate identifying where a pixel in the display image maps to the texture pattern,

LOD generator for generating a LOD value, and

a memory controller for retrieving at least one texel from said clip-map stored in said second texture memory based on said at least one texture coordinate and said LOD value, wherein, said LOD generator includes:

LOD identifying means for identifying an appropriate level of detail representing the level of detail map amongst said multiple level of detail maps where texel dimension is closest in size to said pixel,

texel determining means for determining whether a texel at said at least one texture coordinate is included in a first tile at said appropriate level of detail, said LOD value being set to said appropriate level of detail when said texel is included in said first tile, and

substitute texel determining means for determining a substitute texel in a second tile which includes said at least one texture coordinate, said LOD value being set to the level of detail of said second tile.

15. A computer graphics processing system for providing texture from a texture pattern in a display image, comprising:

first texture memory for storing a texture map, said texture map having multiple level of detail maps for storing texels representing the texture pattern filtered at successively coarser resolutions;

clip-map electing means for selecting a clip-map within said texture map, said clip-map consisting of a set of tiles corresponding to regions of said level of detail maps substantially covering the display image;

second texture memory for storing said clip-map; and

texture processor means for retrieving texels from said clip-map which map to pixel data forming the display image, wherein said set of tiles comprises a cubical part and a pyramidal part;

said one or more tiles in said cubical part comprising one or more arrays of texels, respectively, within said regions of said level of detail maps, and

said one or more tiles in said pyramidal part comprising one or more arrays of texels, respectively, said one or more arrays texels in said pyramidal part consisting of said level of detail maps which are equal to or smaller than said one or more tiles in said cubical part.

16. A method for providing texture from a texture pattern in a display image comprising the steps of:

storing a texture map in a first texture memory, said texture map having multiple level of detail maps for storing texels representing the texture pattern filtered at successively coarser resolutions;

selecting a clip-map within said texture map, said clip-map consisting of a set of tiles corresponding to regions of said level of detail maps, said set of tiles being centered in a field of view extending from an eyepoint location of the display image to substantially cover the display image;

storing said clip-map in a second texture memory;

retrieving texels from said clip-map stored in said second texture memory which map to pixels forming the display image; and

updating texels at edges of said tiles stored in said second texture memory to track changes in the location of the eyepoint location.

17. The method of claim 16, wherein said updating step updates said clip-map to track a change in said eyepoint location for a new display image.

18. The method of 17, further comprising the step of checking whether a texel to be accessed is located in a fringe portion of a tile, said fringe portion including the edges where texels are updated, and using a substitute texel when said texel is located in said fringe portion.

19. The method of claim 16, wherein said updating step update texels at edges of said tiles stored in said second texture memory to track a change to a new eyepoint location for a new display image.

20. The method of claim 19, wherein said updating step includes the steps of:

discarding texels from an edge of a tile located furthest from said new eyepoint location; and

loading texels from said first texture memory to said second texture memory, said texels being loaded next to an edge of a tile closer to the new eyepoint location wherein the number of texels loaded equals the number of texels discarded to maintain the size of said tile constant.

21. The method of claim 16, further comprising the step of retrieving coarser substitute texels from said clip-map stored in said second texture memory.

22. The method of claim 16, wherein said storing step stores texels in said clip-map in a two-dimensional or a three-dimensional texel array.

23. A method for providing texture from a texture pattern in a display image comprising the steps of:

storing a texture map in a first texture memory, said texture map having multiple level of detail maps for storing texels representing the texture pattern filtered at successively coarser resolutions;

selecting a clip-map within said texture map, said clip-map consisting of a set of tiles corresponding to regions of said level of detail maps, said set of tiles being centered in a field of view extending from an eyepoint location of the display image to substantially cover the display image;

storing said clip-map in a second texture memory; and

retrieving texels from said clip-map stored in said second texture memory which map to pixels forming the display image, wherein said retrieving step includes the following steps:

generating at least one texture coordinate identifying where a pixel in the display image maps to the texture pattern,

generating a LOD value, and

retrieving at least one texel from said clip-map stored in said second texture memory based on said at least one texture coordinate and said LOD value, wherein, said LOD value generating step includes:

identifying an appropriate level of detail representing the level of detail map amongst said multiple level of detail maps where texel dimension is closest in size to said pixel,

determining whether a texel at said at least one texture coordinate is included in a first tile at said appropriate level of detail and setting said LOD value to said appropriate level of detail when said texel is included in said first tile, and

when said texel in not included in said first tile, determining a substitute texel in a coarser second tile which includes said at least one texture coordinate and setting said LOD value to the level of detail of said second tile.

24. A texture processor for mapping texels from a clip-map to corresponding pixels, wherein the clip-map consists a set of tiles representing a selected portion of a texture MIP-map which substantially covers the pixels, said texture processor comprising:

a texture memory storing the clip-map;

a texture generator; and

a texture memory manager coupled between said texture generator and said texture memory, wherein

said texture generator includes a texture coordinate generator for generating texel coordinates for each pixel and a LOD generator for generating a LOD value for each pixel, said LOD value identifies the tile which covers the pixel and has texel dimensions closest in size to the pixel, said texel coordinates and LOD value being output for each pixel to said texture memory manager; and

said texture memory manager retrieves texels from the clip-map stored in said texture memory for combining with said pixels to form a textured display image.

25. The texture processor of claim **24**, wherein said texture memory stores said texels in two-dimensional or three-dimensional texel arrays.

26. The texture processor of claim **24**, wherein said texture memory manager includes an address generator for subtracting at least one of a tile offset and an update offset.

27. The texture processor of claim **24**, wherein 2×2 groups of pixels are processed as pixel quads.

28. The texture processor of claim **27**, wherein said texture generator includes a quad coordinate compression block for compressing texel coordinates for each pixel quad sent to said texture memory manager, and said texture memory manager includes a quad coordinate decompression block to decompress the compressed texel coordinates for each pixel quad.

* * * * *

Electronic Patent Application Fee Transmittal

Application Number:	14547148
Filing Date:	19-Nov-2014
Title of Invention:	OPTIMIZED IMAGE DELIVERY OVER LIMITED BANDWIDTH COMMUNICATION CHANNELS
First Named Inventor/Applicant Name:	Isaac Levanon
Filer:	Anatoly Weiser./Jason Berry
Attorney Docket Number:	AP026CON1

Filed as Small Entity

Filing Fees for Utility under 35 USC 111(a)

Description	Fee Code	Quantity	Amount	Sub-Total in USD($)
Basic Filing:				
Pages:				
Claims:				
Miscellaneous-Filing:				
Petition:				
Patent-Appeals-and-Interference:				
Post-Allowance-and-Post-Issuance:				
Extension-of-Time:				

Description	Fee Code	Quantity	Amount	Sub-Total in USD($)
Miscellaneous:				
Submission- Information Disclosure Stmt	2806	1	90	90
		Total in USD ($)		**90**

Electronic Acknowledgement Receipt

EFS ID:	23298300
Application Number:	14547148
International Application Number:	
Confirmation Number:	5188
Title of Invention:	OPTIMIZED IMAGE DELIVERY OVER LIMITED BANDWIDTH COMMUNICATION CHANNELS
First Named Inventor/Applicant Name:	Isaac Levanon
Customer Number:	35938
Filer:	Anatoly Weiser./Jason Berry
Filer Authorized By:	Anatoly Weiser.
Attorney Docket Number:	AP026CON1
Receipt Date:	24-AUG-2015
Filing Date:	19-NOV-2014
Time Stamp:	21:42:56
Application Type:	Utility under 35 USC 111(a)

Payment information:

Submitted with Payment	yes
Payment Type	Deposit Account
Payment was successfully received in RAM	$90
RAM confirmation Number	8207
Deposit Account	503196
Authorized User	WEISER, ANATOLY S

The Director of the USPTO is hereby authorized to charge indicated fees and credit any overpayment as follows:

Charge any Additional Fees required under 37 C.F.R. Section 1.16 (National application filing, search, and examination fees)

Charge any Additional Fees required under 37 C.F.R. Section 1.17 (Patent application and reexamination processing fees)

Charge any Additional Fees required under 37 C.F.R. Section 1.19 (Document supply fees)

Charge any Additional Fees required under 37 C.F.R. Section 1.20 (Post Issuance fees)

Charge any Additional Fees required under 37 C.F.R. Section 1.21 (Miscellaneous fees and charges)

File Listing:

Document Number	Document Description	File Name	File Size(Bytes)/ Message Digest	Multi Part /.zip	Pages (if appl.)
1	Transmittal Letter	AP026CON-IDS_Transmittalfiled.pdf	111065 e98d9f7058afd2279da9860d4912d3a6a481ffe7	no	2
Warnings:					
Information:					
2	Information Disclosure Statement (IDS) Form (SB08)	AP026CON1IDSfiled.pdf	1037802 d5c7bac162955e353ea491543024c1f2133514a5	no	8
Warnings:					
Information:					
3	Non Patent Literature	AP026CONnplDoc10filed.pdf	4339540 2fbf15488e88beed3f4d46a4d3b5603609687ef3	no	29
Warnings:					
Information:					
4	Non Patent Literature	AP026CONnplDoc18filed_1.pdf	16081288 7458860aef111237c58cd26c22a5887f40690bb7	no	73
Warnings:					
Information:					
5	Non Patent Literature	AP026CONnplDoc18filed_2.pdf	17925681 b186c25fd3c3bddeb6f4100b039e5d7c98bbd945	no	73
Warnings:					
Information:					
6	Non Patent Literature	AP026CONnplDoc18filed_3.pdf	18222813 52c5577db0b910d3d9d8ccbb5c64f2222643865d	no	74
Warnings:					
Information:					
7	Non Patent Literature	AP026CONnplDoc18filed_4.pdf	16992132 236191c15c3cb92dc61e01317074071625cbea07	no	74
Warnings:					
Information:					

8	Non Patent Literature	AP026CONnplDoc18filed_5.pdf	17281227 75f11ef208b09a10dad210ae177d66aba8163e88	no	74
Warnings:					
Information:					
9	Non Patent Literature	AP026CONnplDoc18filed_6.pdf	20330470 3979da7d90407121cdc0832373719144323 38175	no	74
Warnings:					
Information:					
10	Non Patent Literature	AP026CONnplDoc18filed_7.pdf	9214028 c185ebd0494e1e4c90ef3b0d7099988a7b66124a	no	74
Warnings:					
Information:					
11	Non Patent Literature	AP026CONnplDoc18filed_8.pdf	10426101 597e7c1b4562d84192cfaf6d59d475d6896c0cb9	no	74
Warnings:					
Information:					
12	Non Patent Literature	AP026CONnplDoc18filed_9.pdf	18116149 7d8c81e0a4c17bb1d6a3b8bc48e52a7a62063c49	no	74
Warnings:					
Information:					
13	Non Patent Literature	AP026CONnplDoc18filed_10.pdf	20795331 6169a2804b0dba8778ab14db3d73c4ac2c28b2f7	no	68
Warnings:					
Information:					
14	Non Patent Literature	AP026CONnplDoc17filedvol1.pdf	19039032 3aa979270a2fcf3760413280dbb41b471ad55caa	no	318
Warnings:					
Information:					
15	Non Patent Literature	AP026CONnplDoc17vol2final__1.pdf	20759224 be8c0fb8d9ca9bb26b2a0e2d42915df07a45ca1f	no	80
Warnings:					
Information:					
16	Non Patent Literature	AP026CONnplDoc17vol2final__2.pdf	23083784 [illegible]	no	120
Warnings:					
Information:					

17	Non Patent Literature	AP026CONnplDoc17vol2final_3.pdf	14836391 8042d7d314e3d6d206f2f750ea10d9799abf2973	no	64
Warnings:					
Information:					
18	Non Patent Literature	AP026CONnplDoc17vol2final_4.pdf	11947155 557379cb881f3ca40717d3303107d3168b5f7d48	no	60
Warnings:					
Information:					
19	Fee Worksheet (SB06)	fee-info.pdf	30717 b5fd923300f72a3e1a053b2251aaa188f5e629d0	no	2
Warnings:					
Information:					
		Total Files Size (in bytes):	260569930		

This Acknowledgement Receipt evidences receipt on the noted date by the USPTO of the indicated documents, characterized by the applicant, and including page counts, where applicable. It serves as evidence of receipt similar to a Post Card, as described in MPEP 503.

New Applications Under 35 U.S.C. 111

If a new application is being filed and the application includes the necessary components for a filing date (see 37 CFR 1.53(b)-(d) and MPEP 506), a Filing Receipt (37 CFR 1.54) will be issued in due course and the date shown on this Acknowledgement Receipt will establish the filing date of the application.

National Stage of an International Application under 35 U.S.C. 371

If a timely submission to enter the national stage of an international application is compliant with the conditions of 35 U.S.C. 371 and other applicable requirements a Form PCT/DO/EO/903 indicating acceptance of the application as a national stage submission under 35 U.S.C. 371 will be issued in addition to the Filing Receipt, in due course.

New International Application Filed with the USPTO as a Receiving Office

If a new international application is being filed and the international application includes the necessary components for an international filing date (see PCT Article 11 and MPEP 1810), a Notification of the International Application Number and of the International Filing Date (Form PCT/RO/105) will be issued in due course, subject to prescriptions concerning national security, and the date shown on this Acknowledgement Receipt will establish the international filing date of the application.