(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
01 February 2018 (01.02.2018)

WIPO | PCT

(10) International Publication Number
WO 2018/022011 A1

(51) International Patent Classification:
G06T 17/00 (2006.01) *B33Y 50/00* (2015.01)
G06T 9/40 (2006.01)

(21) International Application Number:
PCT/US2016/043996

(22) International Filing Date:
26 July 2016 (26.07.2016)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant: **HEWLETT-PACKARD DEVELOPMENT COMPANY, L.P.** [US/US]; 11445 Compac Center Drive W., Houston, Texas 77070 (US).

(72) Inventors: **ZENG, Jun**; 1501 Page Mill Rd., Palo Alto, California 94304-1100 (US). **DEL ANGEL, Ana**; Montemorelos 299, Fraccionamiento. Loma Bonita, Guadalajara, 45060 (MX). **WHITE, Scott**; Cami de Can Graells, 1-21, 08174 Sant Cugat del Valles (ES). **CORTES, Sebastia**; Cami de Can Graells, 1-21, 08174 Sant Cugat del Valles (ES). **DISPOTO, Gary J.**; 1501 Page Mill Road, Palo Alto, California 94304 (US).

(74) Agent: **BURROWS, Sarah E.** et al.; HP Inc., 3390 E. Harmony Road, Mail Stop 35, Fort Collins, Colorado 80528 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

WO 2018/022011 A1

(54) Title: INDEXING VOXELS FOR 3D PRINTING

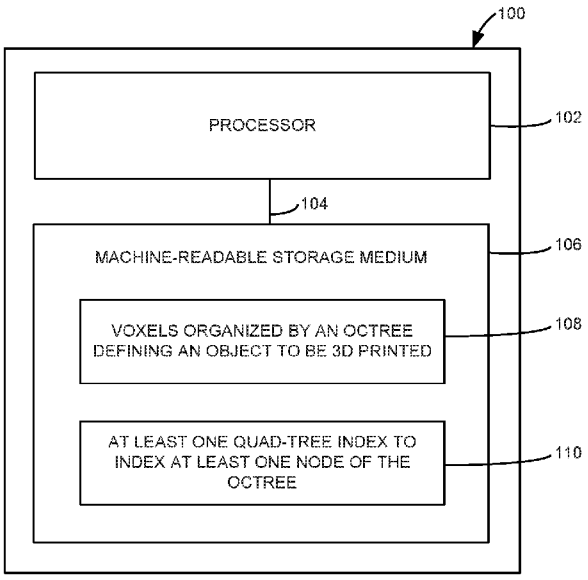

Fig. 1

(57) Abstract: One example includes a non-transitory machine readable storage medium including voxels organized by an octree and at least one quad-tree index. The octree defines an object to be three-dimensionally printed and includes a list of nodes for each depth of the octree where each node includes nodal content representing at least one voxel. The at least one quad-tree index is to index at least one node of the octree and has a depth less than or equal to a maximum resolved depth. The at least one quad-tree index is to be accessed by computer executable instructions to retrieve nodal content from the octree to control a processor to process the object to be three-dimensionally printed.

[Continued on next page]

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *as to the identity of the inventor (Rule 4.17(i))*

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

Published:

— *with international search report (Art. 21(3))*

INDEXING VOXELS FOR 3D PRINTING

Background

[0001] Printing technologies may be used to create three-dimensional objects from data output from a computerized modeling source. For example, a three-dimensional object may be designed using a computer program (e.g., a computer aided design (CAD) application), and the computer may output the data of the design to a printing system capable of forming the solid three-dimensional object. Solid free-form fabrication (or layer manufacturing) may be defined generally as a fabrication technology used to build a three-dimensional object using layer by layer or point-by-point fabrication. With this fabrication process, complex shapes may be formed without the use of a pre-shaped die or mold.

Brief Description of the Drawings

[0002] Figure 1 is a block diagram illustrating one example of a processing system for three-dimensional printing.

[0003] Figure 2 is a diagram illustrating one example of an octree representation.

[0004] Figures 3A-3D are diagrams illustrating each depth of the example octree representation of Figure 2.

[0005] Figure 4A is a diagram illustrating one example of the quaternary nodal numbering rule.

[0006] Figure 4B is a diagram illustrating one example of the mapping of a quad-tree index to a quad-tree.

[0007] Figure 5 is a flow diagram illustrating one example of a method to generate quad-tree indices for an octree.

[0008] Figure 6 is a flow diagram illustrating another example of a method to generate quad-tree indices for an octree.

[0009] Figure 7 is a flow diagram illustrating one example of a method to process an object for three-dimensional printing.

[0010] Figure 8 is a diagram illustrating one example of normalized Z-coordinates indicating which z-slices of an object are defined by each quad-tree index.

[0011] Figure 9 is a flow diagram illustrating another example of a method to process an object for three-dimensional printing.

[0012] Figure 10 is a flow diagram illustrating one example of a method to prepare an image for printing.

[0013] Figure 11 is a diagram illustrating one example of an image to be printed.

[0014] Figure 12 is a flow diagram illustrating one example of a method to perform the micro-slicing of Figure 9.

Detailed Description

[0015] In the following detailed description, reference is made to the accompanying drawings which form a part hereof, and in which is shown by way of illustration specific examples in which the disclosure may be practiced. It is to be understood that other examples may be utilized and structural or logical changes may be made without departing from the scope of the present disclosure. The following detailed description, therefore, is not to be taken in a limiting sense, and the scope of the present disclosure is defined by the appended claims. It is to be understood that features of the various examples described herein may be combined, in part or whole, with each other, unless specifically noted otherwise.

[0016] Heterogeneous objects are three-dimensional (3D) objects whose interior volume is composed of different materials (where a void may be considered a distinctive material type) to meet design objectives (e.g., light-weighting, processing compensation). For example, by spatially arranging different materials and empty spaces, a heterogeneous structure may be obtained that

has mechanical properties that go beyond those of the base constitutive materials.

[0017] One practical means to fabricate heterogeneous objects is 3D printing. Documenting and processing heterogeneous objects (e.g., post-design and pre-print), however, may require highly granular knowledge into the make-up of the interior volume of the objects. There is a computing challenge for providing the highly granular 3D description of the object and for accommodating the runtime and storage constraints, particularly in an industrial setting where objects may be large and complex and there is an overall throughput requirement for the factory's profitability.

[0018] Accordingly, as disclosed herein, a hierarchical data structure to accelerate the processing (e.g., slicing, compensating the systematic bias due to the physical process of the layered manufacturing) of an object to be 3D printed is described. The data structure includes a set of voxels organized by an octree defining an object to be printed, where each node of the octree represents at least one voxel, and a list of quad-tree indices indexing nodes of the octree to enable each indexed node of the octree to be individually addressable. The quad-tree indices may be resolved to a maximum resolved depth less than or equal to the depth of the octree. The greater the maximum resolved depth, the greater the size of the list of quad-tree indices and the lower the runtime latency when the list of quad-tree indices is accessed to 3D print the object. The lower the maximum resolved depth, the smaller the size of the list of quad-tree indices and the greater the runtime latency when the list of quad-tree indices is accessed to 3D print the object.

[0019] Figure 1 is a block diagram illustrating one example of a processing system 100 for 3D printing. System 100 includes a processor 102 and a machine-readable storage medium 106. Processor 102 is communicatively coupled to machine-readable storage medium 106 through a communication path 104. Although the following description refers to a single processor and a single machine-readable storage medium, the description may also apply to a system with multiple processors and multiple machine-readable storage mediums. In such examples, the instructions and/or data may be distributed

(e.g., stored) across multiple machine-readable storage mediums and the instructions and/or data may be distributed (e.g., executed/processed by) across multiple processors.

[0020] Processor 102 includes one or more central processing units (CPUs), microprocessors, and/or other suitable hardware devices for retrieval and execution of instructions and/or retrieval and processing of data stored in machine-readable storage medium 106. As will be described in more detail with reference to the following figures, processor 102 may fetch, decode, and execute instructions to create and/or modify data 108 including voxels organized by an octree defining an object to be 3D printed. The octree includes a list of nodes for each depth of the octree, where each node includes nodal content (e.g., material for the node) representing at least one voxel. Collectively, all the voxels form the 3D object. Processor 102 may fetch, decode, and execute instructions to create and/or modify data 110 including at least one quad-tree index to index at least one node of the octree. The at least one indexed node of the octree has a depth less than or equal to a maximum resolved depth, which indicates the resolved depth of the quad-tree index. Processor 102 may fetch, decode, and execute instructions to access the at least one quad-tree index to retrieve nodal content from the octree to process (e.g., slice) the object to be 3D printed.

[0021] As will be described in more detail with reference to the following figures, in one example, the at least one quad-tree index includes an ordered list of tuples. Each tuple indexes a corresponding node of the octree and includes a corresponding depth of the octree, an offset indicating the location of the corresponding node in the octree at the corresponding depth, and a non-leaf node flag indicating whether the corresponding node is a non-leaf node (i.e., has children nodes) or a leaf node (i.e., does not have children nodes). The at least one quad-tree index includes a resolved depth of the at least one quad-tree index and a flag indicating whether the nodes indexed by the at least one quad-tree index are fully resolved.

[0022] In one example, machine-readable storage medium 106 also includes an index for the at least one quad-tree index. The index includes the depth of the

octree, the maximum resolved depth, and an ordered list specifying the at least one quad-tree index and the resolved depth of the at least one quad-tree index. The octree may be a human readable and editable serial data file and the at least one quad-tree index may be a human readable and editable serial data file.

[0023] Machine-readable storage medium 106 is a non-transitory storage medium and may be any suitable electronic, magnetic, optical, or other physical storage device that stores executable instructions and/or data. Thus, machine-readable storage medium 106 may be, for example, random access memory (RAM), an electrically-erasable programmable read-only memory (EEPROM), a storage drive, an optical disc, and the like. Machine-readable storage medium 106 may be disposed within system 100, as illustrated in Figure 1. In this case, the executable instructions and/or data may be installed on system 100. Alternatively, machine-readable storage medium 106 may be a portable, external, or remote storage medium that allows system 100 to download the instructions and/or data from the portable/external/remote storage medium. In this case, the executable instructions and/or data may be part of an installation package.

[0024] Figure 2 is a diagram illustrating one example of an octree representation 200. An octree is stored in a machine-readable storage medium as a list of a list of nodes. The length of the outer list is the tree depth (e.g., four in the example of Figure 2). The length of each inner list is the number of nodes in the depth (e.g., one node for depth zero, eight nodes for depth one, 16 nodes for depth two, and eight nodes for depth three in the example of Figure 2). In octree representation 200, the root or zero depth includes a single non-leaf node 202. Node 202 has eight children nodes 204 at depth one. Of nodes 204 at depth one, nodes 206 and 208 are non-leaf nodes and the remaining nodes are leaf nodes. Node 206 has eight children leaf nodes 210 at depth two. Node 208 has eight children nodes 212 at depth two where node 214 is a non-leaf node and the remaining nodes are leaf nodes. Node 214 has eight children leaf nodes 216 at depth three.

[0025] Figures 3A-3D are diagrams illustrating each depth of the example octree representation 200 of Figure 2. Figure 3A illustrates an object 300 resolved to the root or zero depth of octree representation 200 (i.e., node 202). Figure 3B illustrates an object 302 resolved to depth one of octree 200 (i.e., nodes 204). At depth one, based on the octree nodal numbering rule, object 302 is divided into eight portions with each portion being defined by a node 204, respectively.

[0026] Figure 3C illustrates an object 304 resolved to depth two of octree representation 200 (i.e., nodes 210 and 212). At depth two, based on the octree nodal numbering rule, two portions (i.e., nodes 206 and 208) of object 304 are further divided into eight portions with each portion being defined by a node 210 and 212, respectively. The eight portions of node 208 are indicated at 306. The eight portions of node 206 are not visible in Figure 3C.

[0027] Figure 3D illustrates an object 308 resolved to depth three of octree representation 200 (i.e., nodes 216). At depth three, based on the octree nodal numbering rule, one portion (i.e., node 214) of object 308 is further divided into eight portions with each portion being defined by a node 216, respectively. The eight portions of node 214 are indicated at 310. Accordingly, any object may be defined by an octree having a depth suitable to represent the object to a desired resolution.

[0028] Each node represents at least one voxel. The payload for each leaf node in the octree includes the nodal content for the node, such as the material composition for the node (i.e., which material(s) to be used to form the at least one voxel represented by the node). The payload for each non-leaf node in the octree is the location of the first child node of the non-leaf node in the node list of the next depth. The nodes are stored sequentially one after the other in an octree (OCT) file in a machine-readable storage medium. Accordingly, in the example octree representation 200 of Figure 2, the nodes are stored in the following order: 202, 204, 210, 212, and 216.

[0029] The OCT file includes a header section and a body section. The header section includes the tree depth and the number of nodes for each depth. The body section includes each node. For each non-leaf node, the node includes a non-leaf node (NLN) flag. For each leaf node, the node includes the nodal

content for the node. The NLN flag may be a Boolean value indicating the node is a non-leaf node. In one example, the NLN flag is "-1". The nodal content may be a JavaScript object notation (JSON) object, which may be a list (e.g., floating point, material identifier (ID)) of tuples indicating the material composition of the node. If the node is filled by a single material, the size of the list may be one.

[0030] In one example, the body section of the OCT file follows two rules: 1) starting from the root depth, there is one line per depth array of the octree; and 2) within the same depth array, each node has seven siblings. The nodes are placed together and sequentially following the octal nodal numbering rule. Accordingly, each 8-node block (e.g., array indices from 8*i to 8*i+7) corresponds to one non-leaf node one line above (e.g., the i^{th} non-leaf node in its node array). Below is an example OCT file for octree representation 200 illustrated in Figure 2.

```
/*header*/
4 /*depth*/
1 8 16 8 /*node count per depth*/
/*body*/
-1 /*root NLN*/
-1 1 2 2 2 -1 2 1 /*depth 1 wherein "1" indicates a first material ID and "2"
        indicates a second material ID*/
1 1 2 2 2 1 1 1 1 1 2 2 1 1 -1 1 /*depth 2*/
1 1 1 2 2 2 2 2 /*depth 3*/
```

[0031] The OCT file may be generated by first writing the header including the tree depth and the number of nodes per depth. Next, a loop over the outer list is performed starting from the root depth (i.e., zero depth) and a loop over the inner list is performed. When a non-leaf node is encountered, the NLN flag (e.g., "-1") is written. When a leaf node is encountered, the content of the node (e.g., tuple of material ID and volume faction) is written (e.g., "1" or "2" in the above example). A line break may be inserted after each inner loop to improve readability. Generating the OCT file is efficient since each node is visited a single time for a single read/write operation. No sorting or other computations are involved.

[0032] Figure 4A is a diagram illustrating one example of the quaternary nodal numbering rule. Area 400 is divided into four quadrants 402, 404, 406, and 408. The (y, x) coordinates for quadrant 402 are (0, 0) = 0 indicating quadrant zero. The (y, x) coordinates for quadrant 404 are (0, 1) = 1 indicating quadrant one. The (y, x) coordinates for quadrant 406 are (1, 0) = 2 indicating quadrant two. The (y, x) coordinates for quadrant 408 are (1, 1) = 3 indicating quadrant three.

[0033] The quad-tree indices (QTI) are organized in a machine-readable storage medium as a list (e.g., a linked list). The quad-tree indices are a set of files including an index file (e.g., index.qti) and a set of quad-tree index files (e.g., *.qti) named sequentially starting from 0.qti. The index.qti file includes the depth of the corresponding octree, a maximum resolved depth for the quad-tree indices, and an ordered list specifying each quad-tree index and the resolved depth of the each quad-tree index. One example of an index.qti file is provided below.

```
8 /* octree depth*/
3 /*maximum resolved depth*/
3, 3, 2, 1 /*the resolved depth for each *.qti file in sequential order*/
```

In this example, there are three *.qti files including 0.qti having a resolved depth of three, 1.qti having a resolved depth of three, 2.qti having a resolved depth of two, and 3.qti having a resolved depth of one.

[0034] The maximum resolved depth (MRD) may be a user input. The maximum resolved depth is a variable that tunes a tradeoff between the storage (e.g., QTI file sizes) and the runtime computing (e.g., slicing) speed. If the maximum resolved depth equals zero, the assistance of the quad-tree indices is effectively turned off and the QTI file size is near zero. In this case, accessing a voxel during runtime will require walking the octree. If the maximum resolved depth equals the octree depth, the QTI file sizes are larger and the QTI files effectively resolve the entire octree maximizing runtime computing (e.g., slicing) speed. In this case, random access of voxels during runtime with constant cost is effectively enabled.

[0035] Each *.qti file is used to write a z-slice of varying thickness of the octree. The thickness of each *.qti file is determined by the resolved depth of the *.qti file. In a computer readable storage medium, a *.qti file may be a node of a linked list, defined as:

```
Class QTI /*z-slice of the octree*/ {
QTI*prev;
QTI*next; /*linked list*/
Int CRD; /*current resolved depth*/
Boolean isFullyresolved;
tuple_list [(depth, offset, NLN) …];
}
```

The above linked list class may be used to create the QTI files and deleted once the QTI files are created. Therefore, once created, the QTI files may be a list of files rather than a linked list. Each tuple indexes a corresponding node of the octree and includes a corresponding depth of the octree, an offset indicating the location of the corresponding node in the octree at the corresponding depth, and a NLN flag indicating whether the corresponding node is a non-leaf node (i.e., not fully resolved) or a leaf node (i.e., fully resolved). The tuple list is ordered as follows: 1) within the same depth array, each node has four siblings, which are placed together and sequentially following the quaternary nodal numbering rule; and 2) a non-leaf node is replaced by its children nodes in place.

[0036] Figure 4B is a diagram illustrating one example of the mapping of a quad-tree index (*.qti file) to a quad-tree. The example *.qti mapped in Figure 4B is provided below.

```
3 /*resolved depth*/
{
        (2, 0, T) /*indicated at 412*/
        (2, 1, T) /*indicated at 414*/
        (2, 2, T) /*indicated at 416*/
                (3, 0, T) (3, 1, T) (3, 2, T) (3, 3, T) /*indicated at 418, 420,
                        422, and 424, respectively*/
} /*first quadrant (0, 0) = 0*/
(1, 1, T) /*second quadrant (0, 1) = 1, indicated at 426*/
(1, 2, T) /*third quadrant (1, 0) = 2, indicated at 428*/
{
```

```
        (2, 8, T) /*indicated at 430*/
        (2, 9, T) /*indicated at 432*/
        (2, 10, T) /*indicated at 434*/
        (2, 11, T) /*indicated at 436*/
} /*fourth quadrant (1, 1) = 3*/
```

The brackets and indentations in the above example are included for improved readability and are not required for the *.qti file. The same quad-tree may be resolved at different depths For example, the above quad-tree may be resolved to depth 1: "1, (1, 0, F) (1, 1, T) (1, 2, T) (1, 3, F)" or depth 2: "2, (2, 0, T) (2, 1, T) (2, 2, T) (2, 3, F) (1, 1, T) (1, 2, T) (2, 8, T) (2, 9, T) (2, 10, T), (2, 11, T)".

[0037] Figure 5 is a flow diagram illustrating one example of a method 500 to generate quad-tree indices for an octree. In one example, method 500 is performed by a processor, such as processor 102 previously described and illustrated with reference to Figure 1. At 502, method 500 includes receiving an octree defining an object to be three-dimensionally printed, the octree including a list of nodes for each depth of the octree. At 504, method 500 includes receiving a maximum resolved depth less than or equal to the depth of the octree. At 506, method 500 includes generating a list of quad-tree indices to index each node of the octree up to the maximum resolved depth, each quad-tree index of the quad-tree indices comprising an ordered list of tuples, each tuple indexing a corresponding node of the octree and including a corresponding depth of the octree, an offset indicating the location of the corresponding node in the octree at the corresponding depth, and a non-leaf node flag indicating whether the corresponding node is a non-leaf node or a leaf node.

[0038] Method 500 may further include generating an index for the list of quad-tree indices, the index comprising the depth of the octree, the maximum resolved depth, and an ordered list specifying each quad-tree index and the resolved depth of each quad-tree index. The greater the maximum resolved depth, the greater the size of the list of quad-tree indices and the lower the runtime latency when the list of quad-tree indices is accessed to 3D print the object.

[0039] Figure 6 is a flow diagram illustrating another example of a method 600 to generate quad-tree indices for an octree. In one example, method 600 is performed by a processor, such as processor 102 previously described and illustrated with reference to Figure 1. At 602, a maximum resolved depth (MRD) is received. MRD is an integer less than or equal to the depth of the octree. At 604, a linked list (QTILIST) of quad-tree indices is initialized and a current resolved depth (CRD) is initialized to zero. At 606, method 600 determines whether CRD is less than MRD.

[0040] If CRD is less than MRD, then at 610 a QTI pointer is reset to the head of the QTILIST. At 612, a QTI is retrieved based on the QTI pointer. At 614, method 600 determines whether the QTI is fully resolved. In one example, determining whether the QTI if fully resolved includes reading the isFullyresolved Boolean value for the QTI. If the QTI is not fully resolved, then at 616, the QTI is duplicated to provide an upper QTI (UQTI) and a lower QTI (LQTI). At 618, the tuples of the QTI are scanned and for each tuple of the QTI that is not resolved, eight children are retrieved and the tuple in the UQTI is replaced with four tuples corresponding to the upper four children and the tuple in the LQTI is replaced with four tuples corresponding to the lower four children. At 620, the QTI in QTILIST is replaced with the LQTI and the UQTI is inserted into the QTILIST after the LQTI.

[0041] If the QTI is fully resolved at 614 or after block 620, at 622 the next QTI in the QTILIST is retrieved. At 624, method 600 determines whether the end of the QTILIST has been reached. If the end of the QTILIST has not been reached, then method 600 returns to decision block 614 and the process continues. If the end of the QTILIST has been reached, then at 626 CRD is set equal to CRD plus one (i.e., incremented) and method 600 returns to decision block 606 and the process continues. If CRD is not less than MRD at 606, then at 608 the QTI files are written since the QTILIST is completed up to the MRD. The index.qti file is generated and each *.qti file is written by walking the QTILIST.

[0042] The following figures describe the runtime operation using the octree and quad-tree indices for 3D printing of an object. The resolution of the octree and

quad-tree indices are hardware (e.g., printer) independent. During runtime, the octree and quad-tree indices are loaded into memory to produce slices (e.g., images with material assignment per pixel) to provide to a printer for printing. At this time, the hardware characteristics may be addressed including additional 3D computation to compensate for process physics and/or the hardware resolution (e.g., printing resolution).

[0043] Prior to printing, the OCT file is loaded into memory. The OCT file is scanned and the header section is read to allocate memory for the list of node lists (per depth). The nodes are then read from the OCT file per depth starting from the root depth and the node content is filled accordingly. For each depth, a non-leaf node counter (NLNC) is initiated to zero. When a leaf-node is encountered, the payload for the node is filled with the content from the OCT file. When a non-leaf node is encountered, the payload for the node is set to NLNC*8, which indicates that the children for the node are in the next depth starting position from NLNC to NLNC+8. NLNC is incremented by one for each non-leaf node. In this way, the octree is recreated in memory. The compute time is linear to the node count and for each node the compute time is small (i.e., largely reading from the OCT file).

[0044] Additional 3D computations may be used to generate auxiliary data to support and/or to compensate for printing physics. For example, in multi-jet fusion printers thermal diffusion is an effect that may be compensated for by active management for geometry fidelity (e.g., sharp corners). Preserving the octree in memory enables this type of computation since the new nodal values are stored in the octree. The quad-tree indices are assistive indices that accelerate nodal content retrieval from the octree.

[0045] Figure 7 is a flow diagram illustrating one example of a method 700 to process an object for 3D printing. In one example, method 700 is performed by a processor, such as processor 102 previously described and illustrated with reference to Figure 1. At 702, method 700 includes receiving an octree defining an object to be three-dimensionally printed, the octree including a list of nodes for each depth of the octree, each node including nodal content. At 704, method 700 includes receiving a list of quad-tree indices indexing each node of

the octree up to a maximum resolved depth, each quad-tree index of the quad-tree indices comprising an ordered list of tuples, each tuple indexing a corresponding node of the octree and including a corresponding depth of the octree, an offset indicating the location of the corresponding node in the octree at the corresponding depth, and a non-leaf node flag indicating whether the corresponding node is a non-leaf node or a leaf node. At 706, method 700 includes rasterizing each slice of the object to be three-dimensionally printed by accessing a corresponding quad-tree index for the slice to retrieve nodal content from the octree.

[0046] Method 700 may further include determining normalized z-coordinates for each quad-tree index between zmin and zmax and slicing sequentially from zmin to zmax based on a thickness for each slice to identify each slice prior to rasterizing. The nodal content of the octree may be modified based on the characteristics of a specified three-dimensional printer prior to rasterizing each slice (e.g., to compensate for thermal diffusion effects).

[0047] Figure 8 is a diagram illustrating one example of normalized z-coordinates indicating which z-slices of an object 800 are defined by each quad-tree index. The index.qti file is loaded into memory and based on the depth list, a qti_marker[], which are the normalized z-coordinates for each QTI slice are computed. The qti_marker[] is used to select the correct QTI file for each slice. The z-coordinates for the object are normalized between -1 and 1 as indicated at 802. The z-resolution (Z_{RES}) is based on the maximum resolved depth (MRD) and is calculated as follows:

$$Z_{RES} = \frac{1}{2^{MRD}}$$

For example, for a maximum resolved depth of three for the example of Figure 8, the z-resolution is 1/8. The qti_marker for each QTI is determined as follows:

```
qti_marker[] = NULL;
marker = -1; /*normalized zmin for octree*/
for (depth in depth_list):
        thickness = 2.0/(2^depth)
        marker += thickness
        qti_marker.append(marker)
```

[0048] In the example of Figure 8, therefore, 0.qti as indicated at 804 having a depth of 3 extends between -1 and -0.75; 1.qti as indicated at 806 having a depth of 3 extends between -0.75 and -0.50; 2.qti as indicated at 808 having a depth of 2 extends between -0.50 and 0; and 3.qti as indicated at 810 having a depth of 1 extends between 0 and 1. Thus, qti_marker[] is qti_marker[-0.75, -0.50, 0, 1] in this example.

[0049] Figure 9 is a flow diagram illustrating another example of a method 900 to process an object for 3D printing. In one example, method 900 is performed by a processor, such as processor 102 previously described and illustrated with reference to Figure 1. At 902, the octree defining the object to be 3D printed is loaded and reconstructed in memory. At 904, 3D processing for hardware compensation is performed. At 906, the index.qti file is loaded and the qti_marker[] is computed. At 908, z is initialized to zmin (i.e., -1). At 910, method 900 determines whether z is less the zmax (i.e., 1). If z is less than zmax, then at 914 the correct QTI for the slice is found based on the qti_marker[]. At 916, method 900 determines whether the slice has already been rasterized. If the slice has not already been rasterized, then at 918 method 900 determines whether the QTI is fully resolved. If the QTI is not fully resolved, then at 920 micro-slicing is performed to fully resolve the QTI. Micro-slicing is further described below with reference to Figure 12. If the QTI is fully resolved at 918 or after performing micro-slicing at 920, the slice or micro-slice is rasterized at 922. Rasterizing the QTI is further described below with reference to Figures 10 and 11. If the slice is already rasterized at 916 or after rasterizing the slice or micro-slice at 922, the image is printed at 924. At 926, z is incremented by the next slice thickness and the process returns to decision block 910. An external program may set the thickness for each slice. The thickness for each slice may be a constant (e.g., 100 μm) or may be different for different slices. If z is not less than zmax at 910, then at 912 the printing of the object is complete.

[0050] Figure 10 is a flow diagram illustrating one example of a method 1000 to prepare an image (i.e., slice) for printing. Method 1000 is performed by a processor, such as processor 102 previously described and illustrated with

reference to Figure 1. At 1002, method 1000 includes initializing an image. In one example, the image is initialized to an NxN image where "N" is a suitable value indicating the length of one side of the image. The initializing of the image may be based on polygon boundaries obtained via vector slices. At 1004, method 1000 includes reading each tuple of the quad-tree index for the slice. At 1006, method 1000 includes computing a pixel array in the image that corresponds to each tuple based on the position of the tuple in the quad-tree index for the slice. At 1008, method 1000 includes retrieving nodal content from the octree based on the offset of each tuple and assigning the nodal content to the pixel array. The nodal content may include the material composition. After scanning all tuples in the quad-tree index, the content composition for the image, and thus the slicing operation, is complete.

[0051] Figure 11 is a diagram illustrating one example of an image 1100 to be printed, which is generated by rasterizing the QTI for a slice. In this example, image 1100 defines a slice of an object having a border indicated at 1101. The slice is partitioned into two-dimensional (2D) tiles based on the resolved depth (i.e., a 2D array of $[2^{resolved\_depth}, 2^{resolved\_depth}]$). In the example of Figure 11, where the resolved depth is three, the slice is partitioned into 8x8 tiles. Each tuple (depth, offset) corresponds to a unique set of tiles. An address index (addr) is used to compute the tuple-tile mapping as follows:

```
addr = 0
for (int i=0; i < tuple_list.size(); i++) {
        addr_of_this_tuple = addr
        addr = addr + 4^(this_tuple_depth - resolved_depth) /*quaternary numbering*/
}
```

The addr compute may happen while scanning the tuple list. For a given tuple, the addr of the tuple may be written as $sum(ai*4^{(resolved\_depth - i - 1)})$ or $[a\_0, a\_1, a\_2]*[4^{(3 - 0 - 1)}, 4^1, 4^0]$, i = 0, 1, … (resolved_depth -1), ai = 0, 1, 2, or 3. Integer ai is further mapped to a tuple of (0, 0), (1, 0), (0, 1), (1, 1) denoted as (ai[0], ai[1]) where [0] corresponds to the x index and [1] corresponds to the y index.

[0052] An addr can translate a list of ai[0] (x-index array) (in this example (a_0[0], a_1[0], a_2[0])) and ai[1] (y-index array) (in this example (a_0[1], a_1[1], a_2[1])) as indicated at 1102, 1104, and 1106, respectively. Therefore:

$$x\text{-addr} = \text{sum}\ (\ a\_i[0] * 2^{(resolved\_depth - i - 1)}\ )$$
$$y\text{-addr} = \text{sum}\ (\ a\_i[1] * 2^{(resolved\_depth - i - 1)}\ )$$

The linear coverage of each tuple is determined by the depth of the tuple, $2^{(this\_tuple\_depth - resolved\ depth)}$. The tiles covered by each tuple is between [x-addr, y-addr] and [x-addr + linear_coverage, y-addr + linear_coverage]. To map to the NxN sliced image, each tuple corresponds to the pixel array between (int(float(x-addr)/($2^{resolved\_depth}$) * N, int(float(y-addr)/($2^{resolved\_depth}$) * N)) and (int(float(x-addr + linear_coverage)/($2^{resolved\_depth}$) * N, int(float(y-addr + linear_coverage)/($2^{resolved\_depth}$) * N)).

[0053] Based on the offset and depth value of each tuple, the octree node corresponding to the tuple may be directly accessed to retrieve the nodal content. If the content (e.g., material ID and/or distribution) is uniform, all pixels in the pixel array are assigned the same value. If sub-node resolution is enabled (e.g., multiple materials reside at different portions of the same node), different pixels with different values may be assigned based on the nodal content (e.g., via interpolation schemes).

[0054] A voxel-based data structure may not be body-fitted. The voxel cubes may not align with complex shape boundaries. Therefore, the resulting slices may have shape error where the size of this error is about the size of a voxel at the boundary. To overcome this, vector slicing may be implemented as part of the rasterization. A very efficient vector slicer can handle 20 million or more triangles and generate a slice in approximately 100 milliseconds.

[0055] The vector slicer is used to generate the slices in the form of boundary polygons. The boundary polygons are body fitted and have zero slicing induced shape error (as opposed to voxel slicing with slicing induced shape error of voxel size). The boundary polygons are used to generate image pixels for the boundaries.

[0056] The boundary pixels on the slices obtain the material properties through voxels. For each boundary pixel, the region of influence (ROI) is identified, which may cover multiple voxels. The value assigned to the pixel is the weighted-average of the values carried by the voxels within the ROI. The weight is the inverse of the distance from the voxel center to the pixel.

[0057] Figure 12 is a flow diagram illustrating one example of a method 1200 to perform the micro-slicing of block 920 of Figure 9. In one example, method 1200 is performed by a processor, such as processor 102 previously described and illustrated with reference to Figure 1. At 1202, a linked list (MICRO_QTILIST) of quad-tree indices is initialized and the current resolved depth (CRD) is initialized to the CRD of the current QTI (QTI_CRD). At 1204, method 1200 determines whether CRD is less than the octree depth.

[0058] If CRD is less than the octree depth, then at 1212, a MICRO_QTI pointer is reset to the head of the MICRO_QTILIST. At 1214, a MICRO_QTI is retrieved based on the MICRO_QTI pointer. At 1216, method 1200 determines whether the MICRO_QTI if fully resolved. If the MICRO_QTI is not fully resolved, then at 1218, the MICRO_QTI is duplicated to provide an upper MICRO_QTI (MICRO_UQTI) and a lower MICRO_QTI (MICRO_LQTI). At 1220, the tuples of the MICRO_QTI are scanned and for each tuple of the MICRO_QTI that is not resolved, eight children are retrieved and the tuple in the MICRO_UQTI is replaced with four tuples corresponding to the upper four children and the tuple in the MICRO_LQTI is replaced with four tuples corresponding to the lower four children. At 1222, the MICRO_QTI in the MICRO_QTILIST is replaced with the MICRO_LQTI and the MICRO_UQTI is inserted into the MICRO_QTILIST after the MICRO_LQTI.

[0059] If the MIRCRO_QTI is fully resolved at 1216 or after block 1222, at 1224 the next MICRO_QTI in the MICRO_QTILIST is retrieved. At 1226, method 1200 determines whether the end of the MICRO_QTILIST has been reached. If the end of the MICRO_QTILIST has not been reached, then method 1200 returns to decision block 1216 and the process continues. If the end of the MICRO_QTILIST has been reached, then at 1228 CRD is set equal to CRD plus one (i.e., incremented) and method 1200 returns to decision block 1204 and the process continues. If CRD is not less than the octree depth at 1204, then at 1206, the correct MICRO_QTI for the micro-slice is found. The micro-slice is then rasterized at block 922 as previously described and illustrated with reference to Figure 9.

[0060] In summary, the *.qti that the slice plane intercepts is rasterized. The rasterized image is cached until the image is obsolete so that no *.qti is rasterized twice. In one example, to rasterize a *.qti, each pixel is visited only once and only relevant nodes are visited and visited only once. The MRD and micro-slicing add elasticity into the balancing between the OCT and QTI file size generated by the pre-print and the additional runtime overhead.

[0061] Although specific examples have been illustrated and described herein, a variety of alternate and/or equivalent implementations may be substituted for the specific examples shown and described without departing from the scope of the present disclosure. This application is intended to cover any adaptations or variations of the specific examples discussed herein. Therefore, it is intended that this disclosure be limited only by the claims and the equivalents thereof.

CLAIMS

1. A non-transitory machine readable storage medium comprising:

voxels organized by an octree defining an object to be three-dimensionally printed, the octree including a list of nodes for each depth of the octree, each node including nodal content representing at least one voxel; and

at least one quad-tree index to index at least one node of the octree having a depth less than or equal to a maximum resolved depth,

wherein the at least one quad-tree index is to be accessed by computer executable instructions to retrieve nodal content from the octree to control a processor to process the object to be three-dimensionally printed.

2. The non-transitory machine readable storage medium of claim 1, wherein the at least one quad-tree index comprises an ordered list of tuples, each tuple indexing a corresponding node of the octree and including a corresponding depth of the octree, an offset indicating the location of the corresponding node in the octree at the corresponding depth, and a non-leaf node flag indicating whether the corresponding node is a non-leaf node or a leaf node.

3. The non-transitory machine readable storage medium of claim 1, wherein the at least one quad-tree index comprises a resolved depth of the at least one quad-tree index and a flag indicating whether the nodes indexed by the at least one quad-tree index are fully resolved.

4. The non-transitory machine readable storage medium of claim 1, further comprising:

an index for the at least one quad-tree index, the index comprising the depth of the octree, the maximum resolved depth, and an ordered list specifying the at least one quad-tree index and the resolved depth of the at least one quad-tree index.

5. The non-transitory machine readable storage medium of claim 1, wherein the octree is a human readable and editable serial data file, and

wherein the at least one quad-tree index is a human readable and editable serial data file.

6. A method to generate quad-tree indices for an octree, the method comprising;

receiving, via a processor, voxels organized by an octree defining an object to be three-dimensionally printed, the octree including a list of nodes for each depth of the octree, each node representing at least one voxel;

receiving, via the processor, a maximum resolved depth less than or equal to the depth of the octree; and

generating, via the processor, a list of quad-tree indices to index each node of the octree up to the maximum resolved depth, each quad-tree index of the quad-tree indices comprising an ordered list of tuples, each tuple indexing a corresponding node of the octree and including a corresponding depth of the octree, an offset indicating the location of the corresponding node in the octree at the corresponding depth, and a non-leaf node flag indicating whether the corresponding node is a non-leaf node or a leaf node.

7. The method of claim 6, wherein generating the list of quad-tree indices comprises:

initializing a linked list of quad-tree indices and initializing a current depth to zero;

iterating the linked list of quad-tree indices until a current depth equals the maximum resolved depth;

for each quad-tree index of the quad-tree indices, determining whether the quad-tree index is fully resolved;

in response to determining that a quad-tree index is not fully resolved, dividing the quad-tree index into two quad-tree indices and updating each of the two quad-tree indices for the current depth to further resolve each of the two quad-tree indices;

in response to determining that the quad-tree index is fully resolved, moving to the next quad-tree index in the linked list of quad-tree indices; and

in response to reaching the end of the linked list of quad-tree indices, incrementing the current depth by one and repeating the iterating of the linked list of quad-tree indices until the current depth equals the maximum resolved depth.

8. The method of claim 7, wherein dividing the quad-tree index into two quad-tree indices and updating each of the two quad-tree indices for the current depth comprises:

duplicating the quad-tree index to provide an upper quad-tree index and a lower quad-tree index;

scanning each tuple of the quad-tree index;

for each tuple indicating a non-leaf node, retrieving eight of the children of the non-leaf node and replacing the tuple in the lower quad-tree index with tuples corresponding to the lower four children and replacing the tuple in the upper quad-tree index with tuples corresponding to the upper four children; and

replacing the quad-tree index with the lower quad-tree index and inserting the upper quad-tree index into the linked list of quad-tree indices after the lower quad-tree index.

9. The method of claim 6, further comprising:

generating an index for the list of quad-tree indices, the index comprising the depth of the octree, the maximum resolved depth, and an ordered list specifying each quad-tree index and the resolved depth of each quad-tree index.

10. The method of claim 6, wherein the greater the maximum resolved depth, the greater the size of the list of quad-tree indices and the lower the runtime latency when the list of quad-tree indices is accessed to three-dimensionally print the object.

11. A method to process an object for three-dimensional printing, the method comprising;

receiving, via a processor, voxels organized by an octree defining an object to be three-dimensionally printed, the octree including a list of nodes for each depth of the octree, each node including nodal content representing at least one voxel;

receiving, via the processor, a list of quad-tree indices indexing each node of the octree up to a maximum resolved depth, each quad-tree index of the quad-tree indices comprising an ordered list of tuples, each tuple indexing a corresponding node of the octree and including a corresponding depth of the octree, an offset indicating the location of the corresponding node in the octree at the corresponding depth, and a non-leaf node flag indicating whether the corresponding node is a non-leaf node or a leaf node; and

rasterizing, via the processor, each slice of the object to be three-dimensionally printed by accessing a corresponding quad-tree index for the slice to retrieve nodal content from the octree.

12. The method of claim 11, wherein rasterizing each slice of the object to be three-dimensionally printed comprises:

determining whether the quad-tree index for the slice is fully resolved;

in response to determining the quad-tree index for the slice is fully resolved, rasterizing the slice;

in response to determining the quad-tree index for the slice is not fully resolved, fully resolving the quad-tree index prior to rasterizing the slice.

13. The method of claim 11, wherein rasterizing each slice of the object to be three-dimensionally printed comprises:

initializing an image based on polygon boundaries obtained via vector slices;

reading each tuple of the quad-tree index for the slice;

computing a pixel array in the image that corresponds to each tuple based on the position of the tuple in the quad-tree index for the slice; and

retrieving nodal content from the octree based on the offset of each tuple and assigning the nodal content to the pixel array.

14. The method of claim 11, further comprising:

determining normalized z-coordinates for each quad-tree index between zmin and zmax; and

slicing sequentially from zmin to zmax based on a thickness for each slice to identify each slice prior to rasterizing.

15. The method of claim 11, further comprising:

modifying nodal content of the octree based on characteristics of a specified three-dimensional printer prior to rasterizing each slice.

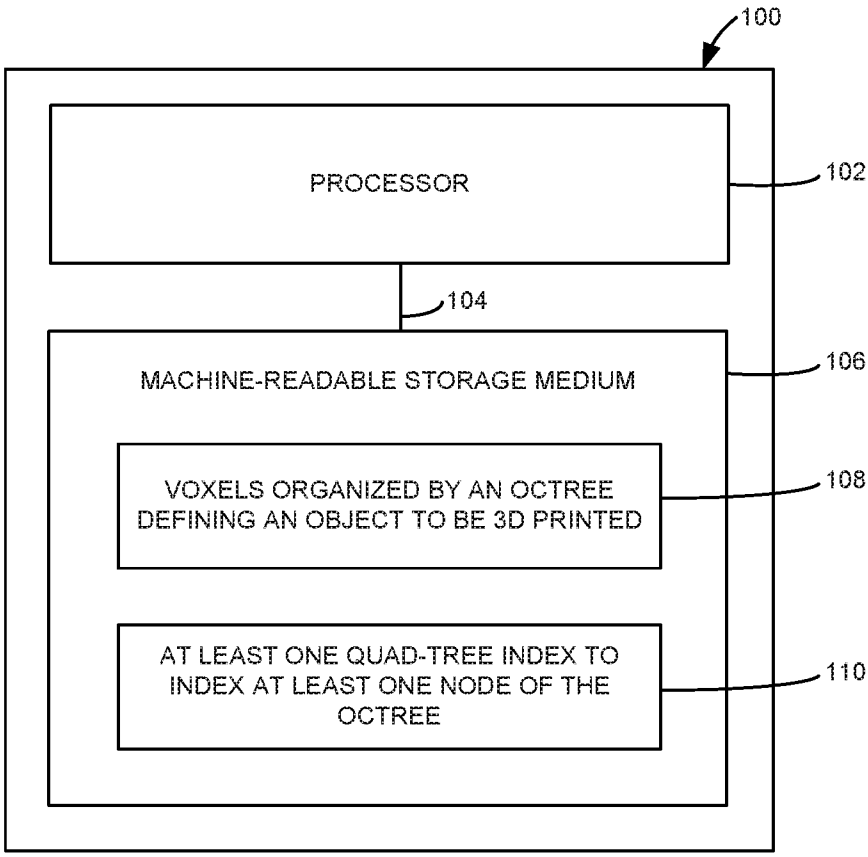
100
PROCESSOR
102
104
MACHINE-READABLE STORAGE MEDIUM
106
VOXELS ORGANIZED BY AN OCTREE DEFINING AN OBJECT TO BE 3D PRINTED
108
AT LEAST ONE QUAD-TREE INDEX TO INDEX AT LEAST ONE NODE OF THE OCTREE
110

Fig. 1

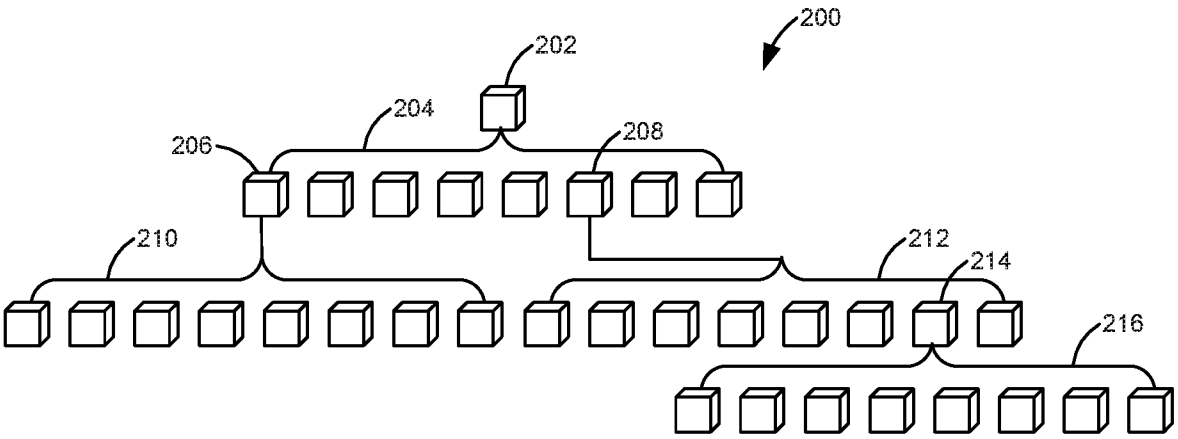

Fig. 2

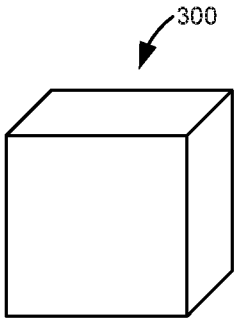

Fig. 3A

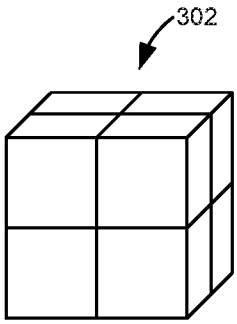

Fig. 3B

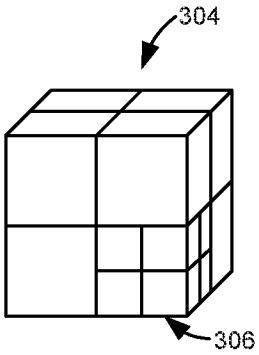

Fig. 3C

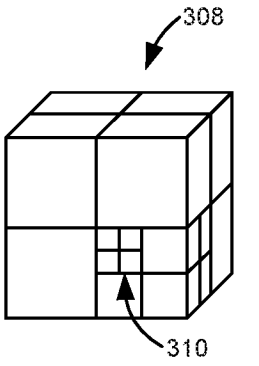

Fig. 3D

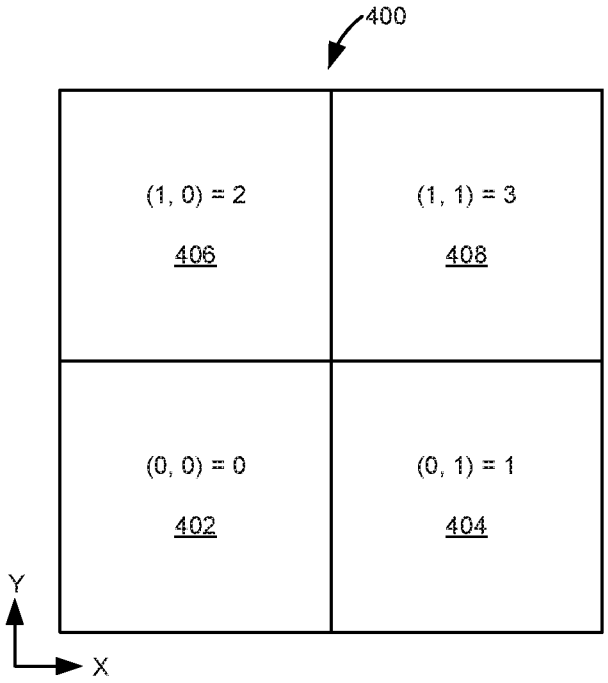
400
(1, 0) = 2
406
(1, 1) = 3
408
(0, 0) = 0
402
(0, 1) = 1
404
Y
X

Fig. 4A

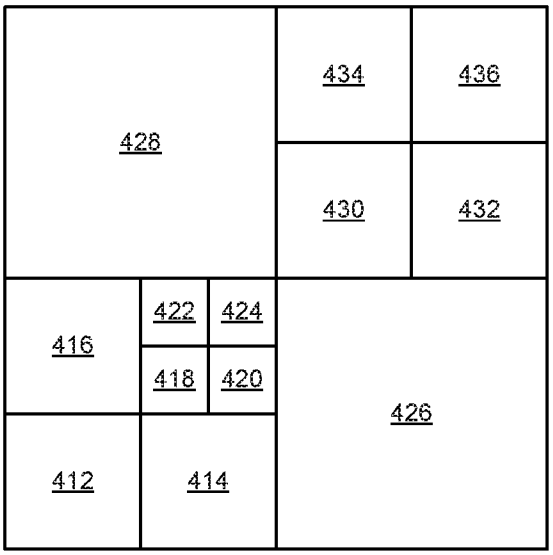
428
434
436
430
432
416
422
424
418
420
426
412
414

Fig. 4B

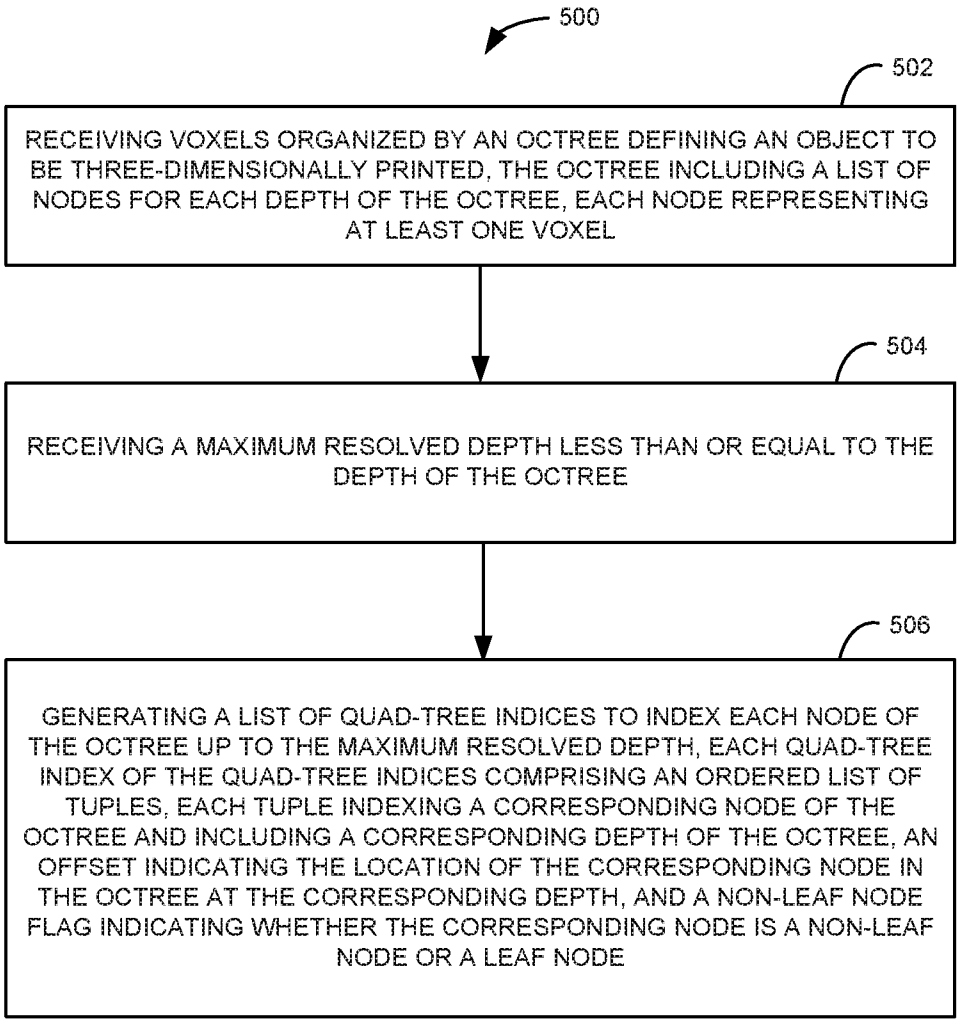
500
502
RECEIVING VOXELS ORGANIZED BY AN OCTREE DEFINING AN OBJECT TO BE THREE-DIMENSIONALLY PRINTED, THE OCTREE INCLUDING A LIST OF NODES FOR EACH DEPTH OF THE OCTREE, EACH NODE REPRESENTING AT LEAST ONE VOXEL
504
RECEIVING A MAXIMUM RESOLVED DEPTH LESS THAN OR EQUAL TO THE DEPTH OF THE OCTREE
506
GENERATING A LIST OF QUAD-TREE INDICES TO INDEX EACH NODE OF THE OCTREE UP TO THE MAXIMUM RESOLVED DEPTH, EACH QUAD-TREE INDEX OF THE QUAD-TREE INDICES COMPRISING AN ORDERED LIST OF TUPLES, EACH TUPLE INDEXING A CORRESPONDING NODE OF THE OCTREE AND INCLUDING A CORRESPONDING DEPTH OF THE OCTREE, AN OFFSET INDICATING THE LOCATION OF THE CORRESPONDING NODE IN THE OCTREE AT THE CORRESPONDING DEPTH, AND A NON-LEAF NODE FLAG INDICATING WHETHER THE CORRESPONDING NODE IS A NON-LEAF NODE OR A LEAF NODE

Fig. 5

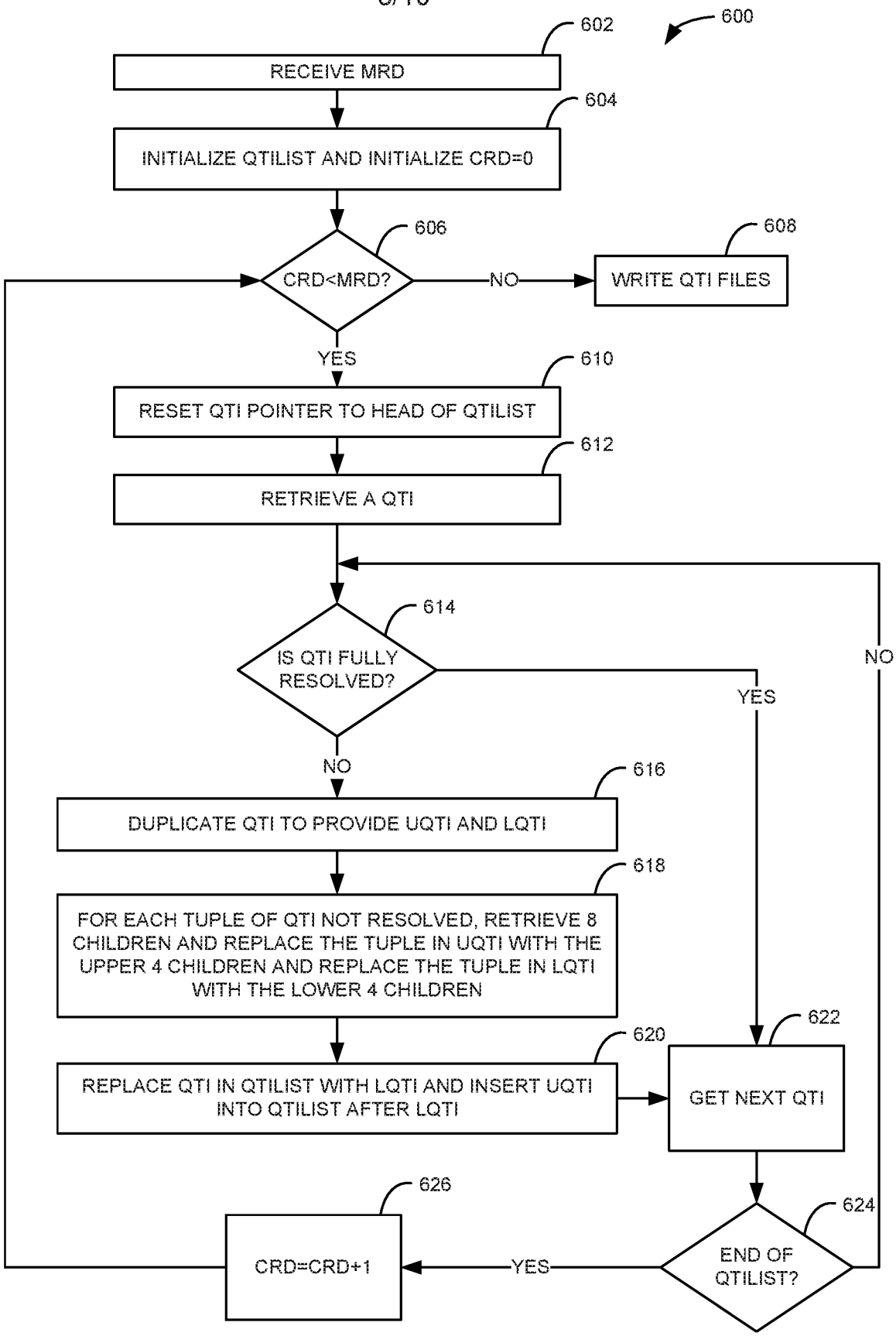
600
602
RECEIVE MRD
604
INITIALIZE QTILIST AND INITIALIZE CRD=0
606
CRD<MRD?
NO
608
WRITE QTI FILES
YES
610
RESET QTI POINTER TO HEAD OF QTILIST
612
RETRIEVE A QTI
614
IS QTI FULLY RESOLVED?
YES
NO
NO
616
DUPLICATE QTI TO PROVIDE UQTI AND LQTI
618
FOR EACH TUPLE OF QTI NOT RESOLVED, RETRIEVE 8 CHILDREN AND REPLACE THE TUPLE IN UQTI WITH THE UPPER 4 CHILDREN AND REPLACE THE TUPLE IN LQTI WITH THE LOWER 4 CHILDREN
620
REPLACE QTI IN QTILIST WITH LQTI AND INSERT UQTI INTO QTILIST AFTER LQTI
622
GET NEXT QTI
624
END OF QTILIST?
YES
626
CRD=CRD+1

Fig. 6

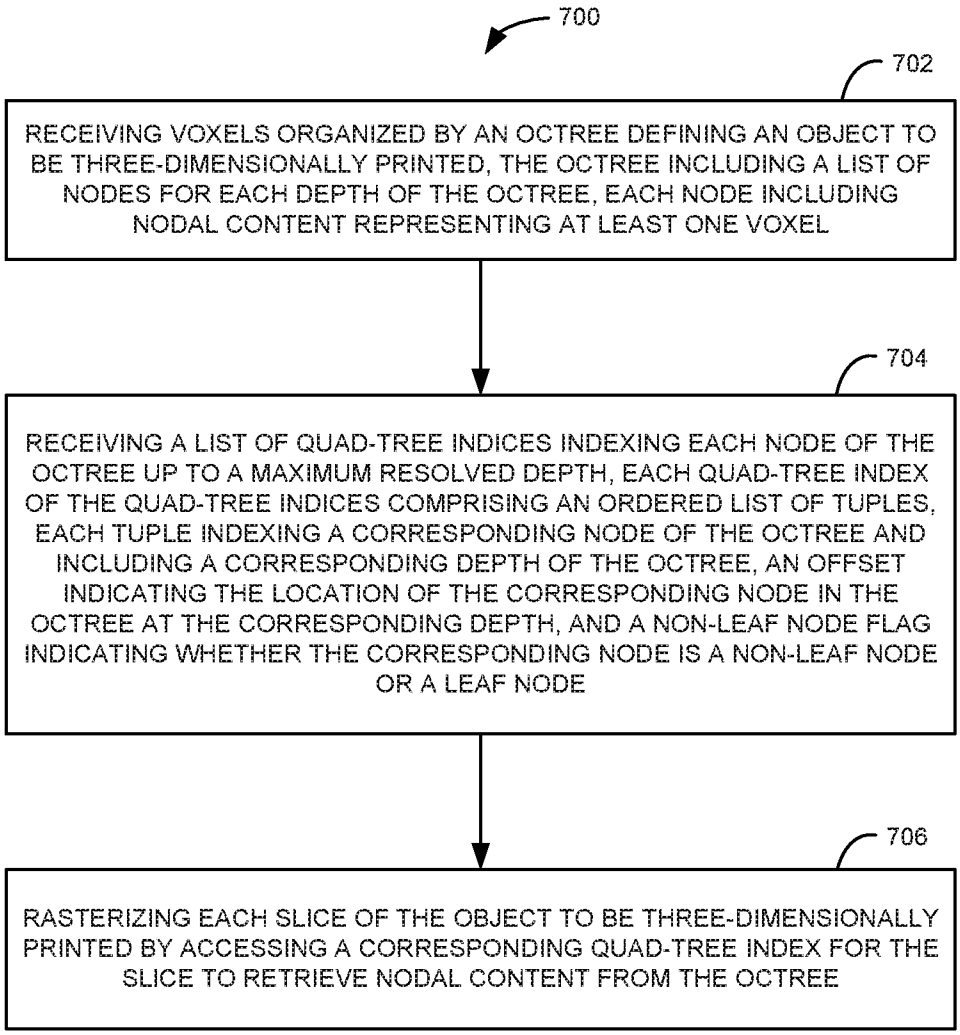
700
702
RECEIVING VOXELS ORGANIZED BY AN OCTREE DEFINING AN OBJECT TO BE THREE-DIMENSIONALLY PRINTED, THE OCTREE INCLUDING A LIST OF NODES FOR EACH DEPTH OF THE OCTREE, EACH NODE INCLUDING NODAL CONTENT REPRESENTING AT LEAST ONE VOXEL
704
RECEIVING A LIST OF QUAD-TREE INDICES INDEXING EACH NODE OF THE OCTREE UP TO A MAXIMUM RESOLVED DEPTH, EACH QUAD-TREE INDEX OF THE QUAD-TREE INDICES COMPRISING AN ORDERED LIST OF TUPLES, EACH TUPLE INDEXING A CORRESPONDING NODE OF THE OCTREE AND INCLUDING A CORRESPONDING DEPTH OF THE OCTREE, AN OFFSET INDICATING THE LOCATION OF THE CORRESPONDING NODE IN THE OCTREE AT THE CORRESPONDING DEPTH, AND A NON-LEAF NODE FLAG INDICATING WHETHER THE CORRESPONDING NODE IS A NON-LEAF NODE OR A LEAF NODE
706
RASTERIZING EACH SLICE OF THE OBJECT TO BE THREE-DIMENSIONALLY PRINTED BY ACCESSING A CORRESPONDING QUAD-TREE INDEX FOR THE SLICE TO RETRIEVE NODAL CONTENT FROM THE OCTREE

Fig. 7

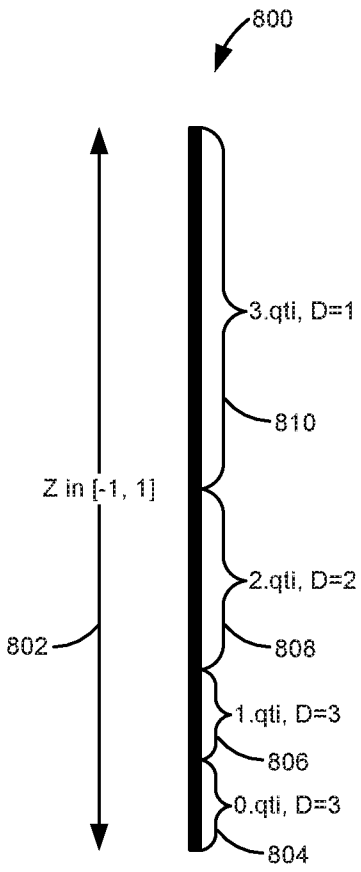

Fig. 8

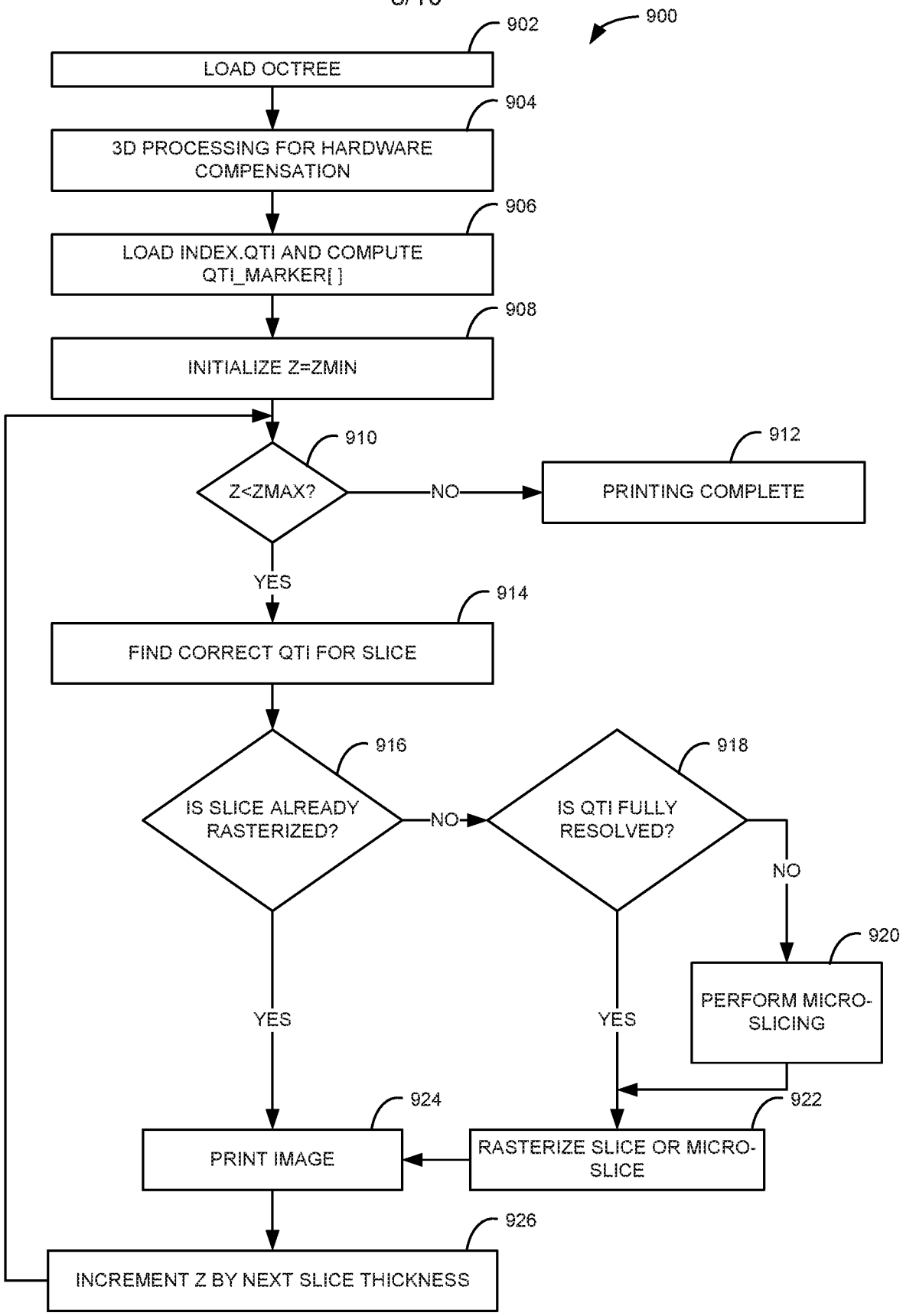
900
902
LOAD OCTREE
904
3D PROCESSING FOR HARDWARE COMPENSATION
906
LOAD INDEX.QTI AND COMPUTE QTI_MARKER[ ]
908
INITIALIZE Z=ZMIN
910
Z<ZMAX?
NO
912
PRINTING COMPLETE
YES
914
FIND CORRECT QTI FOR SLICE
916
IS SLICE ALREADY RASTERIZED?
NO
918
IS QTI FULLY RESOLVED?
NO
920
PERFORM MICRO-SLICING
YES
YES
924
PRINT IMAGE
922
RASTERIZE SLICE OR MICRO-SLICE
926
INCREMENT Z BY NEXT SLICE THICKNESS

Fig. 9

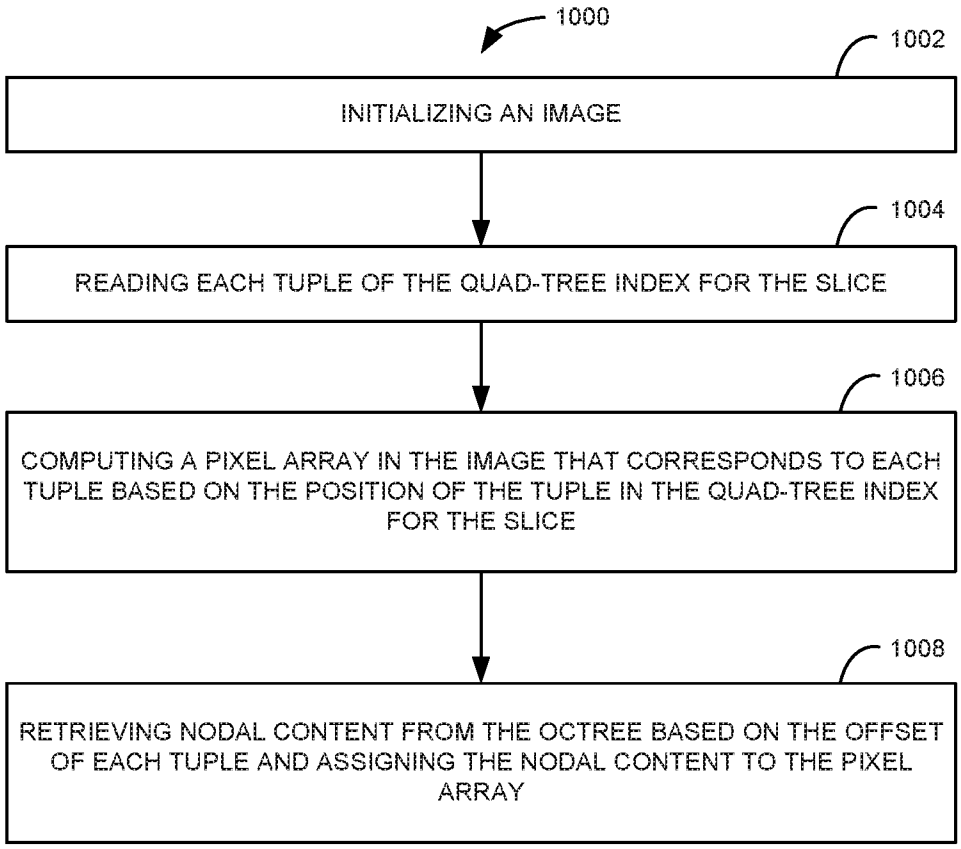
1000
1002
INITIALIZING AN IMAGE
1004
READING EACH TUPLE OF THE QUAD-TREE INDEX FOR THE SLICE
1006
COMPUTING A PIXEL ARRAY IN THE IMAGE THAT CORRESPONDS TO EACH TUPLE BASED ON THE POSITION OF THE TUPLE IN THE QUAD-TREE INDEX FOR THE SLICE
1008
RETRIEVING NODAL CONTENT FROM THE OCTREE BASED ON THE OFFSET OF EACH TUPLE AND ASSIGNING THE NODAL CONTENT TO THE PIXEL ARRAY

Fig. 10

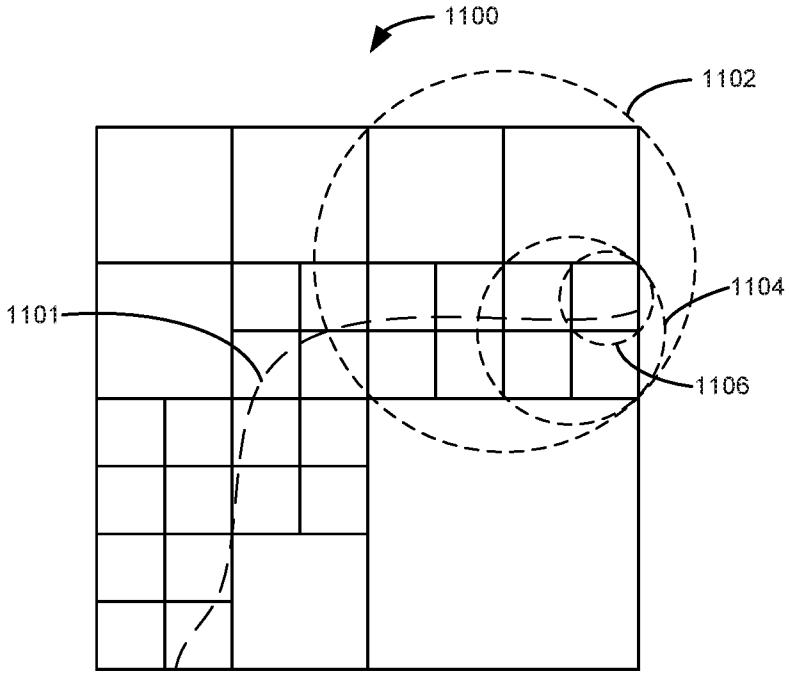
1100
1102
1101
1104
1106

Fig. 11

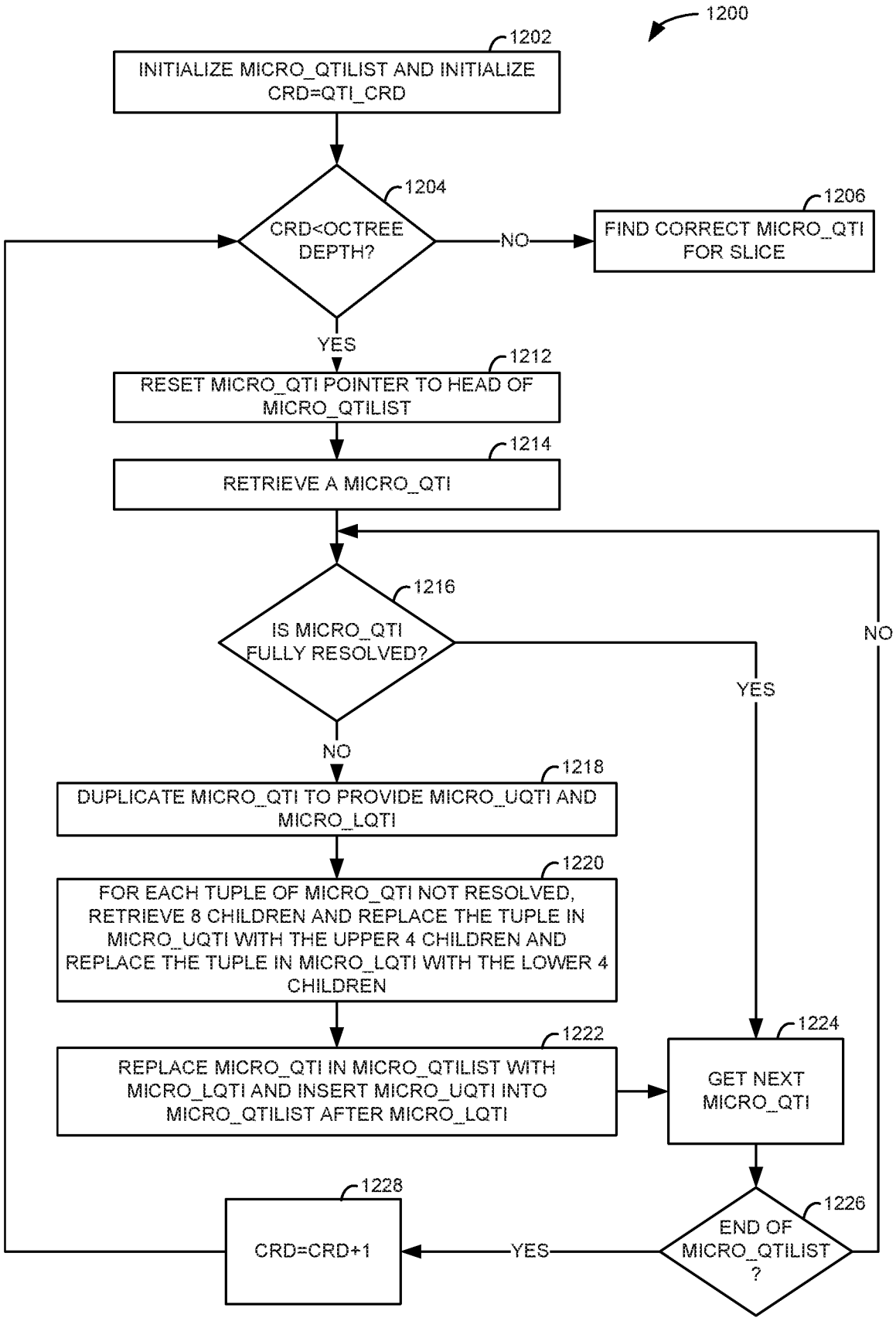
1200
1202
INITIALIZE MICRO_QTILIST AND INITIALIZE CRD=QTI_CRD
1204
CRD<OCTREE DEPTH?
NO
1206
FIND CORRECT MICRO_QTI FOR SLICE
YES
1212
RESET MICRO_QTI POINTER TO HEAD OF MICRO_QTILIST
1214
RETRIEVE A MICRO_QTI
1216
IS MICRO_QTI FULLY RESOLVED?
NO
YES
1218
DUPLICATE MICRO_QTI TO PROVIDE MICRO_UQTI AND MICRO_LQTI
1220
FOR EACH TUPLE OF MICRO_QTI NOT RESOLVED, RETRIEVE 8 CHILDREN AND REPLACE THE TUPLE IN MICRO_UQTI WITH THE UPPER 4 CHILDREN AND REPLACE THE TUPLE IN MICRO_LQTI WITH THE LOWER 4 CHILDREN
1222
REPLACE MICRO_QTI IN MICRO_QTILIST WITH MICRO_LQTI AND INSERT MICRO_UQTI INTO MICRO_QTILIST AFTER MICRO_LQTI
1224
GET NEXT MICRO_QTI
1226
END OF MICRO_QTILIST ?
YES
1228
CRD=CRD+1

Fig. 12

INTERNATIONAL SEARCH REPORT

International application No.

PCT/US 2016/043996

A. CLASSIFICATION OF SUBJECT MATTER

G06T 17/00 *(2006.01)*
G06T 9/40 *(2006.01)*
B33Y 50/00 *(2015.01)*

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F 1/00-17/50, G06T 1/00-17/30, G06Q 10/00-50/34, H04N 9/00-9/38, 19/00-19/98, G09G 5/00-5/42, B29C 67/00-67/24, B33Y 50/00

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

PatSearch (RUPTO internal), USPTO, PAJ, K-PION, Esp@cenet, Information Retrieval System of FIPS

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X Y	US 2011/0087350 A1 (3D M.T.P. LTD) 14.04.2011, abstract, paragraphs [0025], [0050]-[0052], [0062], [0088], [0091], [0097], [0101], [0106], [0109], [0113]-[0115], [0123]-[0124], [0128]-[0129], [0131]	1-2, 5-6, 11, 13-15 3, 4, 7-10, 12
Y	US 2010/0082703 A1 (MICROSOFT CORPORATION) 01.04.2010, paragraphs [0045]-[0051], [0058]-[0060], [0067], [0070], [0082], [0085]	3, 4, 7-10, 12
A	US 2003/0197698 A1 (RONALD N. PERRY et al.) 23.10.2003	1-15
A	EP 1574996 A2 (SAMSUNG ELECTRONICS CO., LTD) 14.09.2005	1-15

☐ Further documents are listed in the continuation of Box C.

☐ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
06 February 2017 (06.02.2017)	09 February 2017 (09.02.2017)
Name and mailing address of the ISA/RU: Federal Institute of Industrial Property, Berezhkovskaya nab., 30-1, Moscow, G-59, GSP-3, Russia, 125993 Facsimile No: (8-495) 531-63-18, (8-499) 243-33-37	Authorized officer V. Zhakovich Telephone No. (499) 240-25-91