PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06T 11/00	A2	(11) International Publication Number: WO 98/43208
		(43) International Publication Date: 1 October 1998 (01.10.98)

(21) International Application Number: PCT/US98/05694

(22) International Filing Date: 23 March 1998 (23.03.98)

(30) Priority Data:
08/822,998 21 March 1997 (21.03.97) US

(71) Applicant *(for all designated States except US)*: NEWFIRE, INC. [US/US]; 12901 Saratoga Avenue #4, Saratoga, CA 95070–4162 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: WOOTTON, Alan, T. [US/US]; 1368 Poe Lane, San Jose, CA 95130 (US). LLOYD, James, E. [US/US]; 806 Castro Street, San Francisco, CA 94114 (US). VASILYEV, Konstantin [RU/US]; 4543 20th Street, San Francisco, CA 94114 (US). SUBRAMANIAM, Srikanth [IN/US]; 237 Arriba Drive #2, Sunnyvale, CA 94086 (US). HAWLEY, Stephen [US/US]; 56 Centre Street #12, Moutain View, CA 94041 (US). WHITTENSTEIN, W., Andreas [US/US]; P.O. Box 604, San Geronimo, CA 94963 (US).

(74) Agents: DURANT, Stephen, C. et al.; Morrison & Foerster LLP, 755 Page Mill Road, Palo Alto, CA 94304–1018 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, GW, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: METHOD AND APPARATUS FOR GRAPHICS PROCESSING

(57) Abstract

The invention is a method and apparatus for graphics processing. One embodiment of the invention is a dynamic visible surface determination process that reduces the number of geometric entities for rendering, by dynamically determining the visibility of node's in a binary space partitioning (BSP) tree. Another embodiment of the invention generates bounding boxes for the node's of a BSP tree. Yet another embodiment of the invention performs a poly differencing process. This process receives information relating to overlapping polys, and generates data structures representing non–overlapping polys. Still another embodiment of the invention performs an abstract rendering process. For each scanline of a display device, one embodiment of the rendering process generates commands to represent each static and non–static data structure on that scanline, and stores these commands in a command buffer for the scanline.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

METHOD AND APPARATUS FOR GRAPHICS PROCESSING

BACKGROUND OF THE INVENTION

Computers have historically been used for simulations especially to provide "experiences" otherwise unattainable. For example, a test pilot can simulate flying the latest fighter jet design before it is even built. The world's fastest computers are often snatched up and pushed to their limits to tackle impossibly hard simulations such as for global weather. Game manufacturers have employed a variety of proprietary optimization techniques to achieve some level of real-time simulation on reasonably priced computing equipment, which may, itself, be proprietary.

Reasonable computing power is now more affordable than ever and widely distributed in homes and offices throughout the world. But the non-standardization brought about by a reliance on proprietary techniques has hampered the widespread use of simulation.

The development of the Virtual Reality Modeling Language (VRML) has rallied support around a standard method for describing three dimensional model worlds for simulation. But simulators capable of simulating VRML worlds achieve frame rates that are too slow to effectively simulate the appearance of smooth, life-like motion. Fast simulators are proprietary or expensive, and standardized simulators are unrealistically slow. Consequently there is a need in the art for a low cost, generalized simulation platform capable of real-time frame rates.

A computer graphics system typically represents a three-dimensional scene as a collection of three-dimensional graphical primitives (i.e., objects such as boxes, spheres, cones, animated characters, etc.). These graphical primitives are defined in a three-dimensional world coordinate system. In addition, these primitives are tessellated into a number of two-dimensional geometric entities (e.g., polygons, triangles, etc.) that are positioned in the three-dimensional world coordinate space. For example, graphical primitives are commonly represented by polygon meshes which are sets of connected, polygonally bounded planar surfaces (such as triangles, quadrilaterals, etc.).

A computer graphics system typically represents a two-dimensional geometric entity by boundary and attribute defining information, such as a data structure containing the coordinate and attribute information for the vertices of the geometric entity. One type of attribute often stored by computer graphic systems for each vertex of the geometric entities is the z (depth) value. A z-value represents the distance between the viewer and the particular point of the geometric entity. A small z-value indicates that the geometric entity is close to the observer, while a large z-value indicates that the geometric entity is far from the viewer. Hence, computer graphic systems use z-values to determine if one geometric entity appears in front or behind another geometric entity.

Computer graphics systems perform a number of operations on the boundary and attribute defining information prior to generating pixel data for display. For example, some systems perform visible surface determination operations to remove surfaces that are back facing or are not in the viewing frustum. The system also performs transformation, projection, and clipping operations to define the geometric entity with

respect to the coordinate system of the display device (e.g., the two-dimensional screen space coordinate system).

After performing these operations, the system then supplies the resulting boundary and attribute defining information to a rendering engine, which generates the pixel data for all of the pixels of a display device that are covered by the geometric entity. The rendering engine stores the generated pixel data in a frame buffer which is read by the display device controller.

Performing the rendering step in real time has been an elusive goal for many graphic system designers and programmers. Computer graphic systems have difficulty accomplishing this goal without using expensive hardware, because typical three-dimensional display scenes are represented by a large number of geometric entities. Consequently, to simplify the rendering process, there is a need for a system that reduces the number of geometric entities that are supplied to the rendering engine.

Several prior art systems reduce the number of geometric entities by performing computationally intensive visible surface determination (VSD) algorithms. One process performs an object-precision algorithm, which compares all the primitives in the scene with each other in order to eliminate entire primitives or portions of them that are not visible. One such algorithm could be expressed by the following pseudo-code:

For each primitive in the scene
- determine parts of the primitive whose view is unobstructed by other parts of it or any other primitive;
- draw those parts that are not obstructed.

This process of comparing n primitives in the scene requires a computational effort proportional to n^2. Hence, such an algorithm is computationally expensive.

Another prior art VSD process creates visibility tables based on leaf nodes of a binary space partitioning (BSP) tree. A BSP tree is a nested hierarchy of grouped partitioned volumes in a reference coordinate space (e.g., a unique coordinate space for the BSP tree or the world coordinate space). Prior art VSD techniques use BSP trees to determine visibility ordering of the primitives, and then uses this ordering to render the primitives. One such prior art technique uses a BSP tree because it assumes that a poly will be scan converted correctly (i.e., will not obscure other polys incorrectly or be obscured incorrectly by other polys) if all polys on the other side of it from the viewer's viewpoint are scan converted first, followed by it, and then all the polys on the same side of it from the viewer's viewpoint.

As mentioned above, one prior art VSD process creates visibility tables based on leaf nodes of a BSP tree. In particular, for each leaf node, the process identifies other leaf nodes that are potentially visible. Such a determination is computationally intensive. In addition, this process is typically performed independent of the camera position within the leaf node. The resulting visibility tables created based on this process are therefore inaccurate at times, since they often are statistically pre-computed for a limited number of camera viewpoints.

In sum, this prior art VSD process is computationally expensive, consumes a large quantity of the memory, and is at times inaccurate. Consequently, there is a need for a computer graphic system that reduces the number of rendered geometric entities in a dynamic, accurate, and computationally inexpensive manner.

Some prior art techniques perform visible surface determinations by using a z-buffer algorithm. To use such an algorithm, a computer graphic system needs an image buffer to temporarily store the pixel data for the geometric entities, and a z-buffer to store the z-values for the stored pixels in the image buffer. Under this approach, the geometric entities are rendered into the image buffer in an arbitrary order. During the rendering process, if the geometric entity's point being rendered is no farther from the viewer than the point whose color and depth are currently in the buffers, then the new point's color and depth replace the old values. Otherwise, the new point is discarded.

Prior art z-buffer systems require no pre-sorting and no object-to-object comparisons of the objects. However, they require the use of an image buffer and a z-buffer. Consequently, they consume a large amount of memory, and therefore add to the expense of the computer graphic system. In addition, typical z-buffered prior art system perform a large number of computations, because they often perform processing for every pixel of every front facing poly within the viewing frustum. Hence, these systems are not ideally suited for real-time rendering applications.

Other prior art systems remove hidden surfaces and assure the correct rendering of the scene by initially sorting the scene's objects in z-order. These systems then render each geometric entity in descending order of smallest z coordinate (i.e., in back-to-front order). This technique is known as the painter's algorithm. While this technique assures the correct rendering of the display scenes, it is computationally expensive, and is therefore not ideal for real-time rendering systems. Consequently, there is a need for a computer graphic system that removes hidden surfaces and assures accurate real-time rendering of displayed scenes, without consuming too much memory.

A system for simulating model worlds may need to detect collisions if there is movement within the world and if some realistic simulation of the physical world is desired. One conventional method detects collisions by comparing the space occupied by every geometric element within the world with every the space occupied by every other geometric element for each frame generation cycle. As the use of 3D model world visualization increases, the desire grows for more complex worlds with more realistic simulation. The work of collision detection by conventional methods grows as the square of the number of geometric elements, and a small growth in world complexity may quickly outstrip the ability of the visualization system to perform collision detection within the time available to achieve a given frame rate. This is especially true when the increased desire for realism necessitates gathering descriptive information about a collision event other than the mere fact it occurred. Consequently, there is a need in the art for an efficient collision detection mechanism with good scalability.

The growth in the complexity of modeled worlds is due in part to an increased use of animation and other visual dynamics. Because so many attributes of objects in the modeled world are represented as numbers, e.g., the height of a bouncing ball, many mathematical techniques are employed to modify those attributes. Animation is, essentially, the modification of those attributes. Linear interpolation has been used in the prior art because of its economic use of the limited computer memory resource. Numbers representing only a few reference data points need to be stored to represent a whole range of values. Conventional interpolation gains its memory efficiency, however, at the cost of computer processor efficiency. Consequently there is a need in

the art for an interpolation mechanism that is efficient in regards to both the amount of memory and computer processor cycles it consumes.

Scripting languages provide the ultimate flexibility in regards to controlling the dynamics of a simulation. Often it is desirable for a simulation program to provide scripting capability using a widely adopted scripting language, e.g., Java, rather than creating a proprietary language. This approach may also allow for a more compact simulation program that takes advantage of external programs and services for its scripting capability. Again, the gains made conventionally come at the price extra computing overhead to comply with the interfacing requirements of the external programs or services. Consequently there is a need in the art for a mechanism to utilize external scripting programs or services without a performance penalty.

SUMMARY OF THE INVENTION

The invention provides a method and apparatus for graphics processing. One embodiment of the invention is a dynamic visible surface determination process that reduces the number of geometric entities for rendering, by dynamically determining the visibility of nodes in a binary space partitioning (BSP) tree. Another embodiment of the invention generates bounding boxes for the nodes of a BSP tree.

Yet another embodiment of the invention performs a poly differencing process. This process receives information relating to overlapping polys, and generates data structures representing non-overlapping polys. Still another embodiment of the invention performs an abstract rendering process. For each scanline of a display device, one embodiment of the rendering process generates commands to represent each static and non-static data structure on that scanline, and stores these commands in a command buffer for the scanline.

Yet another embodiment of the invention performs collision detection utilizing a characterized BSP tree. Another embodiment employs an interpolation mechanism that is both CPU and memory efficient. One more embodiment performs script program processing utilizing a high performance interface mechanism to external script processing services.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features of the invention are set forth in the appended claims. However, for purpose of explanation, several features of the embodiments of the invention are set forth in the following figures.

Figure 1 depicts a high-level block diagram for a simulation system.

Figure 2 presents a computer system upon which one embodiment of the invention is implemented.

Figure 3 presents a diagram for the system architecture of one embodiment of the invention.

Figure 4 presents the operational flow diagram for the embodiment of the invention set forth in **Figure 3**.

Figure 5 presents one example of a visual scene.

Figure 6 presents a scene node graph for the visual scene represented in **Figure 5**.

Figure 7 presents a scene cache graph for the scene node graph of **Figure 6**.

Figure 8 presents one example of a BSP tree structure.

Figure 9 presents the visual scene of **Figure 5** as partitioned by a number of partitioning planes.

Figure 10 presents a diagram of the BSP data structure used in one embodiment of the invention.

Figure 11a presents a specific example of a BSP leaf data structure.

Figure 11b and **10c** present two examples of indexed face sets.

Figure 12 presents a class diagram for a number of classes used in one embodiment of the invention.

Figure 13 presents a flow chart for creating bounding boxes for leaf nodes.

Figure 14 presents one example of bounding boxes being created by the process of **Figure 13**.

Figure 15 presents one embodiment of the dynamic partition visibility detection process of the invention.

Figure 16 presents the update-visibility process for one embodiment of the invention.

Figure 17 presents a state diagram for determining a new state in the update-visibility process of **Figure 16**.

Figure 18a presents one embodiment of the step of **Figure 16** for finding a visible child leaf.

Figure 18b presents the process for determining whether a bounding box is in a mini-frustum, for one embodiment of the invention.

Figures 19-22 present the steps performed during the queuing process of **Figure 13** for one embodiment of the invention.

Figure 23 presents a circularly linked list of vertices.

Figure 24 presents one example of a polygon represented by the linked list of vertices of **Figure 23**.

Figure 25 presents the steps performed by the poly differencing engine of one embodiment of the invention.

Figure 26 presents a poly record structure used in one embodiment of the invention.

Figure 27 presents a sorted list of edges created by the poly differencing engine.

Figure 28 presents a linked list of active polys.

Figure 29 presents a linked list of edge records.

Figure 30 presents a wedge record data structure used in one embodiment of the invention.

Figure 31 presents a displayed representation of the wedge record data structure of **Figure 30**.

Figure 32 presents the process for scanning the sorted lists of edges to initiate the creation of linked lists of active polys and active edges.

Figure 33 presents the process for executing an insert command.

Figure 34 presents the process for closing a wedge record.

Figure 35 presents the process for executing a remove command.

Figure 36 presents the process for executing an exchange command.

Figure 37 presents the process for scanning the active-edge table.

Figure 38 presents the process for calculating the intersection of two edges.

Figure 39 presents an example of wedges created by the polygon clipping process of the invention.

Figure 40 presents an active-edge linked list representing the example of **Figure 39**.

Figure 41 presents a process for determining the portion of a bounding box that resulted in a wedge.

Figure 42 presents the union rectangle process of **Figure 41**.

Figure 43 presents the process for creating a wedge for a non-static geometric entity.

Figures 44 and **45** present one example of the results of the process of **Figure 43**.

Figure 46 presents the process for populating a command buffer during an abstract rendering operation performed by one embodiment of the invention.

Figure 47 presents the process for populating the command buffer with static poly commands.

Figure 48 presents the process for populating the command buffer with the z-fill commands for the static polys.

Figure 49 presents an example of a non-static scanline array used in the process of **Figure 48**.

Figure 50 presents a displayed image with three static wedges and one non-static wedge.

Figure 51 presents one example of a command buffer that has been populated by commands representing the static and non-static wedges of **Figure 50**.

Figure 52 presents the process performed by the screen-writer of one embodiment of the invention.

Figure 53 presents the process for performing a solid-fill operation for one embodiment of the invention.

Figure 54 presents an example of the z-buffer operation performed by one embodiment of the invention.

Figure 55 depicts the high-level software architecture of the collision detection mechanism.

Figure 56 depicts an image of a sample model world.

Figure 57 depicts representative data components utilized during operation of the collision detection mechanism.

Figure 58 depicts a flowchart of the object-to-object collision detection process.

Figure 59 depicts a sample object with its bounding box, projection volume and collision volume.

Figure 60 depicts details of the data structures for the collision volume list.

Figure 61 depicts the sample scene superimposed with collision detection elements.

Figure 62 depicts a flowchart for the camera-to-object collision detection process.

Figure 63 depicts an example of a camera collision volume.

Figure 64 depicts Interpolator Node structures.

Figure 65 depicts a flowchart of the interval table building process.

Figure 66 depicts a flowchart of interpolator node operation.

Figure 67 depicts one embodiment's architectural components that interact to support certain VRML script node processing.

Figure 68 depicts a control flow diagram for one embodiment of an inverted Java environment.

Figure 69 depicts a flowchart detailing the establishment of an inverted Java environment.

DETAILED DESCRIPTION OF THE INVENTION

The invention provides a method and apparatus for graphics and simulation processing. In the following description, numerous details are set forth for purpose of explanation. However, one of ordinary skill in the art would realize that the invention may be practiced without the use of these specific details. In other instances, well-known structures and devices are shown in block diagram form in order not to obscure the description of the invention with unnecessary detail.

I. SYSTEM OVERVIEW

Figure 1 depicts a high-level block diagram for a simulation system. The system comprises a Model Description Source **110,** a Control Data Source **115**, a Simulation Processor **120**, and a Simulation Frame Sink **130**. The Model Description Source **110** delivers a signal via connection **140** to the Model Processor **120**. Signals traversing connection **120** contain information defining the model world for which simulation frames are to be generated. For example, information defining a 3-dimensional model world may consist of the maximum dimensions of the modeled volume, descriptions of the size, appearance, and position of objects in the model world, and other information useful for generating sensory outputs simulating the model world, e.g., the location of sound and light sources for graphic and audio output, or the hardness of an object for tactile output.

An example model world for visualization may have maximum dimensions measured in microns and contain only a 3D representation of a protein molecule that a biochemist wishes to view. An astronomer's example model world may have

dimensions measured in light years and contain 3D representations all of the known stars and planets in a corner of the universe. Regardless of scale, the representation of the objects in the model world in 3D allows the generation of views of the model world from different viewpoints. Generally, a user of a simulation system as depicted in Figure 1, selects a model world and then generates images from many different viewpoints before leaving the world.

Control Data Source **115** delivers a signal via connection **145** to the Model Processor. Signals traversing connection **145** may contain information specifying a sequence of viewpoints for which images of the model world are to be generated. The sequence of viewpoints may jump from location to location within the model world. For example, the biochemist may desire a "top, bottom, right side, left side, front, back" viewpoint sequence. On the other hand, the sequence of viewpoints may each represent only a relatively small change in viewing position and direction from the prior viewpoint. For example, the astronomer may want to simulate a smooth flight through a solar system.

The Control Data Source **115** may generate the information signal to the Model Processor **120** from stored data or from data gathered real-time. For example, it may be advantageous for the biochemist to select his sequence of six specific viewpoints in advance of the image generation process. The astronomer also may find it advantageous to store a specific flight path through the model world in advance of the image generation process, e.g., the planned flight path for a space probe. The astronomer may, however, prefer to provide viewpoint changes in real-time to achieve unrestricted flight through the model world.

The Model Processor **120** generates an output signal to the Simulation Frame Sink **130** via connection **150**. Signals traversing connection **150** may contain information specifying a 2-dimensional image frame. The image frame information represents the projection of the 3d model world signaled by the Model Description Source **110**, onto a viewing plane as would be seen from within the model world at the viewpoint signaled by the Control Data Source **115**. The Simulation Frame Sink **130** operates to store the image frame, immediately present it, or both. Notably, the signal on connection **150** and the Simulation Frame Sink **130**, like the other components of the system, are not limited to operation with graphical information but may be a combination of, e.g., video, audio, and tactile presentation information.

II. THE COMPUTER SYSTEM

Figure 2 presents a computer system upon which one embodiment of the present invention is implemented. Computer system **200** includes bus **205**, processor **210**, read and write memory **215**, read-only memory **220**, mass storage device **225**, display device **230**, frame buffer **235**, hardware graphics rendering engine **240**, alphanumeric input device **245**, cursor controller **250**, hard copy device **255**, and speakers **260**.

Bus **205** collectively represents all of the communication lines that connect the numerous internal modules of the computer. Processor **210** couples to bus **205** and processes digital data. Computer system **200** further includes a read and write memory (RAM) **215**, which also couples to bus **205**. This memory stores digital data and program instructions for execution by processor **210**. Memory **215** also may be used for

storing temporary variables or other intermediate information during the operation of processor **210**.

Computer system **200** also includes a read only memory (ROM) **220** coupled to bus **205** for storing static information and instructions for processor **210**. In addition, mass data storage device **225**, such as a magnetic or optical disk and its corresponding disk drive, may also be included.

In one embodiment of the invention, computer program information (e.g., object code) used in connection with the invention is downloaded from mass data storage device **225** (e.g., downloaded from a hard drive or from a disk via a disk drive) to memory **215** during the operation of the computer. In turn, computer **205** utilizes the software residing in RAM **215** to perform the invention. In another embodiment of the invention, the firmware instructions residing in ROM **220** direct the invention.

Computer system **200** further includes a display device **230**, such as a cathode ray tube (CRT) or a liquid crystal display (LCD) coupled to bus **205**. This device displays images to a computer user. The pixel data representing an image is stored in frame buffer **235**. As shown in **Figure 2**, one embodiment of the invention is implemented in a computer system which includes a hardware graphics rendering engine **240** for generating pixel data. Certain embodiments of the invention do not use the hardware graphics rendering engine to generate pixel data. These embodiments use (1) a software rasterizing engine to generate run-length-encoded (RLE) or quasi-RLE codes for each scanline to represent geometric entities (e.g., polys or wedges on each

scanline), and (2) a screen writer to generate the pixel data by decoding the generated codes.

Bus **205** also couples to alphanumeric input device **245** (e.g., a keyboard) which enables the user to communicate information and command selections to the computer system. Cursor controller **250** is an additional user input device which may be coupled to bus **205**. Input device **245** may take many different forms, such as a mouse, a trackball, a stylus tablet, a joystick, a touch-sensitive input device (e.g., a touchpad), etc. Another device which may be coupled to bus **205** is a hard copy device **255** which may be used for printing a hard copy on paper.

Finally, as shown in **Figure 2**, bus **205** also couples computer **200** to a network 265 through a network adapter (not shown). In this manner, the computer can be a part of a network of computers (such as a local area network ("LAN"), a wide area network ("WAN"), or an Intranet) or a network of networks (such as the Internet).

Any or all of the components of computer system **200** may be used in conjunction with the present invention. However, one of ordinary skill in the art would appreciate that any other system configuration may also be used in conjunction with the present invention.

III. SYSTEM ARCHITECTURE

Figure 3 presents a diagram for the system architecture of one embodiment of the invention. This embodiment is designed to operate as a VRML player, which allows a computer system to process any VRML file and display its contents. Even though many aspects of the invention are described below by reference to the operation of the VRML

system of **Figure 3**, one of ordinary skill in the art would appreciate that the invention extends to the entire field of computer graphics and simulation and is not limited to VRML systems.

System **300** of **Figure 3** includes a VRML engine **305**, a cache manager **310**, a visible scene manager **315**, a software rasterizer **320**, a screen writer **325**, a script manager **330**, Java Handler 396, an Interpolator Processor 290, and a Collision Detection Processor 392.

The VRML engine parses the textual representation of the scene contained in a received VRML file and produces scene node data structures (i.e., a first set of objects) to represent the scene. From the scene node graph data structures, the cache manager then generates the scene cache data structures (i.e., a second set of objects) as an alternative representation of the scene. Part of the cache data structures is a binary space partitioning (BSP) tree data structure, which is used for: (1) performing visible surface determination (such as frustum culling and backface culling), (2) determining the correct rendering order of static polygons, and (3) performing collision detection.

As shown in **Figure 3**, the visible scene manager (VSM) follows the cache manager. The VSM reduces the number of geometric entities (e.g., polygons such as triangles, quadrilaterals, etc.) for rendering by performing visible surface determination (VSD) processes. Some of these processes are conventional VSD processes such as frustum and backface culling processes. The VSM, however, also uses the invention's dynamic VSD process, which is referred to below as the dynamic partition visibility detection (DPVD) process. This process dynamically determines the extent to which the

geometric entities for display occlude one another. In the remainder of the discussion below, all types of geometric entities are generically referred to as polys.

After performing visible surface determination, the VSM supplies information relating to static and non-static polys respectively to a poly differencing engine and the non-static poly creator. These two engines generate data structures for rendering. In one embodiment of the invention, these two engines generate wedge data structures, which are a particular sub-class of poly data structures. Wedges are quadrilateral polys that have horizontal tops and bottoms. Hence, wedges can be represented by a pair of x-values, a pair of slopes, and a pair of y-values.

In one embodiment of the invention, the poly differencing engine receives information relating to overlapping static polys, and generates data structures representing non-overlapping static polys. Some embodiments of the invention use the non-static poly creator to create data structures representing non-static wedges.

In one embodiment of the invention, the poly differencing engine and the non-static poly creator supply their generated data structures to a graphics hardware engine which renders them into the screen buffer. Other embodiments of the invention are employed in computer systems which do not have hardware rendering engines. In such embodiments, the differencing engine and the non-static poly creator supply their generated data structures to a software rasterizer **320**.

This software rasterizer then performs the invention's abstract rendering process. Specifically, for each scanline of the display device, this rasterizer generates commands to represent each static and non-static data structure on that scanline, and stores these

commands in a command buffer for the scanline. Subsequently, screen writer **325** decodes these commands into pixel data for the scanline, and then records the pixel data in a frame buffer.

The aforementioned architectural elements principally perform the visualization task associated with simulation based on an instant state of the objects in the model world. A sophisticated simulation environment, however, has many steps before visualization where the state of objects can be altered in ways that may affect the visualization.

For example, the collision detection processor **392** may anticipate a collision between the "camera" viewing the scene and an object in the scene, e.g., a wall. The collision event may result in the execution of a program script that modifies the position of the camera to compensate for the detected collision. This change in camera position ultimately affects the generated view of the scene. The Script Manager **330** oversees the execution of such program scripts. The Script Manager **330** may use a "handler" module to interface with program services dependent on a particular scripting language implementation but independent of the system **300**, itself. For example, if the program script is written in the Java language, the Script Manager **330** will use the Java handler **396** to effectuate the execution of the program script.

The Interpolator Processor **390** may also operate to affect the generated view of the scene. The VRML Engine **305** may have created interpolator nodes in the scene graph based on definitions in the VRML file that was parsed. An interpolator node, for example, could be defined to alter the position of an object based on the passage of time.

In processing the interpolator node, the Interpolator Processor **390** alters the instant state of the model world by changing the location coordinates, i.e. position, of the object based on the current time. The change in the position coordinates produces a corresponding change in the scene image generated by the visualization components of the architecture.

Figure 4 depicts a life cycle diagram of the VRML player. The life cycle comprises an initialization step **400**, a main operational loop, and a termination step **490**. The main operational loop executes for each generated image frame and comprises a collision detection step **410**, a vehicle position determination step **415**, a sensor processing step **420**, a routing step **425**, a sound processing step **430**, a visible surface determination step **435**, and a rendering step **440**. The rendering step displays the generated image frame and the main operational loop begins again.

The initialization step **400** includes the parsing of the VRML input file that describes the model world to be simulated. The VRML language is a declaration and specification language, rather than a procedural language. VRML defines a model world by describing the nodes and relationships in a directed graph. Methods and data structures well known in the art are used to represent the directed graph, or scene graph, in computer memory. Procedures needed to simulate the operation of the model world, e.g. script programs that move objects, are only identified in the VRML file. They are programmed outside of VRML. A full description of the VRML Language is found in The Virtual Reality Modeling Language Specification, Version 2, August 4, 1996, and is hereafter referred to as the VRML Specification. Additional VRML information can be found on the Internet at San Diego Supercomputer Center's VRML Repository website

(http://www.sdsu.edu/vrml/) or more traditional publications such as Exploring Moving Worlds by Dille (1996, Ventana).

After initialization, the main operational loop begins. The relative order among the steps may not be critical. In one embodiment the main operational loop begins with the collision detection step **410**. As the name implies, collision detection identifies actual and potential collisions between objects in the model world. The vehicle position determination step **415** determines any change in viewpoint from that used to generate the last image frame based on programmatic factors and user inputs. The sensor processing step **420** processes the drawing-independent sensor nodes in the scene graph in accordance with the VRML Specification. The routing step **425** manages the processing of the internodal connection routes specified in the VRML input file. Step **430** then manages the generation and production of any audio output. The visible surface determination **435** and rendering **440** processes described earlier then generate an image frame representing the view of the instant state of the model world from the current viewpoint.

The main operational loop processes repeatedly until the VRML file or the VRML player is closed. This generally results from a user selection. Step **490** then performs termination processing as appropriate. Certain of the life cycle processes, depicted generally in **Figure 4**, are now described in more detail.

IV. INITIALIZATION

The components of **Figure 3** are described below by reference to **Figure 4**, which presents the operational flow for these components. As shown in **Figure 4**, the

initialization process is the first process performed after receiving a VRML file. The VRML engine and the cache manager perform this process.

A. Scene Node Graph Data Structure

Figure 5 presents one example of a visual scene. This visual scene has a textual representation in a VRML file received by the VRML engine. This engine parses the textual representation to obtain scene node graph data structures for representing the scene. One example of these data structures is shown in **Figure 6**. As shown in this figure, the scene node graph includes numerous nodes, each of which corresponds to the types (e.g., scene, transform, shape, geometry, appearance, etc.) specified by the VRML specification. Hence, the scene node graph of **Figure 6** for the visual scene represented in **Figure 5** includes a scene node and five sets of transform, shape, appearance, and geometry nodes for the five primitives in the scene.

Each node includes a cache pointer which initially is set to null by the VRML engine. The scene node also includes a roots field which is a list of pointers pointing to the top-level nodes, in this example the transform nodes. Each transform node includes a number of fields. One of these fields is an operation field for specifying simple operations (such as translation, rotation, and scale) for performing on the primitive to place it in the world coordinate system. Another field is a children field pointing to a set of contained nodes, where in this example each children field contains a single shape node. The shape node, in turn, has a geometry field specifying the geometry of the primitive, and an appearance field specifying the appearance (e.g., texture) of the primitive. Examples of particular geometries include spheres and boxes.

As shown in **Figure 6**, non-static primitives have a similar data structure to static primitives. However, the data structures of non-static primitives include fields that receive routed data that change the values of the fields. For example, the position field of a non-static primitive's transform node is varied when it receives a varying translation vector, and thereby causes it to translate in three-dimensional space. In one embodiment, a Java script generates the varying translation based on varying time signals from a time sensor.

B. Scene Cache Graph Data Structures

After the VRML engine parses the VRML file to create the scene node graph data structures, the cache manager generates the scene cache data structures, which are a different representation of the scene. The cache data structures are created by applying a cache visitor process to the scene graph. A cache visitor process performs a visitor pattern function to each node of the scene graph to create its corresponding cache. A cache visitor process is described in *Design Patterns, Elements of Reusable Object-Oriented Software,* authored by E. Gamma et al., and published by Addison-Wesley.

Figure 7 presents the cache data structures corresponding to the node data structures of **Figure 6**. These cache data structures include transform caches, indexed face set caches, and BSP tree caches. The transform and index face set (IFS) caches are created by the cache manager during initialization, prior to the creation of the BSP tree caches. In addition, the transform and IFS caches respectively correspond to the transform and shape nodes. Each of these two caches connects to its corresponding node via its node's cache pointer and its own original back pointer.

The transform cache also includes: (1) a children field for pointing to its IFS cache, and (2) a transformation matrix for indicating the transformations (translations, rotations, and scaling) that, for a non-static primitive, need to be performed on its indexed face set prior to their display. The IFS cache is a cache representation for the data represented by the shape, geometry, and appearance nodes. Specifically, it includes an appearance field that stores information relating to the appearance (e.g., information pertaining to the texture, material, etc.) of its face sets.

The IFS cache stores indices to tessellated face sets which collectively represent the geometry identified in the geometry node (i.e., stores indices to the coordinates of tessellated polys representing the primitive identified in the geometry node). The tessellated indexed faces either (1) are identical to the received polys that form the received primitive, (2) are further tessellated versions of the received polys that form the received primitive, or (3) are tessellated versions of a received primitive (e.g., sphere) which was not previously tessellated.

One embodiment of the invention relies on the non-static IFS caches for the duration of its operation. For instance, the non-static IFS caches are relied on for rendering the non-static primitives since non-static primitives can be rendered by iterating each of their faces individually. In addition, a non-static IFS cache checks on its corresponding node to determine whether information has been routed to its node. If so, it re-computes its fields in order to change the position and/or appearance of its polys.

On the other hand, reliance on the static IFS caches is short lived because, immediately after their creation, the cache manager creates the BSP tree data structure for spatially sorting the static polys. Thus, individual static caches are broken down into numerous polys that are spatially sorted on the BSP tree. It should be noted that the author of the VRML file, or the authoring tool used to create this file, identifies the primitives in the scene as static or non-static. Criteria for classifying a primitive as static include immovability, simplicity, and occlusion effect (i.e., the extent to which the primitive occludes other primitives in the scene).

C. BSP Tree

One embodiment of the invention uses a BSP tree (1) to perform VSD operations such the invention's DPVD process, (2) to determine the correct rendering order from the camera perspective for queuing static polys to the poly differencing engine, and (3) to perform collision detection. As mentioned above, a BSP tree is a data structure that represents a nested hierarchy of grouped partitioned volumes in a reference coordinate space (e.g., a unique coordinate space for the BSP tree or the world coordinate space). The tree represents the entire space, while each of its nodes represents a convex subspace. Each internal node in this tree (1) stores a splitting plane (also called a hyperplane) which divides the node's space (i.e., the space that the node represents) into two halves, and (2) references two nodes which represent the two halves.

Figure 8 presents one example of a BSP tree structure for the visual scene of **Figure 5** as partitioned by partitioning planes, some of which are shown in **Figure 9**. **Figure 9** does not set forth the constant y partition planes that are co-planar to the top

and bottom surfaces of the three boxes in the scene (i.e., does not present the top facing and bottom facing planes TF1-TF3 and BF1-BF3).

As set forth in **Figures 8-9**, each node in the tree structure represents a particular region of space. In addition, each splitting plane divides a particular region of space into positive and negative spaces which are defined by the direction of the normal of the splitting plane. By convention, a node hanging off a left branch represents the negative space resulting from the division of a parent space into two subspaces, while a node hanging off a right branch represents the positive space resulting from the space division.

As shown in **Figure 8**, each face of the static polys is ultimately used as a splitting plane. Furthermore, each face of the static polys resides as a leaf node for the partitioned space that it faces into. Each leaf node includes a list of non-static indexed face sets within its particular region. It should be noted that a static poly may be further tessellated so that it actually resides in multiple leaf nodes.

A three-step process is used for creating a BSP tree. First, a partition plane is selected. Some prior art authoring tools build BSP trees, and therefore in their textual VRML file identify their splitting planes. After selecting the splitting plane, the set of polys have to be partitioned by the partition plane. Finally, the first two steps are recurred until all the static polys are positioned in the BSP tree. This process is conventional and is described in the following reference.

Figures 10 and **11** present the BSP data structures for one embodiment of the invention. As shown in **Figure 10**, the BSP data structure includes BSP general-node

caches and BSP leaf-node caches. Each BSP node includes left and right fields which are pointers to the node's left and right children. In addition, a BSP leaf node includes static and non-static faces fields which respectively point to static and non-static IFS caches.

A leaf node's IFS caches are internal data structures representing the indexed faces within the leaf node's region of space. Each IFS cache stores indices into a buffer storing the vertex coordinates of the polys within the leaf node's region of space. In particular, as shown in **Figure 11a**, each IFS cache includes a coordinate array pointer for pointing to a coordinate array buffer storing the vertices for one or more faces within the leaf node.

Each IFS cache also includes a face pointer for pointing to a face buffer storing a list of indices (i.e., polys) identifying the particular order of connecting the vertices in the coordinate array buffer to obtain the leaf node's faces. The face buffer's list of indices is a sequence of positive integers for indicating the manner for connecting the vertices, and one negative integer for terminating each sequence of positive integers and thereby closing the face. As shown in **Figures 11a** and **11b**, two IFS caches can share the same coordinate array buffer. Also, as shown in **Figures 11a** and **11c**, a coordinate array buffer can store the vertex coordinates for more than one face in the leaf node, and the face buffer can store indices into the command buffer to generate more than one face.

A static BSP IFS cache is created when the original static IFS caches of the cache data structures of **Figure 7** are broken down. In other words, the static IFS caches of

Figure 7 are split into a number of leaf node static IFS caches representing the individual polys within each leaf node's partition of space.

On the other hand, each non-static BSP IFS cache corresponds to a non-static IFS cache of **Figure 7**. Each non-static BSP IFS cache represents a non-static primitive residing in the leaf node's partition of space. Such a cache points back to its leaf node. In addition, a non-static IFS cache can connect to multiple leaf nodes, because a non-static primitive can be in more than one leaf node's partition of space as it can straddle a splitting plane.

Furthermore, when a non-static primitive moves from one leaf-node partition of space to another, it has to be removed from the first leaf node and added to the second. One embodiment of the present invention removes a non-static IFS cache from all of its leaf nodes, and drops this cache's primitive down the tree to reposition in new leaf nodes, after the primitive moves. Non-static IFS caches also include flags that inform all of their leaf nodes whether they have been rendered, so that their primitive does not get rendered more than once.

Figure 12 presents the class structure used in one embodiment of the invention. As shown in this figure, the class DrawableCache is the parent class of BSP node class, BSP leaf class and IFS class (which itself is the parent class of the static IFS class, non-static IFS class, and BSP static IFS class). From their parent class, these classes inherent the visibility field, the DidDraw field, the bounding box size field, the bounding box center field, and the DrawnRectangle field.

The visibility field stores a visibility flag to indicate whether the node is visible to the camera. The visibility flag can assume one of three states, which are: Visible, Hidden, and Maybe. At the initialization of the BSP tree, all the visibility flags for all the nodes are set to Hidden. The DidDraw flag indicates whether all or part of the node's partition of space remains for display after the poly differencing engine performs its polygon clipping operation (e.g., whether the poly differencing engine creates a wedge for rendering any polys within the region of space represented by the node). As further discussed below by reference to **Figures 41** and **42**, the DrawnRectangle field stores the coordinates representing a rectangular shape recorded by the poly differencing engine to indicate the portion of a bounding box that remained after the polygonal clipping operation of the poly differencing engine.

The bounding box size and center fields are vectors which represent the center and size of orthogonal boxes for bounding the BSP nodes' partitioned spaces and the non-static IFS caches' primitives. Each bounding box is axis aligned with the BSP tree's reference coordinate system (e.g., the world coordinate system if the BSP tree is defined in this coordinate system). Furthermore, the bounding boxes for all the nodes in the BSP tree are pre-computed by the cache graph initialization process. During this process, the bounding boxes of the leaf nodes are initially computed by determining the faces of the partitioned convex volume of spaces of each leaf node. Subsequently, the bounding box for each parent node is computed by combining the bounding boxes of its children nodes. The bounding boxes then remain constant for the remainder of the operation of the system.

Figure 13 presents one process for computing the bounding boxes for the leaf nodes. This process is based on the realization that there is a unique path between any leaf node and the root of the BSP tree. The set of interior nodes (including the root node) on this path define a set of splitting planes. Some or all of these planes may be coincident with faces of the volume of space represented by the leaf node.

Hence, the faces of the leaf node volume can be generated by performing the following two operations for each node on the unique path. First, a huge polygon is generated for a node by intersecting the node's splitting plane with a huge bounding box volume. A huge bounding box is a box which has its center at the origin and has a large size (Huge, Huge, Huge), where Huge is a very large constant (e.g., a constant large enough to contain the navigable world).

Second, the huge polygon is dropped down the unique path of the BSP tree. Generally, at each node, the polygon may be sliced into two pieces, one of which continues down the unique path of the tree. However, the huge polygon is sometimes untouched by a slicing plane and either continues in its entirety down the path, or is eliminated in its entirety from the remainder of the path. When these two steps are completed, the huge polygon either has been eliminated from consideration and falls outside the leaf node volume, or a single polygon remains which represents one face of the leaf node volume.

Consequently, if these steps are performed for each splitting plane of each interior node on the unique path, a set of polygons will be generated to represent all of the faces of the leaf node volume. The bounding box of the leaf node can then be

generated by either (1) performing a min-max operation (analogous to the process set forth in **Figure 42**) to determine the minimum and maximum x, y and z coordinates among the generated polygons; or (2) combining each resulting polygon's bounding box as it is determined by performing a min-max operation.

The above-described process for computing a leaf node's bounding box can be further optimized by noting that there is a unique path between the root node and any other node in the tree. Thus, each node can recursively be treated as a leaf node in the above-described process. In particular, assuming that a node has a number of face polygons, the face polygons for each of the node's child nodes can be computed by performing the following two steps.

First, a huge polygon is generated for the node's splitting plane and is dropped down the tree along the node's unique path from the root to the current node. Dropping the huge polygon down the tree results in a non-empty polygon face, because the polygon corresponds to the most recently created face of each of the child nodes. The resulting polygon is added to the set of faces currently attached to the node.

Second, each polygon in the set of faces for the node is sliced by the node's splitting plane, in order to gather the results into two sets, a left set and a right set. The left set represents the faces of the leaf node volume of the left child, while the right set represents the faces of the leaf node volume of the right child. The polygon corresponding to the node's own splitting plane is the boundary face between the two child nodes, and is placed in both sets. In this manner, the faces for the leaf node volume of a left child and a right child are computed from the faces of their parent node.

Process **1300** of **Figure 13** is based on the above-described realizations. Initially, at step **1305**, a huge bounding box is generated for the root node. The set of node volume faces VFACES is initialized to contain the six faces of the huge bounding box. This set is linked to the root node.

At step **1310**, the process then places the root node on list OpenList. At step **1315**, the process then inquires whether list OpenList is empty. If so, the process transitions to step **1320** to terminate. Otherwise, the process transitions to step **1325** to extract the first node from list OpenList and to define it as the current node. The process then transitions to step **1330**. At this step, it determines whether the current node is a leaf node.

When the root node is initially being processed, the process at step **1330** ordinarily concludes that the current node is not a leaf node, and thereby transitions to step **1340**. At this step, it generates a huge polygon for the current node by computing the intersection of the node's splitting plane and the huge bounding box volume. For instance, in the example provided in **Figure 14**, the process at step **1340** generates the huge polygon AB for the root node by calculating the intersection of this root node's splitting plane and its huge bounding box volume.

From step **1340**, the process transitions to step **1345** to request that the previously created huge polygon be dropped down the tree from the root node to the current node. When the current node is the root node, the length of the path (beyond the implicit six splits of the huge bounding box) is zero node; therefore, the huge polygon does not get dropped down the tree. However, as shown in the example set forth in

Figure 14, the huge polygon of a node other than the root node (e.g., the huge polygon of node 1) is dropped down the nonempty unique path from the root node to the current node so that the huge polygon can be sliced by the splitting planea of the previous nodes on the path (e.g., so that the huge polygon of node 1 is sliced by splitting plane AB of the root node to generate polygon CD).

After step **1345**, the processes transitions to step **1350** to add the resulting polygon (i.e., the polygon resulting from dropping the current node's huge polygon down the unique path) to the node's set of volume faces VFACES. At step 1355, the process then slices the current node's volume faces VFACES into two sets by using the current node's splitting plane. **Figure 14** pictorially presents the operations for the splitting of the huge bounding box of the root node into two open ended boxes for its child nodes 1 and 2, and for the splitting of node 1's volume of faces VFACES into two open ended volumes for its child nodes 3 and 4.

The slicing operation at step **1355** results in a left set of volume faces and a right set of volume faces. At step **1360**, the left set of faces are linked to the current node's left child as its set of volume faces, while the right set of volume faces are linked to the current node's right child as its set of volume faces. The process then transitions to step **1365**, where the left and right child nodes are pushed onto the front of OpenList.

From this step, the process transitions back to step **1315** to determine whether the list OpenList is empty. If so, the process transitions to step **1320** to terminate its operation. If not, the process again transitions through steps **1325** and **1330** to reach the determination whether the current node is a leaf node. If the node is a leaf node, the

process transitions to step **1335** where the current node's volume faces VFACES are set to the volume faces for the leaf node. Finally, at this step, the process computes the bounding box for the leaf node based on its set of volume faces.

As mentioned before, the process can compute the bounding box by either: (1) performing a min-max operation (analogous to the operation performed in **Figures 41** and **42**) to determine the minimum and maximum X, Y and Z coordinates in the set of volume faces; or (2) performing a combining operation to combine each volume face bounding box determined by a min-max operation. One way for forming the union of two bounding box requires the conversion of both input bounding boxes from their center and size notation to a minimum and maximum notation via the following equations:

$$min = center - 0.5 * size$$

$$max = center + 0.5 * size$$

In the above equations, min, max, center, and size are all vectors. Next, vector operation MIN(V1,V2) is defined:

$$MIN(V1,V2) = (min(v1.x,v2.x), min(v1.y,v2.y), min(v1.z,v2.z))$$

Similarly, the vector operation MAX(V1,V2) can be represented as:

$$MAX(V1,V2) = (max(v1.x,v2.x), max(v1.y,v2.y), max(v1.z,v2.z))$$

The union of two bounding boxes is formed by using the MIN operation on each of the input bounding boxes' min vectors, and the MAX operation on each of the input bounding boxes' max vectors, in order to compute the min and max vectors for the

output bounding box. These computed vectors can then be converted back to center and size notation via the following equation:

center = (min+max)*0.5;

size = max-min

V. VISIBLE SCENE MANAGER

One of the processes performed after initialization is the VSD process, as shown in **Figure 4**. The VSM of **Figure 3** performs this process. Although the VSM is described below by reference to its operation in the VRML system of **Figure 3** (i.e., by reference to its operation relating to primitives originating from a VRML file), one of ordinary skill in the art would appreciate that the teachings of this process extend to the entire field of computer graphics. Thus, for example, the VSM's DPVD process applies to any graphics system to reduce the number of polys for rendering.

As mentioned above, the VSM reduces the number of static polys for rendering by not only performing traditional VSD processes (such as frustum and backface culling) but also performing the invention's DPVD process. For any camera direction, the DPVD process culls static polys which are occluded by other static polys. In one embodiment of the invention, this process is performed dynamically for each frame in order to detect changes resulting from variation in the viewpoint. By reducing the number of static polys for rendering, the VSM simplifies and accelerates the rendering process.

The invention's DPVD process reduces the number of poly-to-poly comparisons by reducing the number of polys for comparison. It reduces the number of polys by using bounding boxes to group a number of static polys together for the VSD operation. If a bounding box is not visible, then none of the polys within it need to be submitted to the rendering engine. On the other hand, if the bounding box is visible, then it is broken down into smaller boxes and the process is repeated for smaller boxes.

To determine whether a bounding box is visible, it is submitted as six polygons to the poly differencing engine, which performs polygonal clipping. If a portion of the bounding box remains after the clipping, and is processed into a wedge for rendering, the poly differencing engine (1) delays the rendering process, (2) sets the DidDraw flag for the bounding box's node to True, and (3) awaits the VSD engine's update of the visibility status of the BSP tree. The VSD engine then dynamically re-computes the visibility status of the nodes in the BSP tree.

Figures 15-22 present the steps performed by the VSD engine of one embodiment of the invention. **Figure 15** presents the three major processes performed during the DPVD process. These three processes are: (1) update the visibility states of all nodes in the BSP tree, (2) traverse the BSP tree and queue each visible static poly and each hidden node's bounding box to the poly differencing engine, and (3) perform poly differencing. Each of these process steps is performed during a frame cycle.

A. Update Visibility

Figures 16-18 present the steps performed for each node during the update-visibility process **1505** of one embodiment of the invention. As shown in **Figure 16**, the

first step of this process is step **1605**, during which a determination is made as to the current state of the visibility flag of the particular node. The visibility flags of a node can assume any one of the following three states: Visible, Hidden, Maybe. A Maybe state is provided in order to prevent oscillations from Visible to Hidden during successive frame cycles. Such an oscillation could otherwise occur when the bounding box of an node is bigger than a node's poly, and a portion of the bounding box is visible while the poly itself is occluded.

When this step is performed immediately after initialization of the BSP tree cache, the process transitions to step **1610**, because at initialization the visibility for all nodes are set to Hidden. However, after initialization, the visibility state of a node might no longer be Hidden, and therefore the process might transition to step **1615**. At step **1615**, the process determines if the node has any children. If not, the process transitions to step **1610**. Otherwise, the process transitions to step **1620** to compute the value of this node's DidDraw flag.

Process **1600** determines the value of the DidDraw flag by updating the visibility state (e.g., recursively calling itself to update the visibility) of the node's left child and right child. As the last step performed by the update-visibility process is the return of the DidDraw value, step **1620** computes the DidDraw value for a node by performing an OR function on the values returned by the update-visibility processes for the left and right children. Thus, if a node has children, it will get a DidDraw value from its children through a recursive process that results in the parent getting updated only after its children get updated. On the other hand, if a node does not have any children (i.e.,

is an indexed face set) or is a bounding box, the poly differencing engine sets its DidDraw flag.

At step **1610**, the process determines the new visibility state for the node. **Figure 17** presents the state diagram for this determination. As shown in this figure, if a node is Hidden, and its DidDraw flag is False (i.e., equals 0), the node's visibility state remains Hidden. However, if the node or a portion of it was drawn and thereby had its DidDraw flag set to True (i.e., equal to 1), the visibility state of the node is set to Visible. Once a node is Visible, its visibility state will remain Visible until its DidDraw flag is set to False. At this point, its visibility is set to Maybe. The visibility state of a node remains Maybe until it is set to Hidden after the node is culled or after a predetermined time interval expires.

After step **1610**, the process transitions to step **1625**, where a determination is made as to the old state of the node. If this state was not Hidden, the process transitions to step **1645**. Otherwise, the process determines the new state of the node at step **1630**. If the node's new state is not Visible, the process transitions to step **1645**. However, if the new state is Visible, the process transitions to step **1635** to create a mini-frustum based on the rectangular shape recorded by the poly differencing engine to indicate the portion of the node's bounding box that was drawn into a wedge.

In one embodiment of the invention, process **1600** uses conventional frustum culling processes based on the coordinates of the rectangle generated by the process of **Figure 42,** in order to create a mini-frustum. Specifically, the projection plane, the four coordinates (i.e., X_{MAX}, X_{MIN}, Y_{MIN}, and Y_{MAX}) generated by process **4200**, and the

position of the eye define four faces of the mini-frustum. The other two faces are defined by the near and far clipping plane.

From step **1635**, the process transitions to step **1640** in order to find the front most visible child leaf of the node (which was previously Hidden but now is Visible). Step **1640** will be described below by reference to **Figure 18a**. From step **1640**, process **1600** transitions to step **1645** in order to return the DidDraw value for the node being processed.

Figure 18a presents the process for determining the front most visible child leaf of a node which had its bounding box drawn into a wedge by the poly differencing engine. The first step of this process is step **1802**, where it determines which of the two children of the node at issue is closest to the camera. If the left child is closest to the camera, the process transitions to step **1804** to determine if the left child is in the frustum. For one embodiment of the invention, **Figure 18b** presents the process for determining whether the bounding box of a child node is in a mini-frustum. The process of **Figure 18b** will be described below.

If the left child is not in the frustum, the process then sets the child visibility state to Hidden (at step **1806**) and transitions to step **1814**. On the other hand, if the left child is in the frustum, the process transitions to step **1808** to determine if the left child is a leaf. If the left child is a leaf, the process sets the child's visibility state to Visible (at step **1810**) and quits its recursive process. However, if the left child is not a leaf, the process finds the visible children of the left child (e.g., recursively calls itself to find the visible children of the left child), at step **1812**.

At step **1814**, the process determines if the right child is in the frustum. If not, the process sets the visibility state of the right child to Hidden (at step **1816**), and returns to the function that called it (at step **1824**). However, if the right child is in the frustum, the process decides at step **1818** whether the right child is a leaf node. If so, the process sets the child's state to Visible and quits its recursive operation (at step **1820**). However, if the right child is not a leaf node, the process then finds the visible children of the right child at step **1822**. The process transitions to step **1824** in order to return to the calling function.

If the process determines at step **1802** that the left child is not closest to the camera, the process transitions to steps **1826-1834** to perform almost analogous operations to those performed during steps **1814-1822** described above. One difference between these two sets of steps is that the process transitions to step **1836** after setting the child state to Hidden at step **1828** or finding the visible children of the right child at step **1834**, while the process transitions to return-step **1824** after setting the child states to Hidden at step **1816** or finding the visible children of the right child at step **1822**. Similarly, steps **1836-1844** are identical to steps **1804-1812** described above, except that the process transitions to return-step **1824** after setting the child state to Hidden at step **1838** or finding the visible children of the left child again at step **1844**, while it transitions to step **1814** after setting child state to Hidden at step **1806** or finding the visible children of the left child at step **1812**.

As described above, process **1640** of **Figure 18a** is a recursive algorithm which operates on the children nodes of a previously hidden parent node in order to obtain the

visible child leaf. The process looks for the front most visible child, because it assumes that this child will occlude the other children.

The process of **Figure 18b** will not be described. By the time this process is called, the normal and distance values for each of the six faces of the mini-frustum have been determined. This process receives the bounding box center and size vectors of the child node's bounding box.

Initially, process **1850** sets the value of a variable N (representing the number of the min-frustum face) to one. Next, at steps **1854** and **1856**, the process first determines the value of a sizedistance variable, and then calculates the value of a cornerdistance variable, as shown in **Figure 18b**. At step **1858**, the process determines if the cornerdistance variable is less than zero. If not, the process determines that the child is not in the viewing frustum and returns a value of No.

Otherwise, the process increments the variable N by one, at step **1862**. It then transitions to step **1864**, where it determines if all six face of the frustum have been considered. If so, the process transitions to step **1866** to return the value Yes. However, if all the six faces have not been examined, it transitions back to step **1854**.

B. Queue Visible Polys and Hidden Bounding Boxes

Figures 19-22 present the steps performed during queuing step **1510** of **Figure 15**. As mentioned above, the queuing process traverses the BSP tree and queues visible static polys, as well as bounding boxes for hidden nodes, to the poly differencing engine. The traversal of the BSP tree is depth first and in view (i.e., camera) order. In other words, while traversing down the BSP tree, the process determines the side of the

splitting plane that the viewer is on, and initially traverses down the BSP tree on that side.

At the start of the queuing process, the VSD engine sets the DidDraw flag for all nodes to zero. The engine then performs process **1900** of **Figure 19**. Initially, at step **1905**, this process determines whether this node's bounding box is in the viewing frustum. This determination is performed by using conventional frustum culling processes, and therefore will not be described in order not to obscure the description of the invention with unnecessary detail.

If the bounding box of the node is not in the viewing frustum, the process transitions to step **1910**, where it returns a Don't Queue command signifying that neither the node's bounding box nor the nodes static polys need to be queued to the poly differencing engine for performing polygonal clipping.

If the node's bounding box is in the frustum, the process transition to step **1915**, where a determination is made as to the visibility state of the node. If the node visibility state is Hidden, the process queues the bounding box for the node at step **1920**. **Figure 20** presents the process for queuing the bounding box of a node. At step **2005**, process **2000** sets the DrawnRectangle to its initialized values of negative infinity for the maximum X, Y, and Z values and positive infinity for the minimum X, Y, and Z values. Next, at step **2010**, the process tessellates the bounding box for the node into individual polys. Finally, at step **2015**, the process calls the process of **Figure 22** for each of the tessellated polys of the bounding box, in order to queue these polys.

On the other hand, if process **1900** determines at step **1915** that the node's visibility state is not Hidden, it transitions to step **1925** to determine the dot product of the camera vector and the split-plane normal. This determination allows the process to traverse down to the BSP tree in camera order. If the dot product between the camera vector and the split-plane normal is less than zero, the process first calls itself for the node's left child and then calls itself for the node's right child. Otherwise, the process determines that the right child is in front of the camera, and therefore calls itself first for the right child and then calls itself for the left child.

Figure 21 presents the process for queuing the child node when it is a BSP leaf node. Initially, this process determines (at step **2105**) if the node's bounding box is in the viewing frustum. If not, this process returns (at step **2110**) a Don't Queue command signifying that the bounding box for the leaf does not need to be forwarded to the poly differencing engine.

However, if the node's bounding box is in the frustum, the process transitions to step **2115** where a determination is made as to the visibility status of the node. If the node is Hidden, the process queues the node's bounding box for forwarding to the poly differencing engine at step **2120**. Like step **1920** of **Figure 19** before, step **2120** of **Figure 21** is performed by process **2000** of **Figure 20,** in one embodiment of the invention.

If the node's visibility status is not Hidden, the process transitions to step **2125**. At this step, the process requests the queuing of all the static polys for the leaf node. In one embodiment of the invention, process **2100** performs step **2125** by repeatedly calling

process **2200** at **Figure 22** for each static element of the leaf node. From step **2125** process **2100** transitions to step **2130** in order to command the queuing of the leaf node's non-static polys to the non-static poly creator **335** of **Figure 3**.

Figure 22 presents the process for queuing individual static polys to the poly differencing engine. Initially, at step **2205**, process **2200** performs a conventional back-face culling operation on the static poly. For example, in one embodiment the invention, process **2200** obtains the dot product of the static poly's normal and the camera viewing direction, in order to determine if the poly is facing away from the camera. When the poly is back facing, process **2200** does not queue the poly.

However, if the poly is not back facing, the process transition to step **2215** in order to transform the vertices of the poly from the world coordinate system to the camera coordinate system. Next, the process transitions to step **2220**, where it performs a z-axis clipping operation. After this step, the process determines (at step **2225**) if any portion of the poly remains. If not, it does not queue the poly. However, if a portion of the static poly remains after the z-clipping operation, the process performs a perspective projection operation at step **2235**, by multiplying the X and Y coordinates by 1/Z.

The process then performs (at step **2240**) the screen clipping operation by clipping the portions of the poly that are beyond X_{MAX}, X_{MIN}, Y_{MIN}, and Y_{MAX}. After step **2240**, the process transitions to step **2245** in order to determine whether any portions of the poly remains after the screen clipping operation. If not, the process does not queue the poly for the poly differencing engine. However, if any portion of the poly

does remain after the screen clipping operation, the process queues (at step **2255**) the poly for the poly differencing engine.

C. Poly Differencing

As shown in **Figure 15**, the last process in the DPVD process of **Figure 15** is poly differencing process **1515**. This process is performed by the poly differencing engine. The operation of this engine is described below by reference to **Figures 23-42**. This engine performs a polygon clipping operation which clips received overlapping static polys against each other, in order to generate non-overlapping static polys for display on a display device.

Specifically, in one embodiment of the invention, the poly differencing engine **340** of **Figure 3** (1) receives linked lists of vertices of overlapping polys, (2) generates data structures representing these overlapping polys, and (3) performs polygonal clipping to obtain data structures representing non-overlapping polys. The polygonal clipping operation performed by the poly differencing engine is different than prior art scanline algorithms which clip overlapping polys in order to produce pixel data.

For one embodiment of the invention, the polygonal clipping operation of the poly differencing engine generates data structures representing non-overlapping polys from the information relating to the overlapping polys. Hence, the polygonal clipping operation of this engine is compatible with hardware graphics rendering engines which generate pixel data from the clipped data structures. In addition, this clipping operation simplifies and accelerates the rendering process by removing surfaces that do not need to be rendered.

Also, as further discussed below, the clipped data structures can be run-length encoded or quasi run-length encoded. Furthermore, based on the results of the polygon clipping operation, the poly differencing engine can determine whether a data structure was created for a bounding box, and can feed back this information to the update-visibility process of **Figure 16.** In one embodiment of the invention, the poly differencing engine forwards the static non-overlapping poly data structures to the rendering engine when it either detect no data structures representing bounding-box polys or detects no data structures representing bounding-box polys bigger than a pre-specified value.

Specifically, the poly differencing engine is conditioned to tolerate clipped data structures that represent bounding-box polys of negligible size. Unless this size is surpassed, this engine does not pass the data structures representing the clip static polys to the rendering engine. Instead, it sets a DidDraw flag for the bounding box's node to True, in order to cause the update-visibility process to determine the visible portion of the bounding box during the next frame cycle.

It should be noted that when the poly differencing engine decides not to pass the static non-overlapping data structure to the rendering engine, it also prevents the system from returning to the navigation code, in order to prevent the user from navigating the camera. In this manner, the poly differencing engine prevents the camera from being navigated while the visible surface determination is being re-computed.

In one embodiment of the invention, the static non-overlapping poly data structures are wedge data structures. As mentioned before, wedges are a particular type

of polys. They are quadrilateral polys having horizontal top and bottom. They can be represented by a pair of x-values, a pair y-values, and a pair of slopes. In this document, a slope is not defined in the traditional sense of rise over run, but rather run over rise, because a number of processes of some embodiments use the run-over-rise values to calculate the x-values from the y-values. However, in alternative embodiments of the invention, rise-over-run information is used to derive x-values and/or y-values.

The poly differencing engine's polygon clipping operation and wedge creation operation for one embodiment of the invention is described below by reference to **Figures 23-42**. As shown in **Figures 15** and **25**, the poly differencing engine receives a list of vertices for each static poly in a front to back order. One example of such a linked list is presented in **Figure 23**. This figure provides the linked list of vertices for the screen-clipped poly P1 of **Figure 24**. This engine also receives a pointer to a poly type data structure, which includes a number of fields and a pointer to a number of procedure calls for a number of poly types (such as texture mapped type, gradient-filled type, solid-filled type, etc.).

Based on the received linked list of vertices and the poly type pointer, the poly differencing engine creates a poly record for each poly at step **2505**. As the static polys are received in front-to-back order, the poly differencing engine allocates poly records in front-to-back order. One example of such a record is set forth in **Figure 26**. In this example, the poly record includes next and previous poly pointers NextPoly and PrevPoly. As further discussed below, these pointers are used during the creation of the active-edge table to keep list of all active polys, so that, when a transition from a closer to a farther poly is made, the process can walk down the list created by these pointers.

The NextPoly pointer is also employed to connect the polys in a linked list prior to their rendering, when a hardware rendering engine is used. The polygon record structure of **Figure 26** also includes the poly type pointer originally received from the VSM.

The polygon record further includes other poly data, such as color or shading attributes, or texture map indices. This record also includes the z-value of the poly at the center of the screen and the derivative of the z-value in the x and y directions. These derivatives are used to obtain the z-value of the poly at any point on the screen. The poly record structure further includes a Toggle field to indicate whether an edge in the active-edge table is a left edge or right edge.

The poly record also includes a UseCount field which is used to determine whether the poly has an edge in an active-edge table. Every time one of the poly edges is added to the table, the UseCount variable is incremented by one. Conversely, when an edge is removed from the table, the UseCount is decremented by one. When UseCount reaches zero, the poly differencing engine determines that the poly is no longer in the active-edge table.

After allocating poly records, the poly differencing engine creates (at step **2510**) edge records from the received linked lists of vertices for the screen clipped polys. A pair of contiguous vertices in a linked list creates an edge. Hence, an edge record is created by using the x and y values for two contiguous vertices in a linked list.

Figure 29 presents one example of an edge record. This edge record is for edge e4 in **Figure 24**. Thus, it includes the x and y values for the first and fourth vertices in the linked list of **Figure 23**. In addition, this record includes a poly record pointer

which points to the poly record created by the poly differencing engine at step **2505**. This pointer also serves to identify the priority of the poly (to which the edge e4 belongs to) in the front-to-back ordered group of polys received by the poly differencing engine. The pointer can act as an identifier because the poly differencing engine allocates the poly records in a front-to-back order, as described above.

The edge record also includes the slope of the edge, which is computed based on the x and y vertices, assuming that the two y vertices are not equal. In other words, the poly differencing engine does not maintain edge records for horizontal edges. The edge record further includes pointers to the previous and next edges, as well as a pointer hasAWedge to point to a wedge. The edge record also includes a pointer nextShowing, which points to the next showing edge in the linked list of active edges as further described below. Finally, the edge record includes a Boolean variable needsCalcExchange, which is used to initiate a calculate-exchange process for calculating the intersection between two contiguous edges in the active-edge table.

As shown in **Figure 25**, the poly differencing engine creates (at step **2515**) a sorted list of edges, after creating the edge records. **Figure 27** presents the sorted list of edges for the example presented in **Figure 24**. As shown in this figure, this sorted list includes a table of pointers, with each pointer pointing to a buffer storing the edge data corresponding to a scanline.

For instance, the pointer for scanline Y0 points to a buffer which indicates that edges e5 and e6, corresponding to the background screen polygon, start at scanline Y0. The start of edges e2 and e4 are then indicated in the buffer pointed to by the pointer of

scanline Y2. This sorted list also indicates the termination of edges e2, e4, e5 and e6 at the scanline Y3 in the buffer for the scanlines. Finally, when two edges cross, such a crossing can be represented in the sorted list of edges by inserting a command indicative of such crossing at the scanline for the crossing. As further discussed below, in one embodiment of the invention, crossing commands are inserted into the sorted list of edges dynamically during the creation and scanning of the active-edge table.

After sorting the list of edges, the poly differencing engine performs the invention's active-edge process, at step **2520** of **Figure 25**. In one embodiment of the invention, the active-edge process includes the creation of an active-poly linked list, an active-edge table, and a wedge linked list.

During one embodiment of the active-edge process, the poly differencing engine traverses the array of y-values in the sorted list of edges. Every time it reaches an edge start command, it adds the edge's poly record to the linked list of active polys at the appropriate position, adds the edge to the linked list of active edges at the appropriate position, and initiates a scan of the active-edge linked list in order to create a linked list of wedge records. Alternatively, every time the poly differencing engine reaches an edge stop command, it might remove the edge's poly record from the list of active edges, and removes the edge record from the list of active edges. Hence, the active-edge process is determined on a scanline-by-scanline basis.

Figure 28 presents one example of a linked list of active polys. **Figure 29** presents one example of an active-edge linked list for scanline Y_N of the example set forth in **Figure 24**. As shown in **Figure 29**, the active-edge table is sorted by the

x-values. **Figure 30** presents one example of a wedge record, and **Figure 31** presents a displayed rendition of the wedge record of **Figure 30**. As shown in these figures, in one embodiment of the invention, wedges have horizontal tops and bottoms, and are represented by a pair of x-values, a pair of slopes, and a pair of y values.

Figures 32-38 present the active-edge process performed by one embodiment of the invention. **Figure 32** presents the process for traversing the sorted list of edges to execute the commands in this list and to initiate the scanning of the active-edge table. Initially, at step **3205**, process **3200** sets a Scanline_Count variable to zero. Next, at step **3210**, the process inquires whether the scanline buffer at Scanline_Count of the sorted list of edges has any commands. If so, the process transitions to step **3215**, where it extracts the command and initiates the command's execute process. From step **3215** the process transitions back to step **3210**.

On the other hand, if there are no commands in the scanline buffer at Scanline_Count, the process transitions to step **3220** to increment the Scanline_Count value by one. Next, the process transitions to step **3225** where a decision is made whether there were any commands in the scanline buffer at the Scanline_Count. If so, at step **3230**, the process initiates the active-edge table scan process of **Figure 37**. From step **3230**, the process transitions to step **3235**. If there were no commands in the scanline buffer at the current scanline, the process transitions to step **3235**. At this step, the process determines if the Scanline_Count variable has reached a maximum value. If not, the process transitions back to step **3210**. Otherwise, the process transitions to step **3240** where it terminates.

Figure 33 presents the process for executing an edge start command, such as the ones shown in **Figure 27**. At initial step **3305**, the process determines whether the UseCount variable of the edge's poly record is equal to zero. If not, the process transitions to step **3315**. Otherwise, the process transitions to step **3310**. At this step, the process traverses the linked list of active-polyrecords, compares the priority of the edge's poly with the prioritized addresses in the list of active polys, and inserts the edge's poly record in this list at the appropriate position.

At step **3315**, the process increments the UseCount variable by one. Next, at step **3320**, the process then traverses the linked list of active edges, compares the current edge's x-value with the x-values of the edges in the list, and inserts the current edge into the active-edge table at the appropriate position. Next, at step **3325**, the process determines whether the edge that is behind the current edge in the active-edge table has a wedge. If so, the process transitions to step **3330** to initiate the close-wedge process of **Figure 34**. From step **3330**, the process transitions to step **3335**. The process also transitions to step **3335** from step **3325** when the previous edge in the active-edge table did not have a wedge. At step **3335**, the process sets the needCalcExchange flag of the current edge and the previous edge in the active-edge table. As discussed below, the setting of these flags causes a calculate-exchange process of **Figure 38** to be called to determine whether the current edge and the previous edge intersect. If so, the calculate-exchange process inserts an exchange command in the sorted list of edges at the scanline where the two edges intersect. From step **3335**, the process transitions to step **3340** to terminate its operation.

Figure 34 presents a process for closing a wedge record for one embodiment of the invention. **Figure 30** presents one example of the wedge record structure. As shown in this figure, a wedge record structure includes a pair of x-values, a pair y-values, a pair of slopes, a poly record pointer, and a next wedge pointer. When the wedge is first created between two edges, one edge supplies one x-value and one slope value, and the other edge provides the other x-value and the other slope value. Also, when a wedge is created, one y-value is derived from the current scanline value, and the poly record pointer is derived from the previous showing edge.

The process of **Figure 34** is then used to complete the wedge record by inserting the final y-value and to link the wedge in the linked list of wedges. Initially, at step **3405**, process **3400** sets the second y-value of the edge's wedge to the current scanline's y-value. Next, at step **3410**, the process links the wedge record into the list of wedges. Finally, at step **3415**, the process sets the hasWedge pointer of the edge to null.

Figure 35 presents the process for executing an edge stop command, such as the one shown in **Figure 27**. At step **3505**, process **3500** determines whether the edge has a wedge. If so, the process calls the close-wedge process of **Figure 34,** and then transitions to the step **3515**. Process **3500** also transitions to step **3515** if the edge did not have a wedge.

At step **3515**, the process decrements the UseCount variable of the edge's poly record by one. Next, at step **3520**, the process determines if the UseCount variable is equal to zero. If so, the process unlinks the poly record from the list of active polys, at

step **3525**. The process then transitions to step **3530**. The process also transitions to step **3530** if the UseCount variable does not equal zero. At this step, the process unlinks the edge record from the active-edge table. Next, the process sets the needCalcExchange flag of the previous edge, at step **3535**. Finally, the process transitions to step **3540**, where it terminates.

Figure 36 presents the process for exchanging the order of two edges in the active-edge table. Process **3600** receives as arguments two edge records. At step **3605**, this process unlinks and relinks the two edges in the active-edge table so that their order is exchanged. Next, at step **3610**, the process inquires whether the first edge has a wedge. If so, the process transitions to step **3615** to close the wedge of the first edge. From step **3615**, the process transitions to the step **3620**. The process also transitions to step **3620** from step **3610**, if the first edge did not have a wedge. At step **3620**, the process determines whether the second edge has a wedge. If so, the process closes this wedge at step **3625**. The process then transitions to step **3630** from step **3625**, or from step **3620** if the second edge had no wedge. At this step, the process terminates.

Figure 37 presents the process for scanning the active-edge table in order to generate wedges. Initially, a currentPoly variable is set to infinity, and a previousShowing pointer is set to the head of the active-edge list. Process **3700** uses the currentPoly variable to record the priority of the previous poly. Therefore, at initialization, this variable is set to infinity in order to assure that it is larger than the priority of all the subsequent polys.

Process **3700** starts scanning the active-edge table by setting an edge number variable N to one, at step **3702**. Next, at step **3704**, the process inquires whether the needCalcExchange flag for edge N has been set. If so, the processes transitions to step **3706** to call the calculate-exchange process of **Figure 38**. From step **3702**, the process transitions to step **3708**. The process also transitions to step **3708** from step **3704** if the needCalcExchange flag has not been set.

At step **3708**, the process determines if the priority of edge N's poly is less than or equal to the priority of the current poly. If so, the process determines that the poly of edge N is in front of the old currentPoly. Hence, it transitions to step **3710**, where a determination is made as to the toggle value of the poly of this edge. If the toggle value has been set (i.e., if it is True), the process determines that the edge is a right edge, and transitions to step **3712**. At this step, the process sets the toggle value to zero, traverses active-polylists to find the first poly with its toggle set, and sets a variable newPoly equal to the found poly. From this step **3712**, the process transitions to step **3716**.

If, at step **3710**, the process determines that the toggle value for the poly of edge N is not set (i.e., is False), the process determines that the edge was a left edge and transitions to step **3714**. At this step, the process sets the poly's toggle value to one, and sets the variable newPoly equal to the poly of edge N. From step **3714**, the process transitions to step **3716**.

At step **3716**, the process determines whether the nextShowing pointer of the previousShowing edge points to edge N. If not, the process transitions to step **3718** to determine if the previousShowing edge is the head of the edge list. If so, the process

transitions to step **3722**. Otherwise, the process transitions to step **3720**, where it starts a new wedge for the previousShowing edge and edge N. In addition, at this step, the process sets the nextShowing pointer of the previousShowing edge to point to edge N. From step **3720**, the process transitions to step **3722**.

At step **3722**, the process inquires whether edge N has a wedge and whether this wedge's poly is different than the new poly. If so, the process transitions to step **3724** to close the wedge. It also make a new wedge between edge N and edgeN.nextShowing. From step **3724**, the process transitions to step **3726**. Process 37,000 also transitions to step **3726** from step **3722** when either edge N has no wedge or this wedge's poly is the same as the new poly.

At step **3726**, the process sets the previousShowing pointer to edge N, and sets the currentPoly equal to the newPoly. From step **3726**, the process transitions to step **3728** where it decides whether edge N was the last edge in the active-edge table. If so, the process transitions to step **3732** to terminate. If not, the process increments the edge count variable N at step **3730**, and transitions back to step **3704**.

If the process determines at step **3708** that the poly of edge N is behind the current poly, the process transitions to step **3734** to determine the toggle value of the poly of edge N. If the toggle value is zero, the process transitions to step **3736** to set it equal to one. From step **3736**, the process then transitions to step **3740**. On the other hand, if the toggle value is equal to one, the process transitions to step **3738** to set the toggle value to zero. From this step, the process transitions to step **3740** as well. At step **3740**, the process determines if edge N has a wedge. If so, the process transitions

to step **3742** to call the close-wedge process **Figure 34**. From this step, the process transitions to step **3728**. The process also transitions to step **3728** from step **3740** when edge N does not have a wedge.

Consequently, by scanning the active-edge table, process **3700** generates a number of wedges. One example of the result of the operation of this process is depicted by **Figures 39** and **40**. **Figure 39** presents a number of polys that overlap each other in a scene. The poly differencing engine uses the above-described process to clip these polys into wedges. **Figure 40** presents the active-edge table for the polygons of **Figure 39**.

Figure 38 presents the calculate-exchange process for one embodiment of the invention. This process is called to determine whether two edges intersect in such a manner that will require an exchange command to be inserted in the sorted list of edges. In particular, at step **3805**, the process determines if an edge N and an edge (N + 1) intersect. By setting the line equations for the two edges to equal, a determination can be made as to whether the two edges intersect. If the two edges do not intersect, the process transitions to step **3820** to terminate.

However, if the two edges do intersect, the process transitions to step **3810** to determine if the intersection point is below the lower scanline of either of the edges. If so, the process transitions to step **3820** to terminate. Otherwise, the process transitions to step **3820** where it inserts an exchange command in the sorted list of edges at the calculated intersection point. From step **3820**, the process transitions to step **3820** to terminate.

As described above, the wedge creation process results in the clipping of the polys because, during this process, the poly differencing engine looks at the front-most polys at all times and does not generate wedges for portions of the polys occluded by the other polys. Furthermore, after the wedge creation process, the poly differencing engine determines if a wedge for a bounding-box poly was created. In one embodiment of the invention, the poly differencing engine traverses the linked list of wedges to determine if any wedge having a bounding-box poly type exists. If so, the poly differencing engine initiates the process for traversing the linked list of wedges of Figure 41.

Process **4100** starts (at step **4105**) by examining the first wedge in the linked list. At step **4110**, it determines whether the wedge's poly is a bounding-box poly. If not, the process transitions to step **4120** to inquire whether there are any additional wedge in the linked list. If so, the process transitions to step **4125** to go to the next edge and transitions back to step **4110**. If there are no further wedges in the linked list, the process transitions to step **4130** to terminate.

On the other hand, if, at step **4110**, the process determines that the wedge's poly type is a bounding-box poly type, the process transitions to step **4115** to call the union-rectangle process of **Figure 42**. Process **4200** initially (at step **4205**) extracts the two x-values and the two y-values from the wedge record of the bounding-box. At step **4210**, it then calculates the other two x-values of the wedge, based on the following two equations:

$$X_3 = X_1 + \text{slope } 1 * (Y_2 - Y_1)$$

$$X_4 = X_2 + \text{slope } 2 * (Y_2 - Y_1)$$

Next, at step **4215**, process **4200** performs a min-max operation on each vertex coordinate of the wedge. Specifically, at step **4215**, process **4200** compares each wedge vertex coordinate with the maximum and minimum x and y values, and resets these maximum and minimum values if the vertex coordinates are greater than or less than these values.

D. Non-static Poly Creator

As shown in **Figure 3**, one embodiment of the invention includes a non-static poly creator. Like the poly differencing engine, poly creator **335** receives linked lists of vertices and pointers to poly types. Each linked list of vertices represents a non-static poly. A number of non-static polys create a non-static graphical primitive.

Also, like the poly differencing engine, the non-static poly creator allocates a poly record for each poly that it receives, and creates a number of edge records for each linked list of vertices that it receives. It then sorts the edge records into two lists. It creates an Add list to sort the edge records in an ascending order defined by the first vertex coordinate (i.e., the vertex coordinate with the lowest y-value) of the edge records. It also generates a Remove list to sort the edge records in an ascending order defined by the second vertex coordinate (i.e., the vertex coordinate with the larger y-value) of the edge records. **Figure 45** presents one example of the two lists for the non-static poly presented in **Figure 44.**

The poly creator then scans the two lists in order to create wedge data structures for the non-static polys. **Figure 43** presents this scanning process for one embodiment of the invention. Initially, at step **4302**, process **4300** determines whether both the add

and the Remove lists are empty. If not, the process transitions to step **4304** to determine whether the y-value of the current edge in the Add list is less than the y-value of the current edge in the Remove list.

If the current y-value in the Add list is less, the process transitions to step **4308** where it sets a CurrentY variable equal to the y-value of the current edge in the Add list. The process then transitions to step **4310**. However, if the y-value of the Remove list is less, the process transitions from step **4304** to step **4306**, where it sets the value of the CurrentY variable equal to the y-value of the current edge in the Remove list.

From this step, the process transitions to step **4310** to determine whether the y-value of the current edge in the Add list equals the current y-value. If so, the process transitions to step **4312** in order to unlink the edge from the Add list and insert it into the active-edge table, and to close any wedge of a previous edge in the active-edge table. The process then transitions back to step **4314**.

The process transitions from step **4310** to step **4314** when the y-value of the current edge in the Add list does not equal the value of the CurrentY variable. At step **4314**, the process determines whether the y-value of the current edge in the Remove list equals the value of the CurrentY variable. If so, the process transitions to step **4316** to unlink the edge from the Remove list, to remove the edge from the active-edge table, and to close the wedge of this list. From the step **4316**, the process transitions back to step **4314**.

When the y-value of the current edge in the Remove list does not equal the value of the CurrentY variable, the process transitions from step **4314** to step **4318**. At this

step, the process sets the value of a toggle variable to zero, and sets the value of an edge number N variable to one. The process next transitions to step **4320** to set the value of CurrentEdge to edge N in the active-edge table, and sets the value of NextEdge to the edge pointed to by edge N's nextShowing pointer.

At step **4322**, the process then inquires whether the value of the toggle variable is one. If so, the process sets the value of a toggle variable to zero (at step **4324**), and then transitions to step **4328**. Otherwise, the process sets the value of the toggle variable to one (at step **4326**) and transitions to step **4328**. At step **4328**, the process again inquires as to the value of the toggle variable. If the value is zero, the process transitions to step **4330** to determine if the CurrentEdge has a wedge. If the current edge does not have a wedge, the process transitions to step **4338**. Otherwise, the process transitions to step **4332** to close the wedge of the CurrentEdge, and then transitions to step **4338**.

If, at step **4328**, the process determines that the toggle is set to one, it transitions to step **4334** to determine whether CurrentY has a wedge. If so, the process transitions to step **4338**. If not, the process transitions to step **4336** where it makes a wedge between the CurrentEdge and the NextEdge. From step **4336**, the process transitions to step **4338**. At step **4338**, the process determines if the current edge is the last edge in the active-edge table. If so, the process transitions to step **4302**. If not, the process increments the edge-counts variable by one, and transitions back to step **4320**. By performing the process of **Figure 43**, the non-static poly of **Figure 44** can be split into two wedges w1 and w2.

VI. RENDERING

As shown in **Figure 4**, the rendering process follows the VSD process. When the invention is used with a hardware graphics rendering engine, the poly differencing engine and the non-static poly creator supply their outputs (i.e., supply their linked lists of static and non-static wedges which they create) to the hardware rasterizer which, in turn, creates pixel data and stores this data in the frame buffer.

On the other hand, some embodiments of the invention do not use a hardware graphics engine. Some embodiments use the software rasterizer **320** of **Figure 3**. In these embodiments, the poly differencing engine and the non-static poly creator supply their outputs to the software rasterizer.

The software rasterizer does not create and store pixel data for the wedges that it receives. Instead, it renders each received wedge in the abstract. In other words, rather than rendering the wedges into display pixel data stored in an image buffer and z-values stored in a z-buffer, the software rasterizer creates and stores graphic commands for each scanline of the received wedges. These graphic commands are not pixel data which represent the displayed image at particular pixels. These commands are more abstract representations of the displayed image, and therefore are to be decoded at a later stage into pixel data for display. As shown in **Figure 3**, one embodiment of the invention uses screen writer **325** to decode the generated graphics commands to obtain the pixel data, which is then stored in the frame buffer.

In one embodiment of the invention, the software rasterizer uses a run length encoding (RLE) scheme to reduce each scanline of information in a wedge record to run-

length code. One format for this code is: (pointer to particular command to draw particular poly type) / (scanline start) / (span length) / (other-poly data). The screen writer subsequently decodes such an RLE code in order to generate display data for each pixel.

It should be noted that the other-poly-data field of the stored code can include commands requiring the indexing of other data structures. For example, this field could include texture map indices. Under these circumstances, the generated code is not an RLE code, but rather is a quasi-RLE code, as it is not entirely run-length encoded.

For one embodiment of the invention, **Figure 46** sets forth the process for populating the command buffer during the abstract rendering operation of the software rasterizer. This process will be described by reference to **Figure 50**, which provides an example of four wedges for display, and **Figure 51**, which presents one manner for rendering these four wedges in the abstract in the command buffer.

As shown in **Figure 46**, the first step **4605** in the abstract rendering process of the software rasterizer is the initialization of a command buffer. **Figure 51** presents one example of such a buffer. This buffer includes a table of pointers and a number of individual scanline buffers for each scanline. The table includes a pointer for each scanline. A particular scanline's pointer points to the scanline's individual scanline buffer which can store one or more sets of graphics codes representing the displayed static and non-static polys intersecting the scanline.

The next step in process **4600** of **Figure 46** is step **4610**. At this step, the process reads the received static-wedge linked list and populates the command buffer

with codes representing the displayed static polys. **Figure 47** presents the process for recording graphics commands relating to a particular wedge in the command buffer. Initially, (at step **4705**) this process initializes three variables. It initializes X1var to the wedge's first x-value (x1) at its first y-value (y1), initializes X2var to the wedge's second x-value (x2) at its first y-value (y1), and initializes Scanline_Count to the first y-value (y1).

At steps **4710-4720** process **4700** then inserts the command type (e.g., DrawPolyType1), the scanline start value (i.e., X1var), the span length value (i.e., (X2var - X1var)), and other poly data (such as a texture map index) into the scanline buffer for the current scanline as identified by the variable Scanline_Count. The process then increments the Scanline_Count by one, and increments the X1var and X2var by the two slopes of the wedge. Because the scanline index (i.e., the y-value) is only incremented by one, and because an x-value equals a slope value times a y-value plus a constant, the x-axis values of the wedge can simply be computed by adding the slopes.

If the Scanline_Count does not exceed the second y-value from the wedge record, the process returns to step **4710** to populate the next scanline. Otherwise, the process terminates at step **4730**. By repeatedly performing populate RLE procedures (such as the one set forth in **Figure 47**), all the static wedge structures can be broken down and recorded scanline-by-scanline in the command buffer. **Figure 51** sets forth an example of populating a command buffer with the three static wedges SW1-SW3 of **Figure 50**.

After step **4610**, process **4600** of **Figure 46** transitions to step **4615**, where it determines whether the software rasterizer has received wedges for non-static polys. If

not, the process transitions to step **4630**. Otherwise, the process transitions to step **4620** in order to add z-fill commands for the previously entered static elements on scanlines which include non-static wedges. For one embodiment of the invention, **Figure 48** presents a process for entering z-fill commands for static wedges into the command buffer. As shown in this figure, this process initializes three variables at step **4805**. It initializes X1var to the wedge's first x-value (x1) at its first y-value (y1), initializes X2var to the wedge's second x-value (x2) at its first y-value (y1), and initializes Scanline_Count to the first y-value (y1).

Next, at step **4810**, the process determines whether any non-static wedges reside on the current Scanline_Count. In one embodiment of the invention, process **4800** makes this determination by using a non-static scanline array in which a flag is set for each scanline that contains non-static information. If the current scanline includes non-static information, the process transitions to steps **4815-4825** to insert z-fill commands, scanline start values, and span-length values in the scanline buffer for the current scanline as identified by the variable Scanline_Count. The process then increments (at step **4830**) the Scanline_Count by one, and increments the X1var and X2var by the two slopes of the wedge. If the Scanline_Count exceeds the second y-value from the wedge record, the process terminates at step **4840**. Otherwise, it returns to step **4810** to populate the next scanline.

If, at step **4810**, the process determines that the current scanline does not have non-static information, it transitions to step **4845**. At this step, the process determines the number of scanlines having no non-static elements by examining the non-static scanline array of **Figure 49**. In other words, by examining this array, the process

determines a skip count. It then uses the skip count to increment (at step **4850**) the X1var, X2var, and Scanline_Count. After this step, the process transitions to step **4835**, where it determines if the Scanline_Count has reached the wedge's second y-value (Y2). If so, the process terminates. **Figure 51** sets forth an example of a command buffer that has been populated with z-fill commands for the static wedges of **Figure 50**.

After step **4620**, process 46,00 of **Figure 46** transitions to step **4625**. At this step, the process reads the received non-static wedge linked list and populates the command buffer with codes representing the displayed non-static polys. The process for recording graphics commands relating to a particular non-static wedge is identical to the process presented in **Figure 32** for recording static wedges. Hence, this process will not be described, in order not to obscure the description of the invention with unnecessary detail. The command buffer of **Figure 51** includes commands for the non-static wedge NSW1 of **Figure 50**. Process **4600** lastly transitions to step **4630**, where it appends an END command to the end of each scanline buffer in the command buffer. A scanline END command signifies that no other commands exist for the current scanline. **Figure 51** plays these END commands.

As mentioned above, one embodiment of the invention uses screen-writer **325** of **Figure 3** to decode the generated graphics commands to obtain the pixel data, which is then stored in the frame buffer. **Figure 52** present the screen-writer process for one embodiment of the invention. Initially, at step **5205** process **5200** initializes the Scanline_Count value to zero.

It then determines (at step **5210**) whether the next command in the current scanline buffer is an END command. If not, the process transitions to step **5215** to retrieved a set of code from the command buffer. At step **5220**, the process then uses the draw command type contained in the retrieved set of code to call the draw procedure for the particular poly type of the run-length encoded wedge segment.

As mentioned above, the poly type data structure includes a number of fields and procedure calls for a number of poly types (such as texture mapped type, gradient-filled type, solid-filled type, etc.). **Figure 53** presents one example of a process called through this data structure. This process generates pixel data for a solid-filled poly type from the retrieved code.

As shown in **Figure 53**, process **5300** initially (at steps **5305** and **5310**) extracts the scanline start, span length, and color data (stored under the other-poly-data field) from the retrieved code. It then sets Span_Count to the extracted scanline start value. At step **5320**, process **5300** sets the pixel data value, for the pixel identified by the Span_Count value, to the designated color of the particular solid filled type. The process then increments the Span_Count value. After this step, the process determines if it has performed this operation for the span length extracted from the code. If not, the process transitions back to step **5320**. If so, the process terminates.

After performing step **5220** in **Figure 52**, process **5200** returns to step **5210**. At this step, it decides once again if any additional sets of code remain on the current scanline. If not, the process transitions to step **5225** to increment the Scanline_Count. Subsequently, at step **5230**, it determines whether the Scanline_Count has reached its

maximum value. If not, it transitions back to step **5210**. Otherwise, the process terminates at step **5235**.

When there are more than one set of commands for a particular scanline in the RLE buffer, screen writer **325** performs steps **5210-5220** once for each set of commands. Each time that it performs these steps and generates pixel data by decoding these commands, it writes the pixel data in a row of the frame buffer, as shown in **Figure 54**.

In addition, if there is a non-static poly on the particular scanline, the screen-writer initially extracts the z-fill commands for the static polys, and therefore initially fills a one row z-buffer with the z-values for the static polys. It then extracts the command for the non-static poly. Based on this command, it generates pixel values and z-values for the non-static polys. For each non-static-poly pixel, the screen writer then determines whether the non-static-poly pixel is closer to the viewer than the point which currently has its pixel and depth values stored in the buffers. If so, the screen writer replaces the non-static poly's pixel and depth values with the old values. Otherwise, the non-static poly's values are discarded.

As it is apparent from the foregoing discussion, the RLE abstract rendering process of the invention dramatically simplifies the process of writing pixel values to the frame buffer. In particular, it obviates the need for a separate z-value memory array, since the RLE commands can be decoded into properly-ordered pixel values by using a simple one-row z-buffer. The abstract rendering process also improves performance because it increases the cache coherency of the screen-writing process.

VII. COLLISION DETECTION

A system for visualizing model worlds may need to detect collisions if there is movement within the world and if some realistic simulation of the physical world is desired. One conventional method detects collisions by comparing the space occupied by every geometric element within the world with every the space occupied by every other geometric element for each frame generation cycle. As the use of 3D model world visualization increases, the desire grows for more complex worlds with more realistic simulation. The work of collision detection by conventional methods grows as the square of the number of geometric elements, and a small growth in world complexity may quickly outstrip the ability of the visualization system to perform collision detection within the time available to achieve a given frame rate. This is especially true when the increased desire for realism necessitates gathering descriptive information about a collision event other than the mere fact it occurred. Consequently, there is a need in the art for an improved collision detection mechanism.

Figure 55 depicts the high-level software architecture of the collision detection mechanism. The collision detection mechanism is invoked during each frame-generation cycle in the life of the VRML player as previously discussed in reference to **Figure 4**.

In the presently described embodiment, the collision detection mechanism **410** comprises object-to-object **5510** and camera-to-object **5520** subcomponents, and a set of common routines **5530** shared between them. Object-to-object collision detection **5510**, as the name suggests, identifies collisions that do result, or that are projected to result, from the movement of objects during the viewing of the model world. Camera-to-object

collision detection **5520** identifies collisions that do result, or that are projected to result, from the movement of the viewpoint in the scene.

Common routines **5530** may be employed because the VRML player in this embodiment inserts a logical camera object into the model world space to represent the viewpoint, rather than treating it as a single point in space with an associated direction.

Figure 56 depicts an image of a sample model world. The sample model world is used to explain the operation of the collision detection mechanism for one embodiment employing the present invention. The image reveals a back wall **5610**, a floor **5620**, and a dividing wall **5630** that separates a first compartment **5660** to its left, from a second compartment **5670** to its right. The first compartment **5660** contains 1) a four-faced pyramid object **5640** positioned slightly above the floor plane **5622**, 2) the facing plane **5632** of the dividing wall, and 3) a small box object **5650** suspended above the floor **5620** but below the level of the top of the pyramid object **5640**, near the facing plane **5632** of the dividing wall and between the dividing wall **5630** and the pyramid **5640**.

Figure 57 depicts representative data components utilized during operation of the collision detection mechanism. In order not to obscure an understanding of the invention, **Figure 57** includes only those data components especially helpful to sufficiently and succinctly explain the collision detection mechanism of the presently described embodiment. For example, only selected elements of the earlier described, and much larger, BSP tree and scene graph are shown.

The representative data components depicted include a BSP Root Node **5701**, a BSP Leaf of the BSP tree for the scene being viewed, shape nodes **5720, 5730** of the

scene graph for the scene being viewed, and indexed face sets **5750, 5760, 5770** and associated face elements **5754, 5764-5769, 5774-5777** for the facing wall plane **5632**, box **5650**, and pyramid **5640** of **Figure 56**.

Other data components depicted include the navigation information node **5740** of the scene graph and a collision volume list **5780**. The navigation information node **5740** provides information about a logical camera object, or avatar, viewing the scene. The navigation information node is created as part of the scene graph when the VRML definition of the model world is parsed.

The collision volume list **5780** is repeatedly built, used, and destroyed during the operation of the VRML player embodiment presently described. Each entry in the collision volume list **5780** includes a geometric definition of a collision volume and working storage for variables used during the collision detection process, among other components.

The BSP Leaf **5710** represents the first compartment **5660** of space depicted in **Figure 56**. The compartment is bounded at the back by the facing plane of the back wall **5612**, at the bottom by the floor plane **5622**, and at the right by the facing plane **5632** of the dividing wall **5630**. The compartment of space is further bounded at the top by an invisible plane containing the top edges **5616, 5636** of the back wall **5610** and the dividing wall **5630**, by a second invisible plane containing the front edges **5626, 5634** of the back wall **5610** and the dividing wall **5630**, and a third invisible plane containing the left edges **5614, 5624,** of the back wall **5610** and the floor **5620**.

The BSP Leaf **5710** is so constructed as to point to the members of the set of indexed face sets representing the polygonal faces of static objects in the model world that are coplanar with the bounding planes of the compartment represented by the leaf **5710**. For example, indexed face set **5750** represents the facing plane **5632** of the dividing wall **5630** bounding the right side of the first compartment **5660** depicted in **Figure 56**.

The BSP Leaf **5710** is so constructed as to point to shape nodes **5720, 5730** for non-static objects which to some extent occupy the compartment of space represented by the BSP leaf **5710**. A non-static object may span multiple compartments or reside entirely within a single compartment. For example, the box **5650** and pyramid **5640** of **Figure 57** each reside entirely within the first compartment **5660** depicted in **Figure 56**. Further, a non-static object may occupy all or part of the space within a compartment it occupies. For example, the box **5650** and the pyramid **5640** each occupy only a part of the space within the first compartment **5660**.

Shape nodes **5720, 5730** in turn are so constructed as to point to the indexed face sets representing the polygonal faces of the object the shape node represents. For example, the shape node **5720** for the box **5650** of **Figure 56** points to the indexed face set **5760** having the top **5764**, back **5765**, right **5766**, front **5767**, left **5768**, and bottom **5769** faces which together make up the box.

The indexed face set **5760** also contains a bounding box definition **5763.** The bounding box delimits a volume of space that can contain the object composed of all the faces of the set **5764-5769**. The presently described embodiment employs as the

bounding box, the smallest, axis-aligned, rectangular volume in local space that can contain the object. As such, the bounding box for a rectangular solid has the same geometry as the object itself. The bounding box is represented as a center point plus height, width and depth dimensions.

The box's indexed face set move flag **5761**, for the sake of the ensuing description, is set to the "off" state indicating that the box is stationary for the present operation cycle.

The shape node **5730** for the pyramid **5640** of **Figure 56** points to the indexed face set **5770** having the back **5774**, left front **5775**, right front **5776**, and bottom **5777** faces which together make up the pyramid. The bounding box definition **5773** describes the bounding box appropriate to the pyramid. The move flag **5771**, for the sake of the ensuing description, is set to the on state indicating that the pyramid is changing its position in space from that of the previous operation cycle.

A. Object-to-Object

Figure 58 depicts a flowchart of the object-to-object collision detection process. To better describe the operation of the process, the description includes an example based on the model world space depicted in **Figure 56** and its associated data components depicted in **Figure 57**. Referring to **Figure 56**, the example involves collision detection within the first compartment **5660** for a processing cycle wherein the pyramid **5640** moves toward the facing wall plane **5632,** on a path perpendicular to it. The box **5650**, though movable, is stationary in the current processing cycle.

Step **5804** locates the root node **5701** of the BSP tree via a pointer contained in the scene graph. Step **5810** traverses the BSP tree top-to-bottom and left-to-right using methods well known in the art. As the tree is traversed, nodes not having been marked as visible (by the image generation mechanisms described earlier), are skipped, as are their dependent nodes and leaves. Step **5812** evaluates each BSP leaf subordinate to a visible node to assess whether the compartment represented by the leaf is, itself, visible. If the BSP leaf is not visible, it is skipped and traversal of the BSP tree continues in step **5810**. If the BSP leaf is visible, the process proceeds to step **5814**.

Step **5814** begins the processing cycle performed for each visible leaf discovered in the BSP tree. For example, when the presently described embodiment traverses the BSP tree shown in part in **Figure 57**, it starts at its root node **5701** and works down the branches **5705**. Eventually the collision detection mechanism encounters the BSP Leaf **5710**. This BSP Leaf **5710** represents the volume of space in the first compartment **5660** of **Figure 56** that contains the box and pyramid. Because the visibility flag **5712** for the leaf is in the visible state in this example, collision detection proceeds to step **5814**.

Step **5814** determines whether any non-static objects within the compartment require notification of collision events. This is determined by checking the contents of the collision node notifier component of each shape node pointed to by the BSP Leaf. If the collision node notifier component is not logically empty, collision detection is required for that object and thus for the leaf. The VRML parsing process which builds the scene graph initially determines the contents of the collision node notifier component.

If collision detection is not required, step **5810** continues traversal of the BSP tree. If collision detection is required, control passes to step **5820**. For example, referring to **Figure 57**, the collision detection mechanism tests the contents of the shape nodes **5720, 5730** and determines that the collision mode notifier **5732** in the pyramid's shape node **5730** is filled. As a result, collision detection must be performed for the leaf so control passes to step **5820**.

The collision detection mechanism, having found a compartment which may be visible and which contains some geometry for which collision detection is desired, now sets out to identify collisions within the compartment. Because only moving objects can cause collisions, and only non-static objects can move, the collision detection mechanism first identifies the moving, non-static objects.

Step **5820** traverses the set of non-static shapes in the compartment. For each non-static shape, step **5822** determines whether it moves as part of the current frame-generation cycle. If the shape does not move, it is skipped and traversal of the set continues at step **5820**. If the shape does move, step **5824** adds a collision volume to a list of collision volumes for the compartment and traversal of the set continues at step **5820**. Control passes to step **5830** when the list of non-static objects has been exhausted. In the present embodiment the collision volumes list is constructed in computer memory.

For example, referring to **Figure 57**, the collision detection mechanism sets out to identify the moving, non-static objects in the compartment represented by the BSP Leaf **5710**. The collision detection mechanism follows a first pointer **5718** from the BSP

Leaf **5710** to the shape node **5720** for the box. The collision detection mechanism then follows a pointer **5724** from the shape node **5720** to the indexed face set **5760** for the box. The move flag **5761** in the indexed face set indicates that the box is stationary, so no collision volume is built for the box. The collision detection mechanism then follows a second pointer **5719** to the shape node **5730** for the pyramid, and a subsequent pointer **5734** to the indexed face set **5770** for the pyramid. The move flag **5771** indicates that the pyramid moves, so a first collision volume is built and put in a list for the compartment. The first collision volume in the list is also the last as there are no more non-static objects in the compartment and control passes to step **5830**.

An entry in the collision volume list in the present embodiment includes a geometric definition of the collision volume for an object. The collision volume for an object is closely associated with its bounding box. The bounding box represents the stationary object while the collision volume represents the moving object. The collision volume may result as the logical projection of the bounding box along a travel path.

Figure 59 depicts a sample object with its bounding box, projection volume and collision volume. The pyramid **5640** of **Figure 56** is the example object with its rear, hidden edge illustrated as a broken line **5642**. The object's bounding box **5910** possesses height **5952**, width **5956** and depth **5954** dimensions. Movement vector **5958** indicates the length and direction of movement of the object **5640** for the current processing cycle. In this example, the direction of movement vector **5958** is axis-aligned with the width dimension **5956** of the bounding box **5910**.

The logical projection volume **5920**, possesses the same height **5952** and depth **5954** dimensions as the bounding box **5910**. The projection volume's width dimension is the length of movement vector **5958**.

The collision volume **5930** for the object is the sum of the bounding box **5910** and the projection volume **5920**. In this example, the collision volume **5930** possesses the same height **5952** and depth **5954** dimensions as the bounding box **5910**. The collision volume's width **5960** is, however, the width **5956** of the bounding box **5910** extended by the length of movement **5958**. The collision volume is an approximation of all the space that would be occupied at some point in time during the simulated movement of an object along its movement vector.

Referring again to **Figure 58**, step **5830** traverses the just completed collision volume list. For each collision volume, the collision detection mechanism identifies collisions between it and the static elements in the compartment in steps **5840** to **5846**, and between it and the non-static objects in the compartment in steps **5850** to **5858**.

Step **5830** traverses the list of static elements in the compartment. For each static element in the compartment, step **5842** performs a rough test for collision between the current collision volume and the current static element. If step **5842** determines that no collision between them is possible, control passes back to step **5840**. If step **5842** determines that a collision is possible, step **5844** performs a precise test for collision between the object represented by the current collision volume and the current static element. The rough test looks for intersections of the space occupied by a bounding sphere for the collision volume with the bounding box for the static element. The

precise test looks for intersections of the collision volume with the actual face of the static element.

The two-step approach of collision detection seen in steps **5842** and **5844** is employed in the present embodiment to reduce the work required in total to perform collision detection. The rough test **5842** requires relatively little work as compared to the precise test **5844**. The additional work of performing the rough test is presumed small compared to the work saved by eliminating unnecessary precise tests.

Step **5846** reports the results of the precise test. If no collision was detected, no reporting activity is required and control passes to step **5840**. If a collision was detected, the collision detection mechanism records relevant information. One embodiment records the collision result information in the current entry of the collision volume list.

Figure 60 depicts details of the data structures for the collision volume list. The collision volume list **5780** may contain many individual entries. A representative entry **5782** comprises collision volume geometry **6002**, orientation **6004**, plane equation **6006**, and intersecting splitting plane list **6008** components. The entry **5782** further comprises occupied BSP leaf compartment list reference **6010**, collision flag **6012**, collision distance **6014**, colliding polygon list reference **6016** and work area **6018** components.

The colliding polygon list reference **6016** identifies a list **6040** of the polygons with which a collision is detected. A representative entry **6042** in the colliding polygon list **6040** comprises nearest flag **6052**, indexed face set reference **6054**, face set index **6056**, shape node reference **6058** and opposing direction **6060** components.

One embodiment of a collision detection mechanism reports collision information as follows. The collision flag **6012** is set to the "on" state. A colliding polygon list entry **6040** is created for the specific face involved. The new entry records a reference **6054** to the indexed face set containing the face, the index value **6056** identifying the face within the set, a possible reference **6058** to a shape node representing an object containing the indexed face set, and the opposing direction **6060** at the point of collision so that the angle of collision is known. If the collision represented by the new entry is the nearest collision detected for the collision volume, the "nearest" flag **6052** in the entry is set and the distance is recorded in the distance component **6014** of the collision volume list entry **5782**.

Notably, the information recorded for each collision exceeds the requirements of viewing VRML worlds according to the VRML Specification. The basic information recorded can be used to process VRML Specification-compliant collision nodes. The extended information can be made available, e.g., via a VRML Proto node type. Java programs associated with script nodes defined in a VRML model world description may take advantage of the extended collision information provided in the Proto node type to effectuate more sophisticated collision responses than is possible using the ordinary collision nodes. For example, a projectile object in a game world may be made to accurately ricochet off of another object based on the angle of collision information presented by the Proto node.

All object-to-object collision detection lies, in fact, beyond the base VRML Specification. The ordinary collision nodes only encompass camera-to-object collisions.

Figure 61 depicts the sample scene superimposed with collision detection elements. A dashed outline **6140** represents the post-move position of the pyramid. Radius line **6110** projects from the geometric center point **6112** of the collision volume **5930**, to a vertex **6114** most distant from the center point. The radius line **6110** defines the bounding sphere **6120** of the collision volume for the pyramid. The surface of the bounding sphere **6120** is made up of all points in space with a distance from the center point **6112** equal to the length of the radius line **6110**. The bounding sphere **6120** is used in performing the rough test for collision. An example follows.

Referring to **Figure 57**, the collision detection mechanism sets out to identify collisions between moving objects and the static elements in the compartment represented by the BSP Leaf **5710**. Starting with the first and only member of the set of collision volumes constructed for the compartment, that of the pyramid, the collision detection mechanism follows a first pointer **5717** to the indexed face set **5750** for the facing plane of the dividing wall. The rough test determines that the space occupied by the pyramid's bounding sphere intersects with the space occupied by the bounding box **5753**. **Figure 61** shows this intersection where the right side of the bounding sphere **6120** passes through the wall **5630**. The process for determining such intersections is well known in the art and an example is described in "A Simple Method for Box-sphere Intersection Testing," James Arvo, in Graphics Gems (Andrew Glassner, ed., Academic Press).

The precise test then determines the exact nature of any actual collision between the pyramid and the facing wall plane by looking for intersections, one at a time, between the pyramid's collision volume **5930** and the each of the faces in the facing wall

plane's indexed face set **5750**. The collision detection mechanism determines that the pyramid's collision volume intersects the wall. **Figure 61** shows the rightmost end of the collision volume **5930** extending through the facing wall plane **5632**. Collision points **6132-6138** depict the intersections of the edges of the collision volume with the facing wall plane. Again, the process for determining such intersections is well known in the art and an example is described in Computer Graphics (Hearn and Baker, Prentice Hall).

After the collision detection mechanism checks for collisions between the current collision volume and static elements in the compartment, it compares the current collision volume with the non-static elements in the compartment. Referring to **Figure 58**, steps **5850** through **5858** determine collisions with non-static objects. Step **5850** traverses the list of non-static objects in the compartment. For each object, step **5852** determines whether the object represented by the collision volume and the current object in the non-static list have already been compared. This could occur if a collision volume that appears earlier in the collision volume list represents the current object in the non-static list. If so, the comparison need not be repeated and traversal of the non-statics list continues at step **5850**. If not, control passes to step **5854**.

Step **5854** through **5856** operate to detect collision between the current collision volume and non-static objects, as steps **5842** through **5844** operate for static elements, with the following difference. When the current member of the non-static object list is, itself, a moving object, step **5854** looks for an intersection between the collision sphere and the object's collision volume, rather than its bounding box.

When step **5850** determines that the list of non-static objects in the compartment has been exhausted, control passes to step **5830** to proceed with the next collision volume for the compartment. When step **5830** determines that the list of collision volumes for the compartment has been exhausted, control passes to step **5832**. Step **5832** discards the collision volume list for the compartment and control passes to step **5810** to continue traversal of the BSP tree. When the BSP tree has been completely traversed the object-to-object collision detection process is complete **5816**.

B. Camera-to-Object

The VRML player also detects collisions of the logical camera viewing the scene with the objects that exist in the model world. These collisions fall into two categories. Collisions with objects below the camera, or camera-to-terrain collision, determine whether the camera needs to step up or step down to follow the terrain. Collisions with objects in the direction of motion of the camera, or camera-to-object collisions, determine whether the motion of the camera needs adjustment. For example, a collision with a wall may cause the camera to stop or to slide along the surface of the wall.

Figure 62 depicts a flowchart for the camera-to-object collision detection process. Step **6210** constructs a collision volume for the camera object. The camera object represents a logical vehicle, or avatar, carrying the viewpoint through the model world. The camera object may take on dimensions to represent, e.g., a person walking through the scene, or a space ship flying through the scene.

In the presently described embodiment, the collision volume of the camera object is recalculated for each frame generation cycle. This allows for movement of the camera and for changes in the size of the camera object.

The navigation information node in the scene graph contains some of the dimensional information used to construct the collision object for the camera. Referring to **Figure 57**, the navigation information node **5740** holds an avatar size component **5742**. The avatar size component **5742** further holds 1) an allowable distance component **5743** representing the allowable distance between the viewpoint's position and any collision geometry before a collision is detected, 2) a height component **5744** representing the height above the terrain at which the viewpoint should be maintained (i.e. eye level), and 3) a step height component **5745** representing the height of the tallest object over which the camera object can "step." The navigation information node also contains a "type" component **5748** that specifies the navigation paradigm the player should use, e.g., walking or flying.

Figure 63 depicts an example of a camera collision volume as implemented in the presently described embodiment. Plane **6310** represents the scene terrain. A point in space **6372** and a direction of view **6374** represent the viewpoint vector **6370**. The difference between the values of the height component **5744** and the step height **5745** component from the navigation information node **5740**, depicted in **Figure 63** as **6362** and **6352**, determines the camera object's height dimension **6352**, which projects downward from the viewpoint **6372**. The value of the allowable distance component **5743** of the navigation information node **5740** determines the camera object's length **6356**, which projects in the direction of view **6374**. One half of the camera object's

width projects from each side of the viewpoint. The camera object **6310** is then extended along the direction of view **6374** for the distance of motion **6380** to produce the camera collision volume **6330**.

The presently described embodiment sets the camera object's width **6354** to a fixed value depending on the navigation paradigm type **5748** in the active navigation information node **5740** of the scene graph. If walking, the camera object width takes on a value equal to one fifth of the height component to approximate human proportions. If flying, the camera object width takes on a value equal to the allowable distance component **5743** to approximate the proportions of an envisioned aircraft.

Step **6220** then begins traversal of the BSP tree after the same fashion as used in the object-to-object collision detection process. When a visible leaf is found, step **6222** determines whether the space occupied by the camera object intersects with the compartment represented by the BSP leaf. If not, traversal of the BSP tree continues at step **6220**. If so, control passes to step **6230**.

Notably, the collision detection mechanism does not check whether collision detection is active for other objects within the compartment. The camera object occupies the compartment and the presently described embodiment always detects collisions of the camera object.

Steps **6230** through **6236** identify collisions between the camera object and static elements in the current compartment. Step **6230** traverses the list of static elements in the compartment. For each static element in the compartment, step **6232** performs a rough test for collision between the camera collision volume and the current static

element. If step **6232** determines that no collision between them is possible, control passes back to step **6230**. If step **6232** determines that a collision is possible, step **6234** performs a precise test for collision between the camera object and the current static element. The rough test looks for intersections of the space occupied by a bounding sphere for the camera collision volume with the bounding box for the static element. The precise test looks for intersections of the camera collision volume with the actual face of the static element. Step **6236** reports the results of the precise test. This mimics the operation of steps **5840** through **5846** in the object-to-object collision detection process depicted in **Figure 58** and further details of processing are not repeated here.

After the list of static elements in the compartment is exhausted, control passes to step **6240**. Steps **6240** through **6246** repeat for non-static objects the processing performed in steps **6230** through **6236** for static elements in the compartment. When the non-static element list has been exhausted in step **6240**, control passes to step **6220** where traversal of the BSP tree continues looking for the next compartment containing the camera object. When step **6220** finishes traversing the tree, camera-to-object collision detection is completed **6226**.

The processes described above utilize the subdivision of the space in the model world to reduce the number of comparisons that need to be performed to detect all relevant collision possibilities. Rather than comparing the volume of space traversed by each and every object in the model world with each and every other, the comparison is performed only among the objects traversing the same subdivision of space. This is true for both the object-to-object and the camera-to-object collision detection processes described above.

The number of comparisons for the world is further reduced by increased selectivity. By characterizing a partition, and implicitly all of the objects associated with it, according to indicia relevant to its need for collision detection, entire partitions, and implicitly all of the objects associated with them, can be summarily bypassed by the collision detection mechanism. For example, by characterizing a partition based on its location inside or outside of the viewing frustum, all non-visible partitions can be ignored if the application is only concerned with collisions that immediately impact the generated image. Moreover, in an implementation utilizing a hierarchical data structure to order the subdivisions of space (e.g., a BSP tree), entire sub-hierarchies can be summarily bypassed if the relevant characterizations are recorded above the lowest level in the structure. The higher frame rates made possible through such reductions of collision detection complexity represent a further advantage of the invention.

One skilled in the art recognizes that the foregoing description of one embodiment of a collision detection mechanism sets forth many details for which alternatives can be employed. For example, lists of related data components may be practiced using arrays, singly linked lists, doubly linked lists, or a variety of other data storage and structuring techniques well known in the art. Geometry for rectangular solids may, e.g., be represented as a set of six plane equations, a set of eight vertices, or as a center point and height, width, and depth information. At any particular point, a particular element of geometry may be represented according to a local, global or intermediate coordinate system. Collision volumes may be, e.g., boxes, cross-sectional projections, or precise volumetric projections along the path of motion.

Further, the ordering of individual steps, as well as the ordinal and hierarchical relationships of processing loops may vary. For example, non-static elements could be processed before static elements, or their processing could be interleaved. Collision detection for selected individual objects could be performed on a demand basis rather than processing all objects together, much like camera-to-object collision detection is performed apart from object-to-object collision detection but using common architectural elements. These and other alternatives may be employed without departing from the scope and spirit of the invention.

VIII. FACTOR INTERPOLATION

The VRML Specification recognizes the utility of interpolation in visualizing 3D worlds. Often it is desirable to change the rotation, color, position, or size of an object in direct relation to some other quantity. For example, the height of a ball off of the floor may be related to the percentage of completion for a bounce. Many times the desired relationship between the changing property and the defining quantity is linear. Many times the desired relationship is more complex, e.g. a curve, but capable of adequate representation as a series of linear relationships each defining a segment of the curve. The VRML Specification provides support for such interpolation through its Interpolator Node types (refer to VRML §4.9.3).

Figure 64 depicts Interpolator Node structures. During the parsing and scene graph construction process described earlier, the initialization process inserts an interpolator node **6420** into the scene graph when an interpolator definition **6412** is

discovered in the VRML code **6410**, and constructs a logical key frame table **6430** associated with the node **6420** from information in the VRML definition **6410**. Each key value appearing in the interpolator definition **6412** occupies the key position **6437** in a corresponding key frame. As many key frames are constructed as there are key values in the interpolator definition **6412**. In the example shown, five key frames **6431-6435** result from five key values (0.00, 0.25, 0.50, 0.75, 1.00) **6414** in the interpolator definition **6412**. The key frames **6431-6435** are ordered by their respective key values **6437**. The key frames logically appear in ascending order in one implementation, and the contents of each key frame can be accessed by reference to its ordinal position **6439** in the set.

Each value-field **6416** value appearing in the Interpolator definition **6412** comes to occupy the value position **6438** in a key frame. Value-field values are placed into the key frames in the order in which they appear in the definition **6412**. In the example shown, the five values (1.0, 2.0, 4.0, 7.0, 11.0) **6416** appearing in the Interpolator definition **6412** populate the Value positions **6438** in corresponding key frames **6431-6435**.

Note that **Figure 64** depicts the definition **6412** for a ScalarInterpolator type of interpolator node. The value-field **6416** values are scalar floating point numbers. The simplicity of the ScalarInterpolator type avoids unnecessary detail that might otherwise obscure an understanding of the invention. In any particular instance it may be advantageous to associate more than one value with each key (e.g., PositionInterpolators as defined in the VRML Specification). One skilled in the art will recognize that vector quantities or other complex data types could easily be substituted in the place of scalar

values, performing corresponding operations on the complex data types by methods well known in the art (e.g., vector addition). In an implementation written in the C++ programming language, a complex data type may be advantageously implemented as an object class which includes operator methods in the class definition for any necessary arithmetic operations.

The VRML player, which has parsed the VRML code and built the interpolator node **6420** and key frame table **6430**, also provides a mechanism for performing interpolations for the node **6420**. The interpolator mechanism accepts a fraction input signal (FI_{sig}) **6402** associated with the node **6420**, and produce and interpolated output signal **6404** using the information in the key frame table **6430**.

The conventional interpolator mechanism may be viewed as performing a 3-step process to generate its output signal. First, the key frame table keys (KFKeys) **6437** are scanned to find the first set of two adjacent key frames between whose KFKeys the value of the FI_{sig} **6402** lies. For example, using the sample data structures appearing in **Figure 64**, if the value of the FI_{sig} **6402** is 0.375, the interpolator mechanism selects the key frame at position index 2 **6432** as the low key frame of the set, and the key frame at position index 3 **6433** as the high key frame of the set, because the FI_{sig} **6402**, 0.375, lies between KFKey value 0.25 of Key Frame 2 **6442** and KFKey value 0.50 of Key Frame 3 **6443**.

Second, the conventional interpolator mechanism determines the value for the interpolated output signal **6404** by retrieving values from the low and high key frames selected in the first step and approximating the following calculation:

$$KFValue_{low} + ((KFVALUE_{high} - KValue_{low}) \frac{FI_{sig} - KFKey}{KFKey_{high} - KFKey_{low}}$$

Thirdly, the interpolator mechanism takes the result of the calculation performed in the second step and presents it as the Interpolated Output signal **6404**.

In one embodiment employing the present invention, an interval frame table (IFT) **6460** is constructed in addition to the conventional key frame table **6430**. The IFT comprises a set of interval frames **6461-6466**. The number of interval frames is one more than the number of key frames in the key frames table **6430**. The first and last interval frames **6461, 6466** are known as sentinel frames and are used to provide some consistency of processing and results for fraction input signals which fall outside of the range of key values provided by the interpolator definition **6412**.

Each interval frame comprises a key value (IFKey) **6467**, a value (IFValue) **6468**, and a factor (IFFactor) **6469**. The contents of each interval frame can be accessed by reference to its ordinal position **6470** in the set of interval frames **6461-6466**. Interval frames store information regarding the logical intervals between key frames. For example, the Interval Frame at position 3 **6463** stores information pertaining to the logical interval between the Key Frames at positions 2 and 3 **6432, 6433**.

The IFKey **6467** holds the key value of the corresponding KFKey that bounds the lower end of the interval. The IFValue **6468** holds the corresponding value of the KFValue that bounds the lower end of the interval. For example, the Interval Frame at position 3 **6463** in the Interval Frame Table **6460** stores the KFKey **6442** and KFValue **6452** from the Key Frame at position 2 **6432** in the key frame table **6430**.

An Interval Factor is calculated and stored in the IFFactor **6469** for the Interval Frame as approximated by the following formula:

$$\frac{KFValue_{high} - KFValue_{low}}{KFKey_{high} - KFKey_{low}}$$

Note that for a given Interpolator, the IFFactors are logically of the same data type as the IFValues, and not necessarily the IFKey. Keys match the data type of the Fraction Input signal value. The VRML Specification suggests that keys and fraction input signals should be scalar, floating point numbers. If the values in the values field **6416** of the interpolator definition **6412** are of a complex data type, e.g., a vector, the Factors are of similar complexity as is the resultant Interpolator Output signal.

Figure 65 depicts a flowchart of the interval table building process. Step **6510** sets a variable (N) to the number of key frames in the key frame table for the present interpolator node. Step **6512** then sets the interval frame table position index (iIF) to point to the first interval frame.

In step **6514**, the interval frame table constructor stores a very large negative number, e.g., -1,000,000, in the IFKey for the first (lower sentinel) frame. The number is such as is expected to always be lower than any value received at the interpolator node's fraction input. An implementation may set the lower sentinel IFKey to the largest negative value that can be stored for the relevant data type.

Step **6516** increments the iIF to point to the next, i.e., second, Interval Frame. Step **6518** sets the position index for the key frame table (iKF) to point to the first key frame.

Step **6520** determines whether the iIF is pointing to the last interval frame. If not, a non-sentinel frame is being filled and control passes to step **6522**. Step **6522** copies the iKF-indexed KFKey value into the iIF-indexed IFKey. Similarly, the corresponding KFValue is copied into the IFValue in step **6524**.

Step **6526** then approximates the interval factor according to the formula described earlier and the result is stored in the iIF-indexed IFFactor. In step **6528**, the interval table constructor increments the position indexes, iKF and iIF, to point to the next frames in their respective tables. Processing returns to step **6520** until the upper sentinel frame is reached.

If step **6520** determines that the last (upper sentinel) frame has been reached, the interval frame table constructor stores a very large positive number, e.g., +1,000,000, in its IFKey in step **6530**. The number is such as is expected to always be higher than any value received at the interpolator node's fraction input. An implementation may set the upper sentinel IFKey to the largest positive value that can be stored for the relevant data type.

Step **6532** copies the KFValue from the highest key frame into the upper sentinel frame's IFValue. Lastly, step **6534** sets the IFFactor for the upper sentinel frame to zero. Construction of the interval frame table is complete.

Representative C++ source code for the interval frame table building process depicted in **Figure 65** appears in Table 1.

TABLE 1. Interval Frame Table Build

```
template <class TYPE>
IntervalFrameTable<TYPE>::IntervalFrameTable(Node* node,
        const MxFloat& keys,
        const ValueVector<TYPE>& values,
        const TYPE& zeroValue) : InterpolatorNode(node)

{
const   float   kSentinelKey = 1..0e6;
int     nCount  keys.size();
m_pStart = reinterpret_cast<Frame*>(this+1);
m_pFinish = m_pStart + nCount + 1;

Frame* frameP = m_pStart;

frameP->key = -kSentinelKey;
new(&frameP->value) TYPE(values[0]).
new(&frameP->factor) TYPE(zeroValue);
++ frameP;

int i = 0;
for (; frameP < m_pFinish - 1; ++ frameP, ++ i)
{
        frameP->key = keys[i];
        new(&frameP->value) TYPE(values[i]);
        new(&frameP->factor) TYPE((values[i + 1] - frameP->value) * (1.0 /
                (keys[i + 1] - frameP->key)));
}

frameP->key = kSentinelKey;
new(&frameP->value) TYPE(values[nCount - 1]);
new(&frameP->factor) TYPE(zeroValue);
}
```

Figure 66 depicts a flowchart of interpolator node operation in one implementation employing the present invention. During the routing phase of the VRML player's operation cycle, an interpolator node may be incited to generate a new output signal **6601**. Step **6610** determines whether an interval frame table (IFT) **6460** for the interpolator exists and is valid. The IFT will exist unless this is the very first time in the life of the current scene that the interpolator has been activated by an eventIn during routing. (Activation by an eventIn method is set forth in the VRML Specification.) An existing IFT will be invalid if any of its underlying keys or values have changed in the respective interpolator node in the scene graph since the last activation of the node. If the IFT does not exist or is invalid, step **6620** builds an IFT according to the procedure described above in reference to **Figure 65**.

Once it is established that a valid IFT exists, interpolation processing proceeds to step **6612**, which identifies the interval frame for the logical interval in which the input fraction signal value falls. This is done by scanning through the interval frames looking for the frame with the largest IFKey which is less than or equal to the value from the fraction input signal. The interval frame table position index (iIF) saves the position of the identified frame in working storage **6652**.

Step **6614** retrieves the IFKey value in the iIF-indexed Interval Frame as a subtrahend, retrieves the value of the fraction input signal as a minuend, approximates a subtraction, and retains the resulting difference in working storage **6654**.

Step **6616** retrieves the IFFactor value in the iIF-indexed interval frame as a first factor, retrieves the difference retained from step **6614** as a second factor **6654**, approximates a multiplication and retains the resulting product in working storage **6656**.

Step **6618** retrieves the IFValue in the iIF-indexed interval frame as a first addend, retrieves the product retained from step **6616** as a second addend **6656**, approximates an addition and stores the resulting sum as the new value for the interpolated output signal **6660**. Step **6630** may then present the output signal by invoking the value_changed eventOut method associated with the interpolator node. (Output signaling by an event Out Method is set forth in the VRML Specification.)

An example of interpolator node operation as depicted in the flowchart of **Figure 66**, and using the interpolator node structures illustrated in **Figure 64**, follows. A fraction input signal **6402** representing a value of 0.375 appears to interpolator node **6420** via its set_fraction eventIn method. The interval frame table **6460** already exists so the interpolator mechanism identifies the interval frame for the interval that includes 0.375. The interval frame **6461** at position index 1 has a key **6471** of -1,000,000 that is less than or equal to 0.375. The intervalframe **6462** at position index 2 has a key **6472** of 0.00 that is less than or equal to 0.375. The intervalframe **6463** at position index 3 has a key **6473** of 0.25 that is less than or equal to 0.375. The intervalframe **6464** at position index 4 has a key **6474** of 0.50 that is not less than or equal to 0.375. Scanning stops and the interpolator mechanixm sets the iIF index pointer to 3 representing the interval frame **6463** with the largest key value that is less than or equal to the fraction input signal.

The interpolator retrieves the key value **6473** of 0.25 from the interval frame **6463** at the iIF position. The 0.25 key value is subtracted from the 0.375 fraction input signal value and the resulting difference of 0.125 is retained in working storage **6654**.

The interpolator retrieves the factor value **6493** of 8.00 from the interval frame **6463** at the iIF position. The 8.00 value is multiplied by the 0.125 difference value **6654** that resulted from the previous step. Working storage **6656** retains the resulting product of 1.00.

The interpolator mechanism then retrieves the value **6483** of 2.0 from the interval frame **6463** at the iIF position. The 2.0 value is added to the 1.00 product **6656** that resulted from the previous step. Working storage **6660** retains the resulting sum of 3.00, the interpolated value corresponding to a fraction input signal of 0.375. The interpolator mechanism can then invoke the interpolator node's value_changed eventOut method to signal the new output value. The interpolation process initiated by the fraction input signal is complete.

The process described in relation to **Figure 66** operates to generate an Interpolated Output signal value approximated by the following formula:

$$IFValue + \left(IFFactor\ x\ \left(FI_{sig} - IFKey \right) \right)$$

It is noted that the formula includes addition, multiplication, subtraction, but no division operations. Addition, multiplication, and subtraction represent very fast operations on modern-day computers. Each division operation is very slow, taking on the order of ten times as long as any single one of the other operations. In a graphics

system with many variable aspects whose character can be determined using interpolation, eliminating numerical division operations from Interpolator Output signal generation process can significantly reduce the time per interpolation. This can translate, as compared to conventional interpolation methods, to 1) the ability to process complex scenes requiring more interpolations per time period using the same computing speed, or 2) the ability to process scenes of the same complexity in less time using the same computing speed. This represents a further advantage of the invention.

One skilled in the art recognizes that various data structure constructs well known in the art can be employed in a given embodiment to implement the logical equivalent of the interval frame table which embodies the correspondence and relationship between keys, values, and factors. Some examples are logically composing the interval frame table using an array of storage elements, an array of pointers to storage elements, or a linked list of interval frame storage structures.

Further, the utilization of the interpolation mechanism is not limited to applications affecting the video output of a VRML player. For example, the volume level of an audio program may be interpolated. Users may extend outside of VRML player applications altogether. These and other variations may be employed without departing from the spirit of the invention.

IX. JAVA INVERSION

Figure 67 depicts one embodiment's architectural components that interact to support certain VRML script node processing. Major components include an operating

system **6710**, a web browser **6730**, and a VRML player **6740**. The VRML player **6740** further includes a script manager subcomponent **6750** and a scene graph structure **6742** constructed from the VRML definition of the scene being viewed. The scene graph **6742** may contain script nodes **6744-6748** that may contain a reference to a Java applet **6754** to be executed during the viewing of the scene. The script manager contains a script language subcomponent **6752** that specifically manages the processing of Java scripts.

The web browser **6730** subcomponents may include a Java services interface **6734** and a Java Virtual Machine **6732**. Representative web browsers include Netscape's Navigator and Microsoft's Internet Explorer.

The operating system **6710** subcomponents include OS Java Support **6714** and a Java Virtual Machine **6712**. Representative operating systems include Microsoft's Windows 95 and Windows NT Workstation.

During the course of operation, the VRML player repeatedly invokes the script manager **6750** and its Java subcomponent **6752** to oversee the execution of Java applets associated with the script nodes in the scene graph. The Java applets are actually executed using a Java virtual machine **6712**. In conventional operation, the script manager **6750** requests execution of the Java applet **6754** by making a request **6760** to the web browser's Java Services Interface **6754**. The Java services interface **6734** may run the program module **6754** on the web browser's own Java virtual machine **6732**, or it may run the program module **6754** on the operating system's Java virtual machine **6712** using the OS Java support mechanisms **6714**. In any event, this indirect access to

Java services by way of the web browser's Java services interface **6734** imposes processing overhead that may be desirable to eliminate.

Directly requesting Java services from the OS Java support **6714** is one possible way to eliminate significant processing overhead incurred for each Java applet execution. In some cases, however, the OS Java Support **6714** does not provide the flexibility needed to achieve the objective of overhead reduction.

The Microsoft Windows '95 operating system is one example where the conventions for using the OS Java Support subcomponent restrict the opportunity to eliminate processing overhead. Some of the Java services most useful to a VRML player, e.g., fast execute of a related Java applet **6722**, have restricted access. Services **6722-6726** in the restricted Java service classes area **6720** are enabled by a signal **6718** associated with a particular Java application program module that was started using OS Java Support's execution service **6716**. The signal **6718** is maintained for the duration of the Java applet's execution. In many implementations the VRML player **6740** is not, itself, a Java program, and accordingly has no access to the restricted Java service classes **6720**. As such, it must utilize the execution service **6716** to start each script node Java applet, rather than the Fast Execute Java service class **6722**. The VRML player incurs the higher overhead of the execution service **6716** and possibly performs no better than if the web browser's Java services interface **6734** had been used instead.

An embodiment employing the present invention may establish an inverted relationship between the OS Java Support **6714** and the VRML player application program **6740**. In the inverted relationship, the application program dominates

execution and subordinates OS Java Support as an incidental service provider - a role reversal from the conventional implementation.

Figure 68 depicts a control flow diagram for one embodiment of an inverted Java environment. In the illustrated embodiment, the OS Java Support subcomponent **6714** of the operating system comprises a Java applet execution service **6716**, a restricted set **6720** of Java service classes **6722-6726**, and a signal **6718** which enables a program's access to the restricted Java service classes **6720**. The VRML player application **6740** comprises a start-up program module **6810**, a main program flow module **6816**, a bootstrap applet **6812**, and a bootstrap native method module **6814**. The main program flow module **6810** further comprises the script manager program module with its attendant Java script language subcomponent **6818**. In the presently described embodiment, Microsoft Windows '95 is the relevant operating system **6710**, the OS Java Support **6714** inheres in the Common Object Model implemented in Windows '95, and the VRML player application **6740** is implemented using the C++ programming language. The player application **6740** also includes a bootstrap applet **6812** written in the Java programming language.

When the VRML player application **6740** is initiated, a start-up module **6810** prepares the VRML player application environment. As part of its duties, the start-up routine **6810** passes control along a first path **6831** to the operating system's Java execution service **6716**, requesting execution of the bootstrap applet **6812**. The execution service **6716** passes control along a second path **6832** to the bootstrap applet **6812**. The execution service **6716** generates a signal **6718** enabling access to the set of restricted Java service classes **6720** by the bootstrap applet **6812** and its successors to

control. The bootstrap applet then passes control along a third path **6833** to the bootstrap native method module **6814**.

The bootstrap applet **6812** and bootstrap native method module **6814** sustain their existence and operate together in order to establish and perpetuate access to the set of restricted Java service classes **6720** for the VRML player application **6740**. The bootstrap applet **6812** is a Java-language program module limited in capability by the security and integrity features built into the Java virtual machine in accordance with the Java Specification. The bootstrap native method module **6814** is a C++-language program module not subject to Java limitations. The bootstrap native method module **6814** thus transitions the player application **6740** into the mode of operation enjoying the broadest set of capabilities allowed to an application program by the operating system.

The bootstrap native method module **6814** extends control along a fourth path **6834** to the main program flow module **6816**. The main program flow modules **6816** completes the initialization of the VRML player **6740** and conducts the ongoing operation of the player.

The VRML player's main processing cycle **6817** repeatedly invokes the script manager to process the Script Nodes in the scene graph. When a Java object associated with a script node needs a method executed, the script manager with its attendant Java script language subcomponent **6818** directly utilizes the Fast Execute service class **6722** in the set of restricted Java service classes **6720**. Control passes from the Java Script management **6818**, along a fifth path **6835**, to the Fast Execute service class **6722**, which executes the requested Java method and returns control along a sixth path **6836**. In

normal operation the first, second, third and fourth control paths **6831-6834** are traversed only once in the life of the VRML player **6740**, while the fifth and sixth paths **6835**, **6836** are traversed repeatedly. The VRML player application **6740** thus dominates the execution and subordinates the OS Java Support **6714** as a service provider via use of the fifth and sixth paths **6835**, **6836**. The resulting repeated use of the Fast Execute service class **6722**, as opposed to the execution service **6716**, provides a significant reduction in script node processing overhead. This represents a further advantage of the invention.

Figure 69 depicts a flowchart detailing the establishment of an inverted Java environment in the presently described embodiment. When execution of the VRML player begins **6901**, the Startup method **6870** of a Viewer class object creates an instance of a player-side object to manage the Java inversion. In the presently described embodiment the object belongs to the MSJava class. Step **6910** depicts the creation of the object. The Boot method **6972** of the MSJava object initializes the interface to the operating system in step **6920**. In the case of Windows '95, the interface is the Common Object Model. Initializing the interface makes a preliminary Java application programming interface available to the application program. Step **6922** then establishes a link to Java service class routines.

The Boot method **6972** in step **6924** stores information necessary for the resumption of player initialization in a global data area **6820**. The stored information includes a pointer to the player-side object created in step **6970**, a pointer to the Viewer object representing the executing instance of the player application, and an entry point to

the program code that will continue the player's initialization process. The presently described embodiment refers to the entry point as GoViewer.

In step **6926**, the Boot method **6972** then builds a request to execute the bootstrap applet **6926**, and invokes the execution service to fulfill the request in step **6928**.

Table 2 depicts an example of C++ programming language source code that may be included in the MSJava object's Boot method.

TABLE 2. MSJava Boot Method

Sample Source Code

```
// used by VRML player start-up
MSJava::Boot(t_JavaContinueFuncPtr goViewer, void *cookie)
{
// establish os/com interface to java
if (S_OK != CoInitialize(NULL)) {
        return MSJava::cantFindJavaErr;
}
m_CoInit = 1;
if (S_OK != CoCreateInstance(CLSID_JavaExecute, NULL,
        CLSCTX_ALL, IID_IjavaExecute, (LPVOID *)&m_IJavaExecute)) {
        M_IJavaExecute = 0;
        Return MSJava::cantFindJavaErr;
}
// establish preliminary link to java service classes
if (!InitBootstrapSystem(jvip))
        return MSJava::cantFindJavaErr;
// store information needed to resume player operation
SetJavaContinueFunction(goViewer, cookie)
// build java request to run the java inversion applet
JAVAEXECUTEINFO jeo;
LPCOLESTR      argv[2];
```

```
LPERRORINFO     pErr = 0;
HRESULT         hErr;
// software version number
argv[0] = m_version;
// application library name
argv[1] = m_bootLibName;
jeo.cbSize = sizeof(jeo);
jeo.dwFlags = 0;
// java inversion applet name: jentry
jeo.pszClassName = m_bootClassName
jeo.rgszArgs = &argv[0]
jeo.cArgs - sizeof(argv)/sizeof(char *);
jeo.pszClassPath = m_classPath;
// execute the java request
hErr = m_IJavaExecute->Executre(&jeo, &pErr);
// error handling
// should normally reach this point at player termination
. . .
```

The Startup **6970** and Boot **6972** methods just described represent in pertinent part the start-up program module **6810** of **Figure 68**.

At the beginning of step **6930** control passes from the application program to the operating system. The execution service **6716** of the OS Java Support **6714** depicted in **Figure 68** performs the next series of operations. In step **6930**, the execution service establishes the operating environment for the bootstrap applet. As part of this process, but depicted separately as step **6932**, a signal is generated to permit the bootstrap applet access to the restricted Java service classes. Step **6934** then initiates the execution of the Java applet.

It is noted that much of the work already described and performed by the Boot method **6972** and the OS Java Support **6714** would be repeated for each execution of a

script node Java applet during operation of the VRML player if the presently described Java inversion mechanism were not implemented. The elimination of much of this activity is the overhead reduction noted earlier as an advantage of the present invention.

At the beginning of step **6940**, control passes from the operating system to the application program code. Notably, security and integrity protection features of the Java virtual machine limit the control passed. In step **6940**, the main method of a jentry class object **6974** ascertains whether it is compatible with the VRML player that requested its execution. The bootstrap applet **6812** depicted in **Figure 68** includes the jentry class. In some implementations the bootstrap applet may reside in isolation from other VRML player program code, and the compatibility check offers some protection from player failure in the event that the release level of the player program code is out of synchronization with the release level of the bootstrap applet. If a mismatch exists, the Java inversion is aborted by a return of control from the bootstrap applet to OS Java support **6990**. Error-handling routines may be advantageously placed in the player start-up routine in an embodiment, to detect the inversion failure. Player initialization and operation may then be resumed, but in a mode that relies on conventional methods to obtain Java services.

After any verification of compatibility, the boot method of a jbootstrap class object **6976** invokes a Java service class that directs the Java virtual machine to the library in which the VRML player resides when subsequently searching for native method modules. The library pointer is set in step **6942**. The bootstrap applet **6812** depicted in **Figure 68** also includes the jbootstrap class. The boot method **6976** performs this operation using the System.loadLibrary method set forth in the Java

Language Specification. One skilled in the art recognizes that the System.load method or another method may be used to produce a comparable result.

At this point the bootstrap applet has established itself and the related Java environment. Control passes at the beginning of step **6950** to the doInvert method of a bootstrap class object **6978**. The bootstrap native method **6814** depicted in **Figure 68** includes the bootstrap class. Step **6950** completes the inversion by resuming initialization of the VRML player. The doInvert method acquires the necessary information from global storage where it was stored earlier by the Boot method of the MSJava class object.

In step **6960** control passes to the GoPlayer entry point **6980** where the player begins its main operation. The main program flow module **6816** depicted in **Figure 68** includes the GoPlayer entry point.

The inversion is complete and the player may directly invoke all functionality of the operating system normally available to an application program, in addition to a broad range of Java service classes that includes the set of restricted Java service classes.

It is noted that besides having direct access to Java services via the OS Java support, the player in an inverted-Java configuration is stripped of its dependency on a web browser for Java services. An embodiment of a player so configured, advantageously employing the present invention, may be more easily packaged as a standalone VRML player or as a plug-in module for other applications needing VRML viewing capability. An example may be a video game application that represents its

game space in VRML. This packaging flexibility represents a further advantage of the present invention.

One skilled in the art recognizes that a variety of alternatives exist for many of the details set forth for the illustrative embodiment just described, and that such alternatives may be employed without departing from the scope or spirit of the invention. For example, the target operating system need not be Windows '95. Further, some embodiments may be developed using programming languages other than C++. Further still, pointers may be implemented using any number of referencing techniques well known in the art, e.g., reference by memory address, disk location, or name.

Names used for classes, objects, methods and data items in the illustrated embodiment are generally at the discretion of the programmer except where compliance with specifications is sought, e.g., the System.loadLibrary method set forth in the Java Language specification. Similarly, the processes and data items comprising the application program may be organized into program subcomponents (e.g. object oriented classes) in almost countless ways while still achieving the same result. Further, particular data items may be stored in various types of storage media or classes without losing their functionality. For example, the player resumption data that is stored in a global data area in the above-described embodiment, may alternatively be stored in a file or a resource.

Importantly, the Java Inversion described above may be advantageously employed by applications requiring speedy access to Java service classes other than a

VRML player. Further, a service provider other than Java may be similarly inverted. For example, a VRML player application may support a scripting language other than Java but having a similar implementation. One skilled in the art recognizes these and other variations that may be employed without departing from the spirit of the invention.

While the invention has been described with reference to numerous specific details, one of ordinary skill in the art would recognize that the invention can be embodied in other specific forms without departing from the spirit of the invention. Thus, one of ordinary skill in the art would understand that the invention is not to be limited by the foregoing illustrative details, but rather is to be defined by the appended claims.

CLAIMS:

We claim:

1. In a computer system with a display device having an array of pixels, a method for graphics processing comprising the steps of:

(a) receiving a first set of graphics information for a first geometric entity residing on a first set of rows of pixels in said pixel array, said first set of rows including a first row;

(b) receiving a second set of graphics information for a second geometric entity residing on a second set of rows of pixels in said pixel array, said second set of rows including the first row,

(c) composing graphics commands for the first row of pixels, said commands including graphics information relating to portions of the first and second geometric entities on the first row of pixels.

2. In a computer system with a display device having an array of pixels, a method of processing computer graphics comprising the steps of:

(a) receiving graphics information for a plurality of primitives for being displayed on a number of rows of pixels, wherein at least two primitives are to appear on a common row of pixels;

(b) generating graphics commands for said common row of pixels, said graphic commands including graphics information relating to portions of the first and second geometric entities on the first row of pixels.

3. In a computer system with a display device having an array of pixels, a method of processing computer graphics comprising the steps of:

(a) receiving data structures representing geometric entities for display on the array of pixels;

(b) extracting graphics information for each row of pixels in the array from the data structures, the graphics information for each particular row representing the graphics information in the data structures for the particular row;

(c) storing the graphics information for each particular row.

4. The method of claim 3 further comprising the step of generating pixel data for each unique row of pixels by decoding the graphics information for the unique row.

5. A method for graphics processing comprising the steps of:

(a) receiving a first data structure for a first geometric entity;

(b) receiving a second data structure for a second geometric entity, one of said geometric entities occluding the other geometric entity;

(c) eliminating information from one of the data structure relating to the occluded portion of the occluded geometric entity.

6. A method for graphics processing comprising the steps of:

(a) receiving a first data structure for a first geometric entity;

(b) receiving a second data structure for a second geometric entity, one of said geometric entities occluding the other geometric entity;

(c) clipping said geometric entities against each other to obtain a third data structure representing the occluded geometric entity, wherein the third data structure does not contain information relating to the occluded portion of the occluded geometric entity.

7. For a computer system displaying a plurality of geometric entities in a displayed scene, a method for graphic processing comprising the steps of:

(a) receiving data structures representing at least two overlapping geometric entities, wherein one of the geometric entities occludes the other in the displayed scene;

(b) clipping the two geometric entities against each other to obtain data structures representing two non-overlapping geometric entities.

8. For a binary space partitioning (BSP) tree data structure having a plurality of node data structures, each node data structure corresponding to a volume of space in a displayed scene, a method of generating a face of a bounding box for a volume of space corresponding to a child node in the BSP tree, said child node having at parent node and a grandparent node, said parent and grandparent nodes having splitting planes, said

grandparent node's splitting plane having two side, said parent node on one side of the grandparent node's splitting plane, the method comprising the steps of:

(a) intersecting the splitting plane of the parent with a bounding box to generate a polygon;

(b) determining whether the splitting plane of the grandparent node slices the polygon into two resulting polygons;

(c) if the splitting plane of the grandparent node slices the polygon into two resulting polygons, identifying the face of the child node as the resulting polygon that is on the same side of the grandparent's splitting plane as the parent node.

9. A binary space partitioning (BSP) tree data structure having a plurality of node data structures, each node data structure storing a visibility state, wherein the visibility state of each node is dynamically updated.

10. Method for detecting collisions between virtual objects in a computer system simulating a three dimensional physical world wherein the space of the simulated world is subdivided into a plurality of partitions each referencing any intersecting objects and each partition classified, with respect to itself or its intersecting objects, by one or more states, the steps comprising:

(a) selecting a partition;

(b) characterizing the partition according to its present need for collision determination according to one or more state indicia and proceeding to step (d) if the partition is of a character not requiring collision determination;

(c) determining any intersection between the approximate space occupied or traversed by a first object and the approximate space occupied or traversed by a second object, which first and second objects each intersect the partition;

(d) storing data regarding the character of any intersection detected in step (c).

11. The method of claim 10 wherein the state indicia used to characterize the partition comprises a visibility state having a first visibility character if any portion of the partition is approximately within the range of a current viewing frustum, and the visibility state possessing the first visibility character indicates the partition is of the character requiring collision determination.

12. Method for generating an interpolated output signal value in a computer system having a set of key values, and a corresponding set of output reference values, the steps comprising:

(a) calculating an interpolation factor corresponding to an interval between a first key value and a second key value of greater magnitude, each key value having a corresponding output reference value, as the (SecondOutputReferenceValue – FirstOutputReferenceValue) ÷ (SecondKeyValue – FirstKeyValue);

(b) storing the interpolation factor;

(c) receiving an input signal having a value between the first key value and the second key value;

(d) calculating an output signal value using the previously calculated interpolation factor, as the FirstOutputReferenceValue + (StoredInterpolationFactor × (InputSignalValue – FirstKeyValue));

(e) presenting the output signal value;

(f) repeating steps (c) through (e) for each input signal to be interpolated

13. Method for sustaining an access control signal in a computer system, the steps comprising:

(a) invoking a first initialization component of a principal application program under a first execution manager;

(b) storing data regarding the location of the principal application program's main processing component by the first initialization component;

(c) sending a request from the first initialization component to a second execution manager thereby requesting execution of a second initialization component of the principal application program;

(d) executing the second initialization component thereby generating an access control signal of a character permitting the principal application program components access to one or more restricted service functions;

(e) invoking execution of a third initialization component of a principal application program;

(f) retrieving data stored in step (b), by the third initialization component;

(g) invoking execution of the main processing component using the data retrieved in step (f)

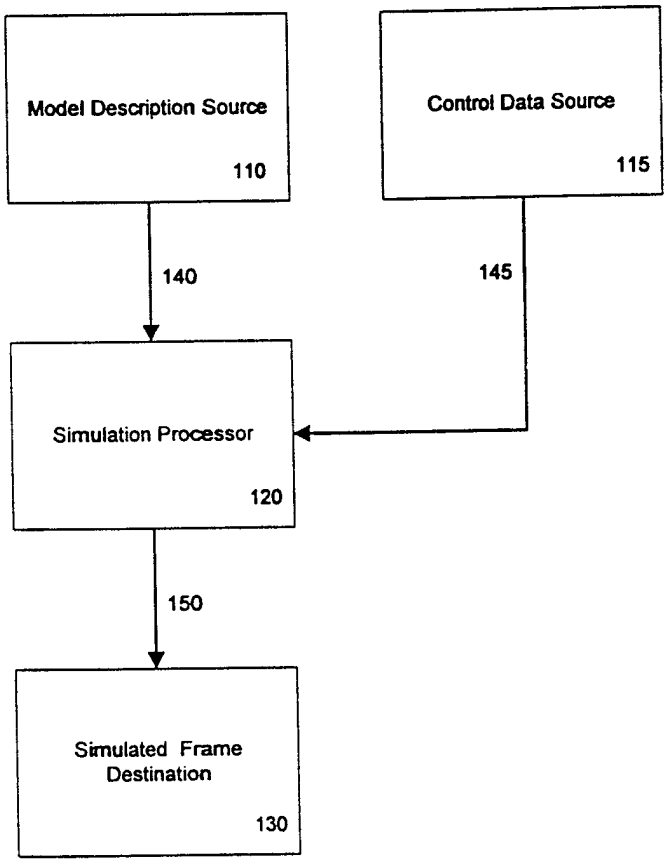
Model Description Source
110
Control Data Source
115
140
145
Simulation Processor
120
150
Simulated Frame Destination
130

Figure 1

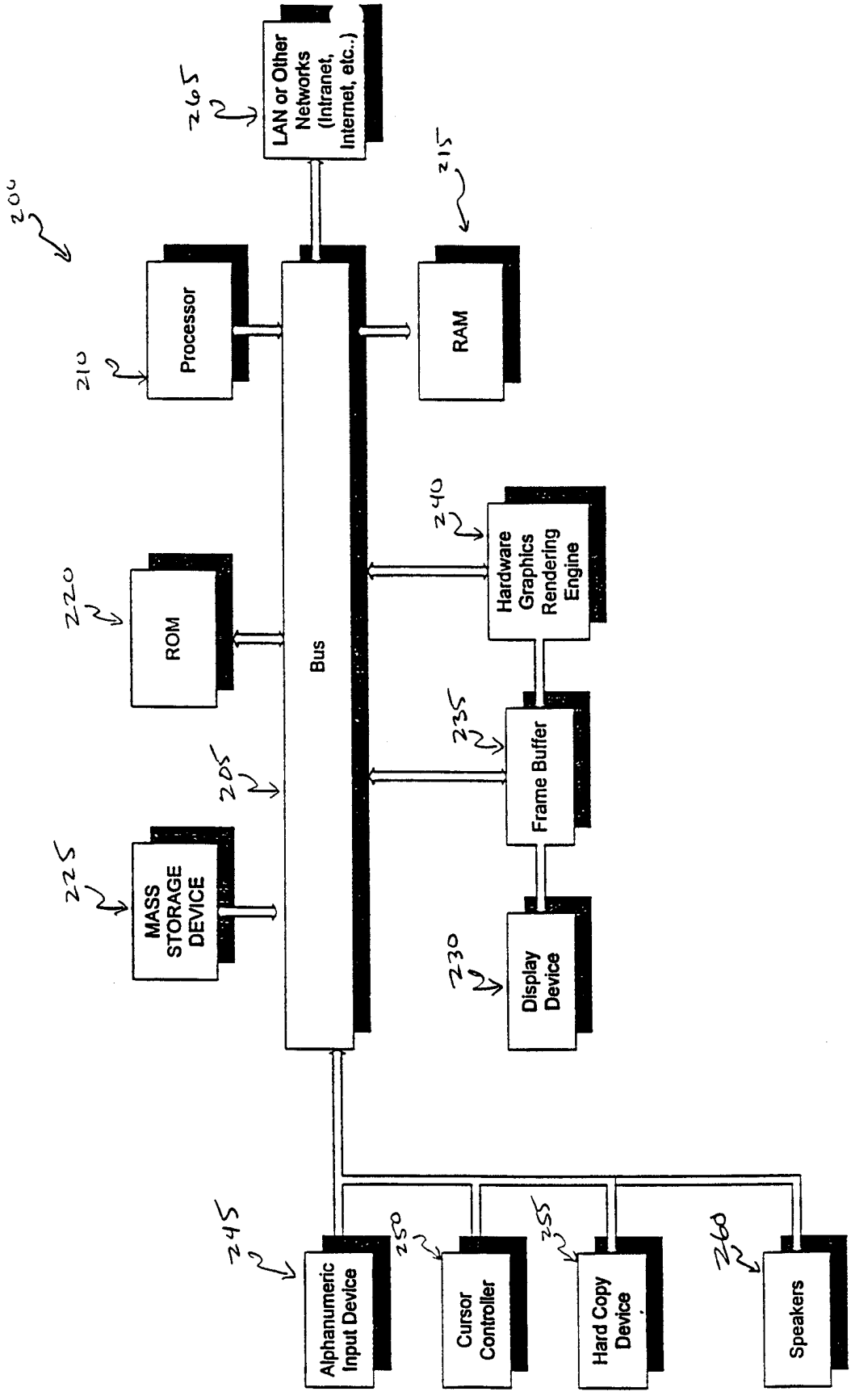
200
265
LAN or Other Networks (Intranet, Internet, etc.)
215
Processor
210
RAM
240
Hardware Graphics Rendering Engine
220
ROM
Bus
235
Frame Buffer
205
225
MASS STORAGE DEVICE
230
Display Device
245
Alphanumeric Input Device
250
Cursor Controller
255
Hard Copy Device
260
Speakers

Figure 2

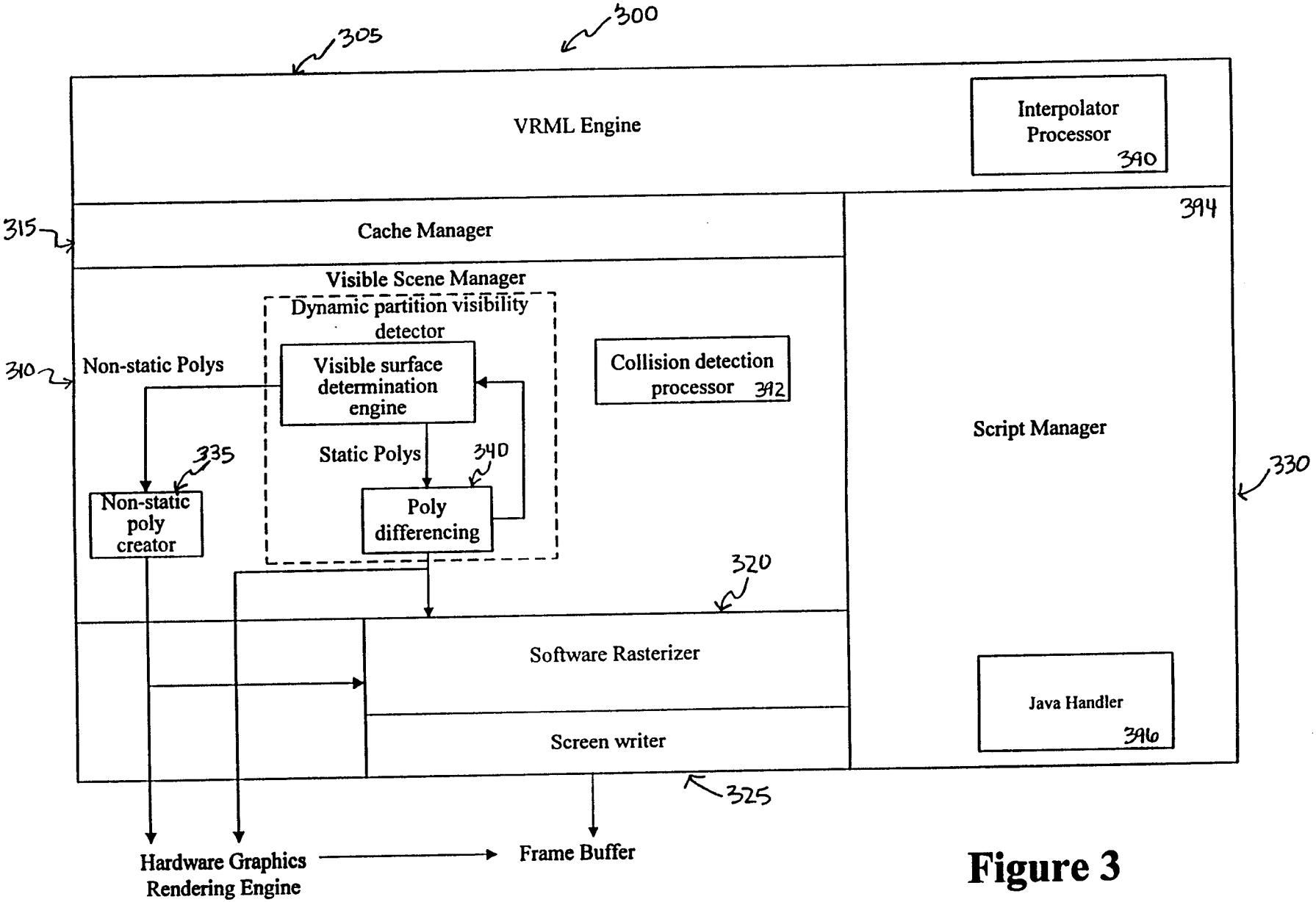
305
300
VRML Engine
Interpolator Processor
390
394
315
Cache Manager
Visible Scene Manager
Dynamic partition visibility detector
310
Non-static Polys
Visible surface determination engine
Collision detection processor 392
Static Polys
340
335
Non-static poly creator
Poly differencing
330
Script Manager
320
Software Rasterizer
Java Handler
396
Screen writer
325
Hardware Graphics Rendering Engine
Frame Buffer

Figure 3

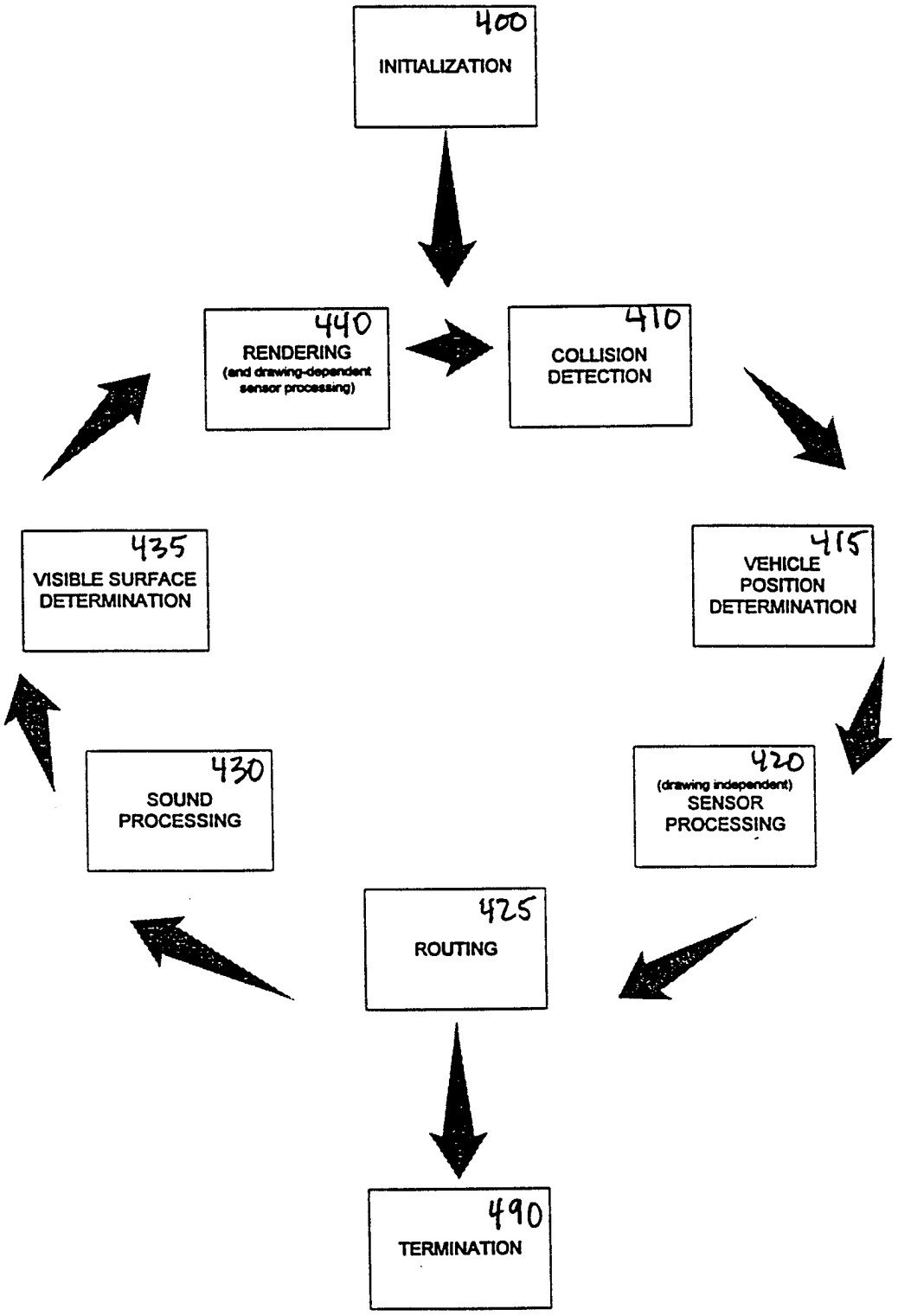
400
INITIALIZATION
410
COLLISION DETECTION
415
VEHICLE POSITION DETERMINATION
420
(drawing independent) SENSOR PROCESSING
425
ROUTING
430
SOUND PROCESSING
435
VISIBLE SURFACE DETERMINATION
440
RENDERING
(and drawing-dependent sensor processing)
490
TERMINATION

Figure 4

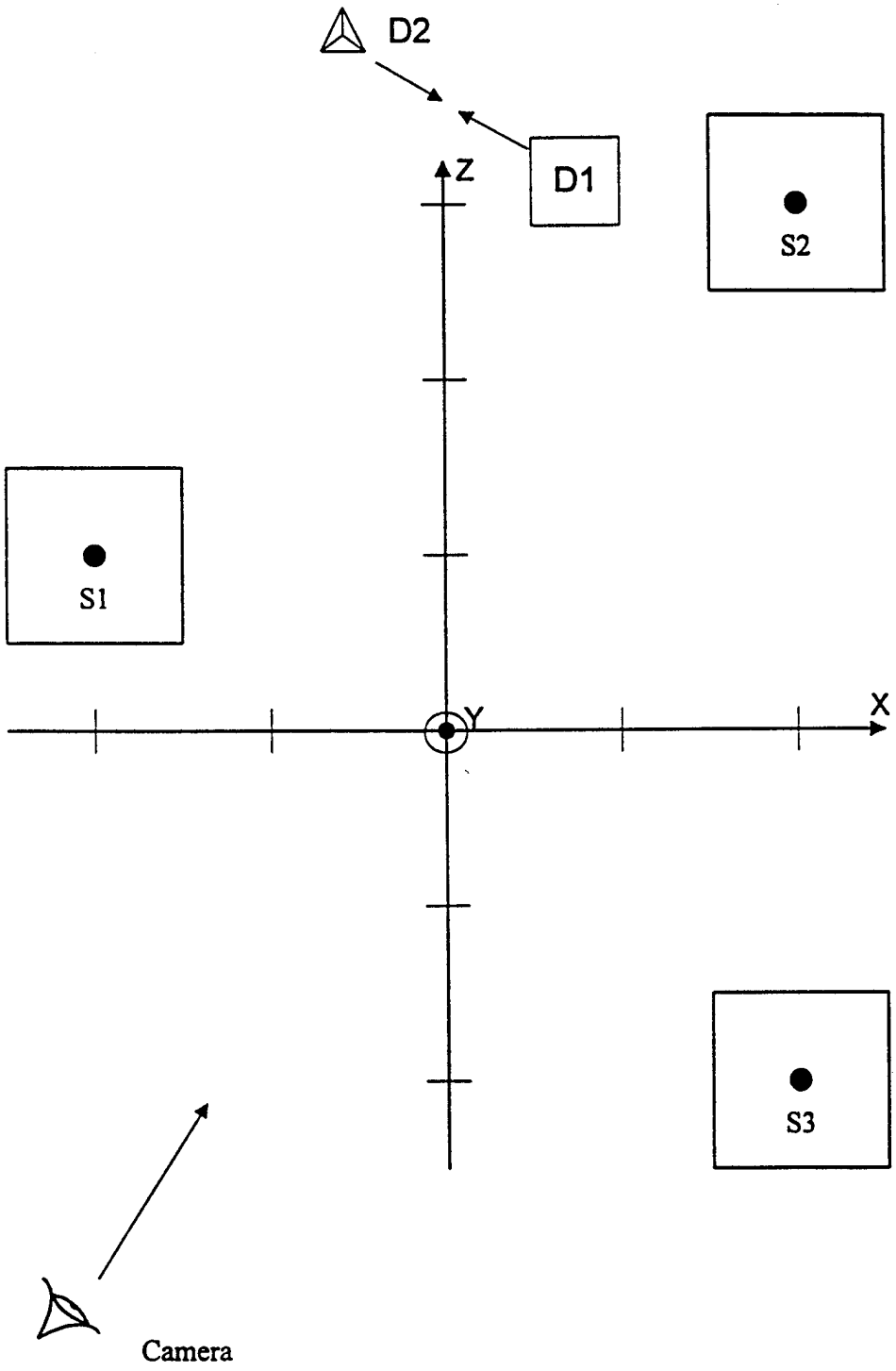

Figure 5

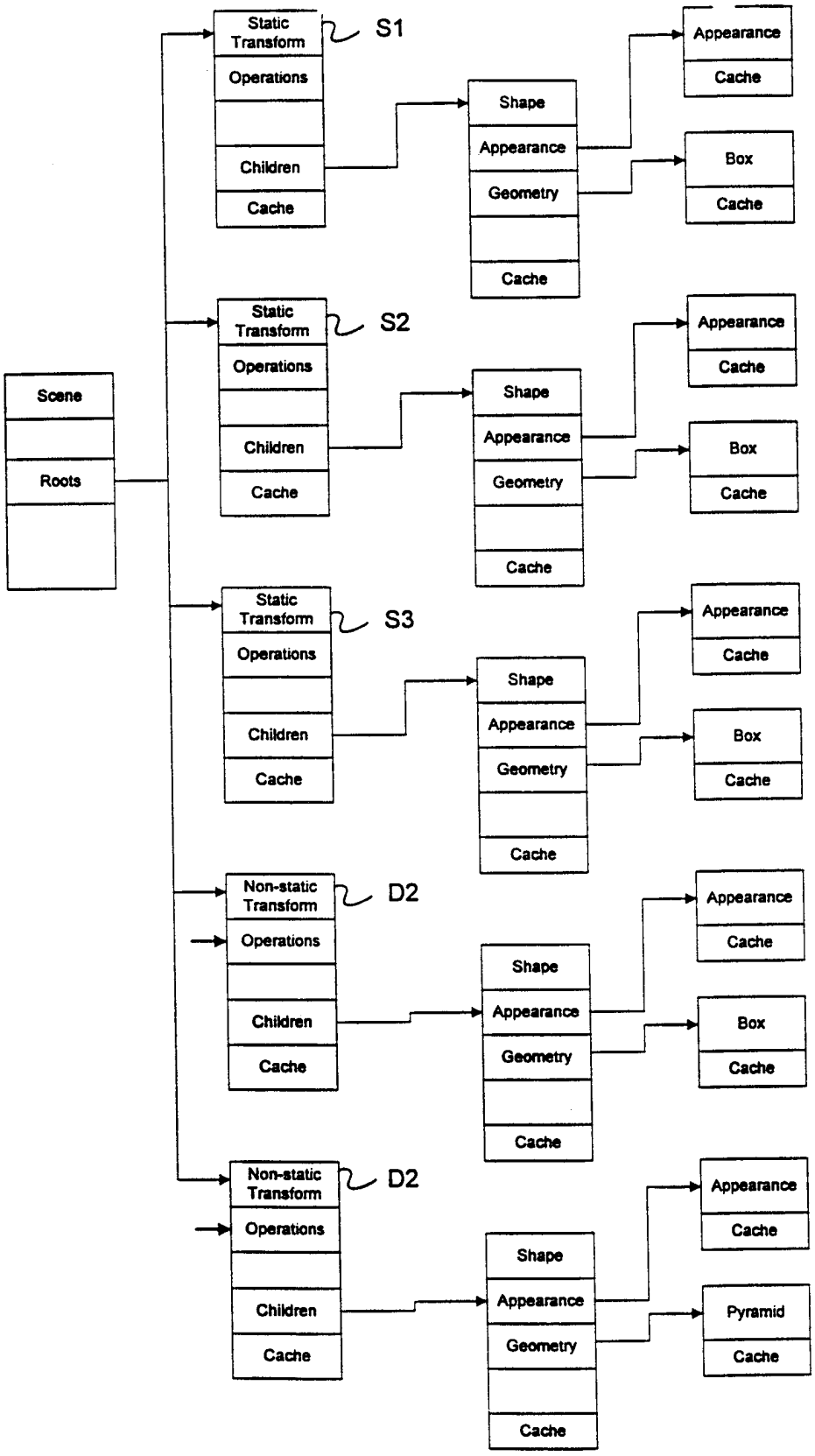
Scene
Roots
Static Transform
Operations
Children
Cache
S1
S2
S3
Non-static Transform
D2
D2
Shape
Appearance
Geometry
Cache
Box
Pyramid

Figure 6

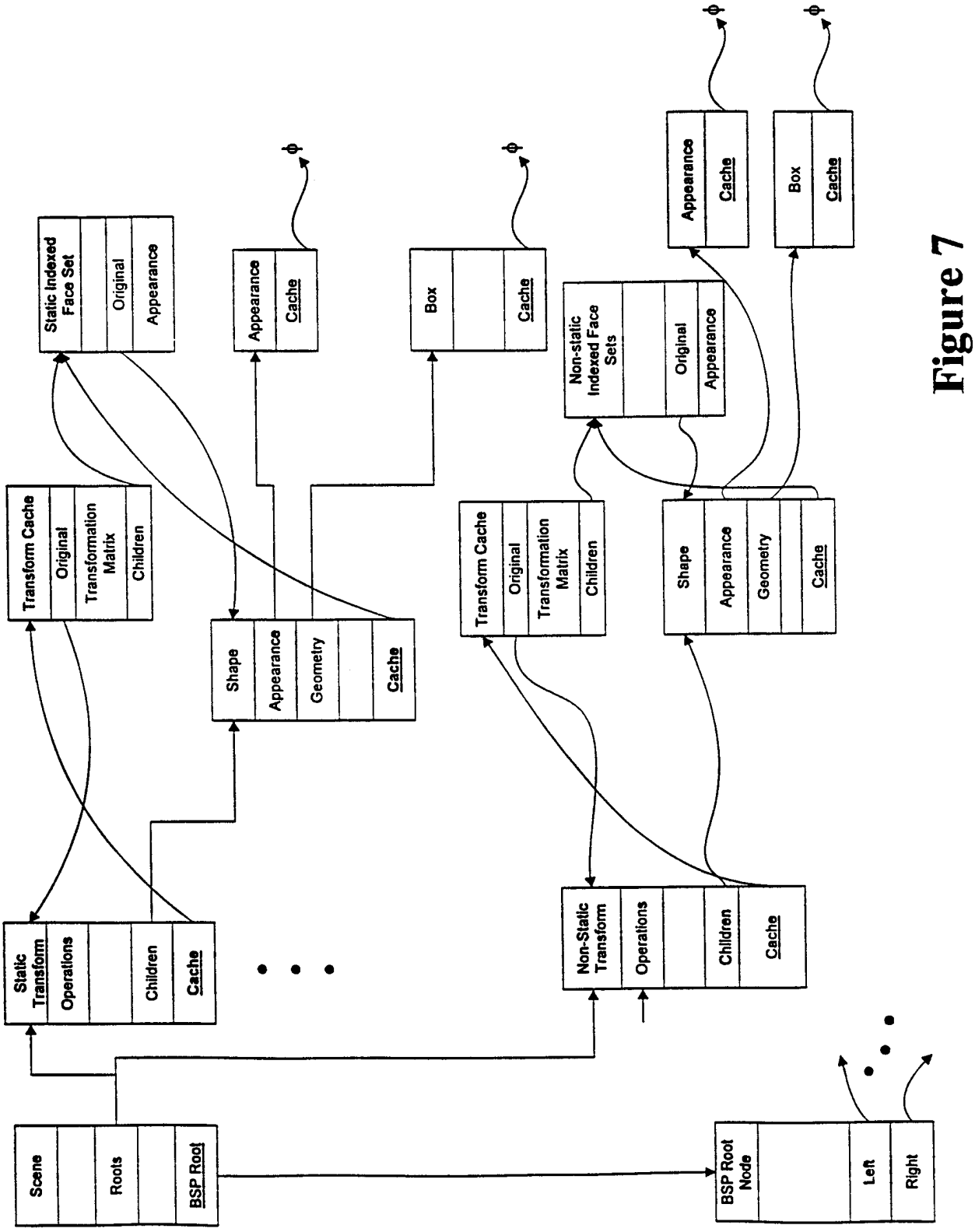
Scene
Roots
BSP Root
Static Transform
Operations
Children
Cache
Transform Cache
Original
Transformation Matrix
Children
Static Indexed Face Set
Original
Appearance
Shape
Appearance
Geometry
Cache
Appearance
Cache
Box
Cache
Non-Static Transform
Operations
Children
Cache
Transform Cache
Original
Transformation Matrix
Children
Non-static Indexed Face Sets
Original
Appearance
Shape
Appearance
Geometry
Cache
Appearance
Cache
Box
Cache
BSP Root Node
Left
Right
pa-165631

Figure 7

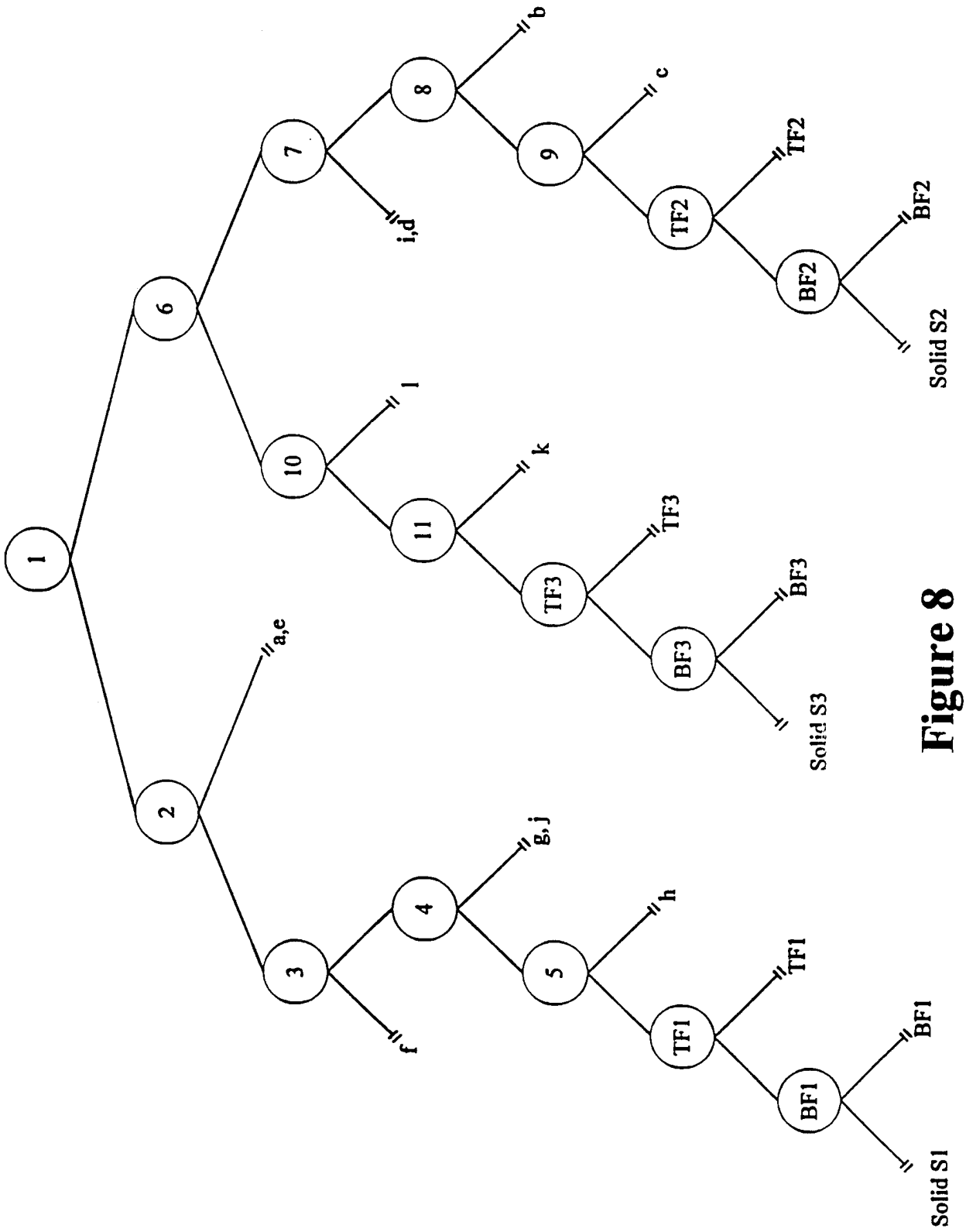
1
2
3
4
5
6
7
8
9
10
11
a,e
f
g, j
h
i,d
b
c
l
k
TF1
TF2
TF3
BF1
BF2
BF3
Solid S1
Solid S2
Solid S3

Figure 8

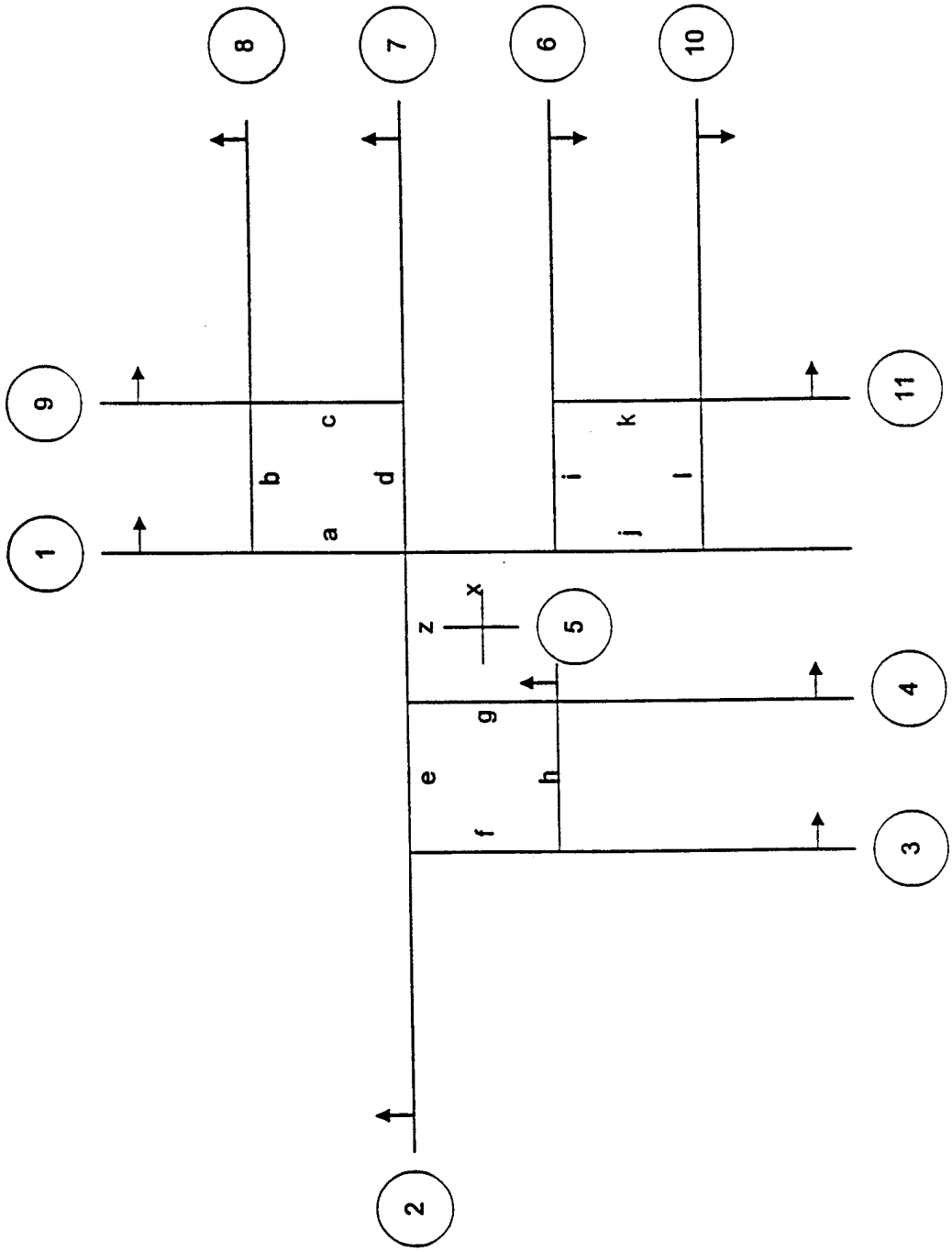

Figure 9

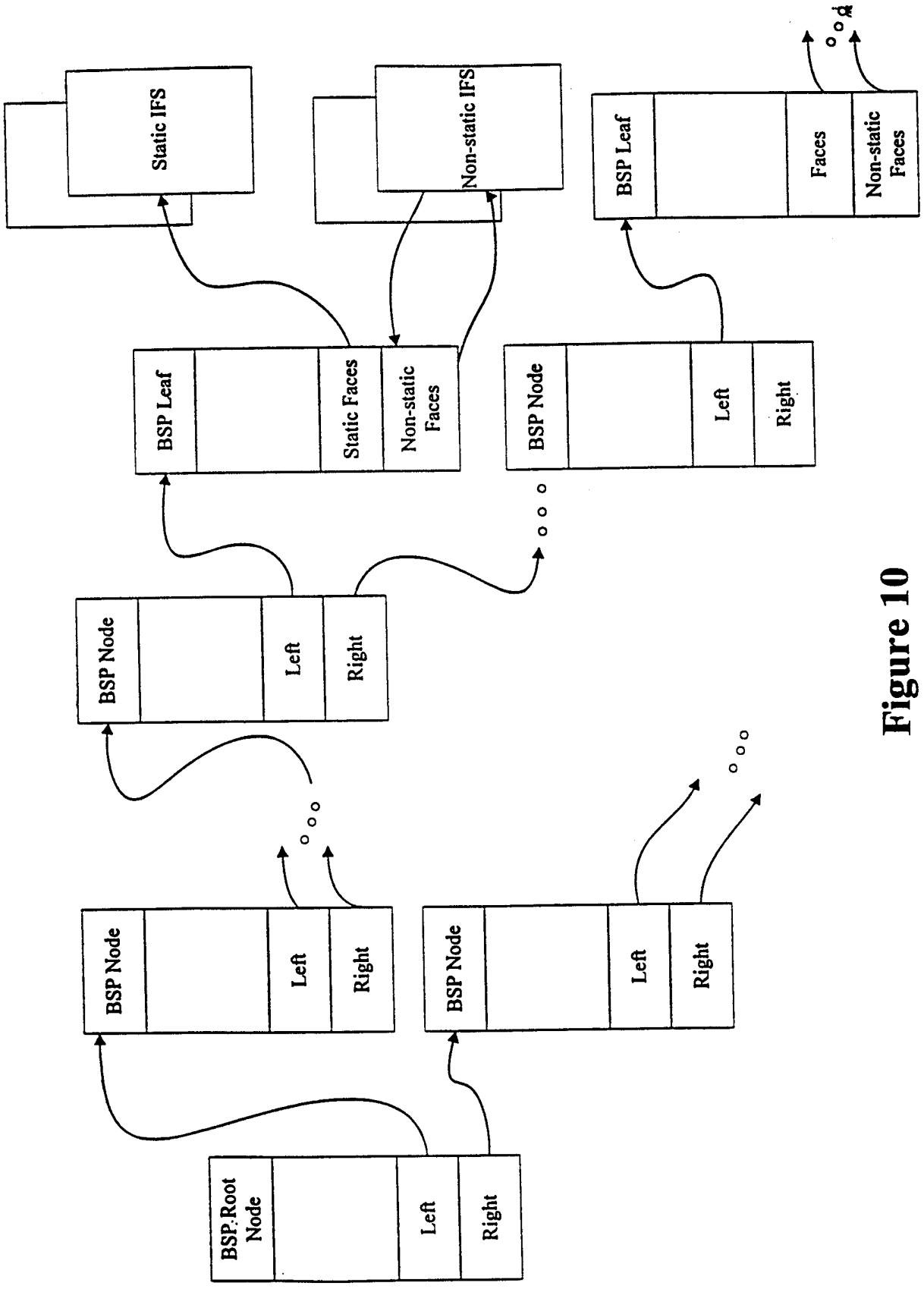
Static IFS
Non-static IFS
BSP Leaf
Faces
Non-static Faces
BSP Leaf
Static Faces
Non-static Faces
BSP Node
Left
Right
BSP Node
Left
Right
BSP Node
Left
Right
BSP Node
Left
Right
BSP Root Node
Left
Right

Figure 10

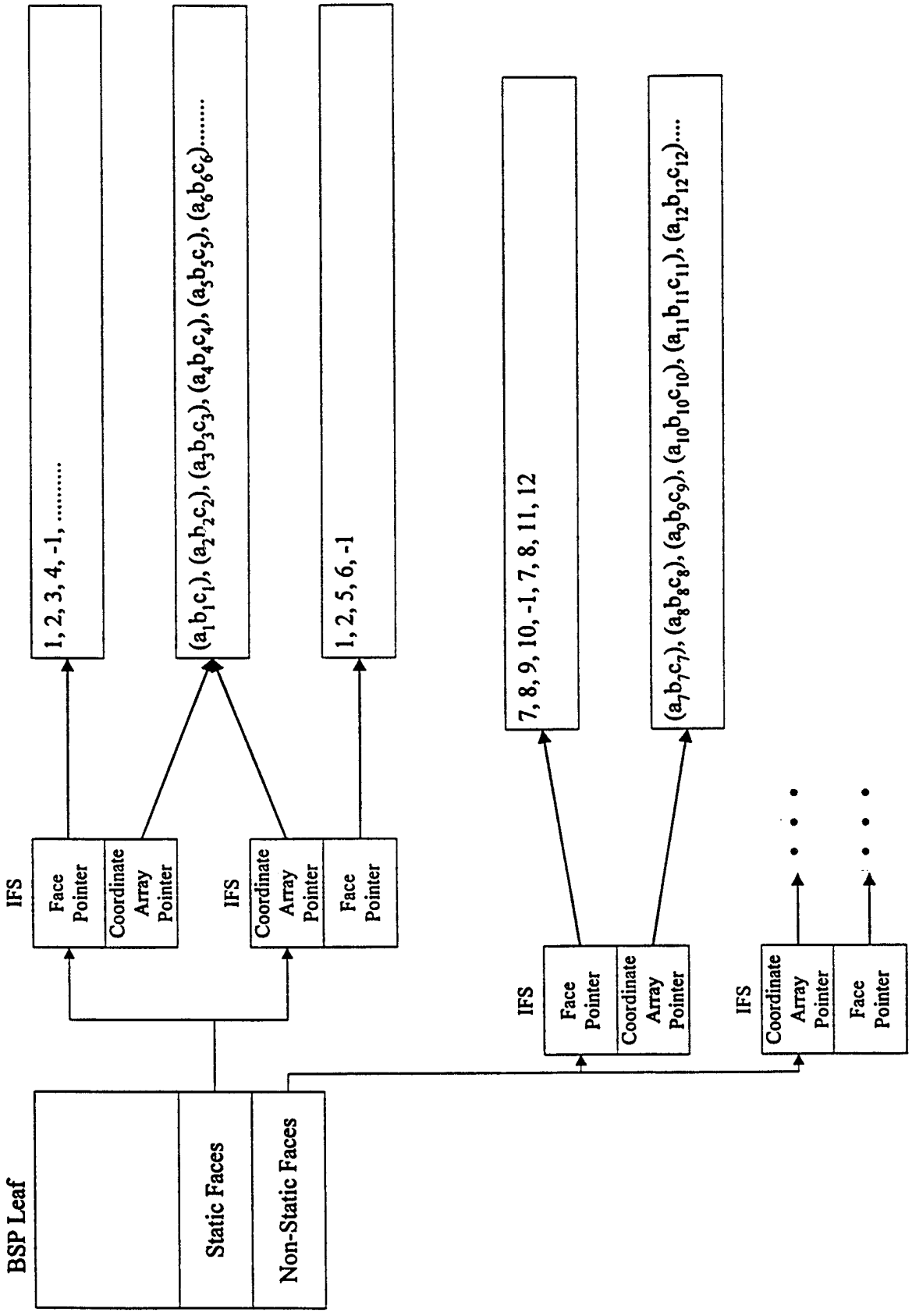
BSP Leaf
Static Faces
Non-Static Faces
IFS
Face Pointer
Coordinate Array Pointer
IFS
Coordinate Array Pointer
Face Pointer
IFS
Face Pointer
Coordinate Array Pointer
IFS
Coordinate Array Pointer
Face Pointer
1, 2, 3, 4, -1, ...
(a1b1c1), (a2b2c2), (a3b3c3), (a4b4c4), (a5b5c5), (a6b6c6).....
1, 2, 5, 6, -1
7, 8, 9, 10, -1, 7, 8, 11, 12
(a7b7c7), (a8b8c8), (a9b9c9), (a10b10c10), (a11b11c11), (a12b12c12)....

Figure 11a

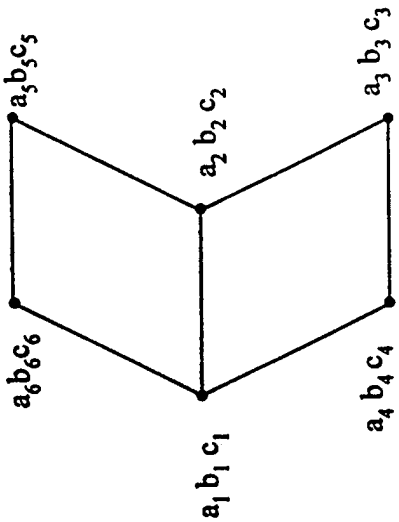

Figure 11b

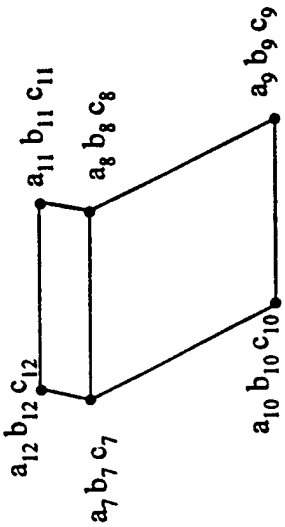

Figure 11c

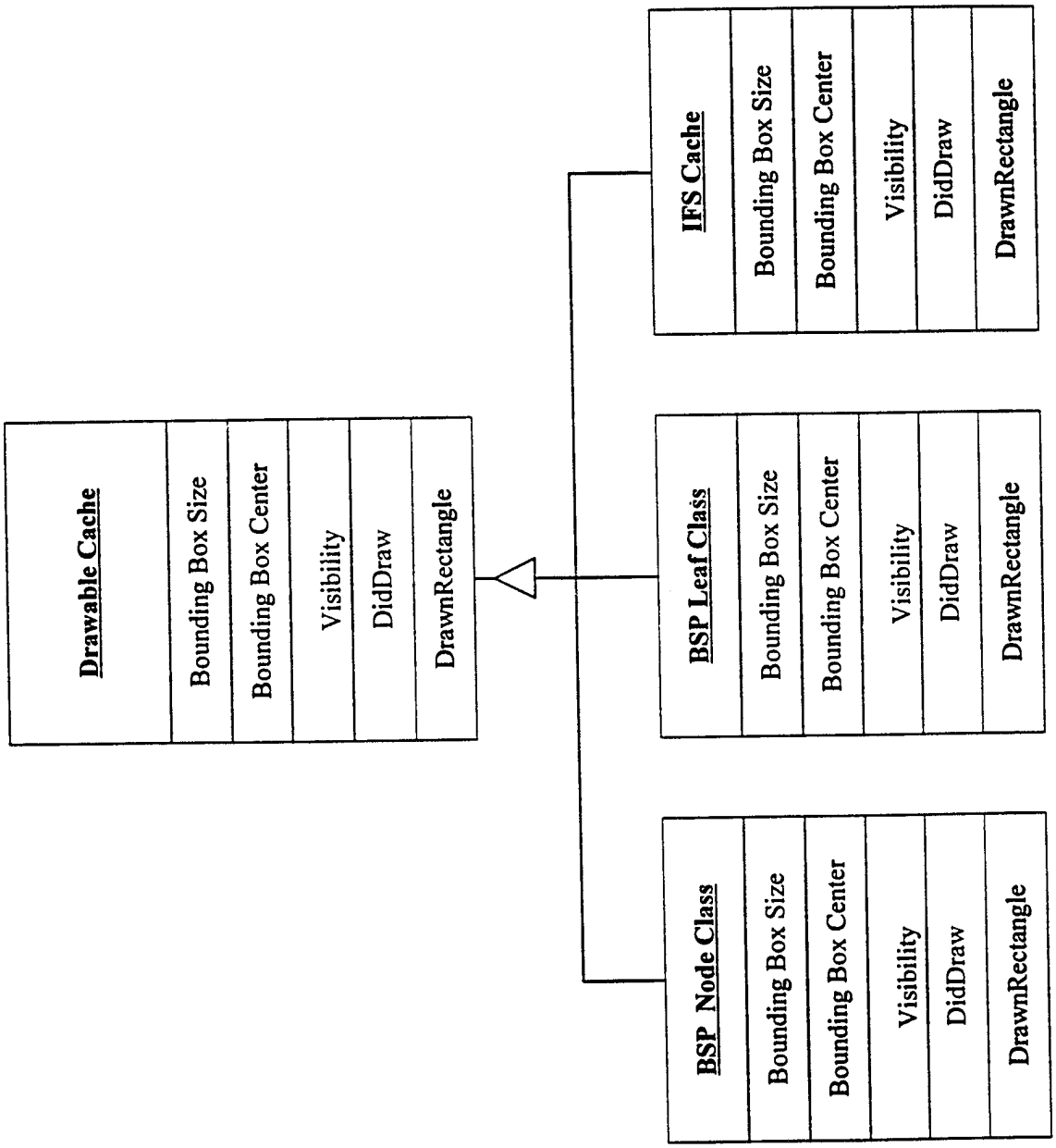
Drawable Cache
Bounding Box Size
Bounding Box Center
Visibility
DidDraw
DrawnRectangle
IFS Cache
Bounding Box Size
Bounding Box Center
Visibility
DidDraw
DrawnRectangle
BSP Leaf Class
Bounding Box Size
Bounding Box Center
Visibility
DidDraw
DrawnRectangle
BSP Node Class
Bounding Box Size
Bounding Box Center
Visibility
DidDraw
DrawnRectangle

Figure 12

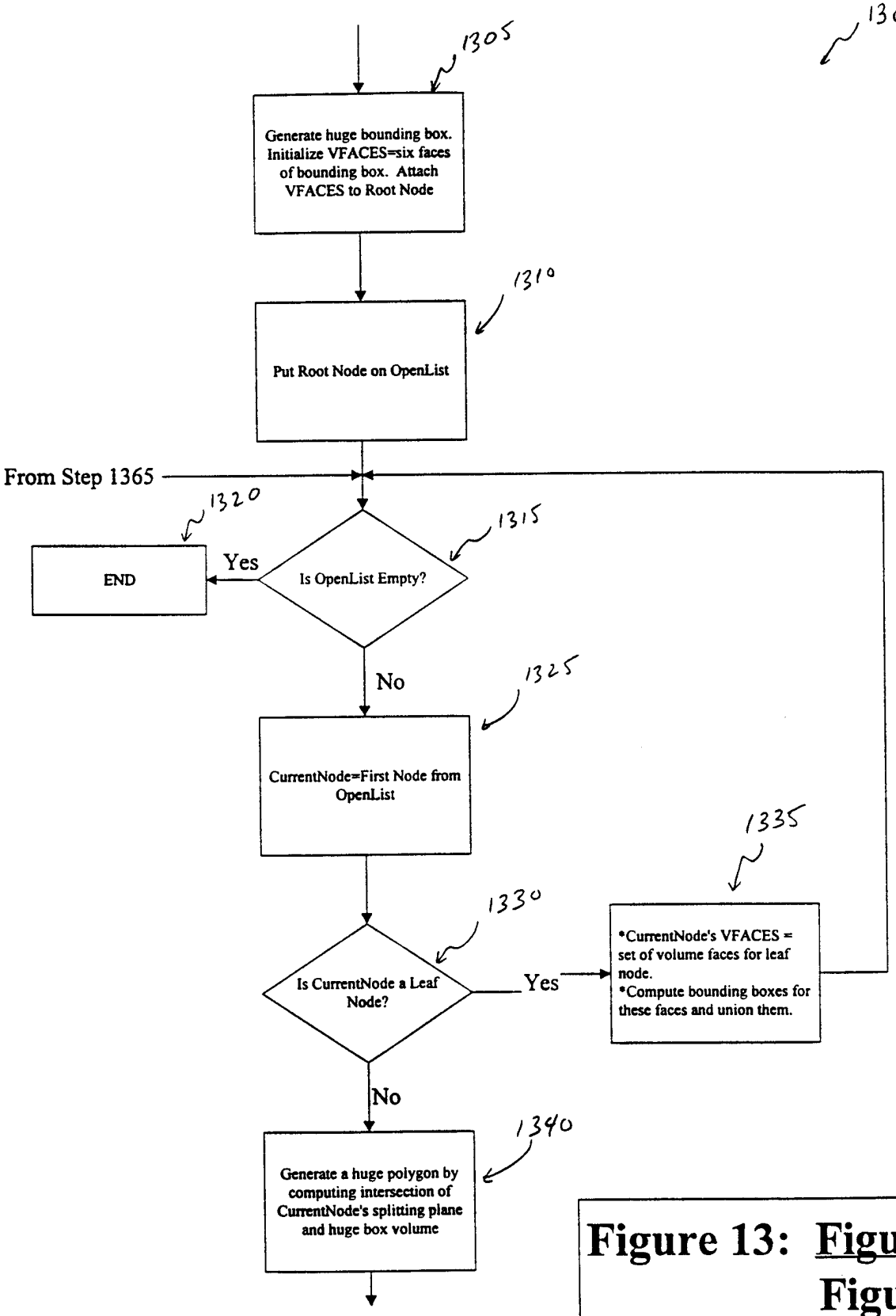

Figure 13: Figure 13a Figure 13b

Figure 13a

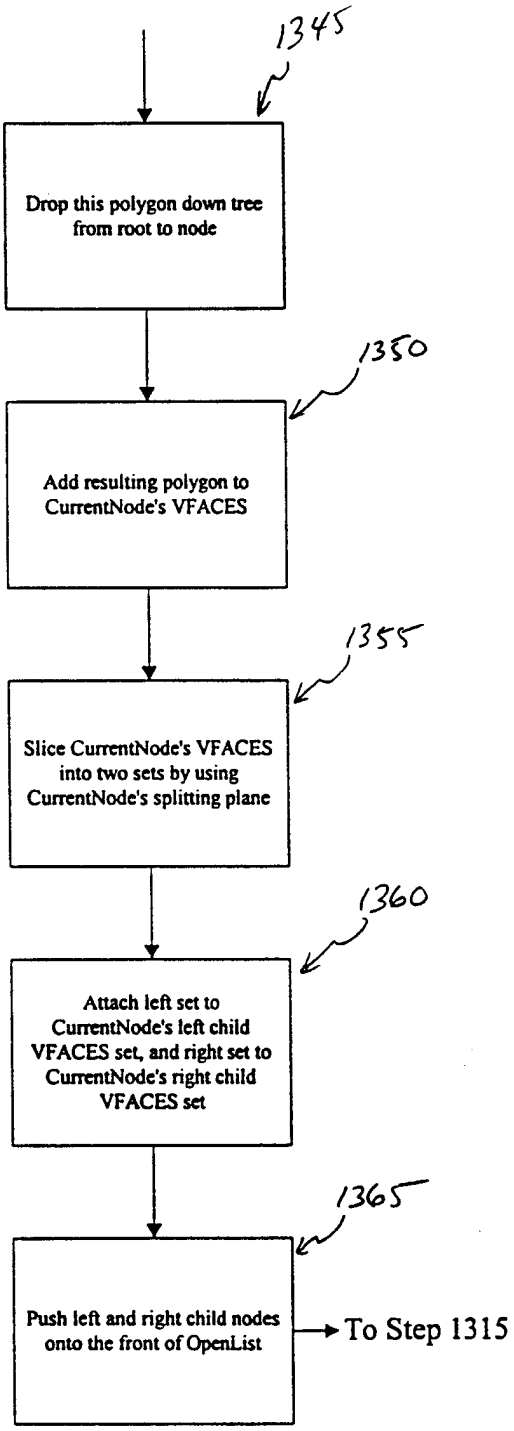
1345
Drop this polygon down tree from root to node
1350
Add resulting polygon to CurrentNode's VFACES
1355
Slice CurrentNode's VFACES into two sets by using CurrentNode's splitting plane
1360
Attach left set to CurrentNode's left child VFACES set, and right set to CurrentNode's right child VFACES set
1365
Push left and right child nodes onto the front of OpenList
To Step 1315

Figure 13b

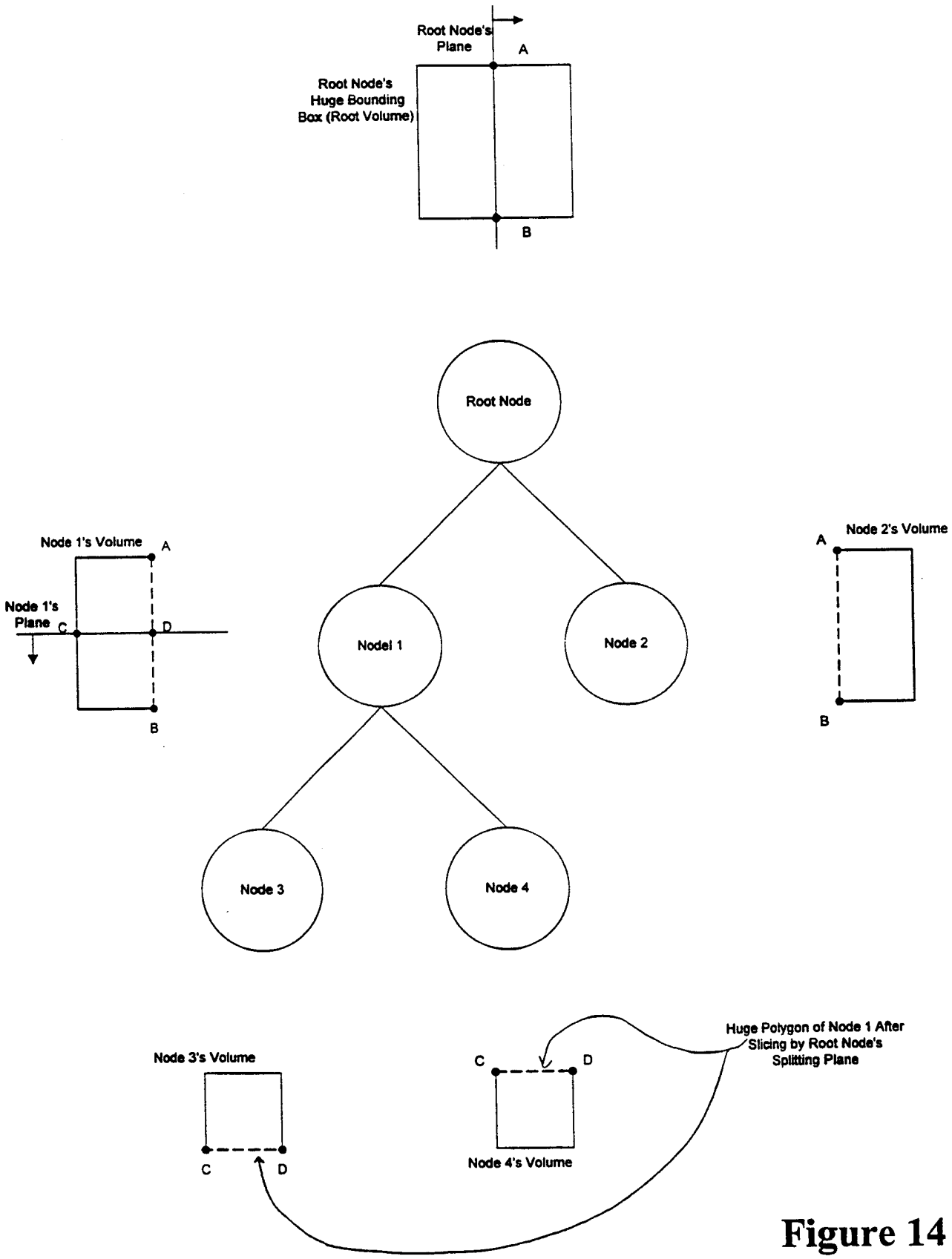
Root Node's Plane
A
Root Node's Huge Bounding Box (Root Volume)
B
Root Node
Node 1's Volume
A
Node 1's Plane
C
D
B
Nodel 1
Node 2
Node 2's Volume
A
B
Node 3
Node 4
Node 3's Volume
C
D
C
D
Node 4's Volume
Huge Polygon of Node 1 After Slicing by Root Node's Splitting Plane

Figure 14

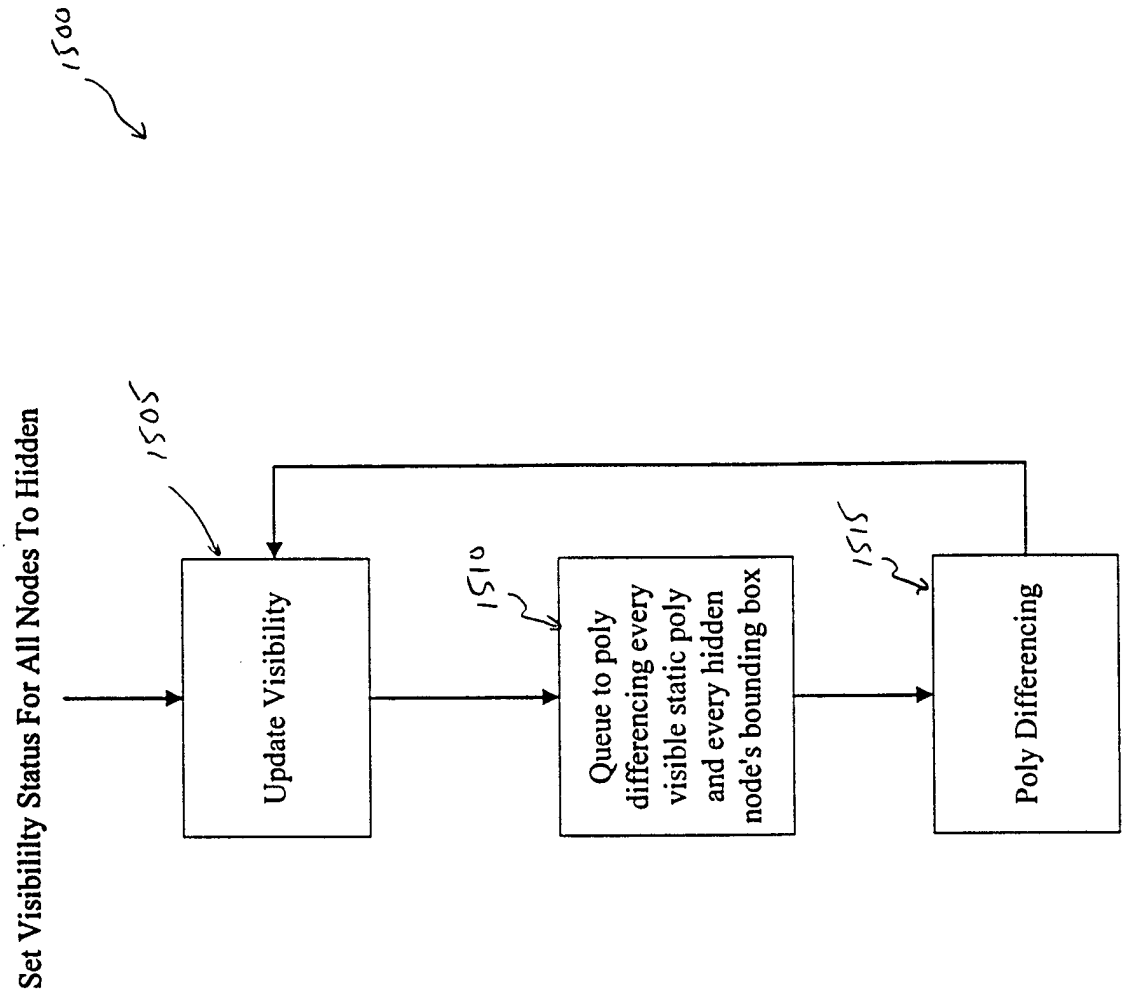
1500
Set Visibility Status For All Nodes To Hidden
1505
Update Visibility
1510
Queue to poly differencing every visible static poly and every hidden node's bounding box
1515
Poly Differencing

Figure 15

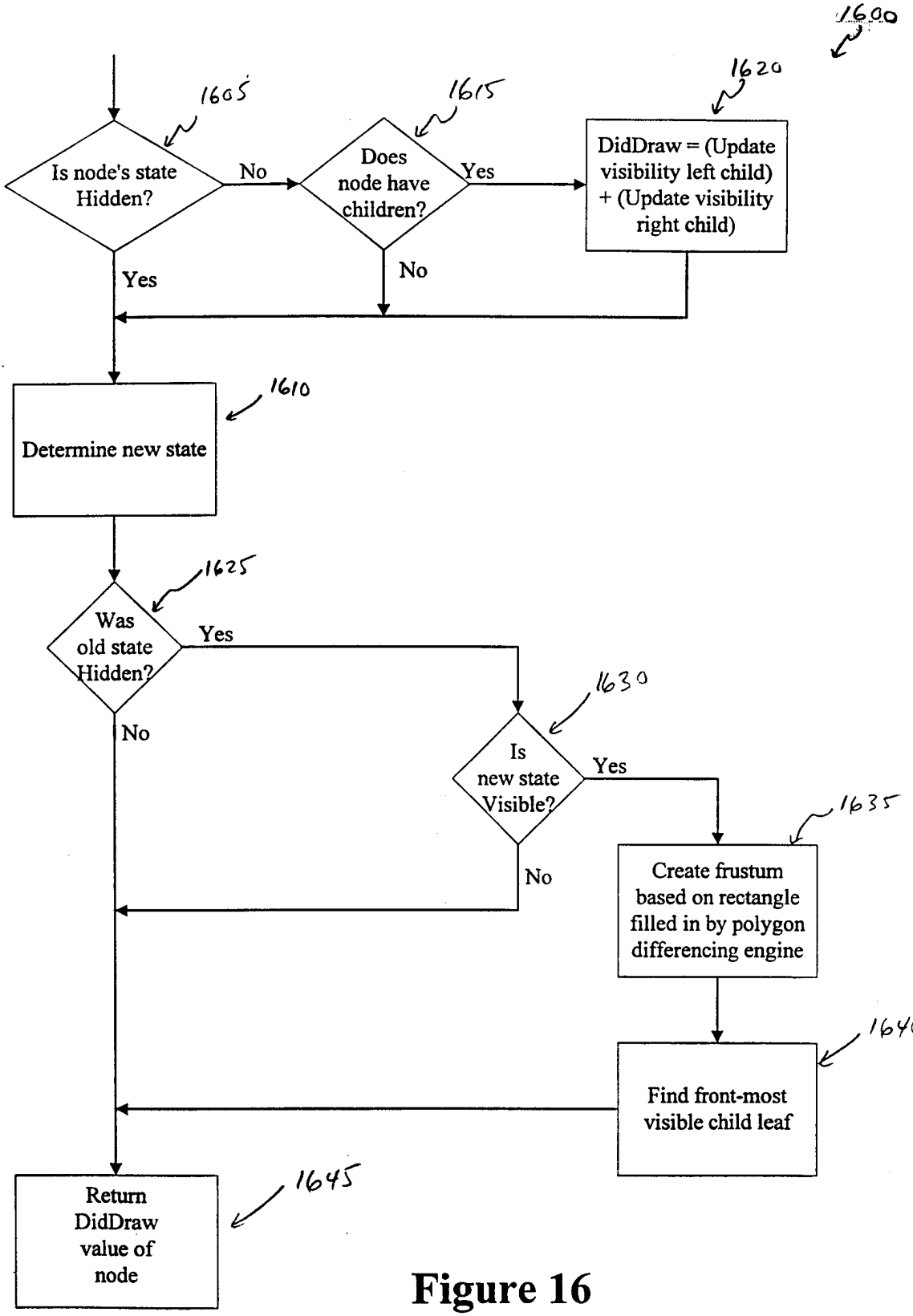
1600
1605
Is node's state Hidden?
No
1615
Does node have children?
Yes
1620
DidDraw = (Update visibility left child) + (Update visibility right child)
Yes
No
1610
Determine new state
1625
Was old state Hidden?
Yes
No
1630
Is new state Visible?
Yes
No
1635
Create frustum based on rectangle filled in by polygon differencing engine
1640
Find front-most visible child leaf
1645
Return DidDraw value of node

Figure 16

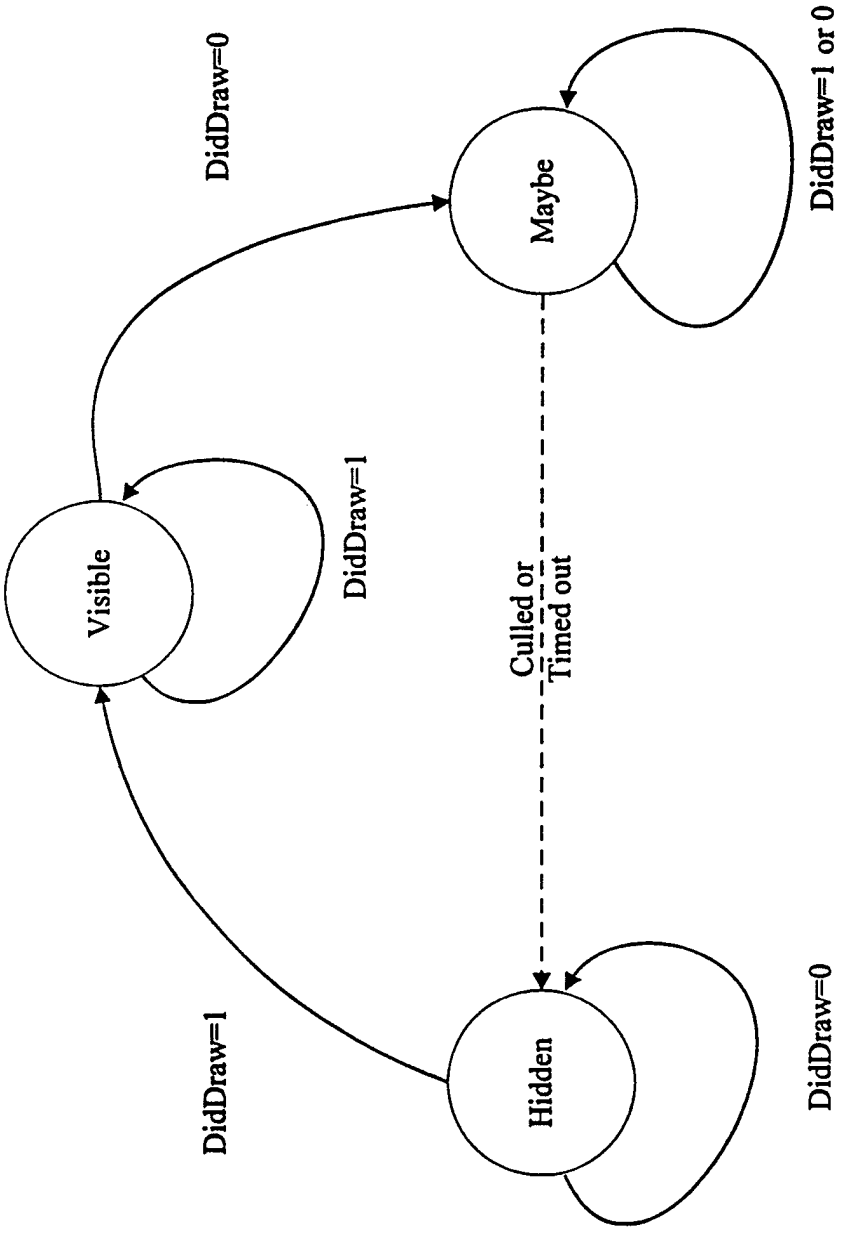
Visible
Maybe
Hidden
DidDraw=0
DidDraw=1
DidDraw=1 or 0
Culled or
Timed out
DidDraw=1
DidDraw=0

Figure 17

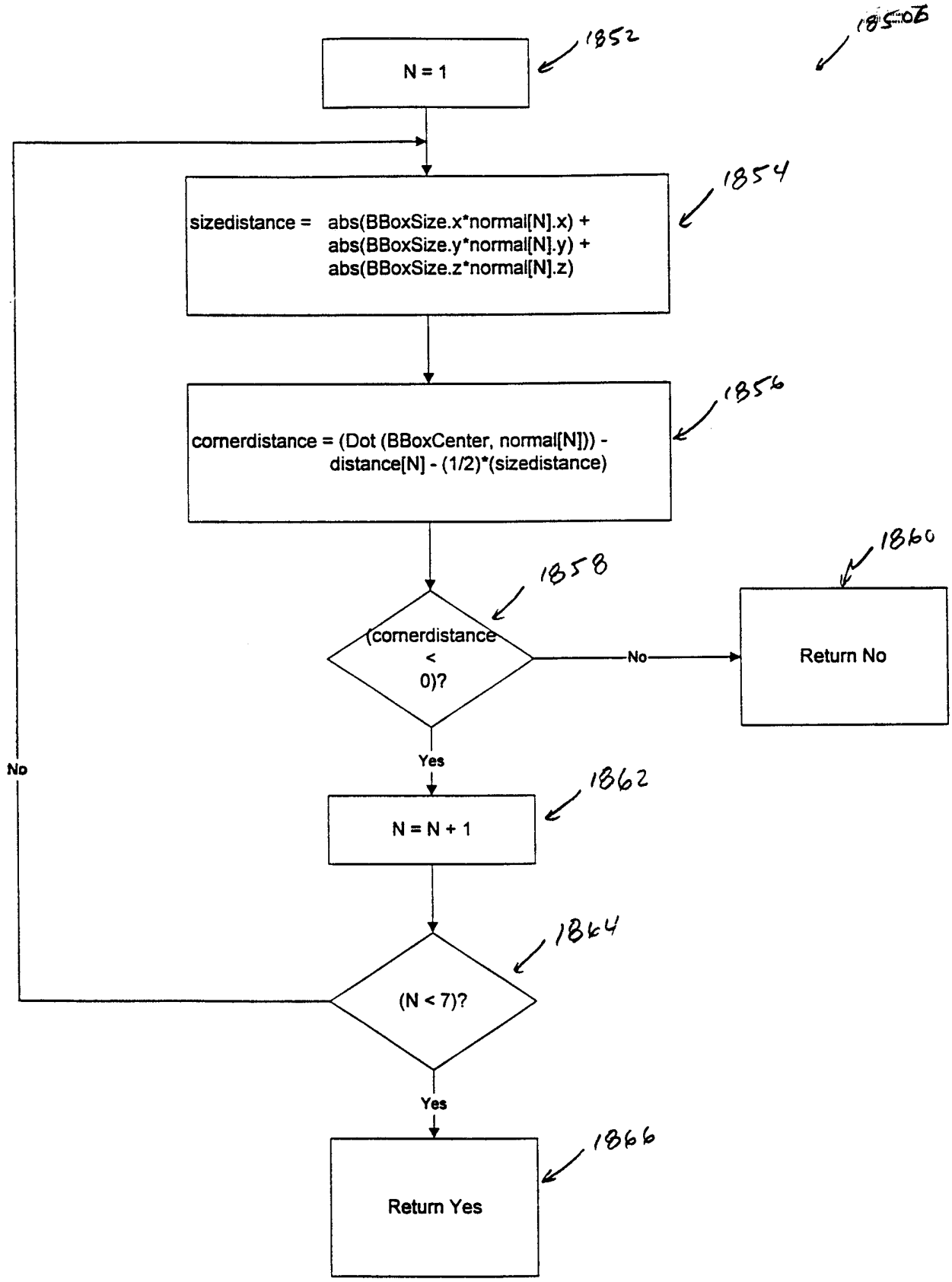
1850
1852
N = 1
1854
sizedistance = abs(BBoxSize.x*normal[N].x) +
abs(BBoxSize.y*normal[N].y) +
abs(BBoxSize.z*normal[N].z)
1856
cornerdistance = (Dot (BBoxCenter, normal[N])) -
distance[N] - (1/2)*(sizedistance)
1858
(cornerdistance < 0)?
No
1860
Return No
Yes
No
1862
N = N + 1
1864
(N < 7)?
Yes
1866
Return Yes

Figure 18b

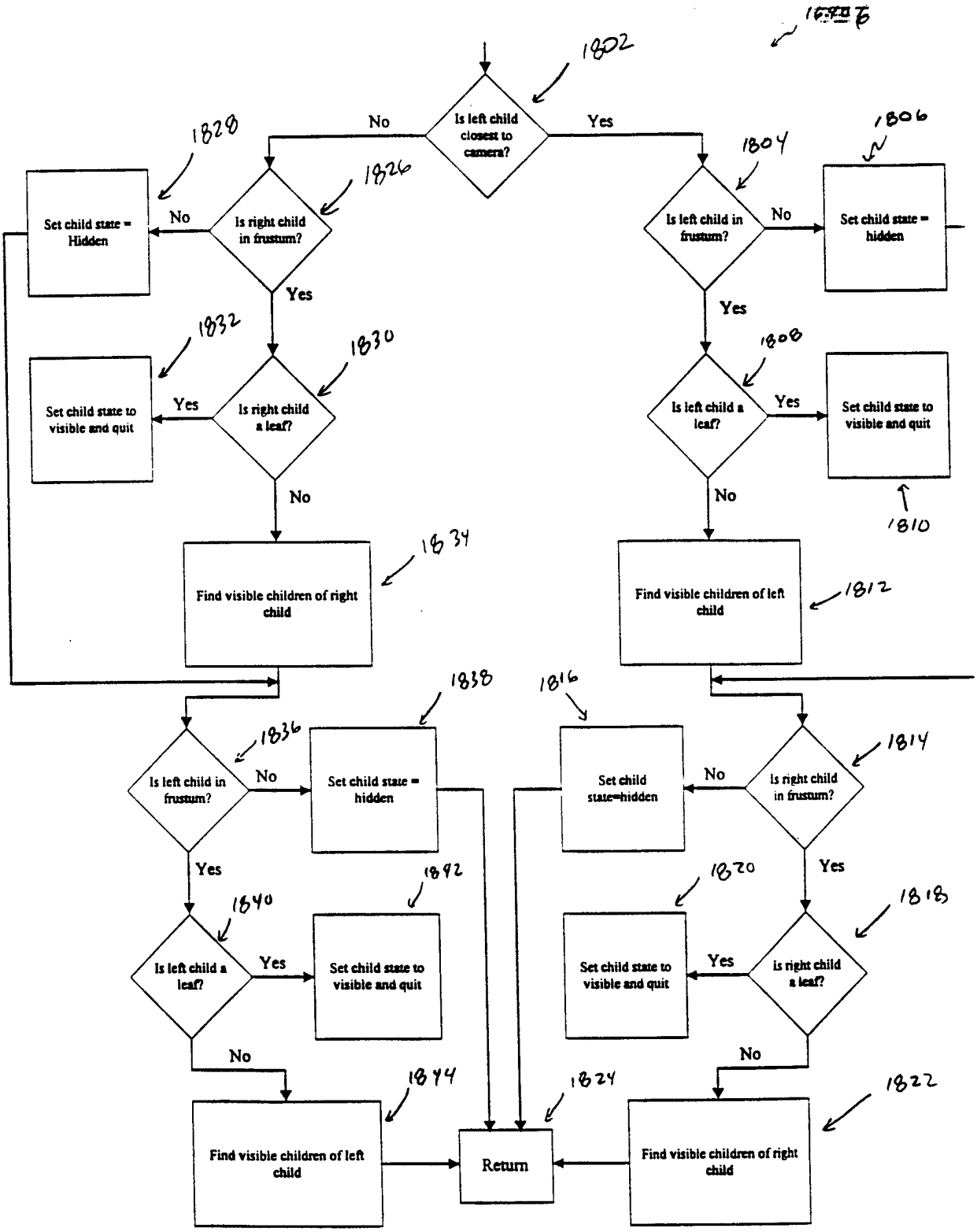
1802
Is left child closest to camera?
No
Yes
1828
Set child state = Hidden
1826
Is right child in frustum?
No
Yes
1832
Set child state to visible and quit
Yes
1830
Is right child a leaf?
No
1834
Find visible children of right child
1804
Is left child in frustum?
No
Yes
1806
Set child state = hidden
1808
Is left child a leaf?
Yes
Set child state to visible and quit
1810
No
Find visible children of left child
1812
1838
1836
Is left child in frustum?
No
Set child state = hidden
Yes
1840
Is left child a leaf?
Yes
1842
Set child state to visible and quit
No
1844
Find visible children of left child
1816
Set child state=hidden
No
1814
Is right child in frustum?
Yes
1820
Set child state to visible and quit
Yes
1818
Is right child a leaf?
No
1824
Return
1822
Find visible children of right child

Figure 18a

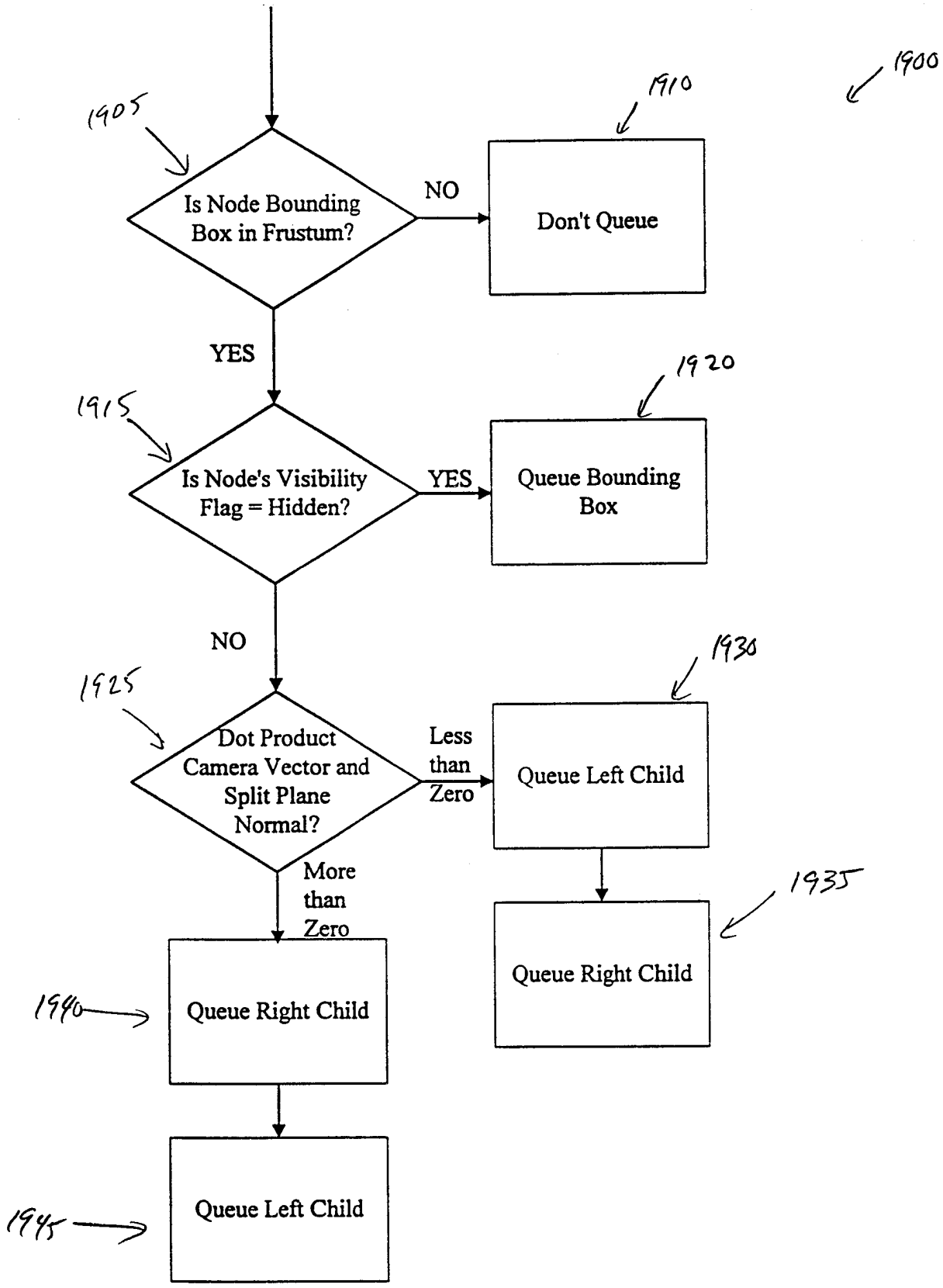
1900
1905
Is Node Bounding Box in Frustum?
NO
1910
Don't Queue
YES
1915
Is Node's Visibility Flag = Hidden?
YES
1920
Queue Bounding Box
NO
1925
Dot Product Camera Vector and Split Plane Normal?
Less than Zero
1930
Queue Left Child
1935
Queue Right Child
More than Zero
1940
Queue Right Child
1945
Queue Left Child

Figure 19

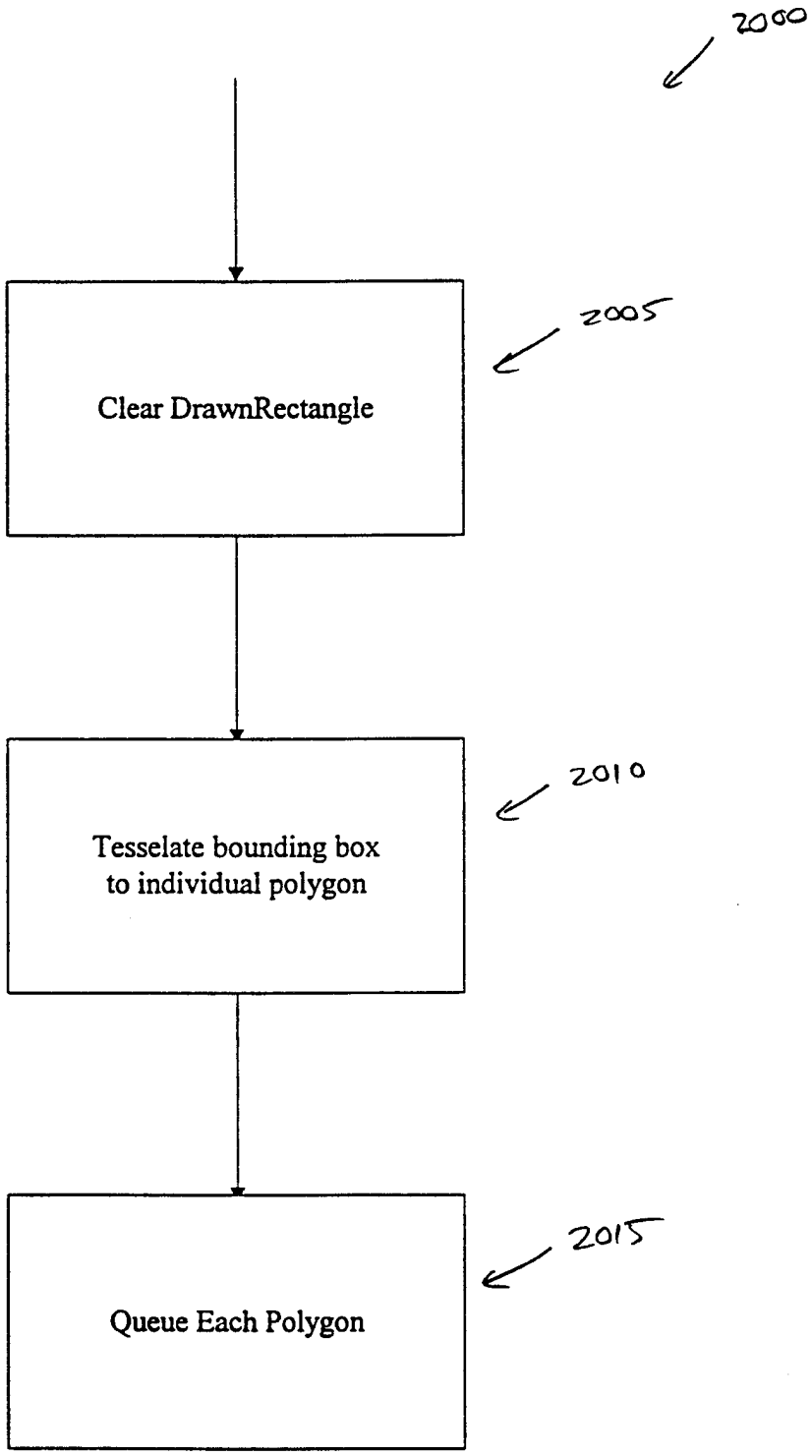
2000
Clear DrawnRectangle
2005
Tesselate bounding box
to individual polygon
2010
Queue Each Polygon
2015

Figure 20

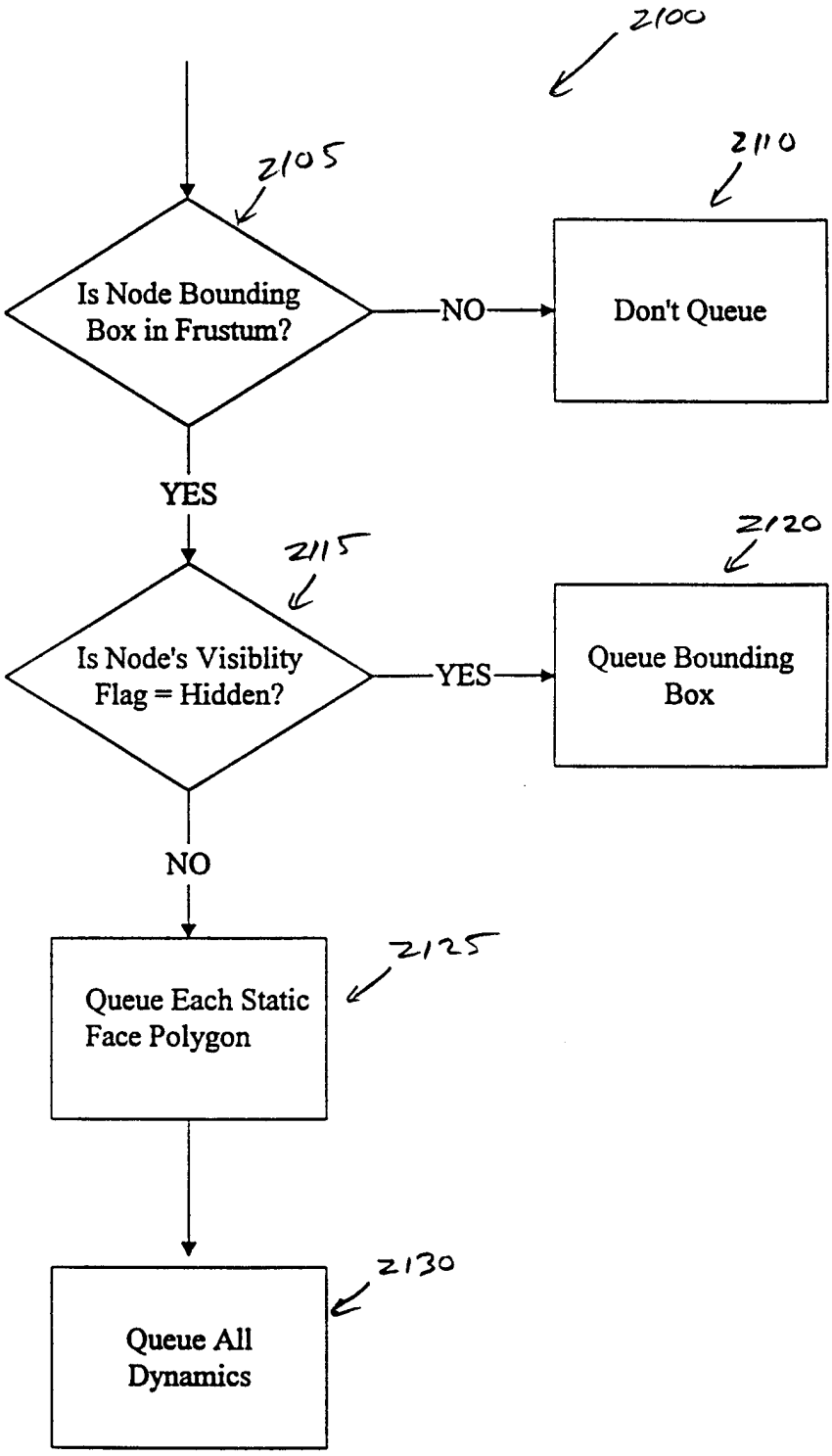
2100
2105
Is Node Bounding Box in Frustum?
NO
2110
Don't Queue
YES
2115
Is Node's Visiblity Flag = Hidden?
YES
2120
Queue Bounding Box
NO
2125
Queue Each Static Face Polygon
2130
Queue All Dynamics

Figure 21

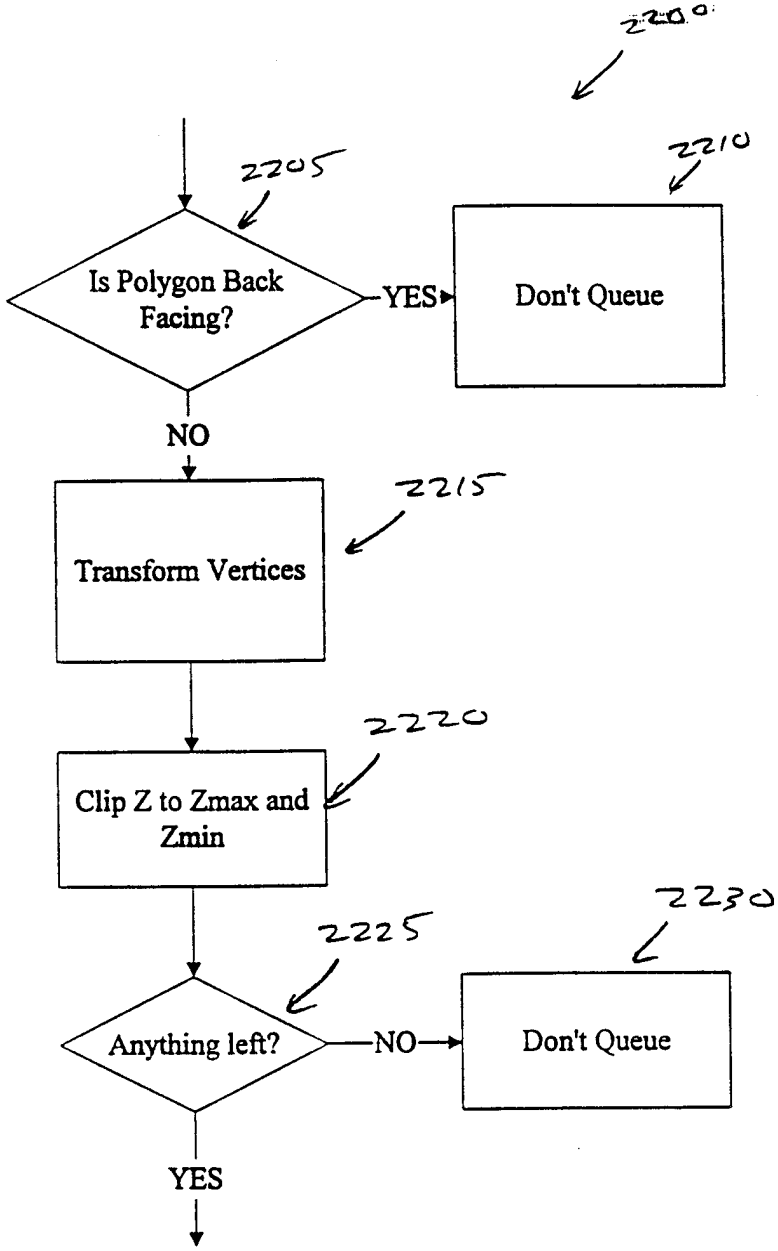

Figure 22: Figure 22A Figure 22B

Figure 22A

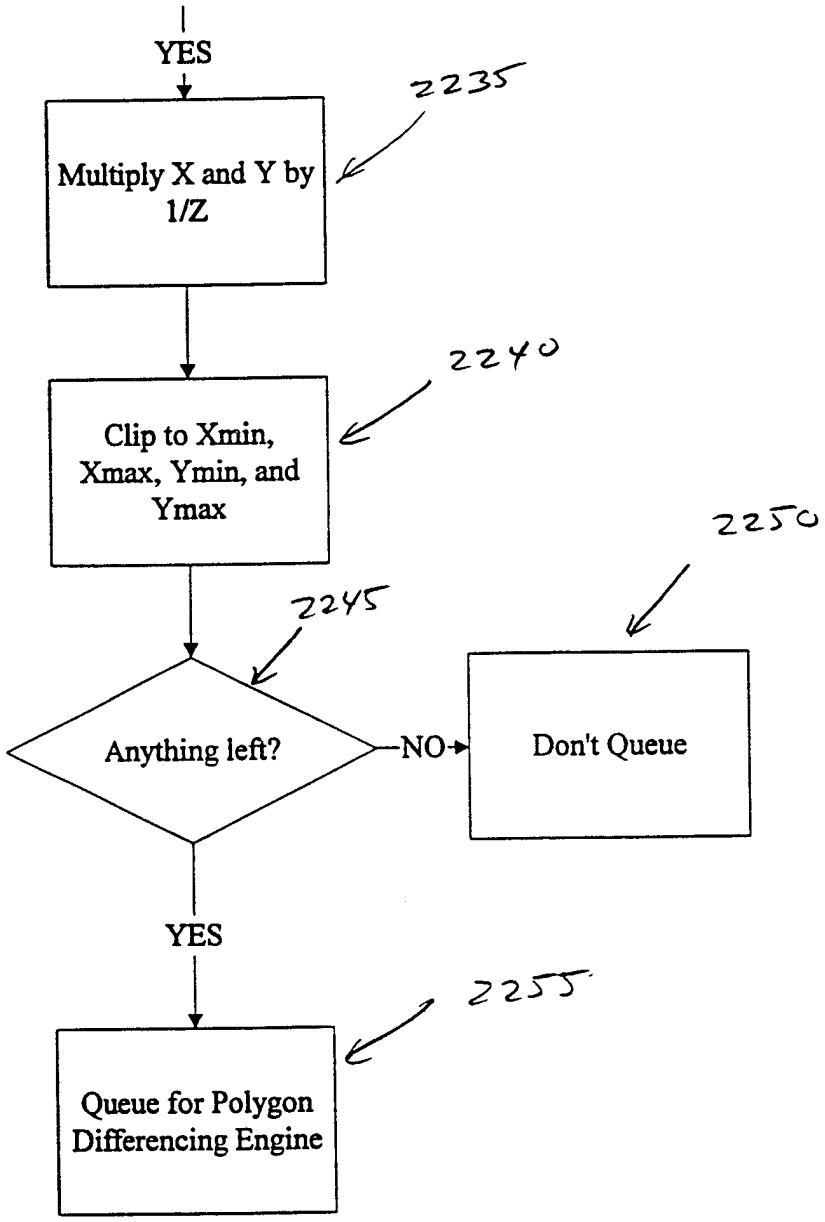
YES
2235
Multiply X and Y by 1/Z
2240
Clip to Xmin, Xmax, Ymin, and Ymax
2250
2245
Anything left?
NO
Don't Queue
YES
2255
Queue for Polygon Differencing Engine

Figure 22B

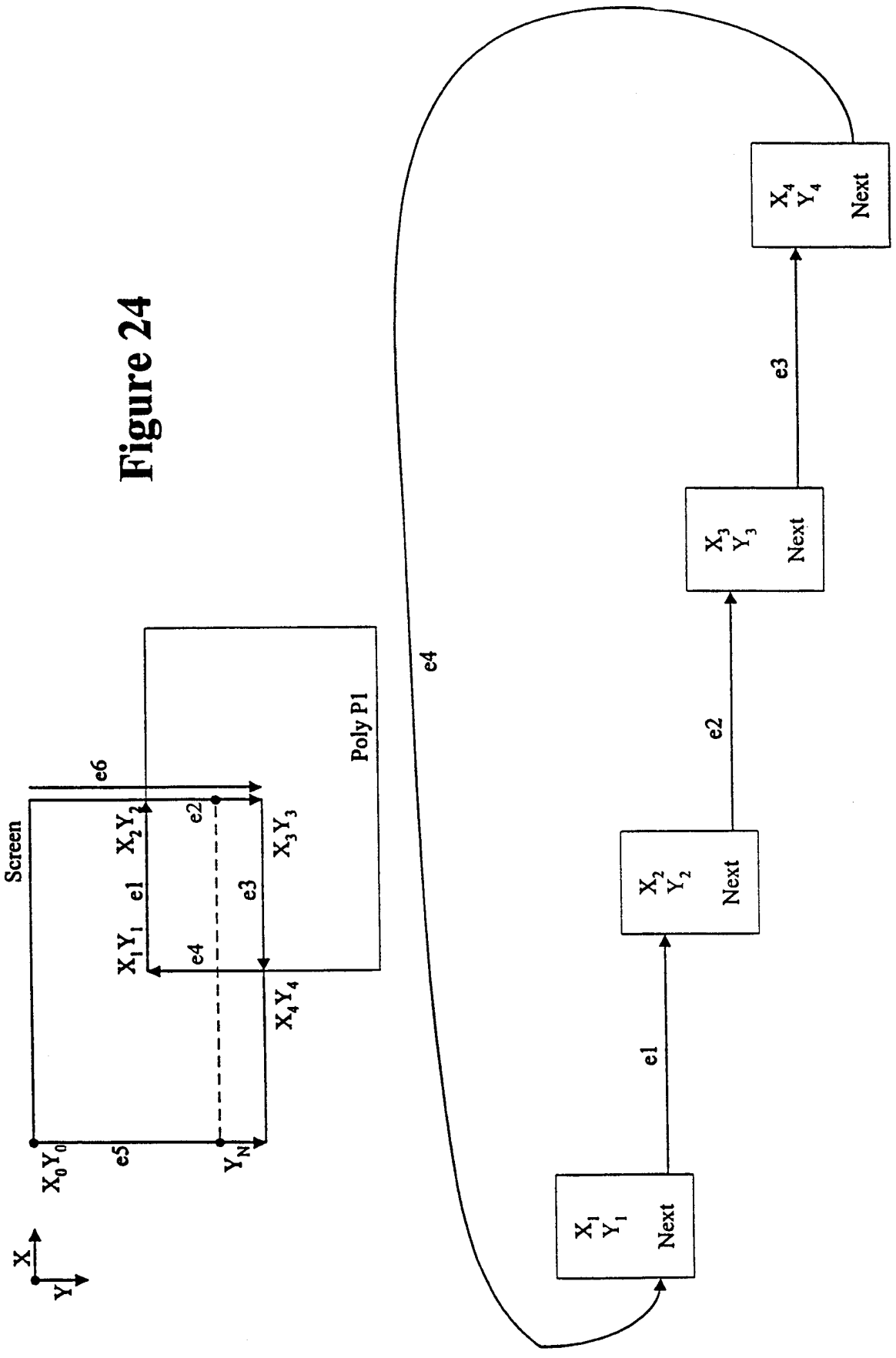

Figure 24

Figure 23

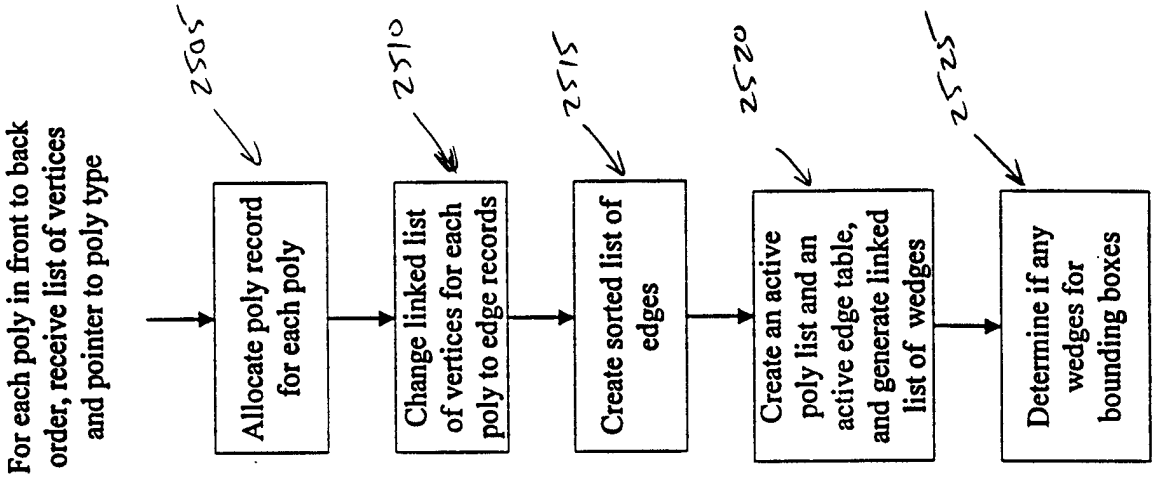

Figure 25

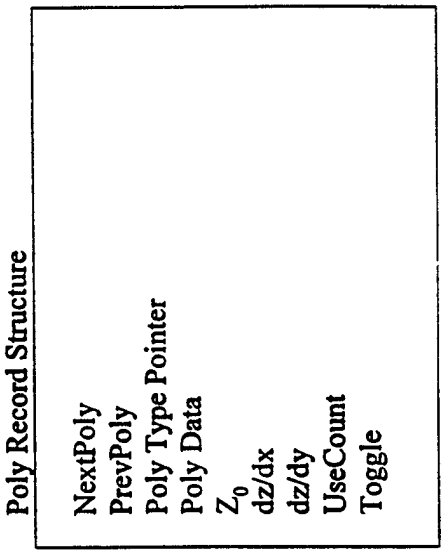

Figure 26

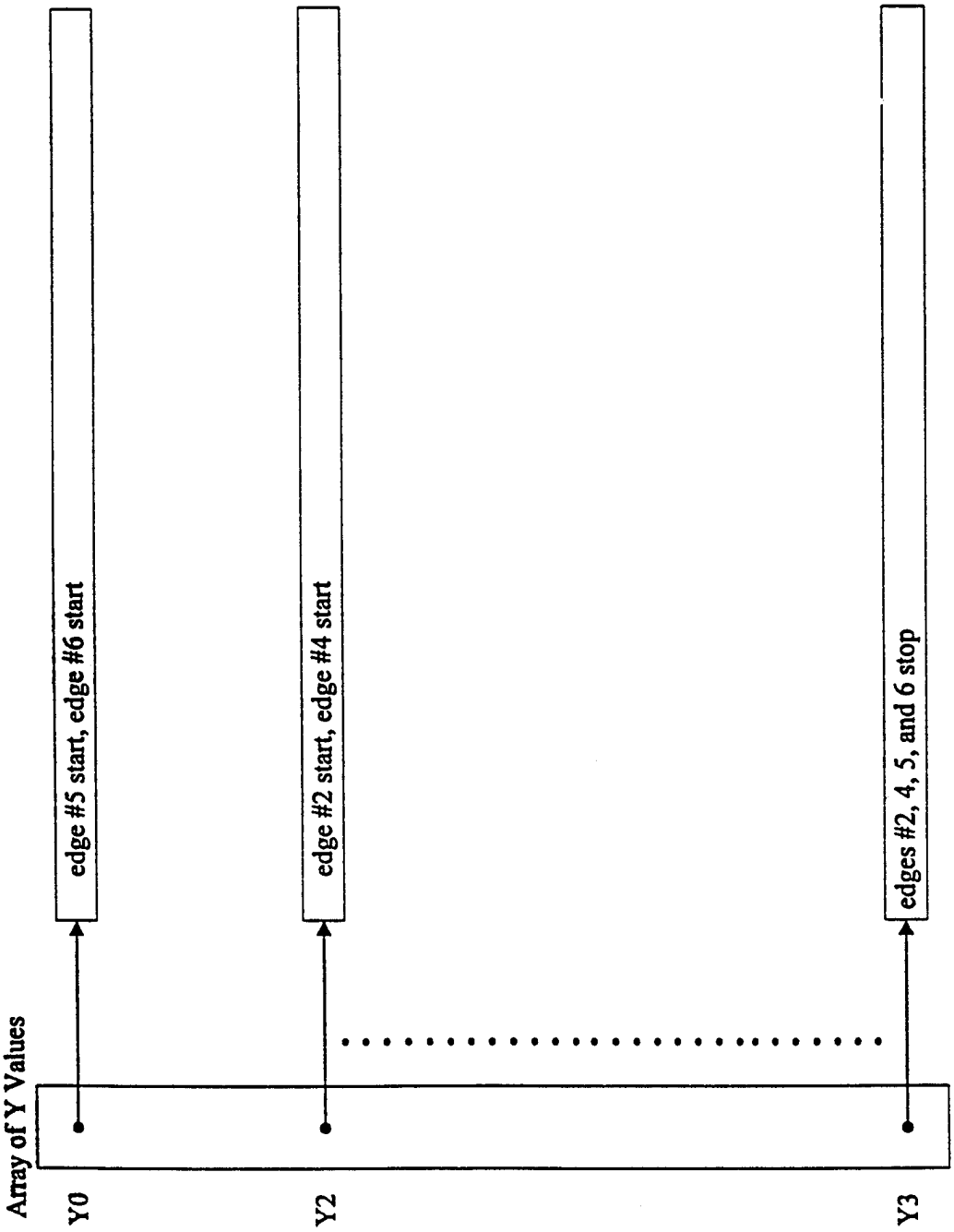
Array of Y Values
Y0
Y2
Y3
edge #5 start, edge #6 start
edge #2 start, edge #4 start
edges #2, 4, 5, and 6 stop

Figure 27

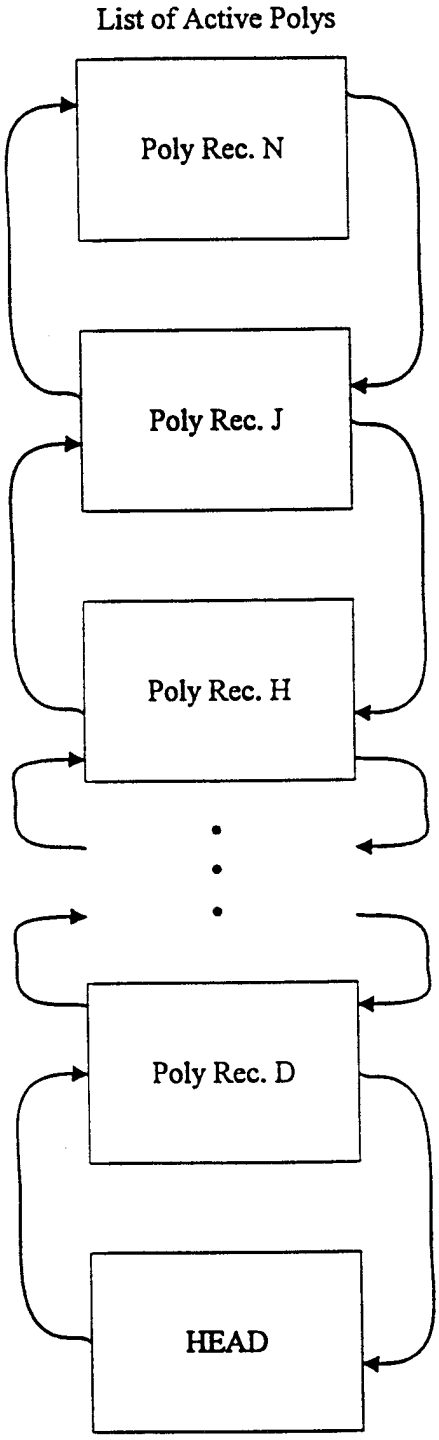
List of Active Polys
Poly Rec. N
Poly Rec. J
Poly Rec. H
Poly Rec. D
HEAD

Figure 28

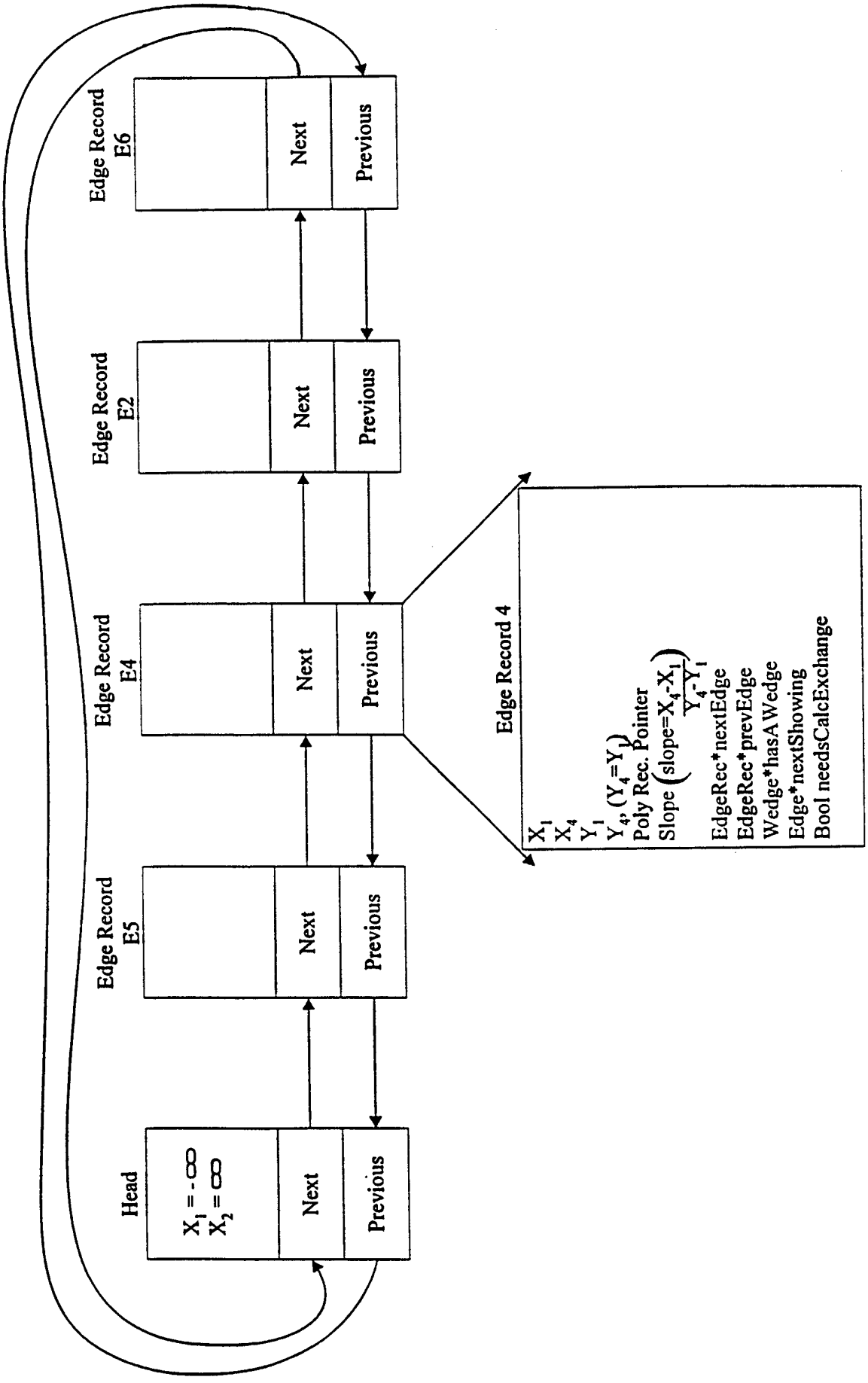
Head
$X_1 = -\infty$
$X_2 = \infty$
Next
Previous
Edge Record E5
Edge Record E4
Edge Record E2
Edge Record E6
Edge Record 4
X_1
X_4
Y_1
Y_4, $(Y_4 = Y_1)$
Poly Rec. Pointer
Slope $\left(\text{slope}=\frac{X_4-X_1}{Y_4-Y_1}\right)$
EdgeRec*nextEdge
EdgeRec*prevEdge
Wedge*hasAWedge
Edge*nextShowing
Bool needsCalcExchange

Figure 29

Wedge Record Structure

X1
X2
Y1
Y2
Slope 1
Slope 2
Poly Rec. Pointer
Next Wedge

Figure 30

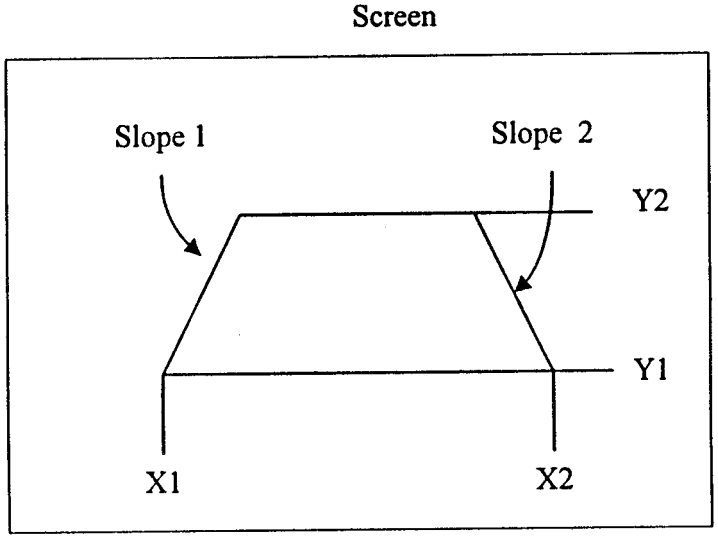

Figure 31

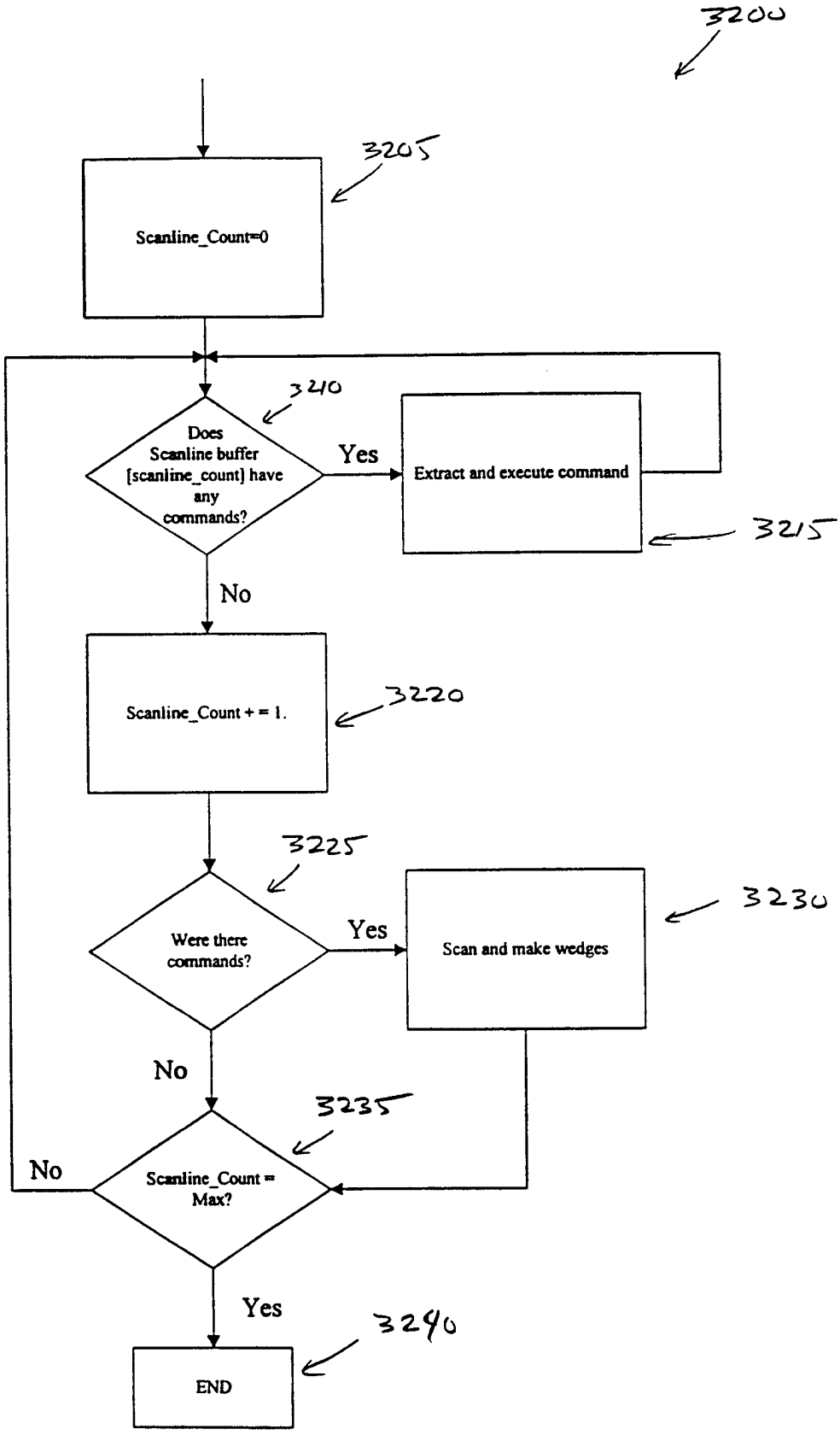
3200
3205
Scanline_Count=0
3210
Does Scanline buffer [scanline_count] have any commands?
Yes
Extract and execute command
3215
No
Scanline_Count + = 1.
3220
3225
Were there commands?
Yes
Scan and make wedges
3230
No
3235
Scanline_Count = Max?
No
Yes
END
3240

Figure 32

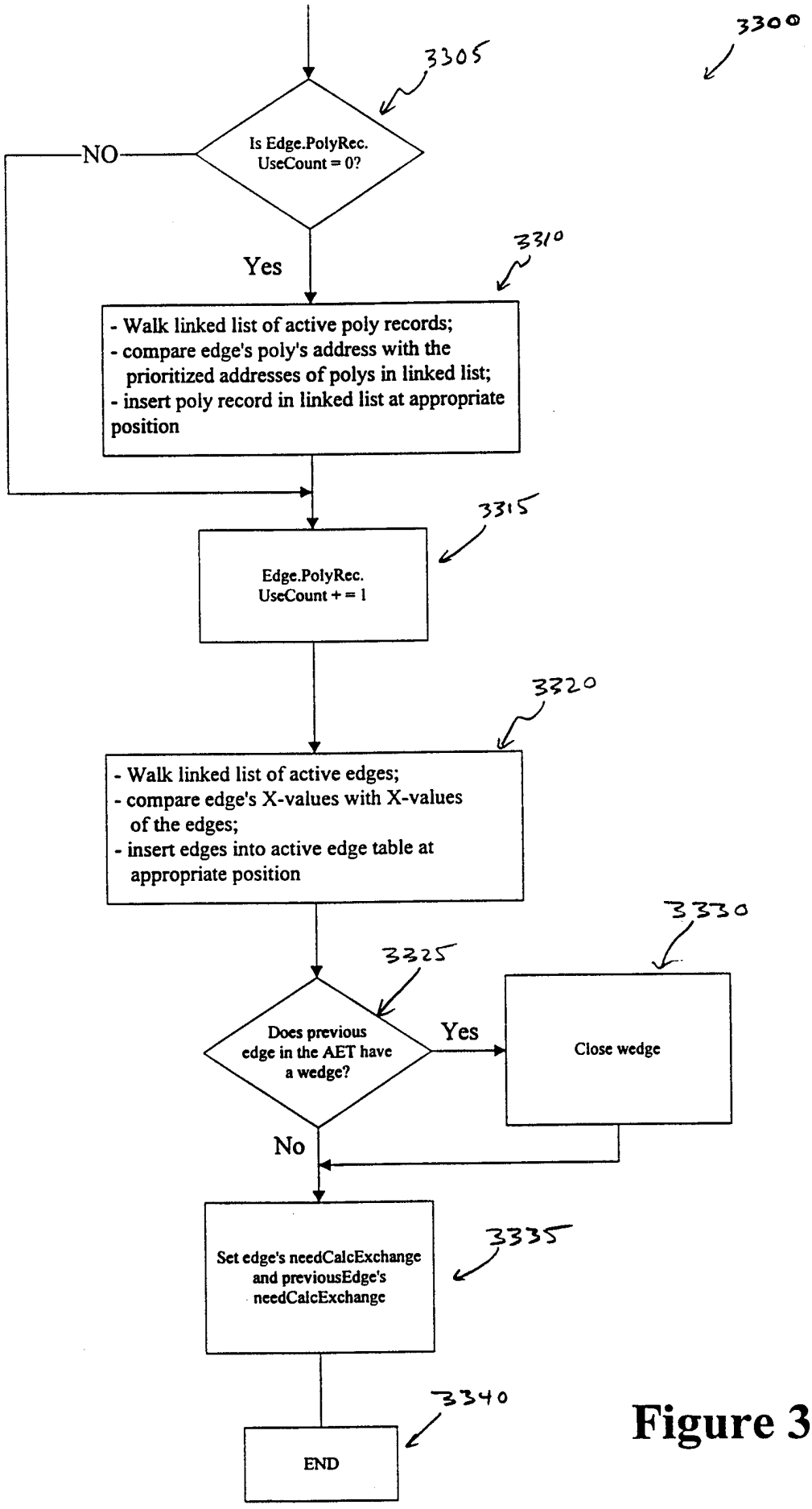
3300
3305
Is Edge.PolyRec. UseCount = 0?
NO
Yes
3310
- Walk linked list of active poly records;
- compare edge's poly's address with the prioritized addresses of polys in linked list;
- insert poly record in linked list at appropriate position
3315
Edge.PolyRec. UseCount + = 1
3320
- Walk linked list of active edges;
- compare edge's X-values with X-values of the edges;
- insert edges into active edge table at appropriate position
3330
3325
Does previous edge in the AET have a wedge?
Yes
Close wedge
No
Set edge's needCalcExchange and previousEdge's needCalcExchange
3335
3340
END

Figure 33

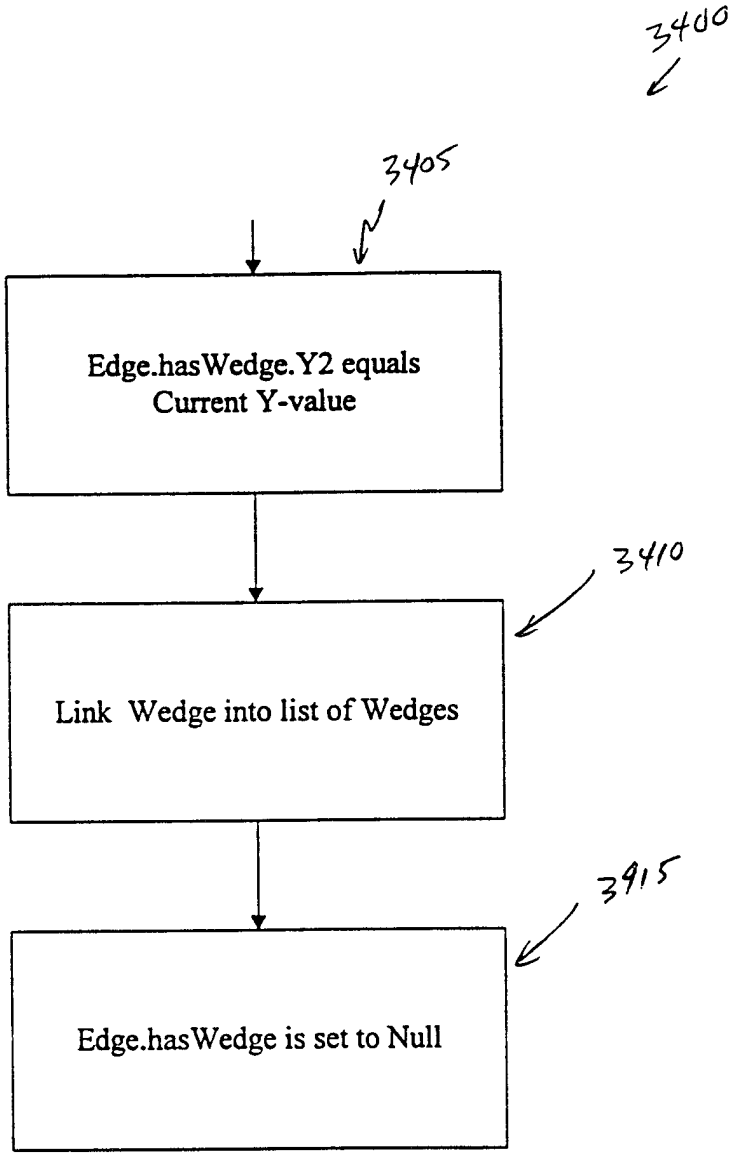
3400
3405
Edge.hasWedge.Y2 equals Current Y-value
3410
Link Wedge into list of Wedges
3415
Edge.hasWedge is set to Null

Figure 34

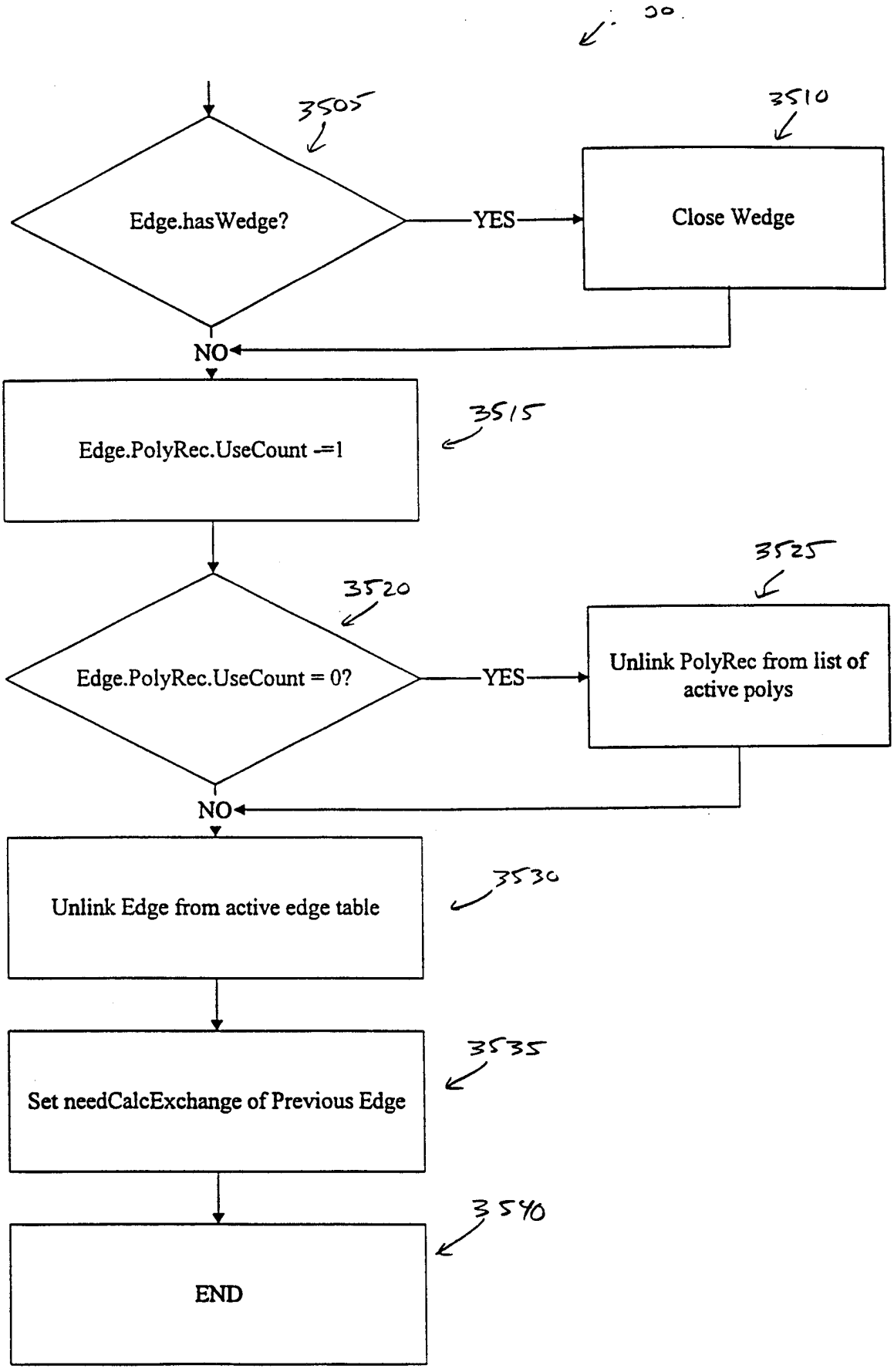
3500
3505
Edge.hasWedge?
YES
3510
Close Wedge
NO
Edge.PolyRec.UseCount -=1
3515
3520
Edge.PolyRec.UseCount = 0?
YES
3525
Unlink PolyRec from list of active polys
NO
Unlink Edge from active edge table
3530
Set needCalcExchange of Previous Edge
3535
END
3540

Figure 35

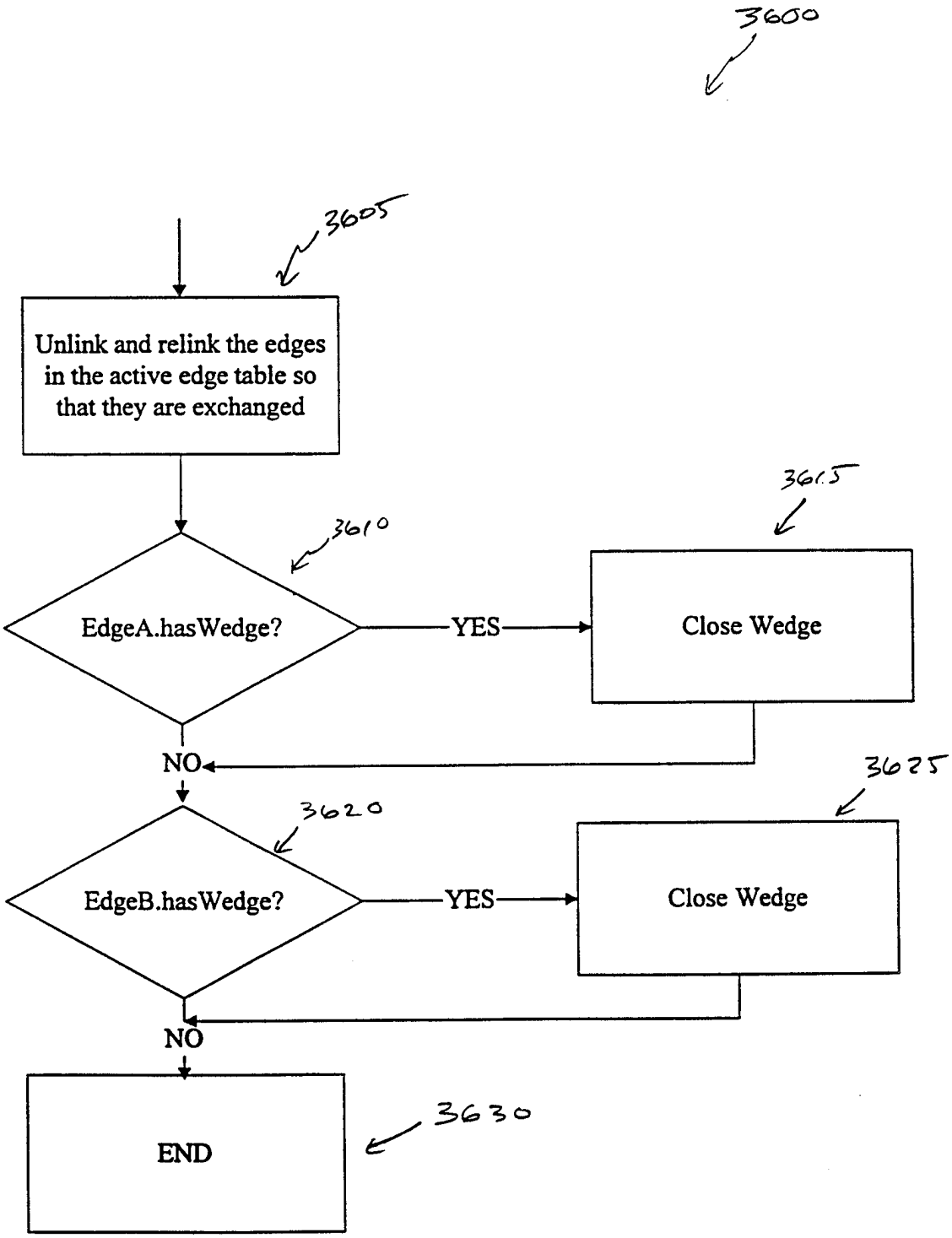
3600
3605
Unlink and relink the edges in the active edge table so that they are exchanged
3610
EdgeA.hasWedge?
YES
3615
Close Wedge
NO
3620
EdgeB.hasWedge?
YES
3625
Close Wedge
NO
END
3630

Figure 36

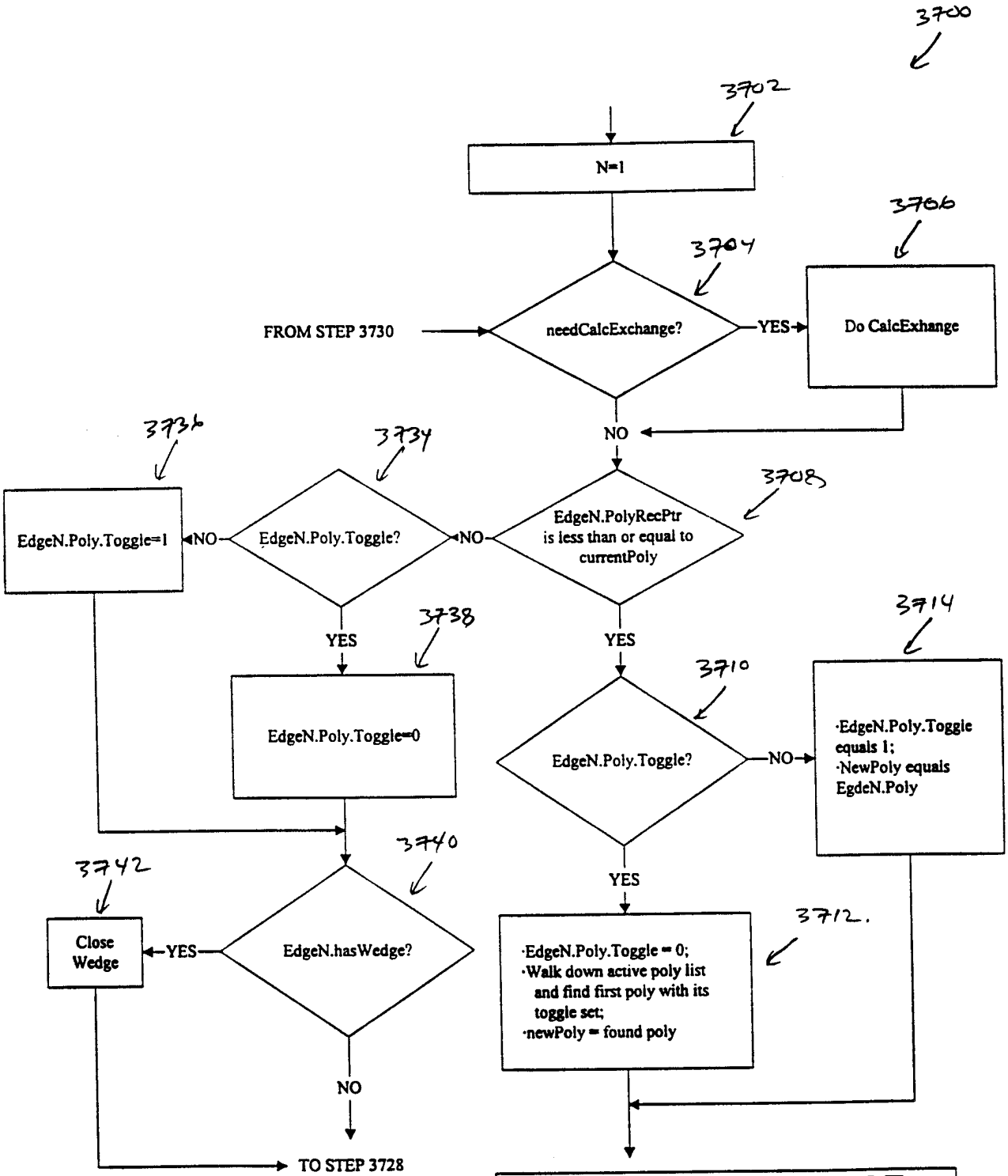

Figure 37: Figure 37a Figure 37b

Figure 37a

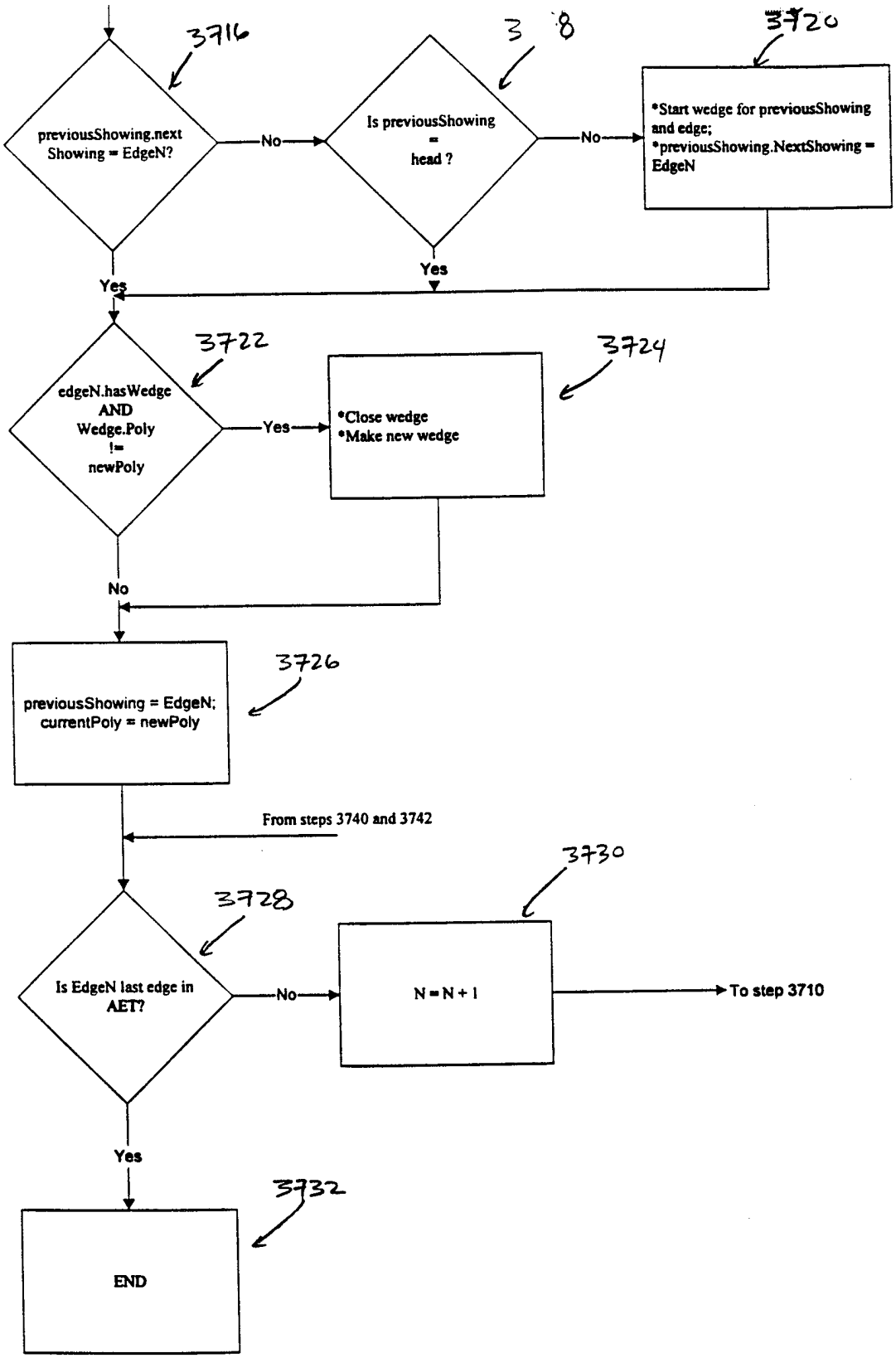
3716
previousShowing.next Showing = EdgeN?
No
3718
Is previousShowing = head ?
No
3720
*Start wedge for previousShowing and edge;
*previousShowing.NextShowing = EdgeN
Yes
Yes
3722
edgeN.hasWedge AND Wedge.Poly != newPoly
Yes
3724
*Close wedge
*Make new wedge
No
3726
previousShowing = EdgeN; currentPoly = newPoly
From steps 3740 and 3742
3728
Is EdgeN last edge in AET?
No
3730
N = N + 1
To step 3710
Yes
3732
END

Figure 37b

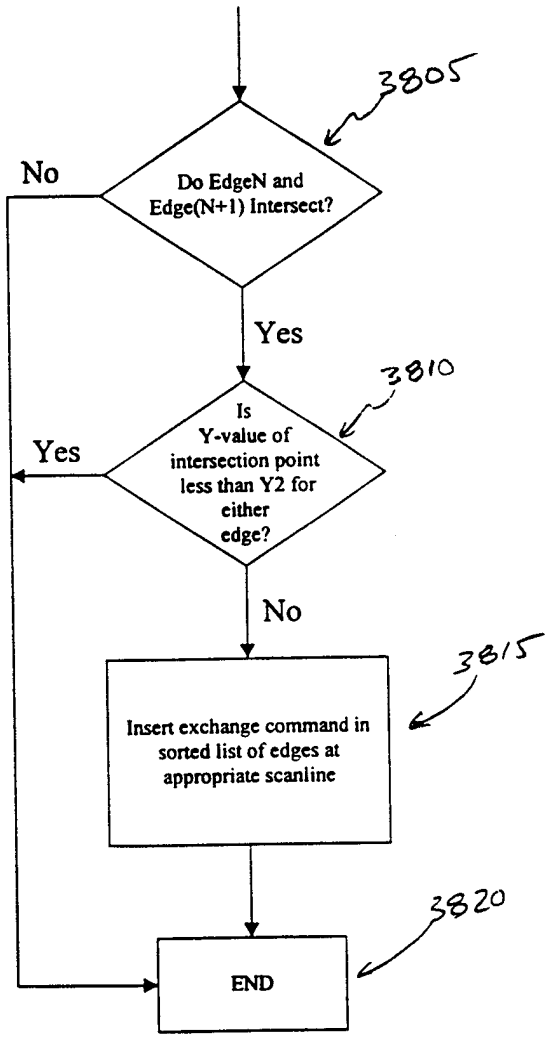
3800
3805
Do EdgeN and Edge(N+1) Intersect?
No
Yes
3810
Is Y-value of intersection point less than Y2 for either edge?
Yes
No
3815
Insert exchange command in sorted list of edges at appropriate scanline
3820
END

Figure 38

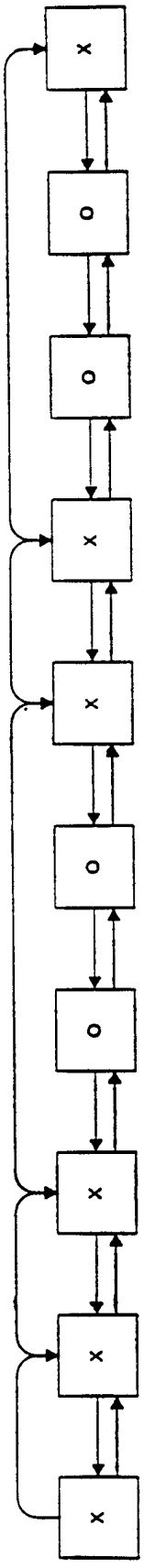

Figure 40

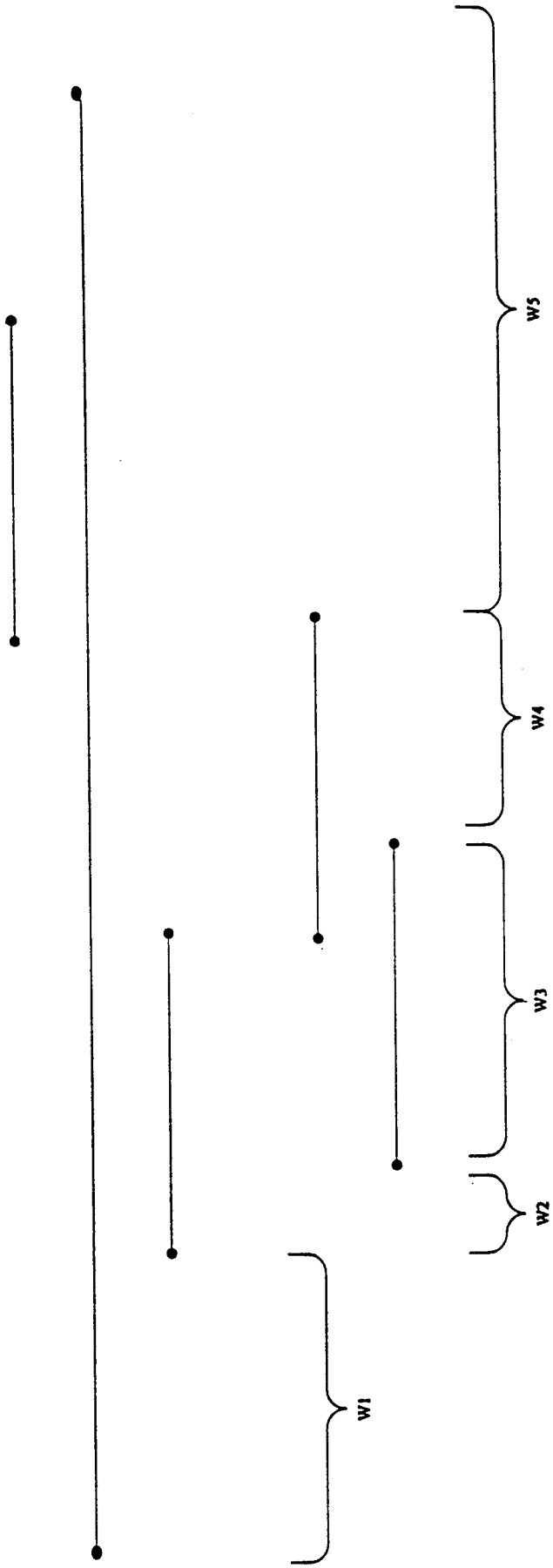

Figure 39

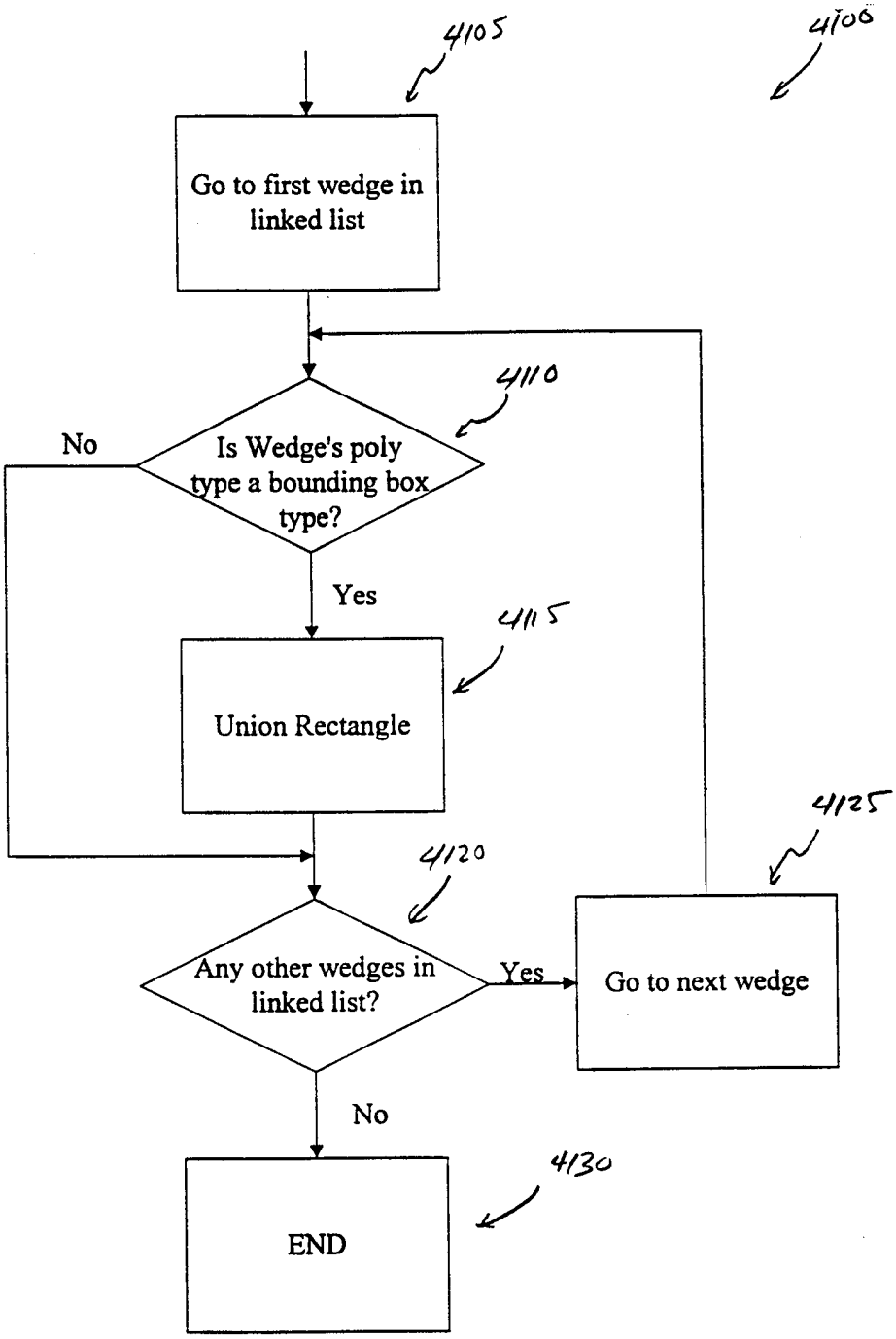
4105
4100
Go to first wedge in linked list
4110
No
Is Wedge's poly type a bounding box type?
Yes
4115
Union Rectangle
4125
4120
Any other wedges in linked list?
Yes
Go to next wedge
No
4130
END

Figure 41

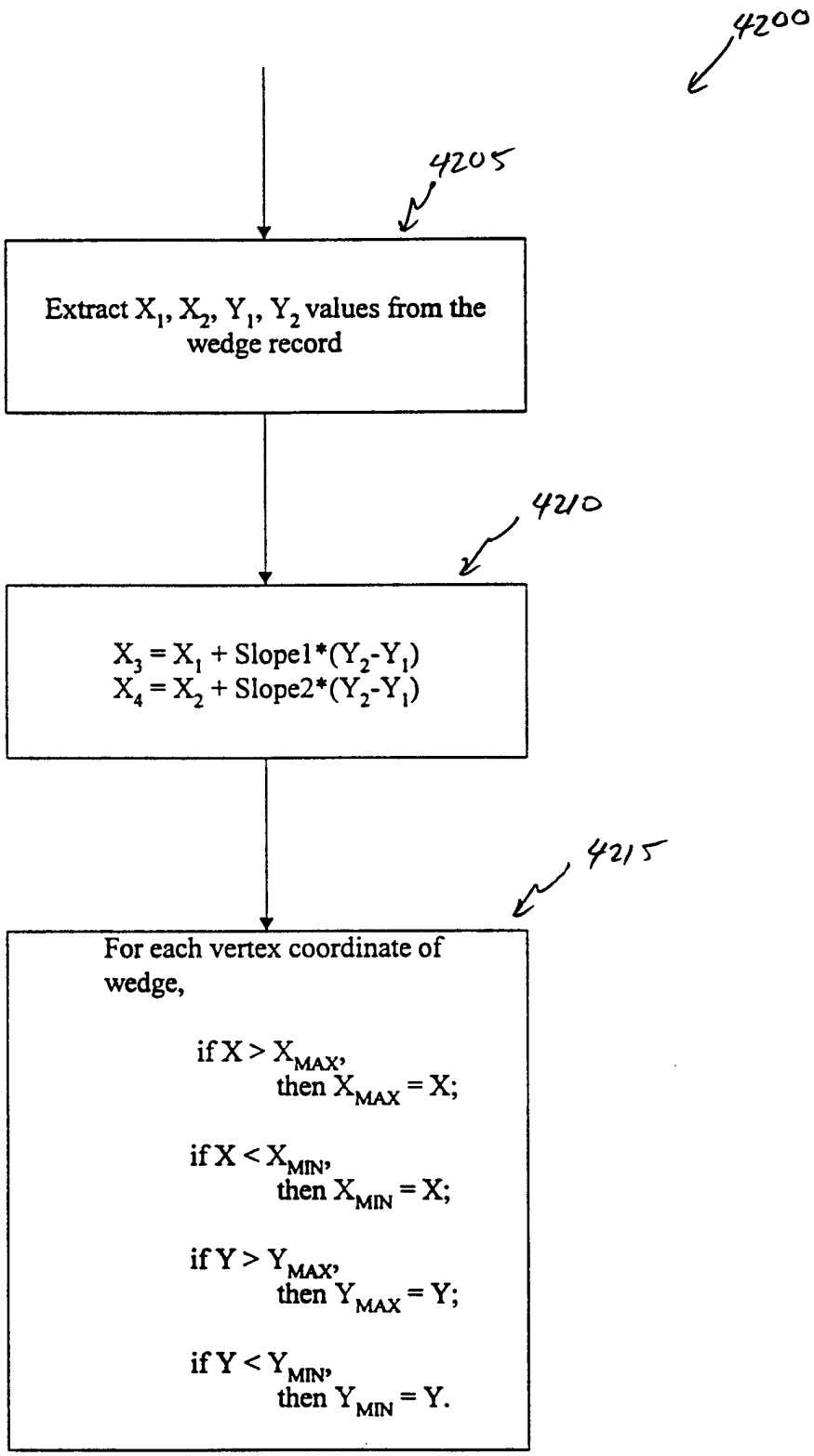
4200
4205
Extract X_1, X_2, Y_1, Y_2 values from the wedge record
4210
$X_3 = X_1 + Slope1*(Y_2-Y_1)$
$X_4 = X_2 + Slope2*(Y_2-Y_1)$
4215
For each vertex coordinate of wedge,
if $X > X_{MAX}$, then $X_{MAX} = X$;
if $X < X_{MIN}$, then $X_{MIN} = X$;
if $Y > Y_{MAX}$, then $Y_{MAX} = Y$;
if $Y < Y_{MIN}$, then $Y_{MIN} = Y$.

Figure 42

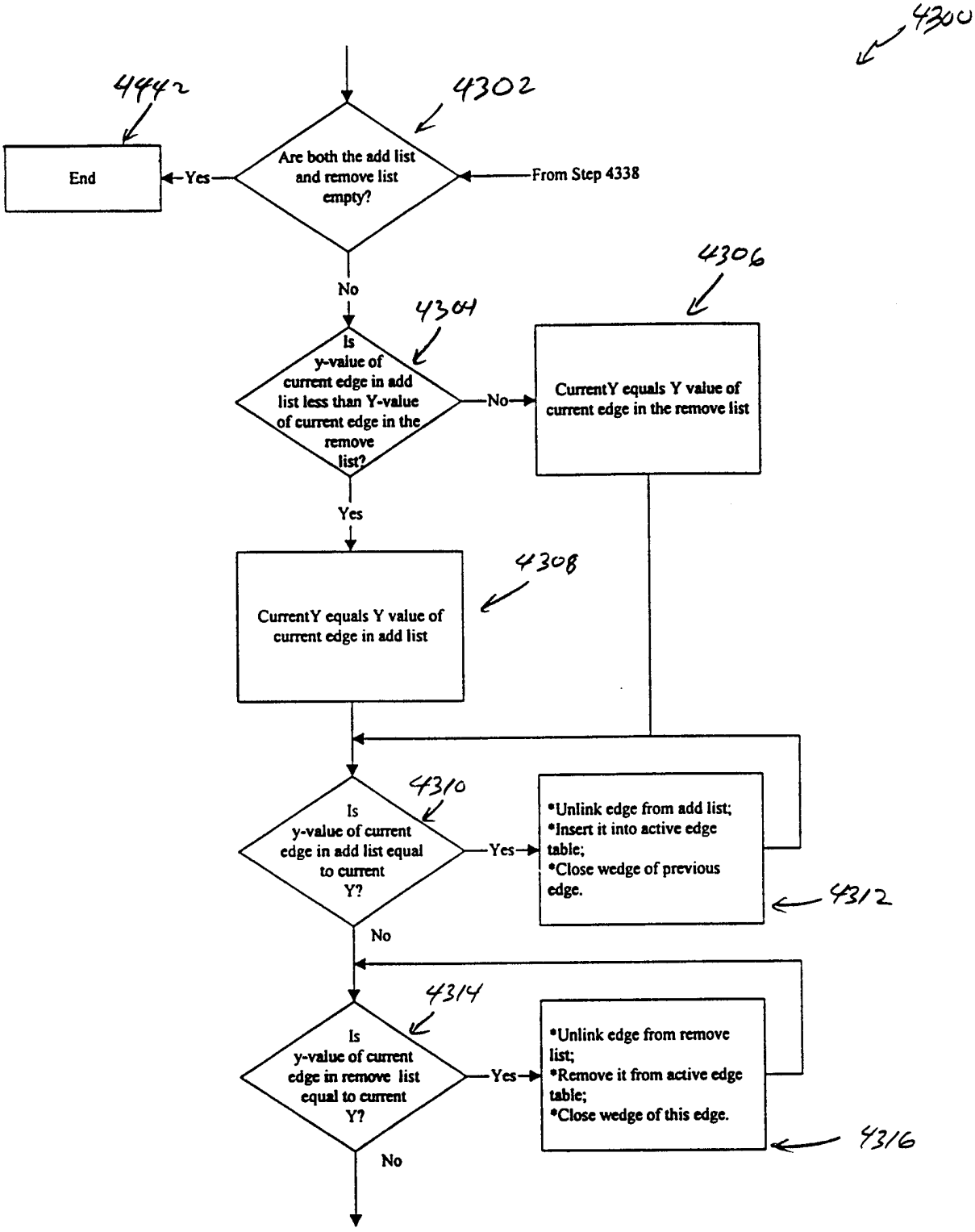
4300
4302
Are both the add list and remove list empty?
From Step 4338
Yes
4442
End
No
4304
Is y-value of current edge in add list less than Y-value of current edge in the remove list?
No
4306
CurrentY equals Y value of current edge in the remove list
Yes
4308
CurrentY equals Y value of current edge in add list
4310
Is y-value of current edge in add list equal to current Y?
Yes
*Unlink edge from add list;
*Insert it into active edge table;
*Close wedge of previous edge.
4312
No
4314
Is y-value of current edge in remove list equal to current Y?
Yes
*Unlink edge from remove list;
*Remove it from active edge table;
*Close wedge of this edge.
4316
No

Figure 43a

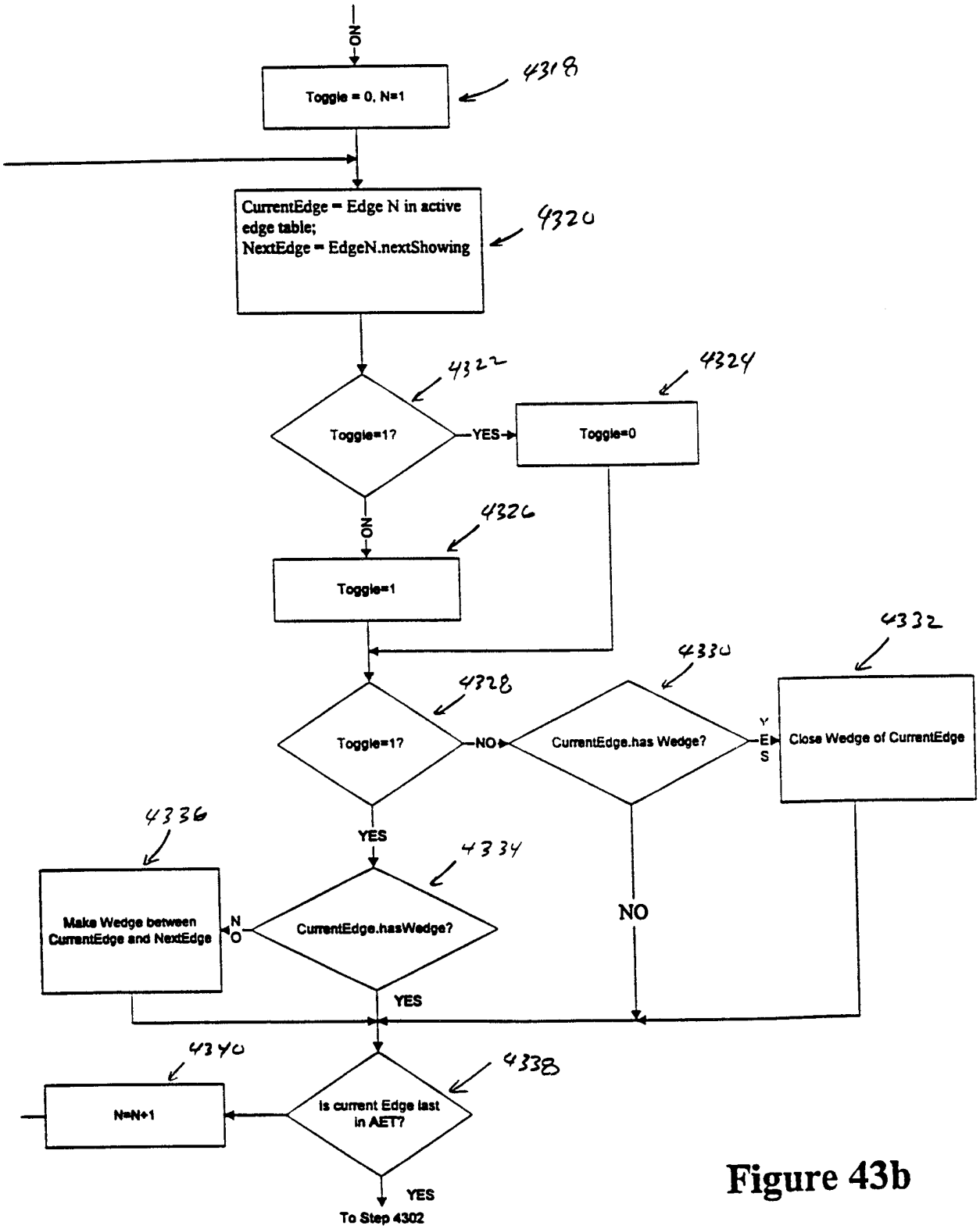
ON
Toggle = 0, N=1
4318
CurrentEdge = Edge N in active edge table;
NextEdge = EdgeN.nextShowing
4320
4322
Toggle=1?
YES
4324
Toggle=0
NO
4326
Toggle=1
4328
Toggle=1?
NO
4330
CurrentEdge.has Wedge?
YES
4332
Close Wedge of CurrentEdge
YES
4334
CurrentEdge.hasWedge?
NO
4336
Make Wedge between CurrentEdge and NextEdge
NO
NO
YES
4338
Is current Edge last in AET?
4340
N=N+1
YES
To Step 4302

Figure 43b

Y-Values	Add list
Y1	e1
Y1	e3
Y2	e2

Y-Values	Remove list
Y2	e1
Y2	e2
Y3	e3

Figure 45

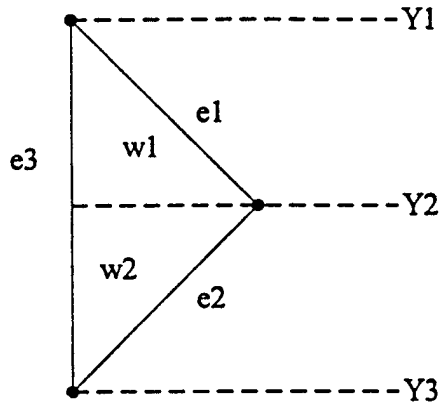

Figure 44

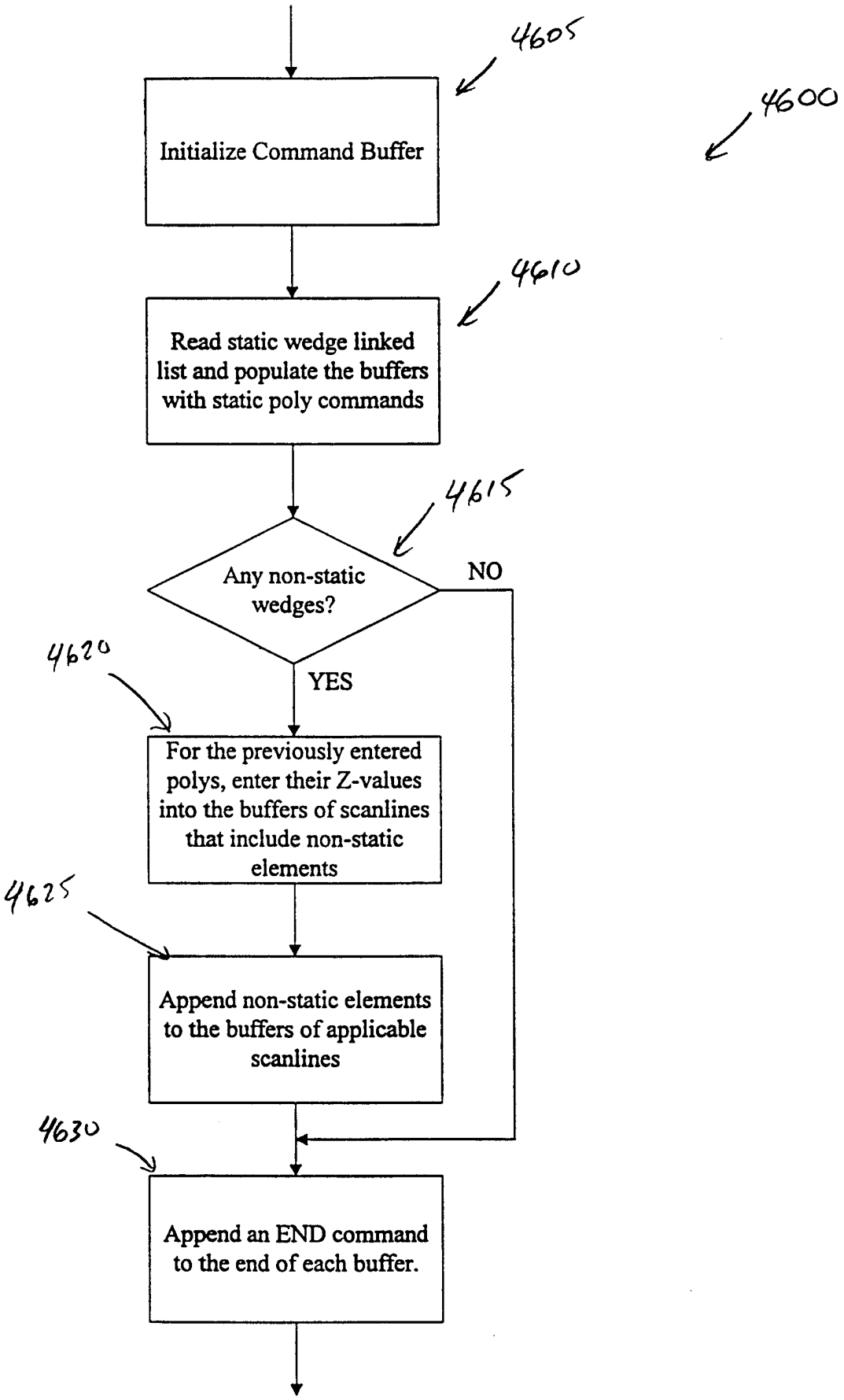
4605
4600
Initialize Command Buffer
4610
Read static wedge linked list and populate the buffers with static poly commands
4615
Any non-static wedges?
NO
YES
4620
For the previously entered polys, enter their Z-values into the buffers of scanlines that include non-static elements
4625
Append non-static elements to the buffers of applicable scanlines
4630
Append an END command to the end of each buffer.

Figure 46

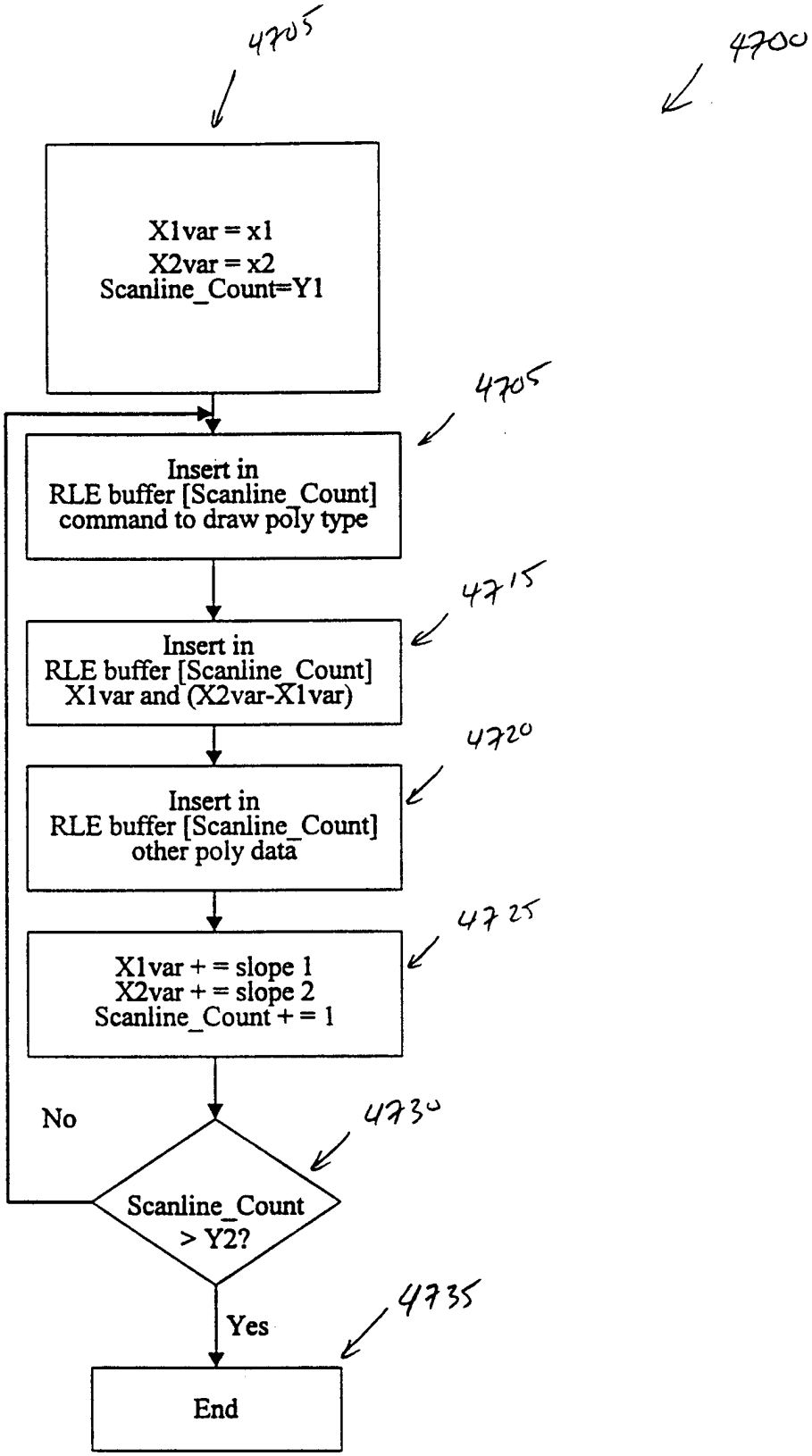
4705
4700
X1var = x1
X2var = x2
Scanline_Count=Y1
4705
Insert in
RLE buffer [Scanline_Count]
command to draw poly type
4715
Insert in
RLE buffer [Scanline_Count]
X1var and (X2var-X1var)
4720
Insert in
RLE buffer [Scanline_Count]
other poly data
4725
X1var + = slope 1
X2var + = slope 2
Scanline_Count + = 1
No
4730
Scanline_Count
> Y2?
Yes
4735
End

Figure 47

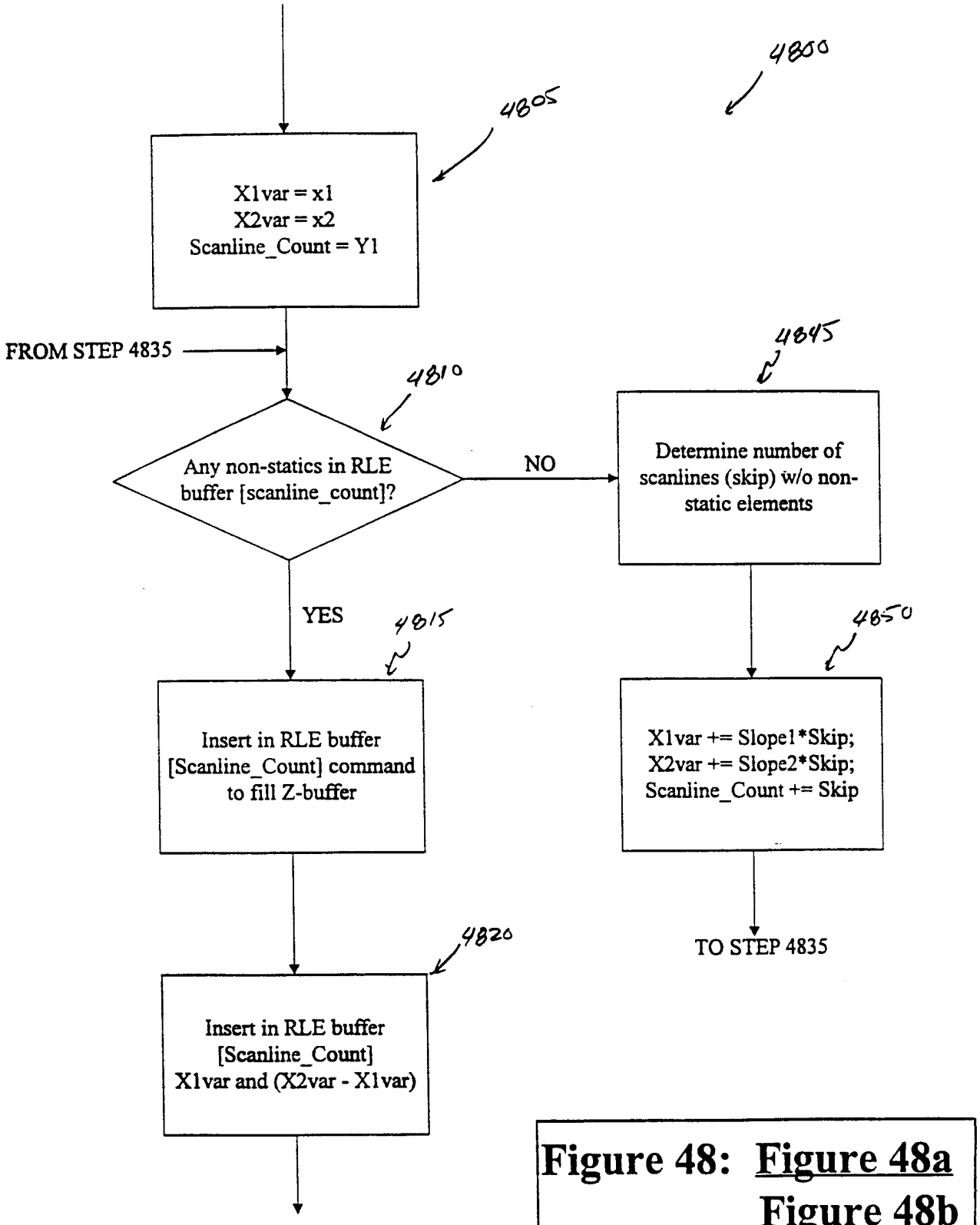

Figure 48: Figure 48a Figure 48b

Figure 48a

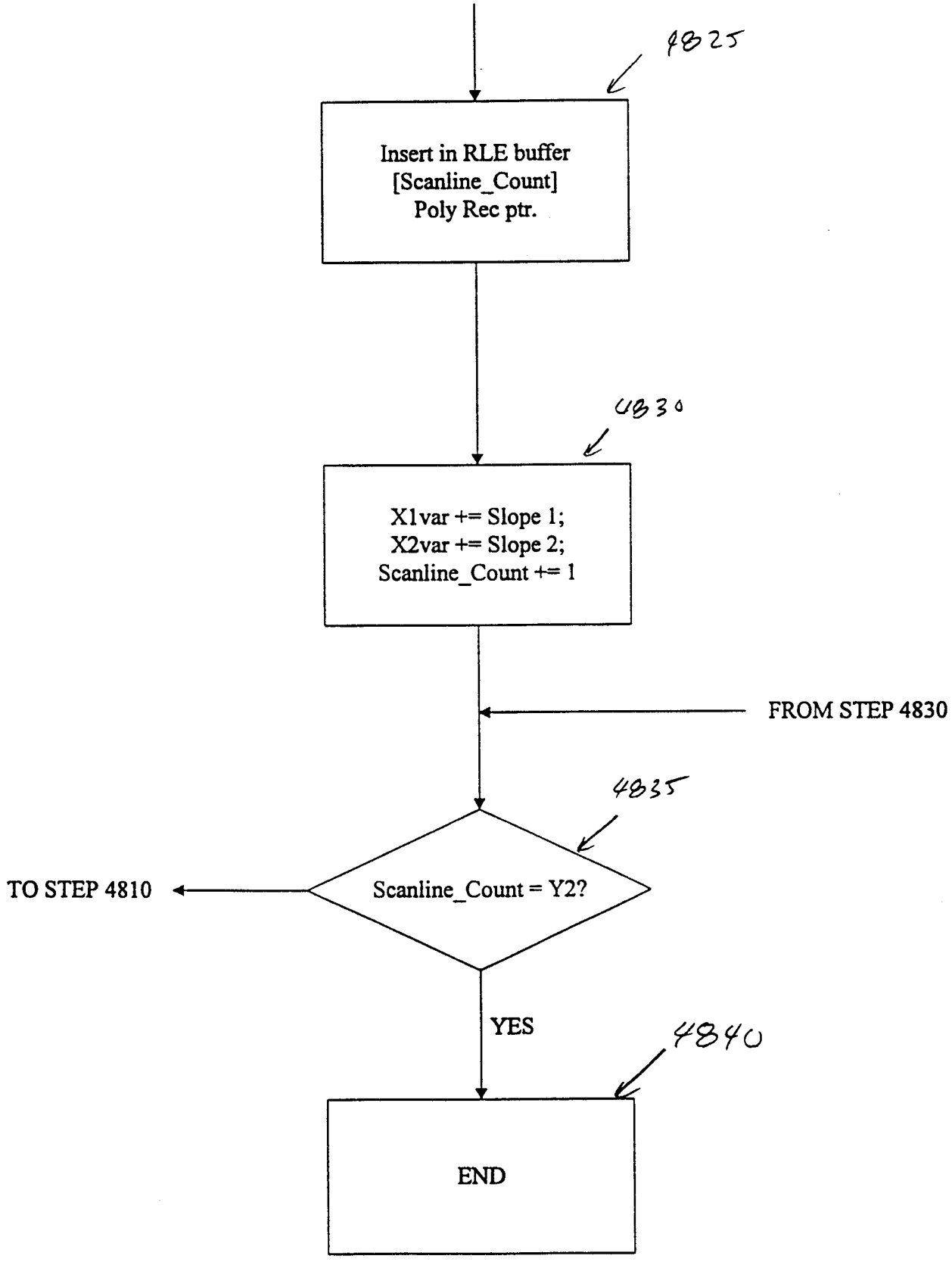
4825
Insert in RLE buffer
[Scanline_Count]
Poly Rec ptr.
4830
X1var += Slope 1;
X2var += Slope 2;
Scanline_Count += 1
FROM STEP 4830
4835
Scanline_Count = Y2?
TO STEP 4810
YES
4840
END

Figure 48b

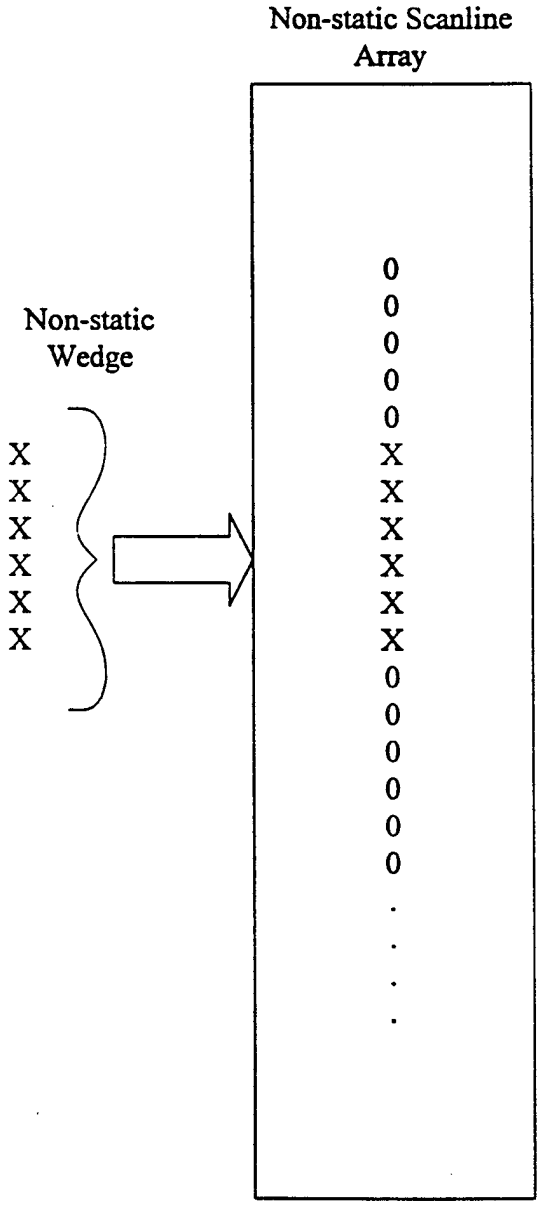
Non-static Scanline Array
Non-static Wedge
X
X
X
X
X
X
0
0
0
0
0
X
X
X
X
X
X
X
0
0
0
0
0
0
0
.
.
.
.

Figure 49

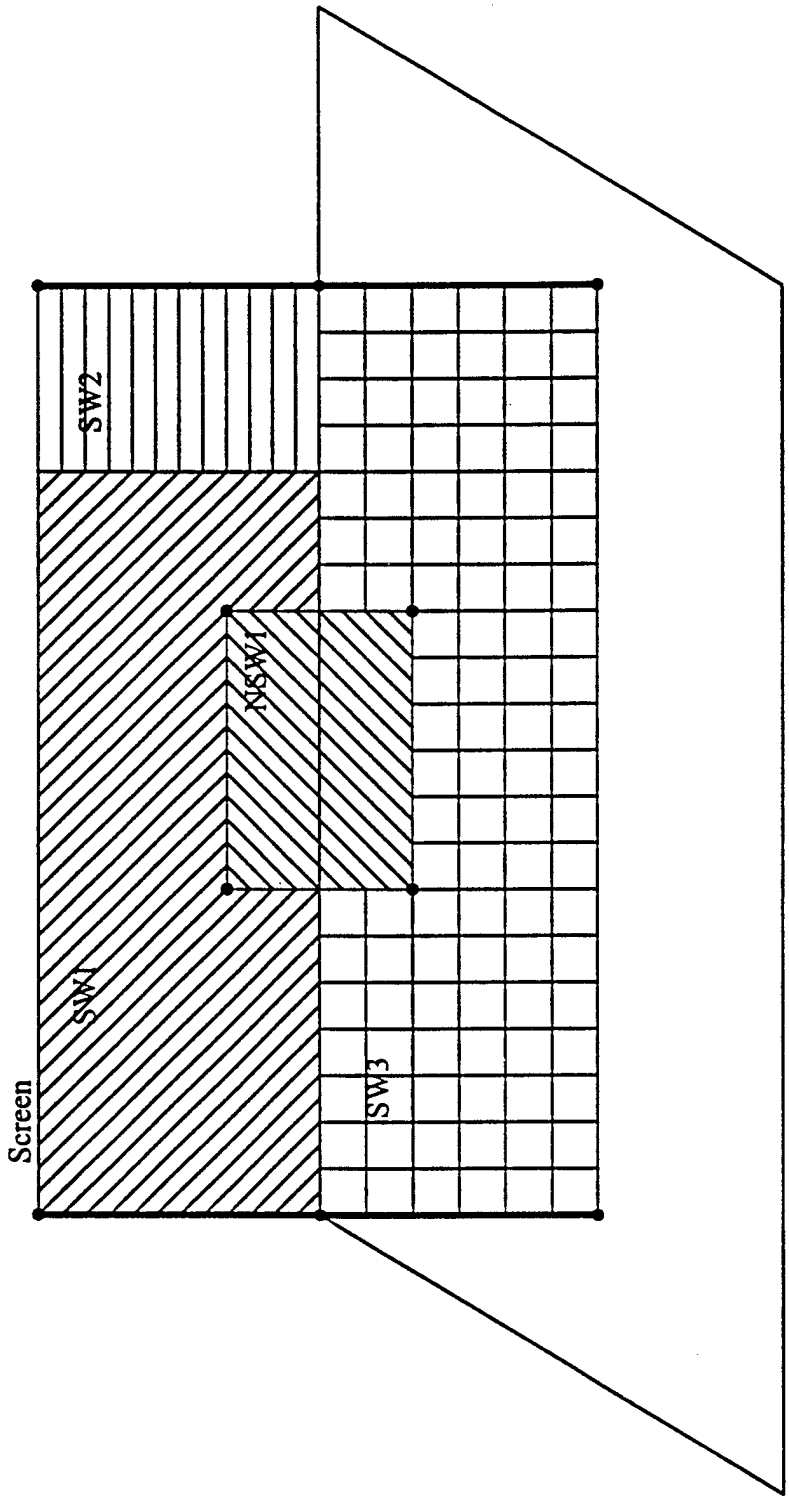

Figure 50

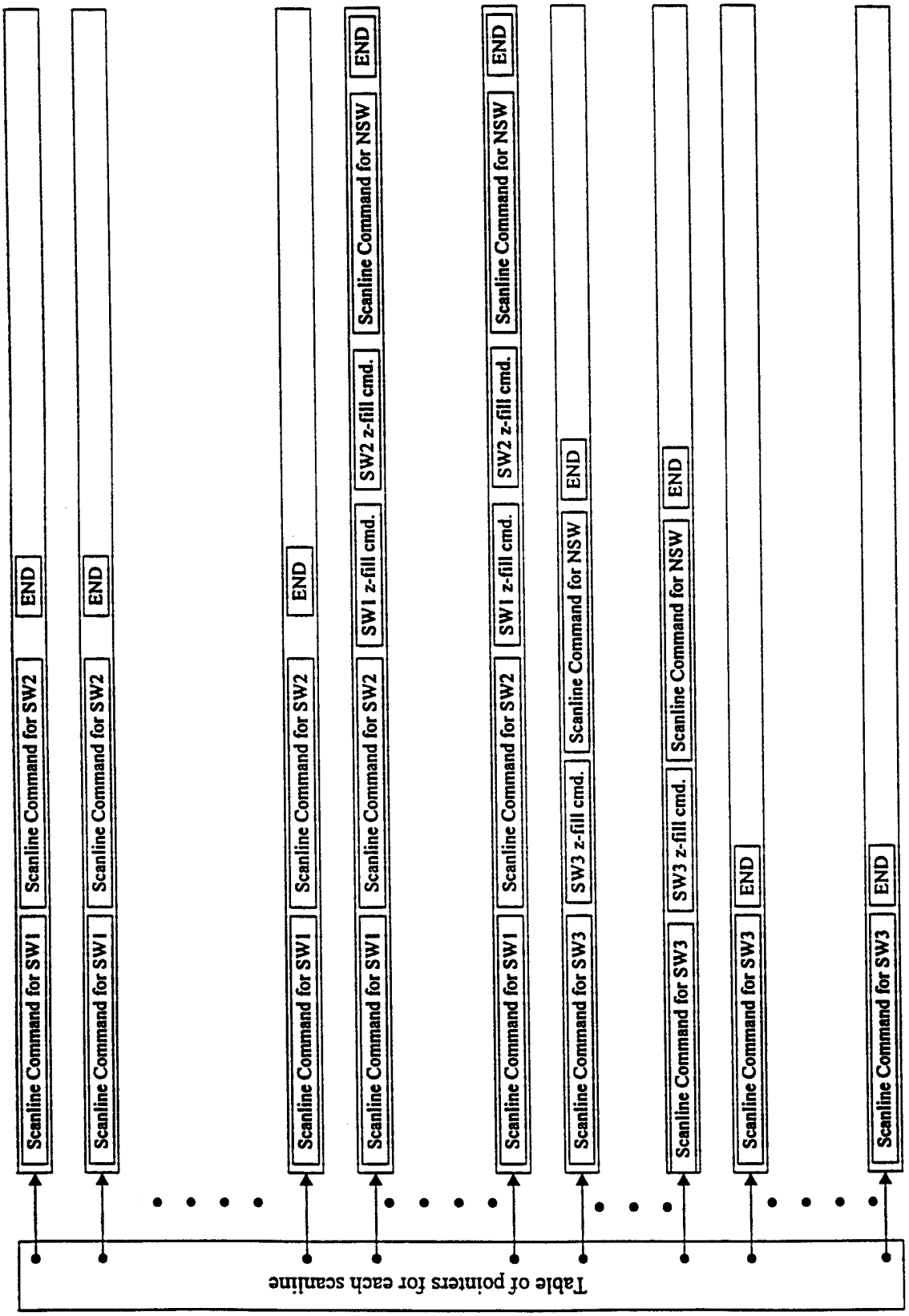
Table of pointers for each scanline
Scanline Command for SW1
Scanline Command for SW2
END
Scanline Command for SW1
Scanline Command for SW2
END
Scanline Command for SW1
Scanline Command for SW2
END
Scanline Command for SW1
Scanline Command for SW2
SW1 z-fill cmd.
SW2 z-fill cmd.
Scanline Command for NSW
END
Scanline Command for SW1
Scanline Command for SW2
SW1 z-fill cmd.
SW2 z-fill cmd.
Scanline Command for NSW
END
Scanline Command for SW3
SW3 z-fill cmd.
Scanline Command for NSW
END
Scanline Command for SW3
SW3 z-fill cmd.
Scanline Command for NSW
END
Scanline Command for SW3
END
Scanline Command for SW3
END

Figure 51

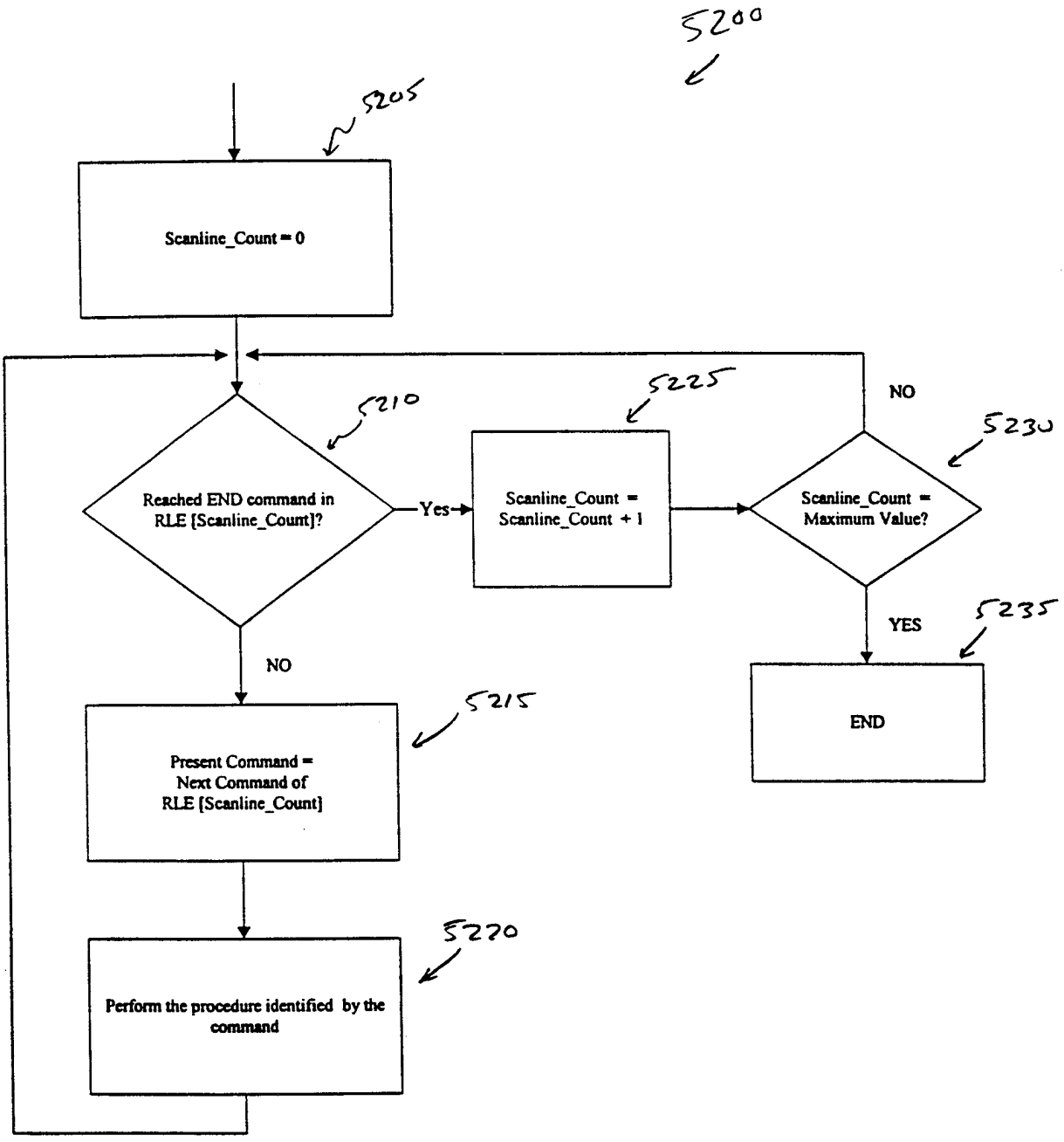
5200
5205
Scanline_Count = 0
5210
Reached END command in RLE [Scanline_Count]?
Yes
5225
Scanline_Count = Scanline_Count + 1
5230
Scanline_Count = Maximum Value?
NO
YES
5235
END
NO
5215
Present Command = Next Command of RLE [Scanline_Count]
5220
Perform the procedure identified by the command

Figure 52

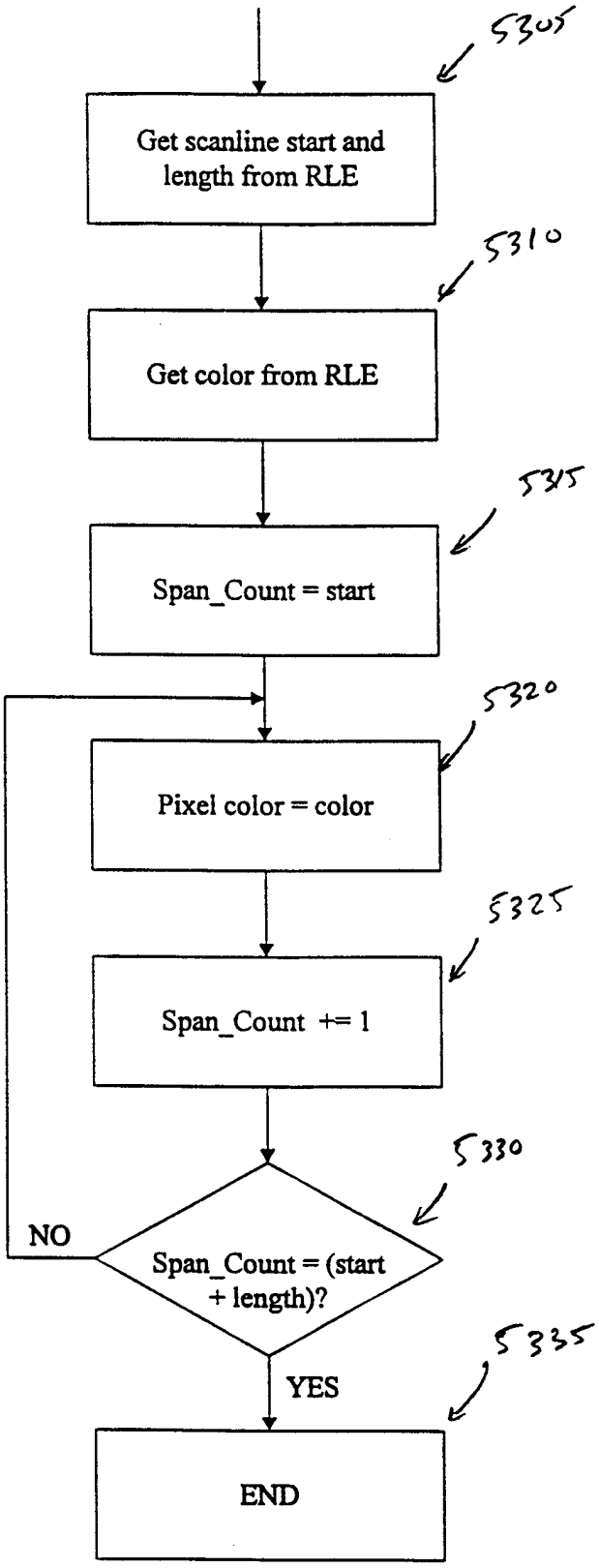
5300
5305
Get scanline start and length from RLE
5310
Get color from RLE
5315
Span_Count = start
5320
Pixel color = color
5325
Span_Count += 1
5330
Span_Count = (start + length)?
NO
YES
5335
END

Figure 53

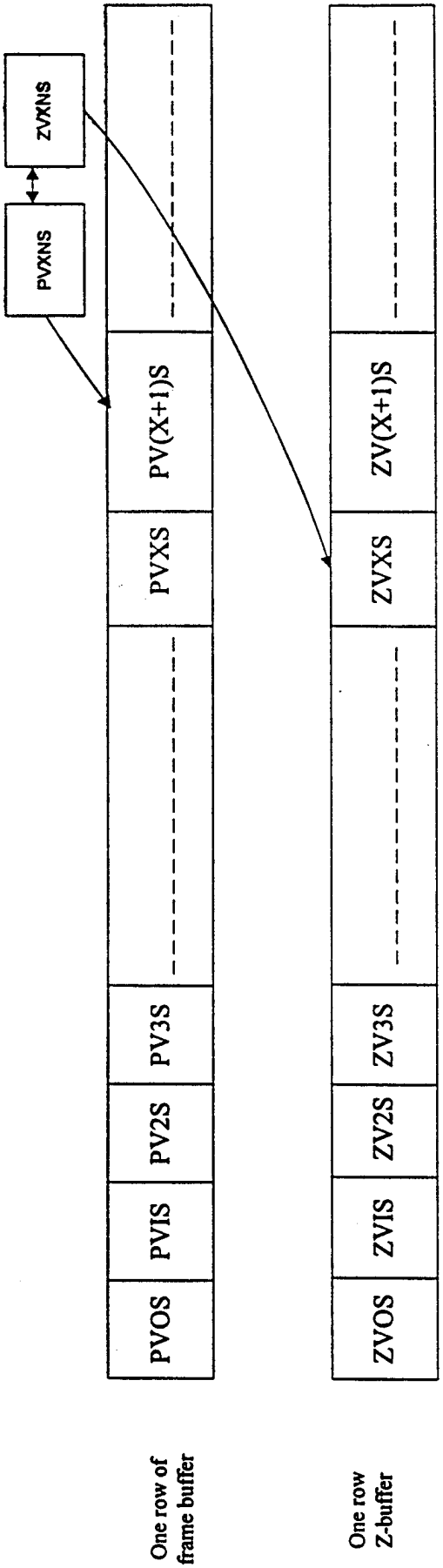
PVXNS
ZVXNS
PVOS
PVIS
PV2S
PV3S
PVXS
PV(X+1)S
ZVOS
ZVIS
ZV2S
ZV3S
ZVXS
ZV(X+1)S
One row of frame buffer
One row Z-buffer

Figure 54

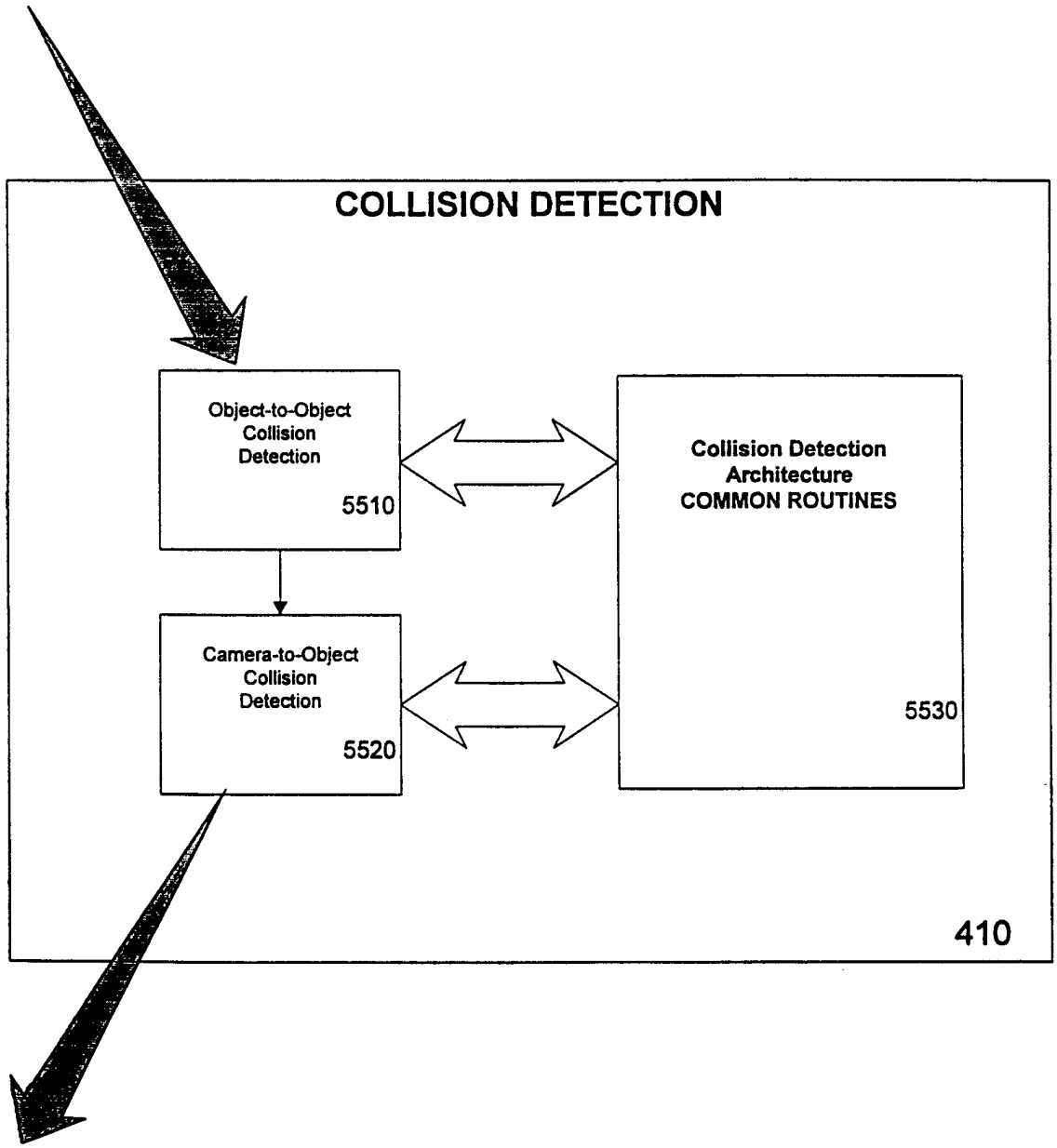
COLLISION DETECTION
Object-to-Object Collision Detection
5510
Camera-to-Object Collision Detection
5520
Collision Detection Architecture COMMON ROUTINES
5530
410

Figure 55

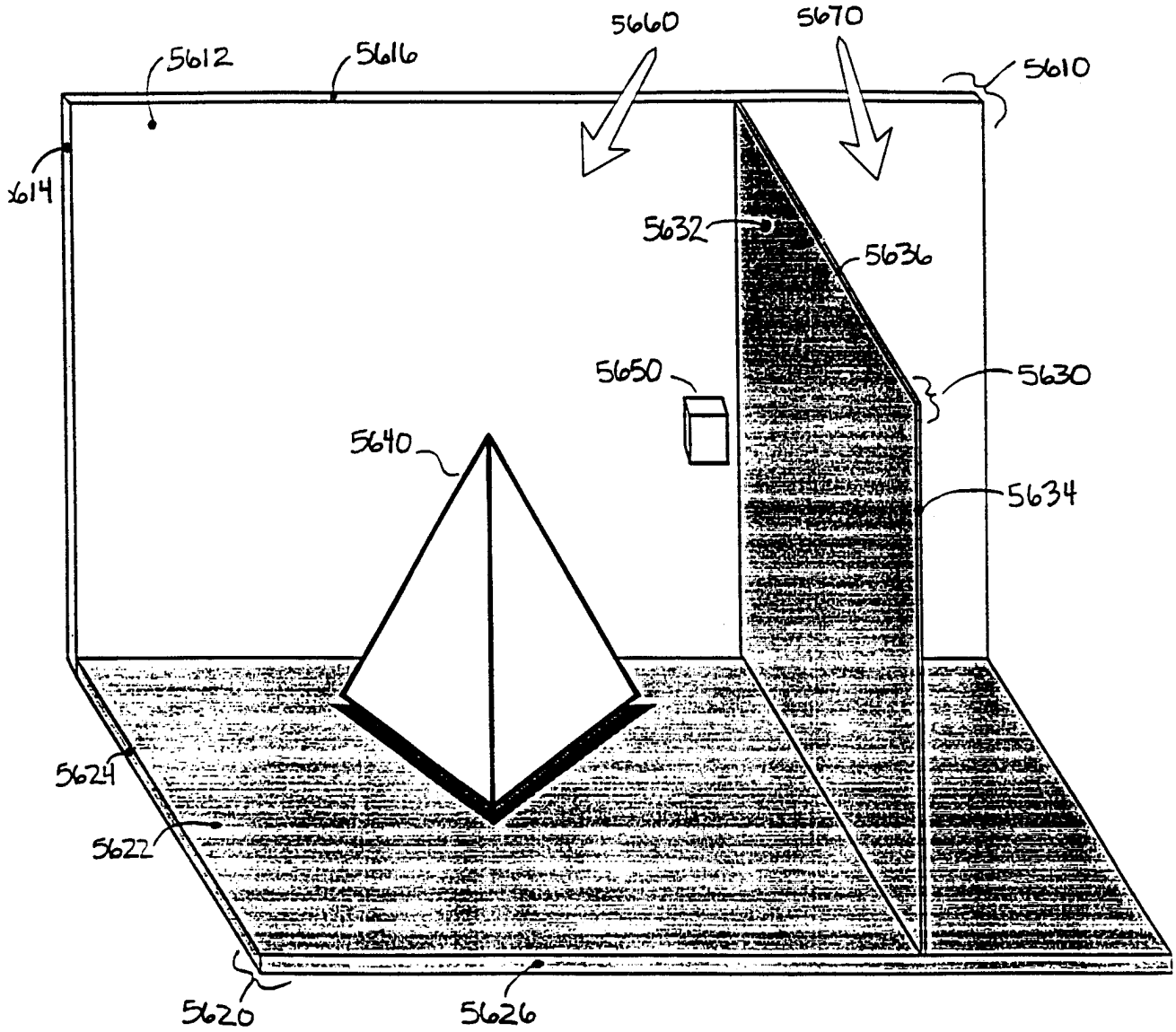
5660
5670
5612
5616
5610
5614
5632
5636
5650
5630
5640
5634
5624
5622
5620
5626

Figure 56

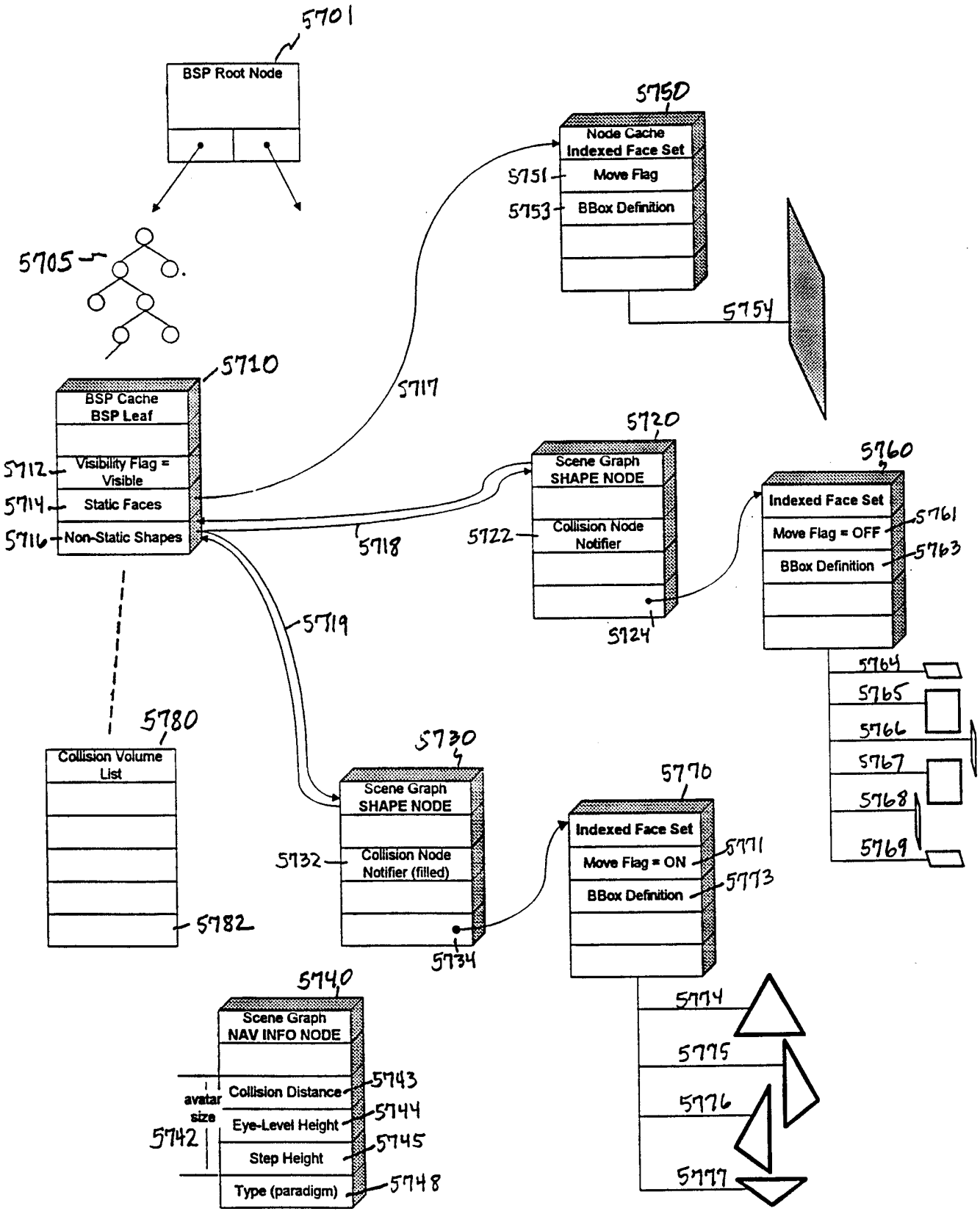
5701
BSP Root Node
5705
5710
BSP Cache
BSP Leaf
5712
Visibility Flag = Visible
5714
Static Faces
5716
Non-Static Shapes
5717
5718
5719
5750
Node Cache
Indexed Face Set
5751
Move Flag
5753
BBox Definition
5754
5720
Scene Graph
SHAPE NODE
5722
Collision Node Notifier
5724
5760
Indexed Face Set
Move Flag = OFF
5761
BBox Definition
5763
5764
5765
5766
5767
5768
5769
5780
Collision Volume List
5782
5730
Scene Graph
SHAPE NODE
5732
Collision Node Notifier (filled)
5734
5770
Indexed Face Set
Move Flag = ON
5771
BBox Definition
5773
5774
5775
5776
5777
5740
Scene Graph
NAV INFO NODE
avatar size
5742
Collision Distance
5743
Eye-Level Height
5744
Step Height
5745
Type (paradigm)
5748

Figure 57

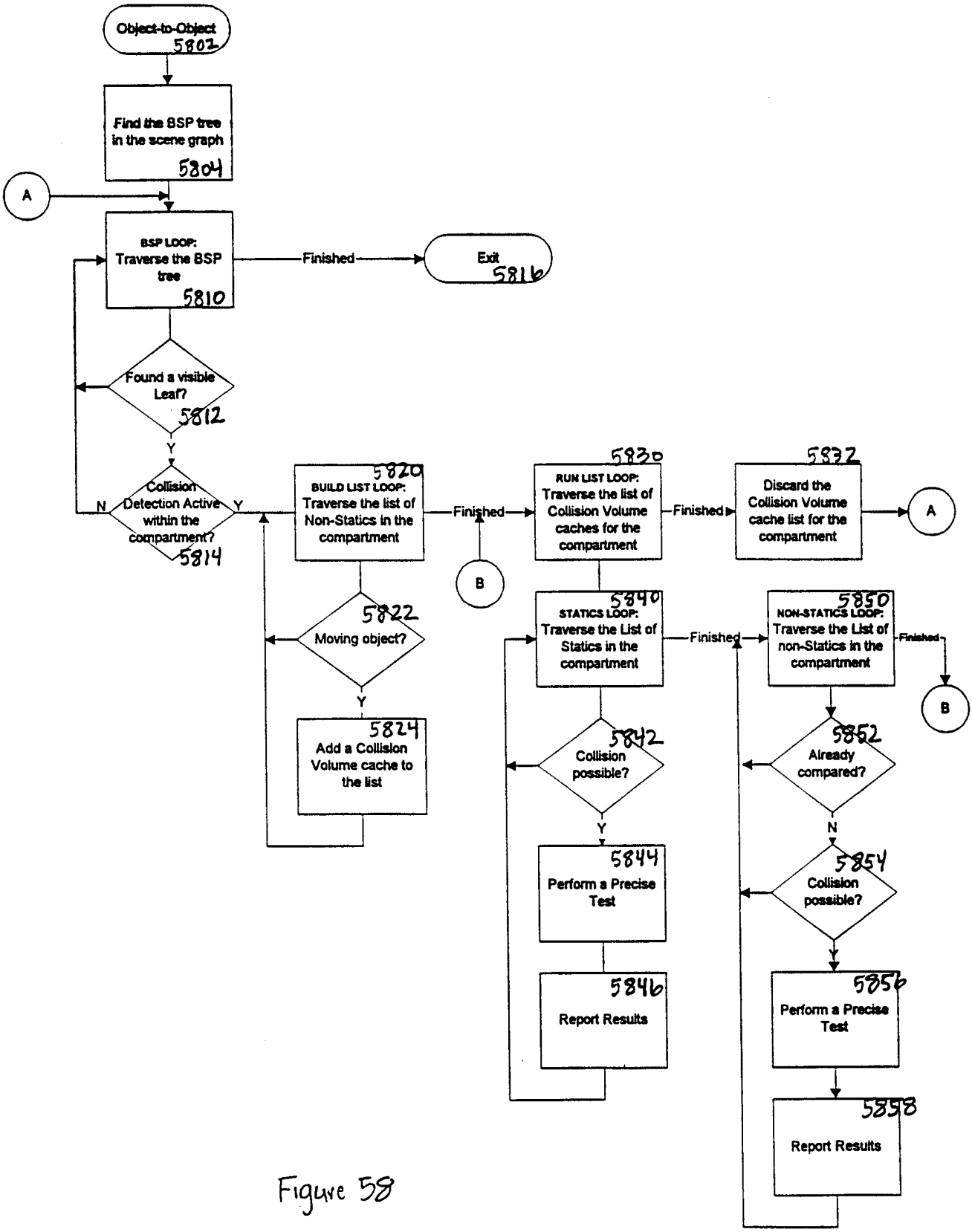
Object-to-Object
5802
Find the BSP tree in the scene graph
5804
A
BSP LOOP:
Traverse the BSP tree
5810
Finished
Exit
5816
Found a visible Leaf?
5812
Y
N
Collision Detection Active within the compartment?
5814
Y
5820
BUILD LIST LOOP:
Traverse the list of Non-Statics in the compartment
Finished
B
5822
Moving object?
Y
5824
Add a Collision Volume cache to the list
5830
RUN LIST LOOP:
Traverse the list of Collision Volume caches for the compartment
Finished
5832
Discard the Collision Volume cache list for the compartment
A
5840
STATICS LOOP:
Traverse the List of Statics in the compartment
Finished
5850
NON-STATICS LOOP:
Traverse the List of non-Statics in the compartment
Finished
B
5842
Collision possible?
Y
5844
Perform a Precise Test
5846
Report Results
5852
Already compared?
N
5854
Collision possible?
Y
5856
Perform a Precise Test
5858
Report Results

Figure 58

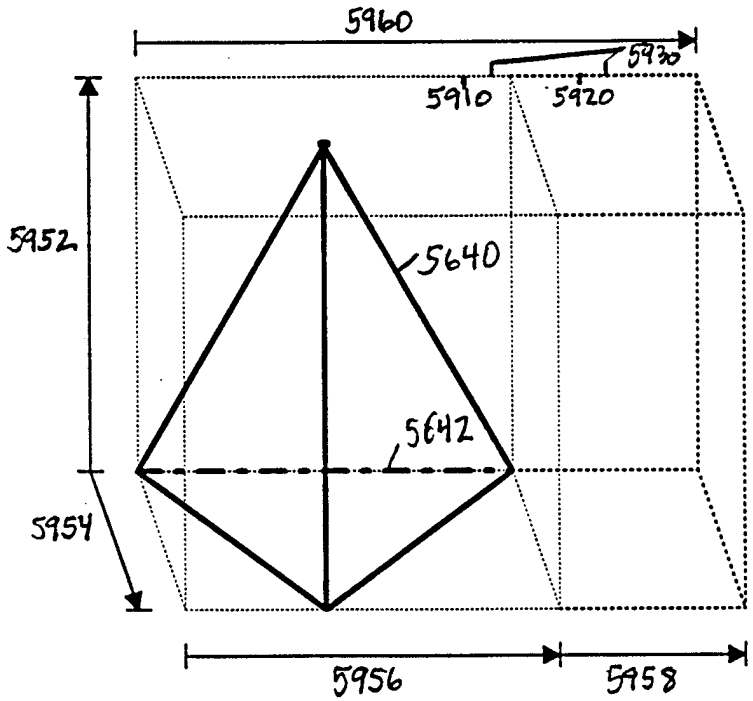

Figure 59

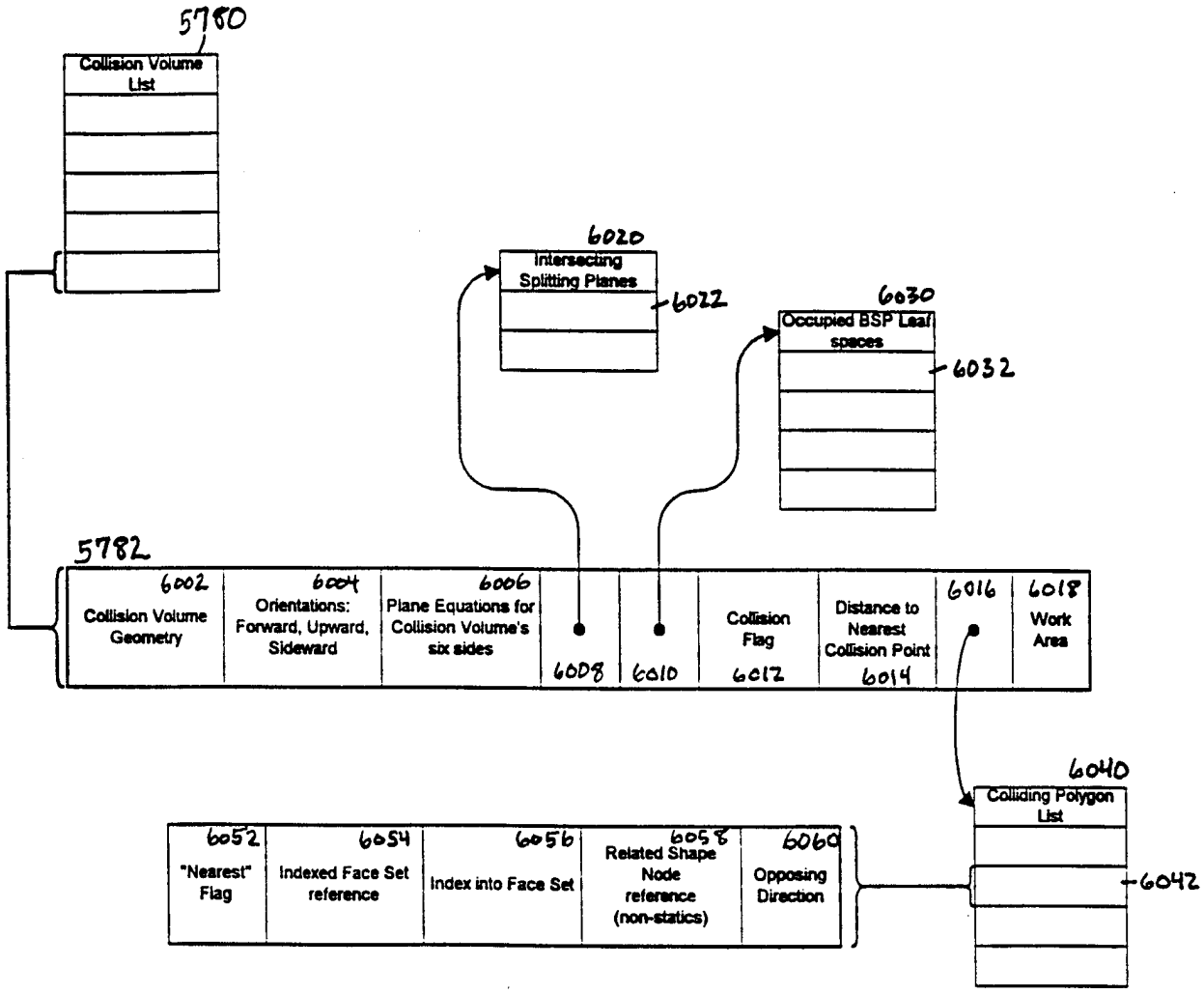
5780
Collision Volume List
6020
Intersecting Splitting Planes
6022
6030
Occupied BSP Leaf spaces
6032
5782
6002
Collision Volume Geometry
6004
Orientations: Forward, Upward, Sideward
6006
Plane Equations for Collision Volume's six sides
6008
6010
Collision Flag
6012
Distance to Nearest Collision Point
6014
6016
6018
Work Area
6040
Colliding Polygon List
6042
6052
"Nearest" Flag
6054
Indexed Face Set reference
6056
Index into Face Set
6058
Related Shape Node reference (non-statics)
6060
Opposing Direction

Figure 60

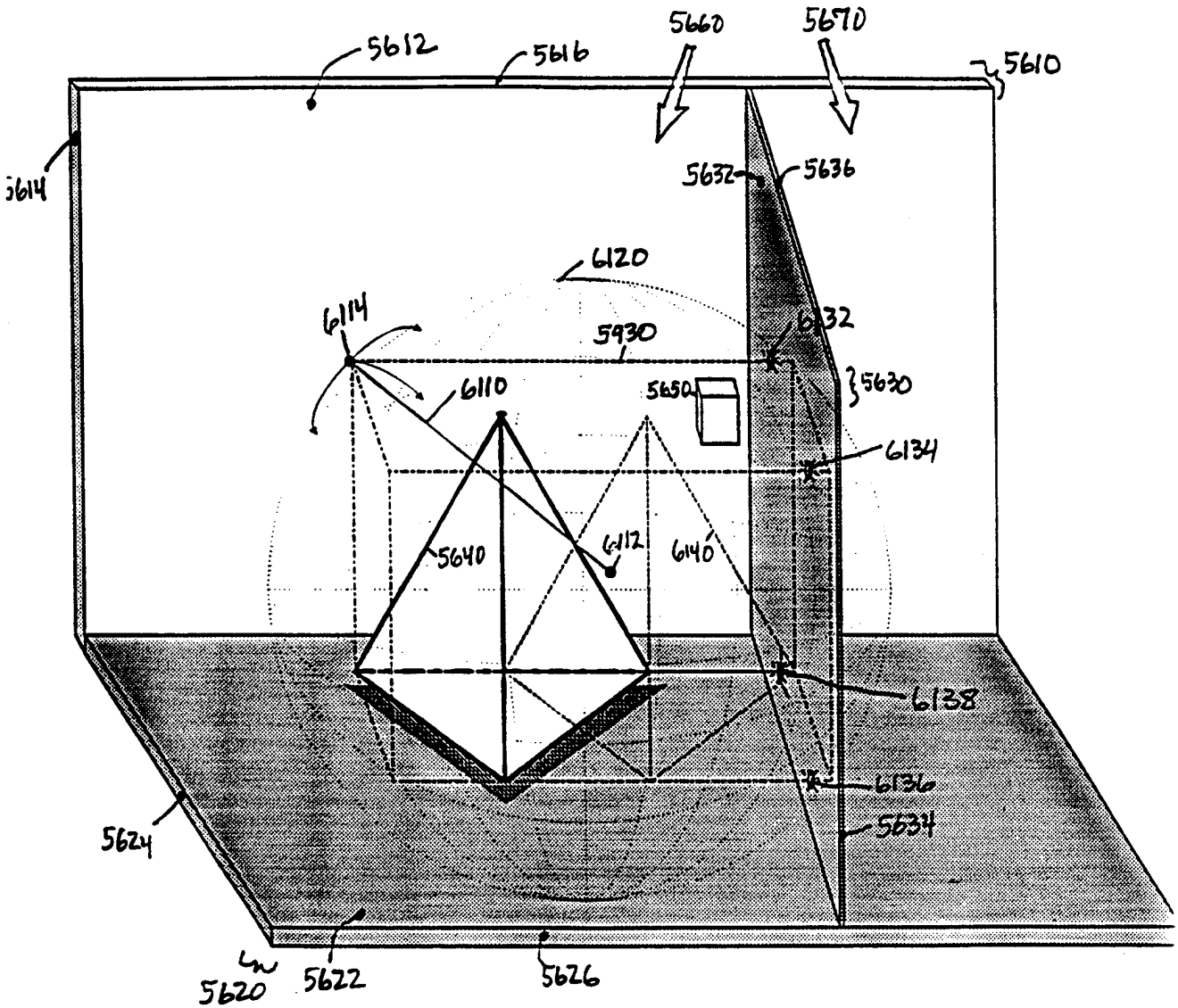
5612
5616
5660
5670
5610
5614
5632
5636
6120
6114
5930
6132
6110
5650
5630
6134
5640
6112
6140
6138
6136
5634
5624
5620
5622
5626

Figure 61

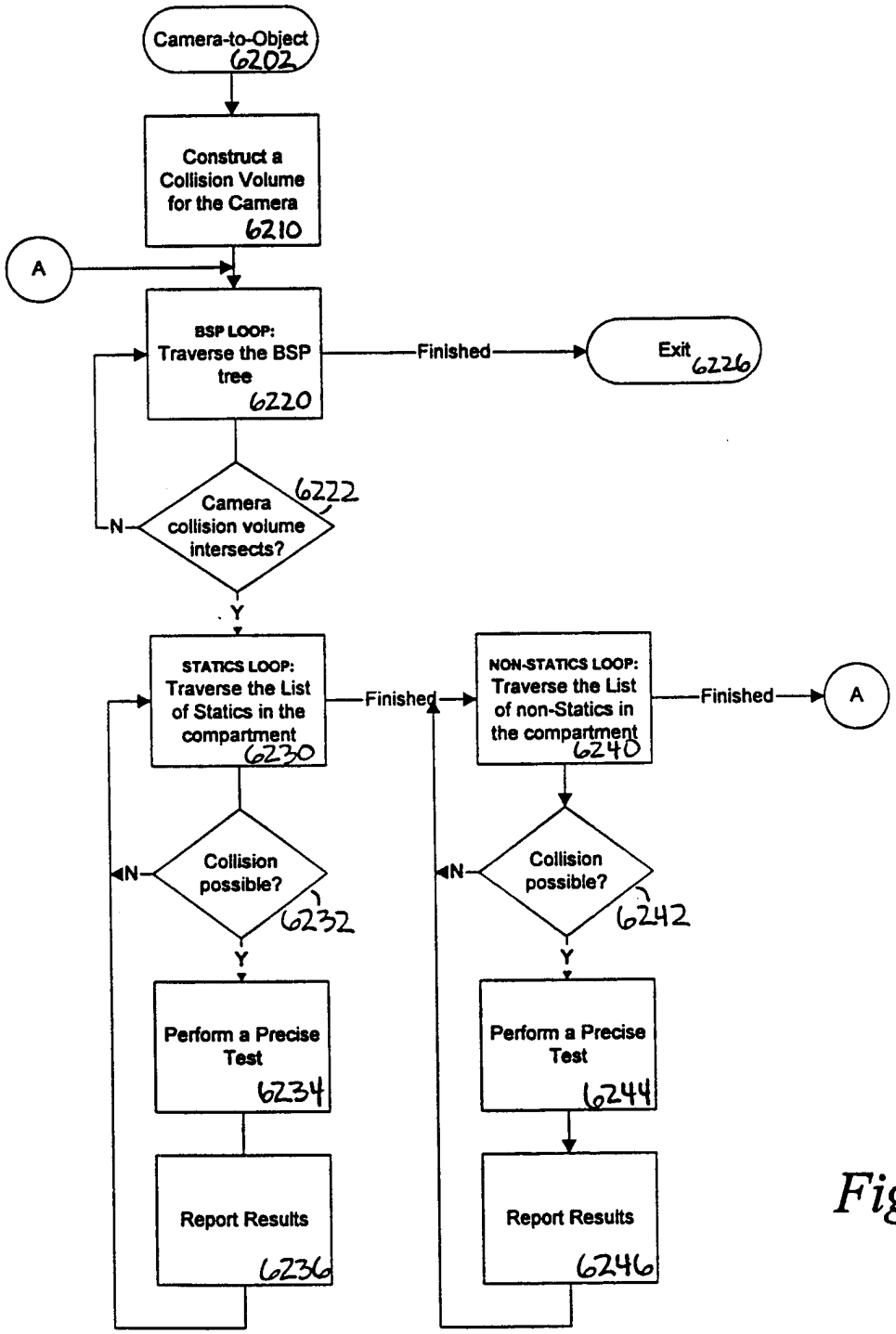
Camera-to-Object
6202
Construct a Collision Volume for the Camera
6210
A
BSP LOOP:
Traverse the BSP tree
6220
Finished
Exit
6226
Camera collision volume intersects?
6222
N
Y
STATICS LOOP:
Traverse the List of Statics in the compartment
6230
Finished
NON-STATICS LOOP:
Traverse the List of non-Statics in the compartment
6240
Finished
A
Collision possible?
6232
N
Y
Collision possible?
6242
N
Y
Perform a Precise Test
6234
Perform a Precise Test
6244
Report Results
6236
Report Results
6246

Figure 62

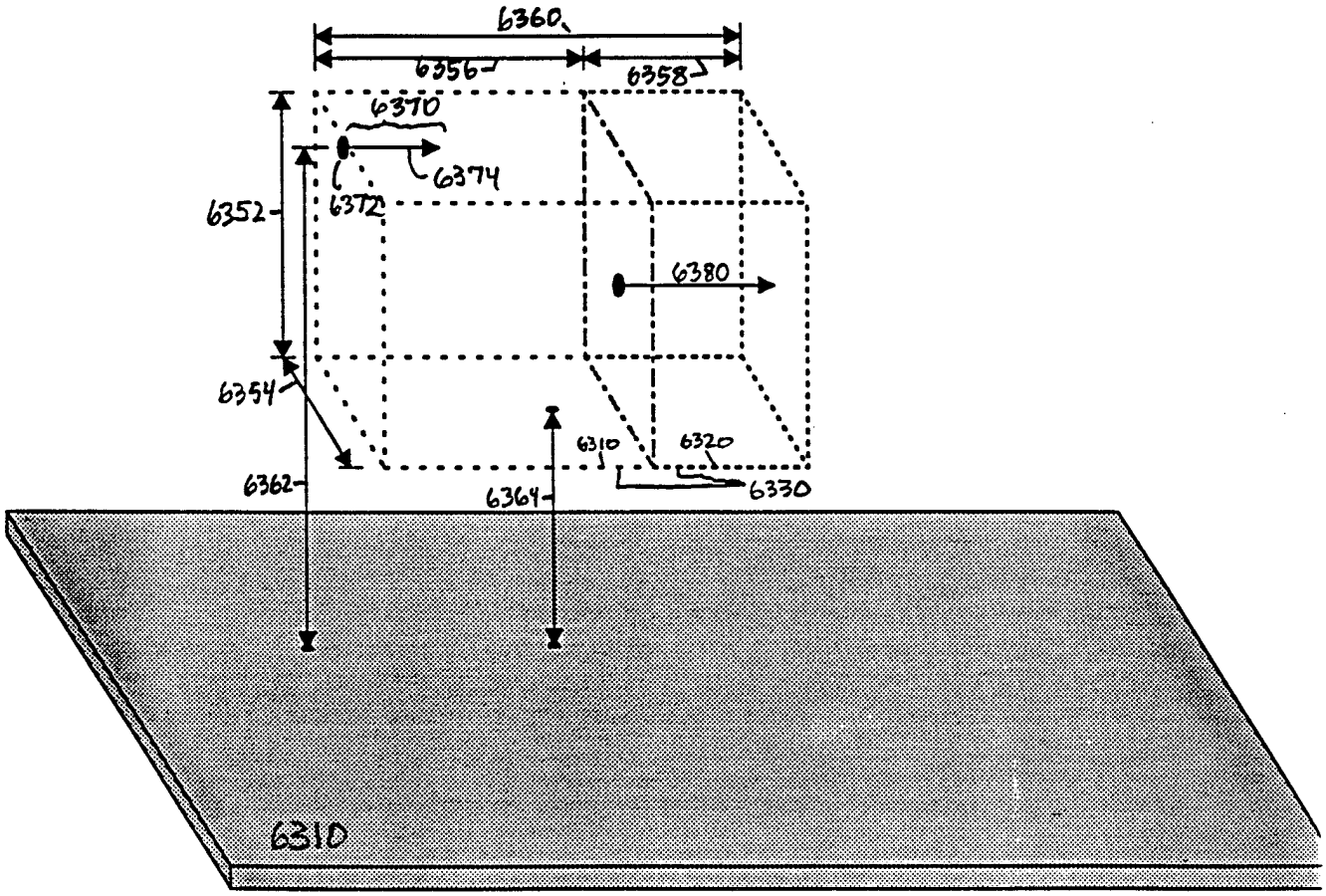
6360
6356
6358
6370
6374
6372
6352
6380
6354
6310
6320
6362
6364
6330
6310

Figure 63

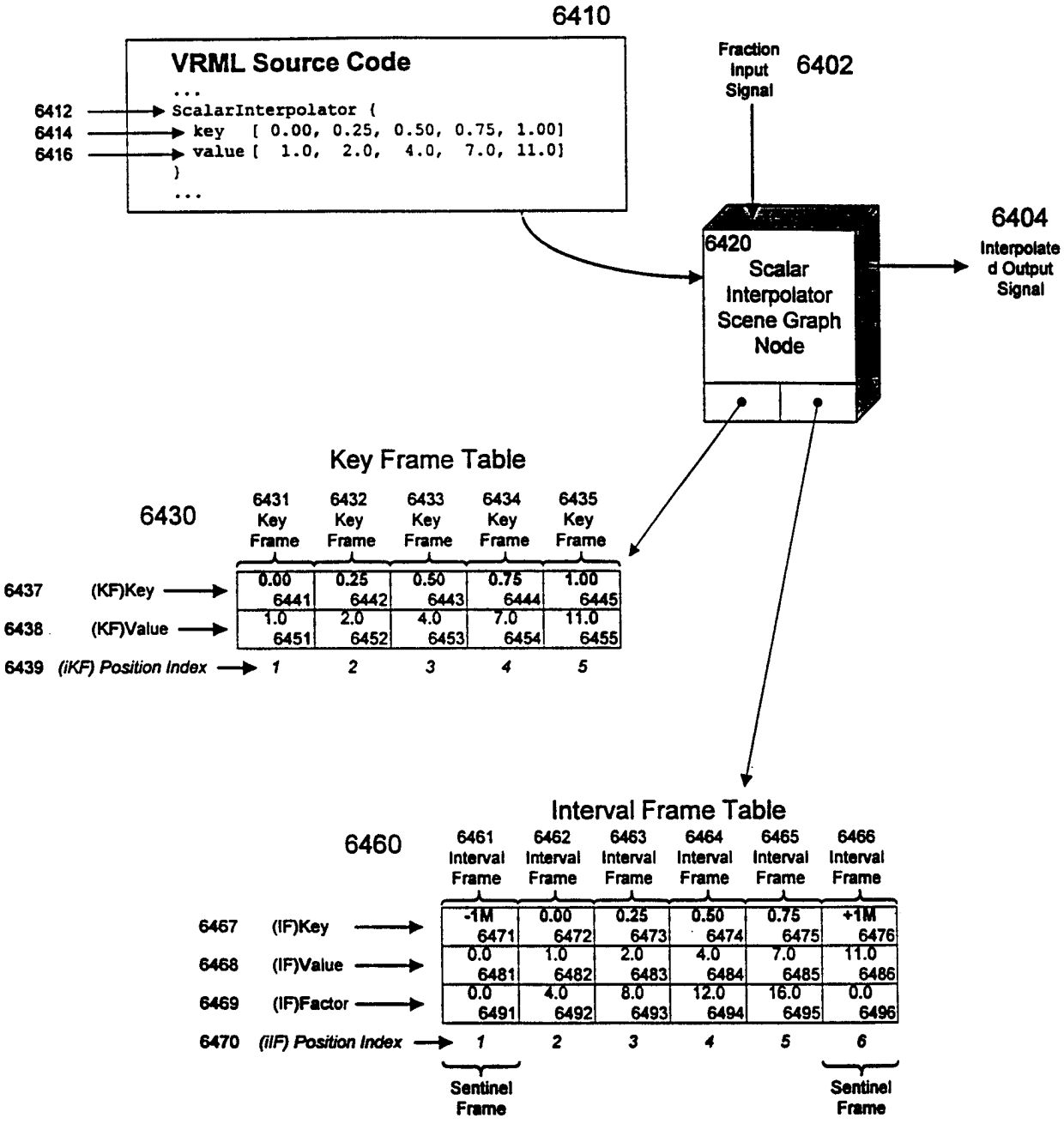
6410
VRML Source Code
...
6412 ScalarInterpolator {
6414 key [0.00, 0.25, 0.50, 0.75, 1.00]
6416 value [1.0, 2.0, 4.0, 7.0, 11.0]
}
...
Fraction Input Signal
6402
6420 Scalar Interpolator Scene Graph Node
6404 Interpolated Output Signal
Key Frame Table
6430
6431 Key Frame
6432 Key Frame
6433 Key Frame
6434 Key Frame
6435 Key Frame
6437 (KF)Key
0.00 6441
0.25 6442
0.50 6443
0.75 6444
1.00 6445
6438 (KF)Value
1.0 6451
2.0 6452
4.0 6453
7.0 6454
11.0 6455
6439 (iKF) Position Index
1 2 3 4 5
Interval Frame Table
6460
6461 Interval Frame
6462 Interval Frame
6463 Interval Frame
6464 Interval Frame
6465 Interval Frame
6466 Interval Frame
6467 (IF)Key
-1M 6471
0.00 6472
0.25 6473
0.50 6474
0.75 6475
+1M 6476
6468 (IF)Value
0.0 6481
1.0 6482
2.0 6483
4.0 6484
7.0 6485
11.0 6486
6469 (IF)Factor
0.0 6491
4.0 6492
8.0 6493
12.0 6494
16.0 6495
0.0 6496
6470 (iIF) Position Index
1 2 3 4 5 6
Sentinel Frame
Sentinel Frame

Figure 64

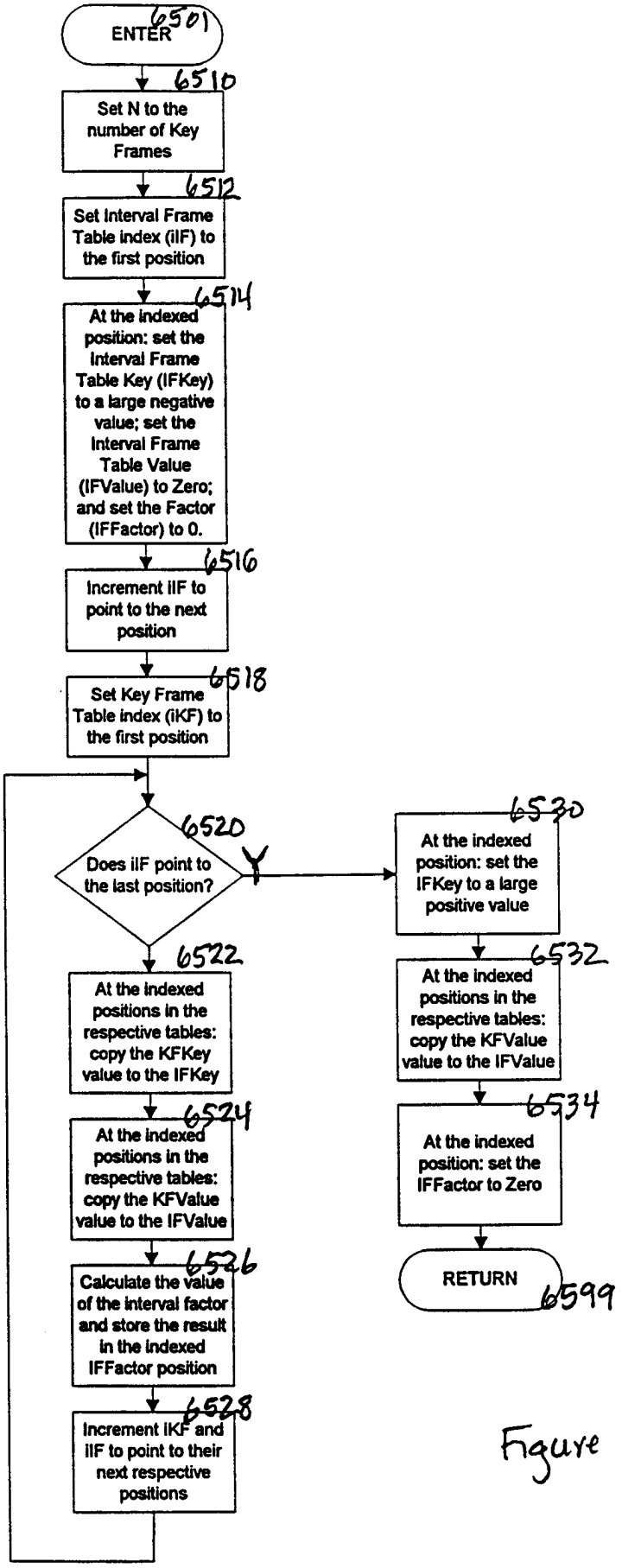
ENTER
6501
6510
Set N to the number of Key Frames
6512
Set Interval Frame Table index (iIF) to the first position
6514
At the indexed position: set the Interval Frame Table Key (IFKey) to a large negative value; set the Interval Frame Table Value (IFValue) to Zero; and set the Factor (IFFactor) to 0.
6516
Increment IIF to point to the next position
6518
Set Key Frame Table index (iKF) to the first position
6520
Does iIF point to the last position?
Y
6530
At the indexed position: set the IFKey to a large positive value
6532
At the indexed positions in the respective tables: copy the KFValue value to the IFValue
6534
At the indexed position: set the IFFactor to Zero
RETURN
6599
6522
At the indexed positions in the respective tables: copy the KFKey value to the IFKey
6524
At the indexed positions in the respective tables: copy the KFValue value to the IFValue
6526
Calculate the value of the interval factor and store the result in the indexed IFFactor position
6528
Increment IKF and IIF to point to their next respective positions

Figure 65

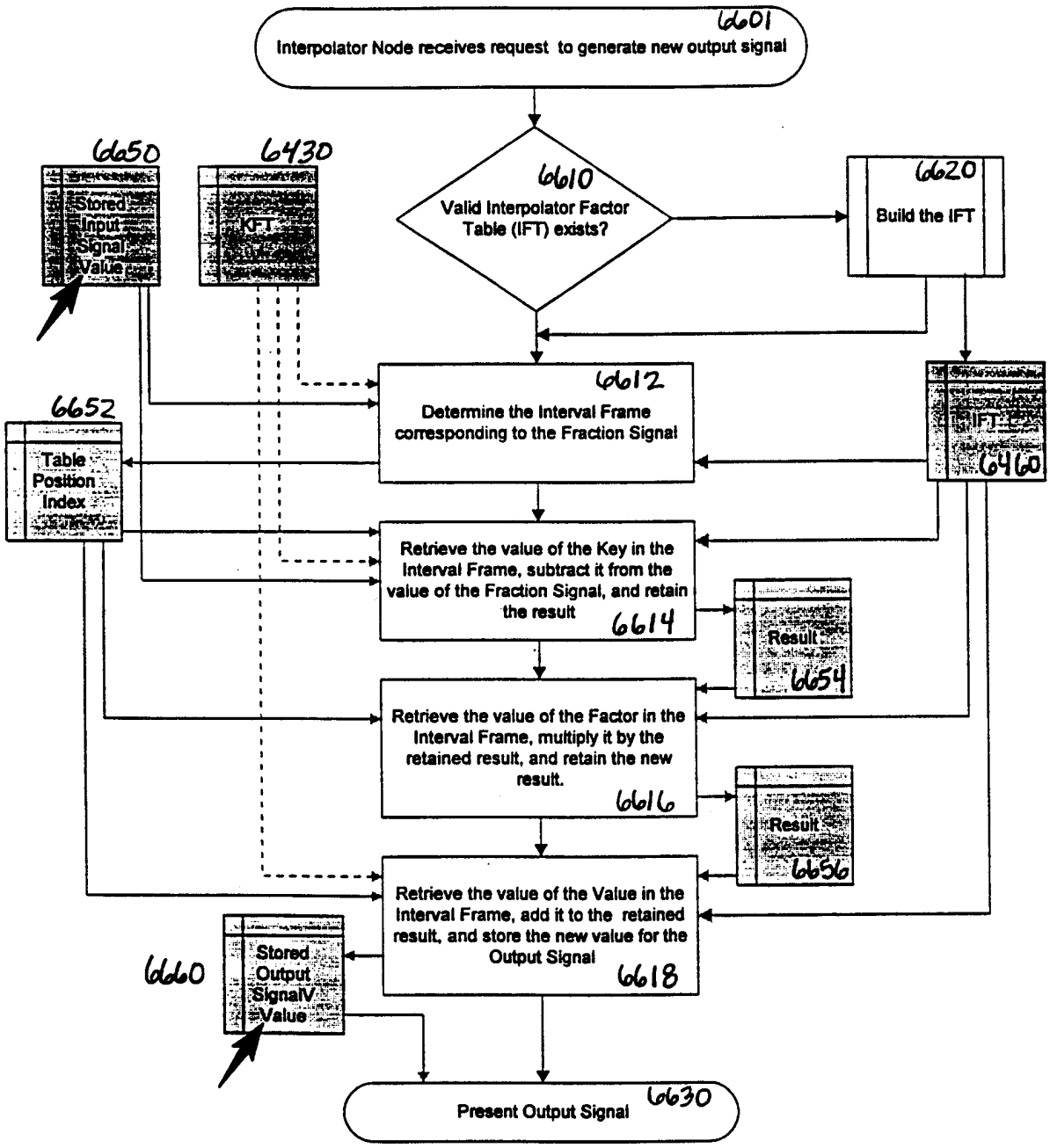
6601
Interpolator Node receives request to generate new output signal
6650
Stored Input Signal Value
6430
IFT
6610
Valid Interpolator Factor Table (IFT) exists?
6620
Build the IFT
6612
Determine the Interval Frame corresponding to the Fraction Signal
6652
Table Position Index
IFT
6460
Retrieve the value of the Key in the Interval Frame, subtract it from the value of the Fraction Signal, and retain the result
6614
Result
6654
Retrieve the value of the Factor in the Interval Frame, multiply it by the retained result, and retain the new result.
6616
Result
6656
Retrieve the value of the Value in the Interval Frame, add it to the retained result, and store the new value for the Output Signal
6618
6660
Stored Output SignalV Value
Present Output Signal
6630

Figure 66

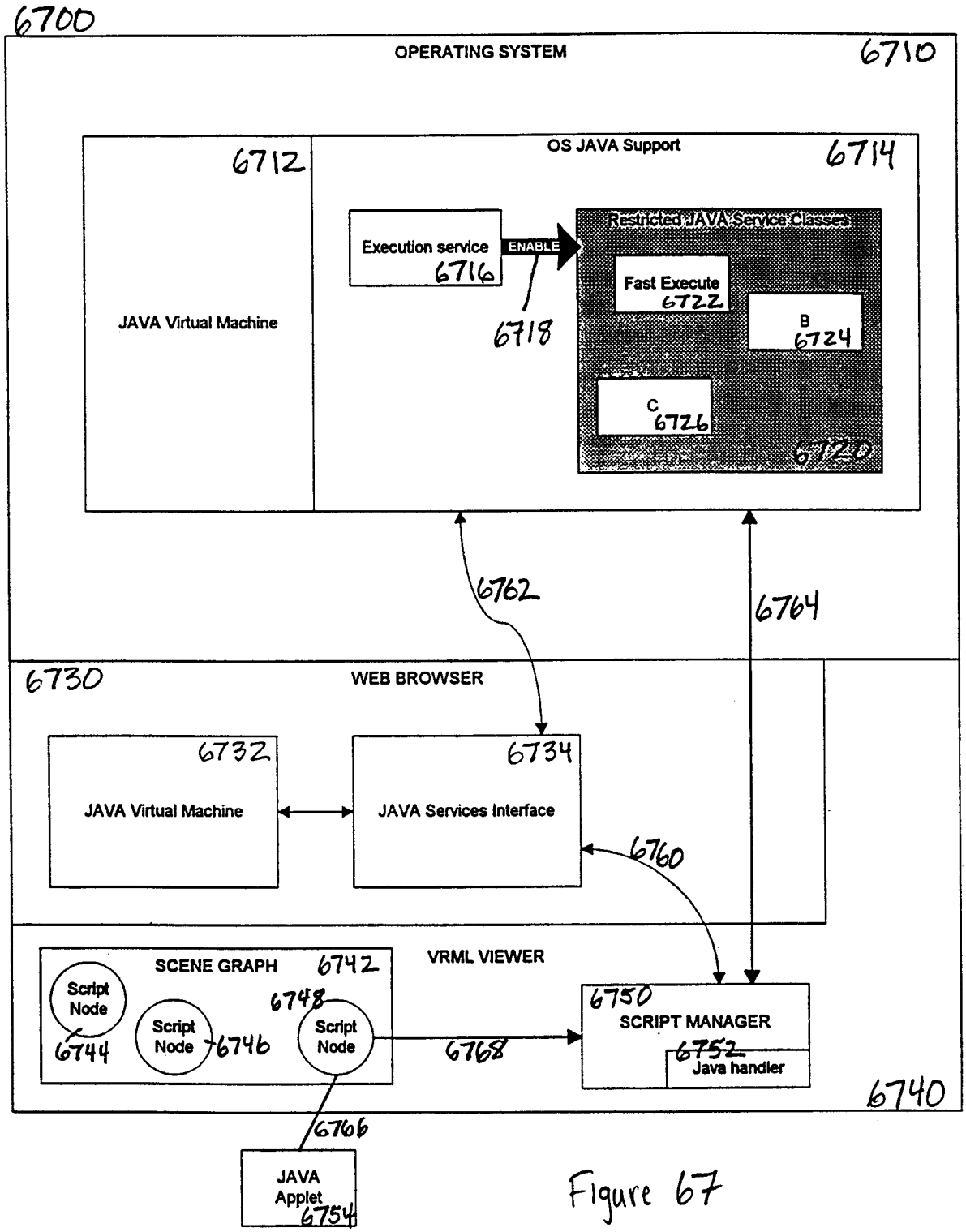
6700
OPERATING SYSTEM
6710
6712
OS JAVA Support
6714
JAVA Virtual Machine
Execution service
6716
ENABLE
6718
Restricted JAVA Service Classes
Fast Execute
6722
B
6724
C
6726
6720
6762
6764
6730
WEB BROWSER
6732
JAVA Virtual Machine
6734
JAVA Services Interface
6760
SCENE GRAPH
6742
Script Node
6744
Script Node
6746
6748
Script Node
VRML VIEWER
6750
SCRIPT MANAGER
6752
Java handler
6768
6740
6766
JAVA Applet
6754

Figure 67

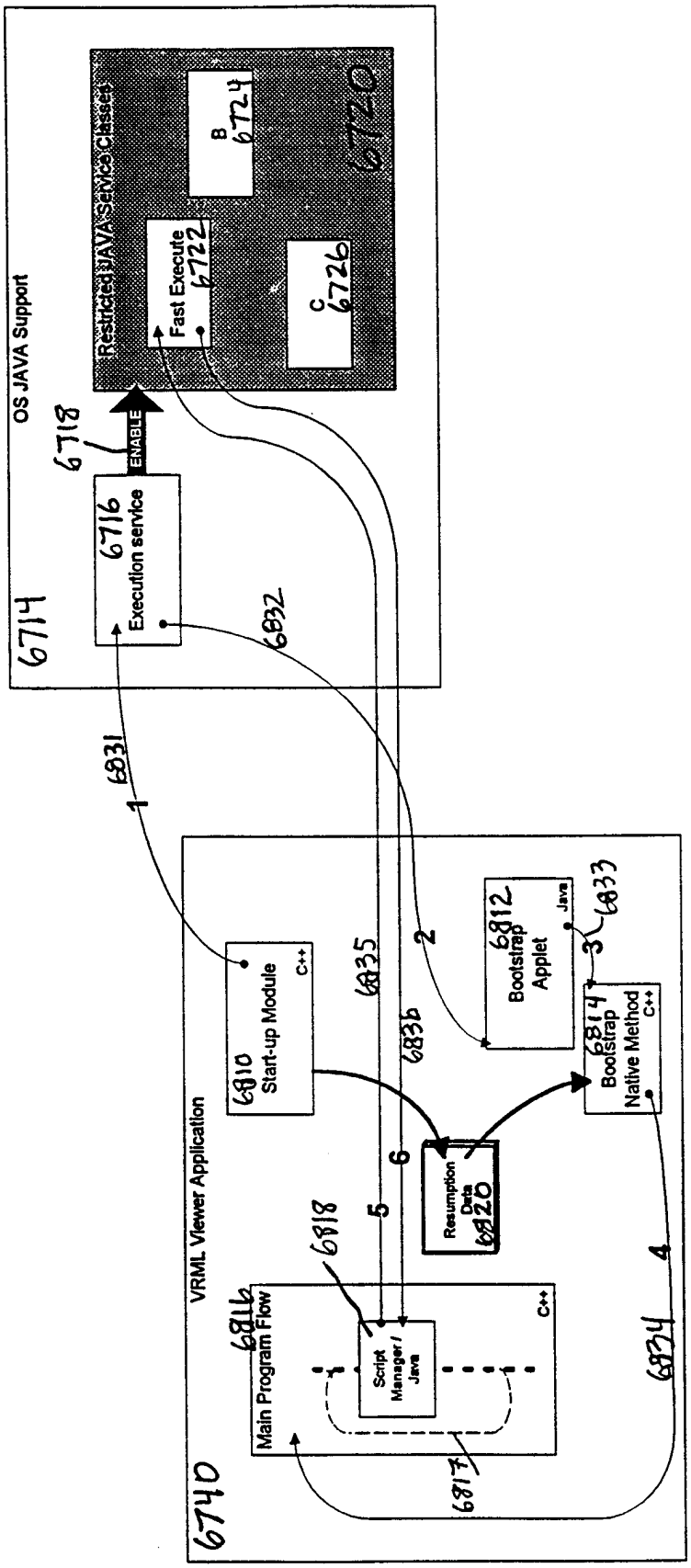
OS JAVA Support
6714
Restricted JAVA Services Classes
6720
B
6724
Fast Execute
6722
C
6726
6718
ENABLE
6716
Execution service
6832
6831
1
VRML Viewer Application
6740
6810
Start-up Module
C++
6812
Bootstrap
Applet
Java
6833
6814
Bootstrap
Native Method
C++
6835
2
3
6836
6820
Resumption
Data
5
6
6818
6816
Main Program Flow
Script
Manager /
Java
C++
6817
6824
4

Figure 68

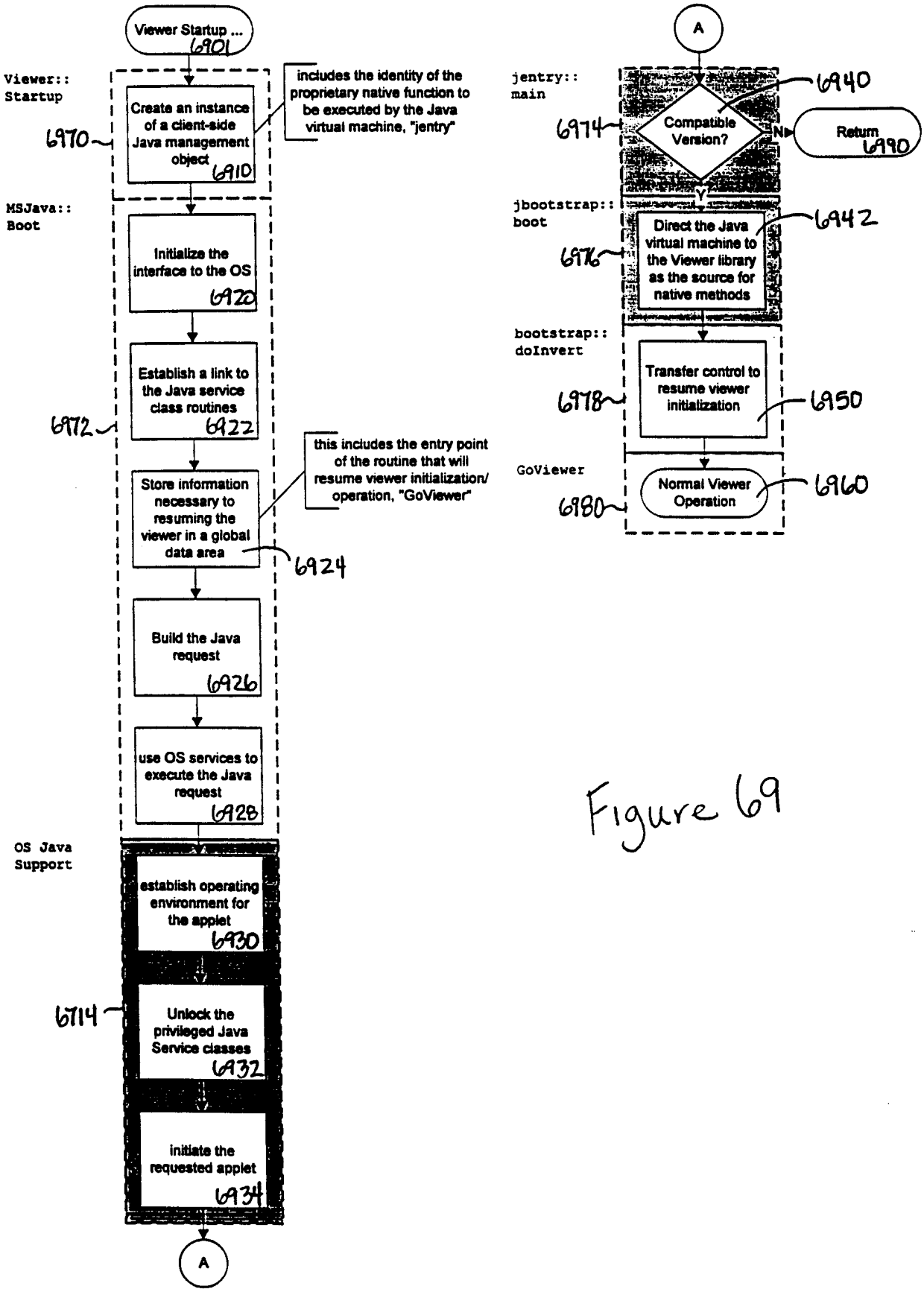
Viewer Startup ...
6901
Viewer:: Startup
Create an instance of a client-side Java management object
6910
6970
includes the identity of the proprietary native function to be executed by the Java virtual machine, "jentry"
MSJava:: Boot
Initialize the interface to the OS
6920
Establish a link to the Java service class routines
6922
6972
Store information necessary to resuming the viewer in a global data area
6924
this includes the entry point of the routine that will resume viewer initialization/ operation, "GoViewer"
Build the Java request
6926
use OS services to execute the Java request
6928
OS Java Support
establish operating environment for the applet
6930
6714
Unlock the privileged Java Service classes
6932
initiate the requested applet
6934
A
A
jentry:: main
Compatible Version?
6940
6974
N
Return
6990
Y
jbootstrap:: boot
Direct the Java virtual machine to the Viewer library as the source for native methods
6942
6976
bootstrap:: doInvert
Transfer control to resume viewer initialization
6978
6950
GoViewer
Normal Viewer Operation
6980
6960

Figure 69