ISSN 2007-9737

Compact Union of Disjoint Boxes: An Efficient Decomposition Model for Binary Volumes

Irving Cruz-Matías[1], Dolors Ayala[2]

[1] Universidad de Monterrey, San Pedro Garza García, Mexico

[2] Universitat Politècnica de Catalunya, Barcelona, Spain

irving.cruz@udem.edu, dolorsa@lsi.upc.edu

Abstract. This paper presents in detail the Compact Union of Disjoint Boxes (CUDB), a decomposition model for binary volumes that has been recently but briefly introduced. This model is an improved version of a previous model called Ordered Union of Disjoint Boxes (OUDB). We show here, several desirable features that this model has versus OUDB, such as less unitary basic elements (boxes) and thus, a better efficiency in some neighborhood operations. We present algorithms for conversion to and from other models, and for basic computations as area (2D) or volume (3D). We also present an efficient algorithm for connected-component labeling (CCL) that does not follow the classical two-pass strategy. Finally we present an algorithm for collision (or adjacency) detection in static environments. We test the efficiency of CUDB versus existing models with several datasets.

Keywords. Binary volumes, orthogonal polyhedra, connected-component labeling, collision detection.

1 Introduction

The model used to represent digital binary volumes (or 3D binary images) is one of the most important research topic in computer graphics, looking for a better compression for storage, analysis or visualization purposes. Moreover, there are several demanding neighborhood operations on binary volumes such as connected-component-labeling (CCL), connectivity, collision detection, between others, where the performance variability of the existing algorithms is mainly caused by the number of basic geometric elements to analyze (voxels, triangles, planes, vertices, etc.).

In most of the reported literature, the operations to study binary volumes are performed directly on the classical voxel model [31, 41]. However, in the field of volume analysis and visualization, several alternative models have been devised for specific purposes. For instance, Hierarchical structures such as octrees and kd-trees have been used for Boolean operations [46], CCL [17], and thinning [40, 58]. Octrees are used as a means of compacting regions and getting rid of the large amount of empty space in the extraction of isosurfaces [57]. Kd-trees have been used to extract two-manifold isosurfaces [24].

There are other models that store surface voxels, thereby gaining storage and computational efficiency. The semi-boundary representation affords direct access to surface voxels and performs fast visualization and manipulation operations [25]. Certain methods of erosion, dilation and CCL use this representation [20, 51].

On the other hand, a binary voxel model represents an object as the union of its foreground voxels and its continuous analog is an orthogonal pseudo-polyhedra (OPP) [33]. OPP have been used in 2D to represent the extracted polygons from numerical control data [39]. Some 3D applications of OPP are: general computer graphics applications such as geometric transformations and Boolean operations [10, 19], skeleton computation (instead of iterative peeling techniques) [18, 37], and orthogonal hull computation [8, 9]. OPP have been also used in theory of hybrid systems to model the solutions of reachable states [10, 16].

The Extreme Vertices Model (EVM) and the Ordered Union of Disjoint Boxes (OUDB) [1, 3] represent OPP in a compact way. EVM stores only a sorted subset of vertices of the OPP boundary, whereas OUDB keeps a sorted list of boxes that compose the whole object. EVM has been used to prove the suitability of OPP as geometric bounds in CSG [2, 1], and for many other 3D applications such as: erosion and dilation operations [42], skeleton computation [4], virtual porosimetry without skeleton computation [44], biomaterials structural parameters computation [54] and model simplification [15]. OUDB has been used to perform CCL [5, 42] and also for biomaterials structural parameters computation such as connectivity [7] and center of gravity [6].

In this paper we present in detail the Compact Union of Disjoint Boxes (CUDB), a representation model for OPP, and thus for binary volumes, that has been recently introduced, but without going into details of its implementation [14, 44]. CUDB is a special kind of cell decomposition representation which performs a spatial partition of the volume in a non-hierarchical sweep-based way consisting of a set of disjoint boxes. This model is an improved version the OUDB. We test and report the performance of CUDB and also present new algorithms for CCL and collision detection.

The paper is arranged as follows. The next section reviews related work. Section 3 introduces CUDB using the object-oriented paradigm presenting algorithms for conversion to and from other models, and for basic computations as area and volume. Section 4 presents a new CUDB-based algorithm for CCL based on the detection of connected components in graph theory. Section 5 presents a collision detection algorithm for scenarios with multiple CUDB-represented objects. Sections 6 shows the performance of all the proposed algorithms by discussing experimental results. Finally, Section 7 concludes the paper and outlines future work.

2 Background and Related Work

The classical voxel model is based on a regular decomposition of the 3D space into a set of identical cubic cells called voxels. In a voxel model, voxels are all the same size and their edges are parallel to the main axes. Formally, a voxel model V of size $n_x \times n_y \times n_z$ is defined as:

$$V = \{v_{i,j,k} \mid 0 \leq i \leq n_x, 0 \leq j \leq n_y, 0 \leq k \leq n_z\},$$

where $v_{i,j,k}$ is a voxel in location (i, j, k).

From V, all geometric and topological information can be obtained. On each voxel $v_{i,j,k}$, there is a set of associated values. For a binary voxel model, the associated value of its voxels is limited to $v_{i,j,k} \in \{0, 1\}$, where 0 corresponds to the background and 1 to the foreground.

Algorithms for the voxel model are straightforward to implement. However, just because of the size of the source models, it has the drawbacks of the loss of geometric information and high memory and computational power requirements [32]. To reduce the memory footprint and the computation time of the algorithms, many alternative models have been proposed such as hierarchical data structures and boundary representations.

Hierarchical structures made recursive subdivisions of the volume. In bintrees [47] an axis-aligned hyperplane intersects the interior of the volume producing two equal parts. The Binary Space Partition (BSP) trees [50] are similar to bintrees except that the position and direction of the subdivision hyperplanes are usually selected following an optimization heuristic. Octrees [11], like bintrees, begin with an initial volume, but it create eight sub-volumes of equal size. And kd-trees [24] are a special case of bintree, where each level is asymmetrically divided in alternate directions according to a discriminant (e.g., X, Y or Z-coordinate). In this structures, volume subdivision is repeated until a certain satisfactory level of refinement or until achieving the maximum level of recursion.

When working with binary volume datasets, it is not necessary to keep the exact scanned density values. Therefore, boundary models are also adequate to represent these kind of datasets. The basic model for representing the polygonal surface of an object is the Boundary Representation (B-Rep) model [38] that keeps explicitly all of the relationships between geometric elements such as vertices, edges and faces. However, there are many proposed models for representing the object boundary in a more compact way.

Semi-boundary (SB) [52] is a data structure which stores only the boundary voxels of the volume keeping the information of the interior voxels in an implicit way. Shell representation [53] has the same indexing scheme used in the SB representation, but the set of voxels, which belongs to the list that contains all the boundary voxels of the volume, was redefined in order to represent objects with fuzzy boundaries. Cell-boundary [34] is also a very similar representation to SB, which consists of a set of boundary cells with their voxel configurations, so, the points of the sample in SB become vertices of the cells in the cell-boundary representation.

2.1 EVM and OUDB models

The Extreme Vertices Model (EVM) [2, 3] is a very concise representation scheme in which any OPP can be described using a subset of its vertices: the extreme vertices. EVM is actually a complete solid model with very fast Boolean operations, it is an implicit B-Rep model, i.e., all the geometry and topological relations concerning faces, edges and vertices of the represented OPP can be obtained from the EVM [43] and therefore represents OPP unambiguously [55].

Let Q be a finite set of points in $\mathbb{R}^3$, the ABC-sorted set of Q is the set resulting from sorting Q according to A-coordinate, then to B-coordinate, and then to C-coordinate. Let P be an OPP, a *brink* is the maximal uninterrupted segment built out of a sequence of collinear and contiguous two-manifold edges of P and its ending vertices are called *extreme vertices* (EV). An OPP can be represented in a concise way with the ABC-sorted set of its EV and such representation scheme is called EV.

The Ordered Union of Disjoint Boxes (OUDB) [1, 42] is a special kind of spatial partitioning representation derived from EVM, where an OPP is decomposed in a list of disjoint boxes. EVM can

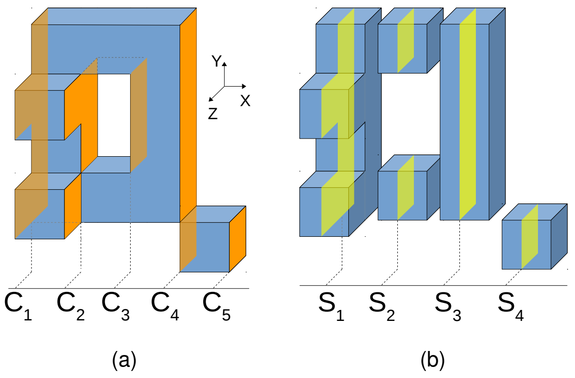


Fig. 1. (a) An orthogonal polyhedron with 5 cuts. (b) Its sequence of 4 prisms with the representative sections (X direction)

be obtained from the voxel model, in turn, OUDB is obtained from EVM. The conversions algorithms have been published [42]. To introduce later the CUDB model, we first briefly introduce OUDB.

Let P be an OPP and Π_c a plane whose normal is parallel, without loss of generality, to the X axis, intersecting it at $x = c$, where c ranges from $-\infty$ to $+\infty$. Then, this plane sweeps the whole space as c varies within its range, intersecting P at certain intervals. Let us assume that this intersection changes at $c = C_1, ..., C_n$. More formally, $P \cap \Pi_{C_i - \delta} \neq P \cap \Pi_{C_i + \delta}, \forall i = 1, ..., n$, where δ is an arbitrarily small quantity. Then, $C_i(P) = P \cap \Pi_{c_i}$ is called a *cut* of P and $S_i(P) = P \cap \Pi_{C_s}$, for any C_s such that $C_i < C_s < C_{i+1}$, is called a *section* of P.

Fig. 1 shows an OP with its *cuts* and *sections* perpendicular to the X axis. Since we work with bounded regions, $S_0(P) = \emptyset$ and $S_n(P) = \emptyset$, where n is the total number of *cuts* along a given coordinate axis.

Cuts and sections are orthogonal polygons embedded in the 3D space. For each main direction, sections can be computed from cuts and cuts from sections by applying simple XOR operations. These operations are actually performed on the projections of cuts and sections onto the main plane parallel to them. From now on, we thus call the projection of the ith cut and ith section onto the main plane parallel to them C_i and S_i respectively.

The two following expressions relate cuts and sections:

$$S_i(P) = S_{i-1}(P) \otimes C_i(P), \forall i = 1 \ldots n-1, \quad (1)$$

$$C_i(P) = S_{i-1}(P) \otimes S_i(P), \forall i = 1 \ldots n, \quad (2)$$

where $\otimes$ denotes the regularized XOR operation. An OP can be represented with a sequence of orthogonal prisms represented by their section (see Fig. 1(b)). Moreover, if we apply the same reasoning to the representative section of each prism, an OP can be represented as a sequence of boxes.

The OUDB model represents an OPP with such a sequence of boxes. OUDB is axis-aligned like octrees and bintrees, but the partition is done along the object geometry as in BSP. Depending on the order of the axes along which we choose to split the data, an OPP P can be decomposed into six different ABC-OUDB, i.e., P is subdivided by planes perpendicular to the A-axis first, and then by planes perpendicular to the B-axis. Their corresponding sets of disjoint boxes are generally different. Fig. 2 shows the possible OUDB decompositions for the OPP in Fig. 1(a).

2.2 Connected Component Labeling

Connected Component Labeling (CCL) is a very important operation for managing volume datasets where multiple disconnected components that compose the volume need to be identified. Traditional voxel-based methods have been widely used [45].

The typical implementation of many approaches, even of the recent ones [27, 28, 35] is based in the classical two-pass strategy [45]: the *labeling pass* and the *renumbering pass*. In short, in the *labeling pass* all elements are scanned and labeled according to their already labeled neighbors. Some labeling ambiguities can be produced in this step which are properly registered in a set of equivalence classes. Then, the renumbering pass solves these ambiguities and the elements are relabeled.

There are other labeling algorithms for special volume representations as hierarchical structures or semi-boundary representations looking for a

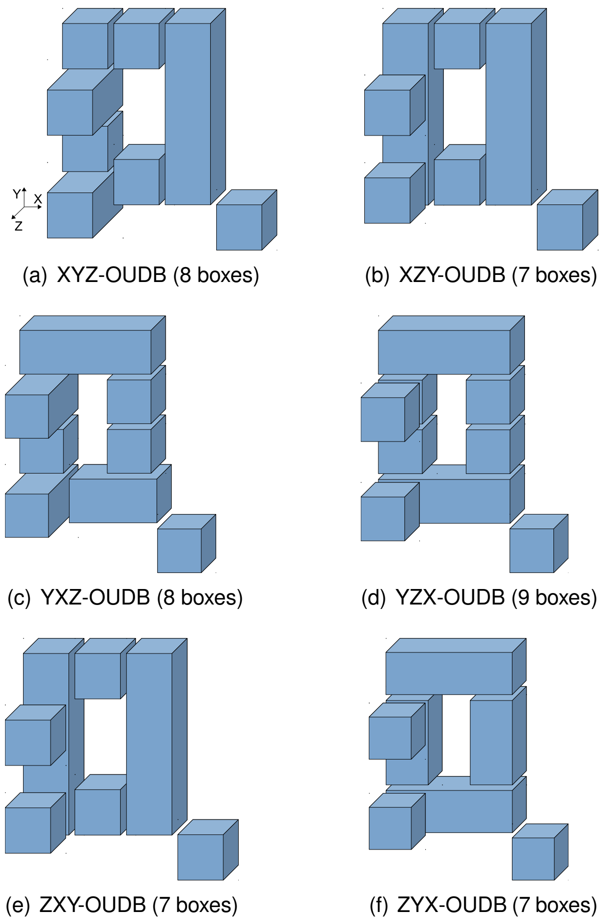


Fig. 2. The six possible OUDB decompositions for the OPP in Figure 1(a)

better efficiency of CCL. In this sense, OUDB has been proved to be efficient for CCL [42, 43]. With regard to semi-boundary representations, it has been concluded that CCL is better in OUDB than in semi-boundary representations when the number of boxes in the OUDB is less than the number of boundary voxels, which generally occurs.

In the OUDB-CCL process, the traversal of the boxes is performed orderly, so, checking the neighborhood of the current box involves those boxes in the immediate previous B-slice and those boxes in the immediate previous A-slice. An improvement of the OUDB-based CCL has been already proposed [5], where the so-called OUDB-extended is computed, which allows jumping directly to the required box that needs to be tested, instead of querying and skipping several intermediate boxes.

Because of its similarity to OUDB, the two-pass strategy for CCL has been also applied in CUDB [44]. As generally the CUDB-represented object contains less boxes than its respective OUDB-representation, this approach is faster. However, the main drawback of all the aforementioned approaches is the large size of the equivalence table, because they need one entry per each new detected label.

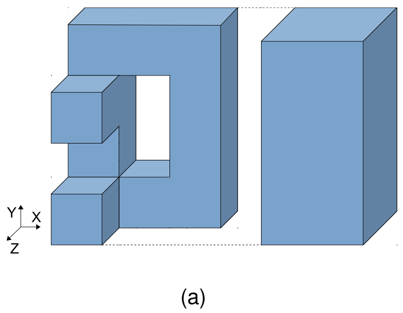
2.3 Collision Detection

Collision detection is an important characteristic in computer graphics and simulation. Sometimes we want to determine if two or more objects collide or are adjacent. In collision detection [30] when exact accuracy is not required, typical bounding volumes like axis-aligned bounding boxes (AABB) [59], spheres [22], oriented polytopes [13] or hybrid bounding volumes are used [56].

However, when accuracy is important, a thorough analysis of the contact between the involved objects needs to be done. In complex scenes there might be several objects interacting. In such cases, an early detection phase can be applied to discard collisions between objects which are not close enough. As bounding volumes present simpler features, they are used as a preliminary test for collisions. The absence of bounding volume collisions guarantees the absence of collision between some objects and thus it significantly reduces the number of objects to be checked. Sweep and prune algorithms [12, 23] sort the objects according to the lower and upper bounds of their bounding volumes, and when a pair of objects are very close, it can be tested to exact collision.

3 Compact Union of Disjoint Boxes

Like OUDB, CUDB is also a union of disjoint boxes but a more compact one as several contiguous boxes are merged into one in several parts of the model. Let P be an OPP, to obtain the



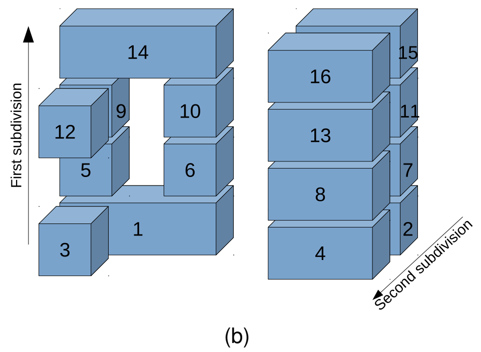


Fig. 3. (a) An OPP. (b) YZX-OUDB with 16 boxes labeled according to their YZX-coordinates

ABC-OUDB model, P is subdivided by planes perpendicular to the A-axis first, and then by planes perpendicular to the B-axis, at each cut C_i of P. Thus, every C_i splits all the geometry of P along the corresponding plane, and therefore some local regions of P, with which C_i actually has no relationship, are further unnecessarily divided. Fig. 3 shows this situation for the YZX-OUDB of an OPP P, where some cuts force unnecessary divisions. For OUDB this constraint is mandatory to keep sorted the resulting boxes. However, in order to subdivide just the pieces of P related with the cut which induces the splitting, this constraint can be relaxed.

Formally, let P be an OPP. The $CUDB(P)$ can be obtained by merging boxes in several parts of the corresponding $OUDB(P)$. Then, this model is the set of boxes obtained according to the next properties:

1. Let β_1 and β_2 be two adjacent boxes of $OUDB(P)$ in B-direction, and let $\overline{\beta_1}^B$ and $\overline{\beta_2}^B$ be their projections respectively onto the plane

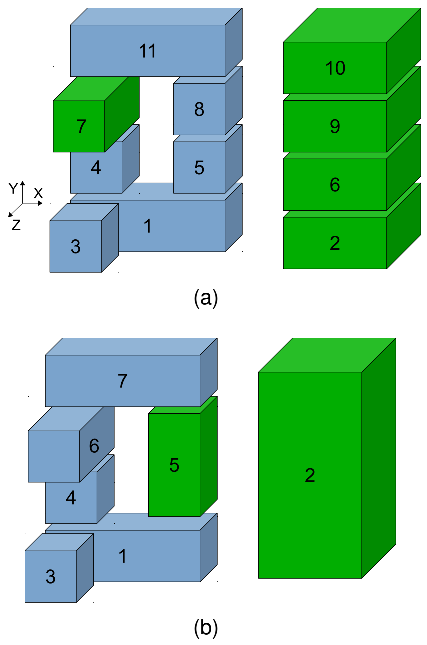


Fig. 4. (a) Result after first merging in Z-direction. (b) Resulting YZX-CUDB with 7 boxes after merging in Y-direction

perpendicular to the B-axis, then β_1 and β_2 can be merged as a single box if $\overline{\beta_1}^B = \overline{\beta_2}^B$.

2. Let β_1 and β_2 be two adjacent boxes of $OUDB(P)$ in A-direction, and let $\overline{\beta_1}^A$ and $\overline{\beta_2}^A$ be their projections respectively onto the plane perpendicular to the A-axis, then β_1 and β_2 can be merged as a single box if $\overline{\beta_1}^A = \overline{\beta_2}^A$. Note that A-direction in this property is different of B-direction of the first property.

The first property merges all unnecessary subdivisions along B-axis. Following with the previous example, Fig. 4(a) shows that the pairs of boxes (2, 4), (7, 8), (9, 12), (11, 13) and (15, 16) depicted in Fig. 3(b) can be merged applying this property for the Z-axis.

The second property merges the remaining unnecessary subdivisions along A-axis. Following with the same example, the resulting model depicted in Fig. 4(b) shows that applying this second property for the Y-axis, not only the pairs of boxes (5, 8), but also the set of boxes (2, 6, 9, 10) depicted in Fig. 4(a) can be merged.

Then, the CUDB-representation of an OPP P, is the set of disjoint boxes of the corresponding OUDB, conveniently reduced by applying the two previous merging properties. Let β_i be a box in $CUDB(P)$:

$$P = \bigcup_{i=1}^{n_b} \beta_i(P) \qquad (3)$$

where n_b is the number of boxes, which is less or equal than the number of boxes in $OUDB(P)$.

Like OUDB, there are six different ABC-CUDB models for a given OPP and the number of obtained boxes depends on the order of the axes along which we choose to split the data, but we cannot know it a priori. Boxes in CUDB are sorted according to its coordinate A, then to coordinate B, and finally to coordinate C of its lower bound.

Although the implicit order among boxes in OUDB that defines their adjacency is lost, preserving the adjacency information in the CUDB model is easy with a tiny storage effort. Each box has neighboring boxes in only two orthogonal directions: A and B-direction, and for each one there are two opposite senses, so, four arrays of pointers to the neighboring boxes (two for each direction) are enough to preserve the adjacency information that is required for future operations. We define these arrays as A-backward neighbors (ABN), A-forward neighbors (AFN), B-backward neighbors (BBN) and B-forward neighbors (BFN).

CUDB has been implemented as an object with a set of properties and methods. Next, we describe the CUDB object and the algorithms for conversion to and from EVM.

3.1 CUDB Structure

Let β and Q be a box and a CUDB object respectively:

Box properties:

— point3D V_0, V_1: point3D objects representing the two diagonally opposed vertices of β, the ones with lowest and highest coordinate values.

— array ABN, AFN, BBN, BFN: Arrays of pointers to the neighbors of β.

— integer $label$: The label of β.

Box methods:

— Box(point3D V_0, point3D V_1): Constructor for a new box β with vertices V_0 and V_1. $label$ is set to 0 (undefined) and arrays ABN, AFN, BBN and BFN to $\{\emptyset\}$.

— $getLabel()$: Returns the label of β.

— $setLabel$(integer $label$): Sets the label of β as $label$.

CUDB properties:

— boolean $nonManifolds$: Flag to indicate if the boxes adjacency allows non-manifold configurations.

— array $boxes$: Vector containing the sorted boxes of Q.

— dimType dim: Dimension of Q. dimType={0D, 1D, 2D, 3D}

— integer $nBoxes$: Number of boxes in Q.

— sortingType $sort$: Sorting of Q. sortingType={XYZ, XZY, YXZ, YZX, ZXY, ZYX}

CUDB methods:

— $CUDB$(dimType dim, sortingType $sort$, boolean non_m): Constructor for a CUDB object Q of dimension dim and sorting $sort$, and flag $nonManifolds$ set as non_m.

— $getBox$(integer id): Returns the box at position id in the array $boxes$.

— $getDimension()$: Returns the dimension of Q.

— $getNBoxes()$: Returns the number of boxes of Q.

— $getNextBox$(Box β): Returns the next box to β in the array $boxes$.

— $getSorting()$: Returns the sorting of Q.

— $insertBox$(Box β): Inserts the box β into Q, at the end of the array $boxes$.

3.2 CUDB Computation

In order to obtain the CUDB-representation it is not necessary to have or compute the OUDB model. CUDB can be computed directly from the EVM, and the merging of boxes is performed on the fly. The strategy is similar to the process to compute OUDB [42]. Cuts of the EVM-represented OPP are obtained sequentially, and sections are computed from them. When this process is performed in 2D, the corresponding sections result in the boxes of the OUDB model.

For the EVM to CUDB conversion method, the same set of boxes is computed on the fly and a box is stored in the CUDB model if it is not possible to merge it with previously computed boxes applying the aforementioned Properties 1 and 2.

For a given box, the set of previous boxes that have to be considered for merging with it are those boxes belonging to the previous A-slice, which can be adjacent in A-direction, and those boxes belonging to the previous B-slice, which can be adjacent in B-direction. To facilitate this process, temporary arrays of box pointers of the current and previous slices are maintained. The corresponding algorithm is detailed next.

As most of the algorithms dealing with EVM, the corresponding conversion algorithm is also recursive over the dimension. The main function $EVMtoCUDB()$ (Algorithm 1) receives an EVM-represented OPP P and a flag to indicate if the adjacency relationship among boxes allows non-manifold configurations, and returns the CUDB-represented OPP Q containing the neighborhood information. This function initializes the temporary arrays of box pointers $prevBBoxes$, $currentBBoxes$, $prevABoxes$ and $currentABoxes$, which are defined as global variables throughout the whole conversion process, and starts the recursion by calling the function $doConversion()$ (Algorithm 2) with the original EVM-represented object P.

In function $doConversion()$, when dimension is 3D, the object is split at each cut in A-direction obtaining a set of 3D A-slices, $\Upsilon_A = \{\Upsilon_A^1, \Upsilon_A^2, \ldots, \Upsilon_A^n\}$, where n is the number of A-slices. Then the algorithm applies recursively to the 2D section representing each slice, which is split at every internal cut in B-direction obtaining a set of 2D B-slices, $\Upsilon_B = \{\Upsilon_B^1, \Upsilon_B^2, \ldots, \Upsilon_B^m\}$, where m is the number of B-slices represented by their 1D sections, which are composed by a set of collinear brinks in C-direction. Each of these brinks defines a box. Then, each 2D slice Υ_B^i defines one o more boxes, and each 3D slice Υ_A^j contains all the boxes defined in its 2D slices.

Algorithm 1: EVMtoCUDB(**Input** P:EVM, **Input** nonManifolds:boolean, **Output** Q:CUDB)

```
$dim \leftarrow P.getDimension()$;
$Q =$
  $CUDB(dim, P.getSorting(), nonManifolds)$;
$prevBBoxes \leftarrow \{\emptyset\}$; $currentBBoxes \leftarrow \{\emptyset\}$;
$prevABoxes \leftarrow \{\emptyset\}$; $currentABoxes \leftarrow \{\emptyset\}$;
$doConversion(P, Q, dim, \emptyset, \emptyset)$; // First call
```

In the base case, when dimension is 1D, each brink in the current slice Υ_B^i results in a box, which is inserted into $currentBboxes$. Boxes in a 2D slice Υ_B^i can be merged with boxes in the previous slice Υ_B^{i-1}, then, in the backtracking step of the recursion when dimension is 2D, function $mergeB()$ (Algorithm 3) is called, which compares all boxes β_1 in $currentBboxes$ with all boxes β_2 in Υ_B^{i-1} (stored in $prevBBoxes$) for merging. In this process, when merging property 1 is accomplished, $\beta_2 = \beta_2 \cup \beta_1$, and it is inserted into a array called $activeBoxes$. Otherwise, β_1 is inserted into $currentABoxes$ and $activeBoxes$. When the process finishes, the array $activeBoxes$ becomes $prevBBoxes$ in order to be compared with boxes in Υ_B^{i+1} in the next call.

Similar to the merging case in B-direction, boxes in a 3D slice Υ_A^j can be merged with boxes in the previous slice Υ_A^{j-1}. Once all the boxes of the current slice Υ_A^j have been computed and conveniently merged in B-direction, they are in $currentAboxes$. Then, in the backtracking step of the recursion when dimension is 3D, function $mergeA()$ (Algorithm 4) is called, which compares all boxes β_1 in $currentAboxes$ with all boxes β_2 in Υ_A^{j-1} (stored in $prevAboxes$) for merging. The steps in this function are quite similar to those in function $mergeB()$, but in this case, the merged boxes are finally inserted into the CUDB model Q.

Algorithm 2: doConversion(**Input** P:EVM, **Input/Output** Q:CUDB, **Input** dim:dimType, **Input** V_0, V_1:point3D)

```
if $dim$ =1D then
    foreach brink $br \in P$ do
        $V_0.C, V_1.C \leftarrow br.readBrink()$;
        $\beta = Box(V_0, V_1)$;
        Add $\beta$ to $currentBBoxes$;
    end
else // $dim$ =2D or 3D
    $Sec \leftarrow \emptyset$;
    $Cut, coordIni \leftarrow P.getNextCut()$;
    $Sec \leftarrow Sec \otimes Cut$;
     $Cut, coordFin \leftarrow P.getNextCut()$;
    while $Cut \neq \emptyset$ do
        if $dim$ =3D then
            $V_0.A \leftarrow V_1.A$; $V_1.A \leftarrow coordFin$;
        else // $dim$ =2D
            $V_0.B \leftarrow V_1.B$; $V_1.B \leftarrow coordFin$
        end
        $doConversion(Sec, Q, dim-1, V_0, V_1)$;
        if $dim$ =3D then
            $mergeA()$;
        else // $dim$ =2D
            $mergeB()$;
        end
        $Sec \leftarrow Sec \otimes Cut$;
         $Cut, coordFin \leftarrow P.getNextCut()$;
    end
end
```

Note that, the adjacency information of boxes (ABN, AFN, BBN, BFN) is computed on the fly when tests for merging are performed.

3.3 CUDB to EVM Conversion

As in OUDB, all of the boxes in CUDB are disjoint, and according to the following EVM property:

— Let P and Q be two OPP such that $P \cap^* Q = \emptyset$, having $EVM(P)$ and $EVM(Q)$ as their respective models, then $EVM(P \cup^* Q) = EVM(P) \otimes^* EVM(Q)$.

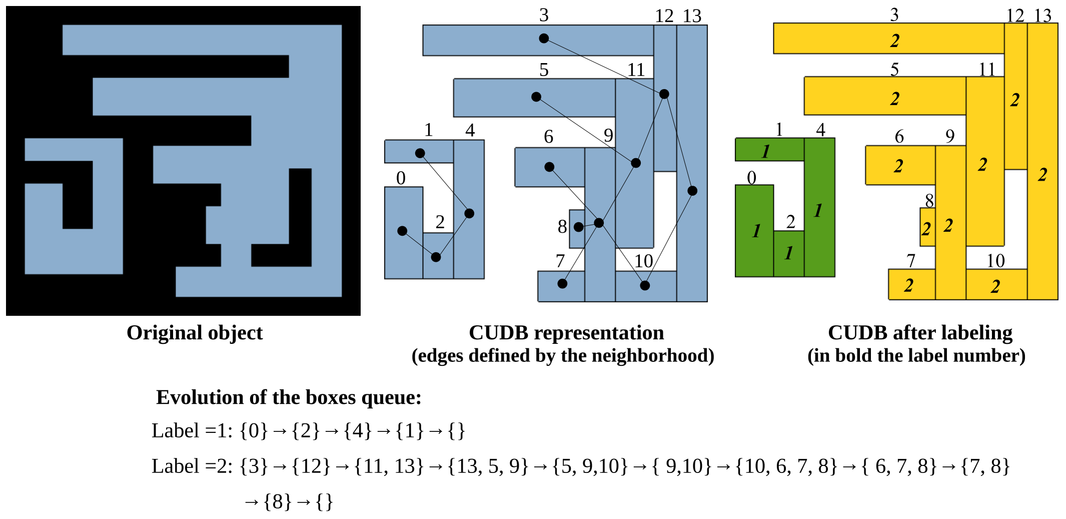

3
12 13
5
11
1 4
6
9
0
2
8
7
10
2
1
Original object
CUDB representation
(edges defined by the neighborhood)
CUDB after labeling
(in bold the label number)
Evolution of the boxes queue:
Label =1: {0}→{2}→{4}→{1}→{}
Label =2: {3}→{12}→{11, 13}→{13, 5, 9}→{5, 9,10}→{ 9,10}→{10, 6, 7, 8}→{ 6, 7, 8}→{7, 8}
→{8}→{}

Fig. 5. 2D example of CUDB-CCL

Algorithm 3: mergeB()

```
$activeBoxes \leftarrow \{\emptyset\}$;
foreach $\beta_1 \in currentBBoxes$ do
    foreach $\beta_2 \in prevBBoxes$ do
        if $\overline{\beta_1}^B = \overline{\beta_2}^B$ then // Merging property 1
            $\beta_2 \leftarrow \beta_1 \cup \beta_2$;
            Add $\beta_2$ to $activeBoxes$;
            break;
        else if $\overline{\beta_1}^B \cap \overline{\beta_2}^B \neq \emptyset$ then
            Add $\beta_2$ to $\beta_1.BBN$;
        end
    end
    if $\beta_1$ was not merged then
        Add $\beta_1$ to $currentABoxes$;
        Add $\beta_1$ to $activeBoxes$;
        Add $\beta_1$ in BFN of each box in $\beta_1$.BBN;
    end
end
$prevBBoxes \leftarrow activeBoxes$;
$currentBBoxes \leftarrow \{\emptyset\}$;
```

Algorithm 4: mergeA()

```
$activeBoxes \leftarrow \{\emptyset\}$;
foreach $\beta_1 \in currentABoxes$ do
    foreach $\beta_2 \in prevABoxes$ do
        if $\overline{\beta_1}^A = \overline{\beta_2}^A$ then // Merging property 2
            $\beta_2.BBN = \beta_2.BBN \cup \beta_1.BBN$;
            $\beta_2.BFN = \beta_2.BFN \cup \beta_1.BFN$;
            $\beta_2.ABN = \beta_2.ABN \cup \beta_1.ABN$;
            $\beta_2.AFN = \beta_2.AFN \cup \beta_1.AFN$;
            $\beta_2 \leftarrow \beta_1 \cup \beta_2$;
            Add $\beta_2$ to $activeBoxes$;
            break;
        else if $\overline{\beta_1}^A \cap \overline{\beta_2}^A \neq \emptyset$ then
            Add $\beta_2$ to $\beta_1.ABN$;
        end
    end
    if $\beta_1$ was not merged then
        $Q.insertBox(\beta_1)$;
        Add $\beta_1$ to $activeBoxes$;
        Add $\beta_1$ in AFN of each box in $\beta_1$.ABN;
    end
end
$prevABoxes \leftarrow activeBoxes$;
$currentABoxes \leftarrow \{\emptyset\}$;
```

Therefore, a simple XOR operation of all the EVM-represented boxes is necessary to get the EVM-represented object. That is, let β_i be a box in the CUDB-represented OPP P

$$EVM(P) = \bigotimes_{i=1}^{n_b} EVM(\beta_i), \tag{4}$$

where $EVM(\beta_i)$ is the EVM-representation of β_i and n_b is the number of boxes in CUDB.

3.4 Area and Volume Computation

Computing the volume (3D) or area (2D) is straightforward by doing a traversal of the boxes, and making a summation of each volume or area depending on the dimension:

$$Volume(P) = \sum_{i=1}^{n_b} Volume(\beta_i). \tag{5}$$

4 CUDB-based Connected Component Labeling

Since a CUDB-represented object contains the boxes neighborhood information, it can bee seen as an undirected graph. Thus, we proporse a new CUDB-CCL process based on the detection of connected components in graph theory.

Let $G = (V, E)$ be an undirected graph without self loops, with V being a set of vertices (the CUDB boxes) and E a set of edges defined by the neighborhood information (ABN, AFN, BBN and BFN). A connected component in G is a maximal subgraph $g = (V^g, E^g)$ in which for any two vertices $v, u \in V^g$ there exists an undirected path in G with v as start and u as end vertex [48]. A maximal subgraph means that for any additional vertex $w \in (V \backslash V^g)$ there is no path from any $v \in V^g$ to w.

Thus, the CUDB-CCL process has linear complexity, in terms of the sum of the numbers of vertices and edges of the graph, using either depth-first search or breadth-first search [29]. In either case, a search that begins at some box β, will find the entire connected component containing β. To detect all the connected components, a traversal of the boxes is performed, starting a new breadth-first search or depth-first search whenever a box that has not been already labeled is detected.

Algorithm 5: CCL(**Input/Output** Q:CUDB, **Output** cc:Integer)

```
$cc \leftarrow 0$ ; // Number of connected components
$\Phi = \{\emptyset\}$ ; // Queue of box pointers
$currentLabel \leftarrow 1$;
foreach $\beta \in Q$ do
    if $\beta.getLabel() = \emptyset$ then
        $\beta.setLabel(currentLabel)$ ;
        Add $\beta$ to $\Phi$;
        while $\Phi \neq \emptyset$ do
            $\beta_1 = \Phi.pop()$;
            foreach $\beta_2 \in (\beta_1.ABN \cup \beta_1.AFN \cup \beta_1.BBN \cup \beta_1.BFN)$ do
                if $\beta_2.getLabel() = \emptyset$ then
                    $\beta_2.setLabel(currentLabel)$;
                    $\Phi.push(\beta_2)$;
                end
            end
        end
        $currentLabel++$;
    end
end
$cc \leftarrow currentLabel - 1$;
```

Algorithm 5 details the steps for the CUDB-CCL process using a breadth-first search strategy. Fig. 5 depicts the CUDB-CCL process for a 2D example, where the evolution of the boxes queue used by the algorithm is shown.

5 CUDB-based Exact Collision Detection

In this section, we present a CUDB-based collision detection algorithm. Given an environment composed of n objects, we assume that each object is its CUDB-representation and have the same ABC-ordering; otherwise, a preprocesing must be performed first in order to set the same ordering.

Due to the nature of the CUDB model, the proposed algorithm works only in static environments. The goal of this method is to check which objects overlap or are adjacent.

Since the CUDB-representation contains fewer elements than other representations, a straightforward solution could be to iteratively compare each of the boxes in an object with all of the boxes in the other objects (brute force). Nevertheless, taking advantage of the implicit order of the boxes in the CUDB model, unnecessary analysis can be avoided.

In the presented method, a discarding of those objects whose AABB do not collide is performed first. Then, as boxes in CUDB are ABC-sorted, all the remaining potentially colliding objects can be tested jointly, instead of testing them in pairs.

Let $\Theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$ be a finite sequence of n CUDB-represented potentially colliding objects, and let $\Phi = \{\beta_1, \beta_1, \ldots, \beta_n\}$ be a set of box pointers, where each β_i points to a box in the object θ_i. Initially each β_i points to the first box of the corresponding object.

A collision detection between all of the boxes in Φ is performed first, followed by an iterative process. This process obtains the box β_{min} in Φ (β_i with the minimum ABC-position of its vertex V_0) and updates it with the next box in the corresponding object θ_{min}. If there are no more boxes in θ_{min}, this object is marked as not active.

Otherwise, β_{min} is compared for collision with all boxes $\beta_i \in \Phi, \forall i \neq min$ and with the subsequent neighboring boxes of each β_i, say β_t, until $\beta_t.V_0$ has an A-coordinate greater than $\beta_{min}.V_1$. Note that we do not need to compare B and C-coordinate. The main iterative process finishes when β_{min} cannot be defined, which means that all θ_i have been marked as not active. At the end, a set Δ, with object pairs (θ_i, θ_j) that collide or are adjacent has been defined.

Algorithms 6 and 7 detail the steps of this process. Function *getIndexMinBox()* returns the index min of the box in Φ with the minimum ABC-position of its vertex V_0, such that θ_{min} is marked as active. If all θ_i are marked as not active, this function returns $\emptyset$. The worst case time-complexity of the CUDB-based exact collision detection is $O(n \cdot m \cdot T)$, where n is the number of objects, m the number of boxes of the object having the maximum number of boxes, and T the total number of boxes in the n objects. In any case it holds that $n \leq m \leq T$.

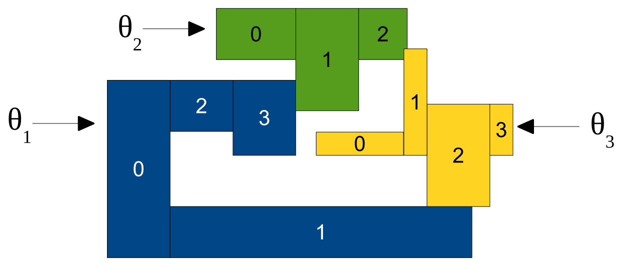
Algorithm 6: detectCollision(**Input** Θ:array or CUDB, **Output** Δ:array of CUDB pairs)

```
$\Delta = \{\emptyset\}$ ; // Set of colliding object pairs
$A \leftarrow \{\emptyset\}$ ; // Vector of boolean for active $\theta_i$
$\Phi \leftarrow \{\emptyset\}$ ; // Vector of Box pointers
foreach $\theta \in \Theta$ do
    $\beta \leftarrow \theta.getBox(0)$ ;
    Add $\beta$ to $\Phi$ ;
    Add true to $A$ ; // Mark all as active
end
foreach $\beta_i \in \Phi$ do // First collision test
    $testCollision(\Phi, \beta_i, \Delta)$
end
$min \leftarrow getIndexMinBox(\Phi, Act)$;
while $min \neq \emptyset$ do
    $\beta_{min} \leftarrow \Theta[min].getNextBox()$;
    if $\beta_{min} \neq \emptyset$ then
        $\Phi[min] = \beta_{min}$;
        $testCollision(\Phi, \beta_{min}, S)$;
    else
        $A[min] =$ false ; // Mark as not active
    end
    $min \leftarrow getIndexMinBox(\Phi, Act)$;
end
```

Algorithm 7: testCollision(**Input** Φ: Vector of Box pointers, **Input** β_i: Box, **Input** / **Output** Δ: array of CUDB pairs)

```
foreach $\beta_j \neq \beta_i \in \Phi$ do
    $\beta_t \leftarrow \beta_j$;
    while $\beta_t \neq \emptyset$ do
        if $\beta_t.V0.A > \beta_i.V1.A$ then break; end;
        if $\beta_i \cap \beta_t \neq \emptyset$ then
            Add pair $(\theta_i, \theta_j)$ to $\Delta$; break;
        end
        $\beta_t \leftarrow \theta_j.getNextBox(\beta_t)$;
    end
end
```

Figure 6 depicts the process for a 2D example with three objects. It shows the evolution of the variables used by the algorithm.



Evolution of the variables →

$\Phi=\{0_1,0_2,0_3\}$	$\Phi=\{1_1,0_2,0_3\}$	$\Phi=\{2_1,0_2,0_3\}$	$\Phi=\{3_1,0_2,0_3\}$	$\Phi=\{3_1,1_2,0_3\}$	$\Phi=\{3_1,2_2,0_3\}$	$\Phi=\{3_1,2_2,1_3\}$
min=1	*min*=1	*min*=1	*min*=2	*min*=2	*min*=3	*min*=3
$\Delta=\{\}$	$\Delta=\{(1,3)\}$	$\Delta=\{(1,3)\}$	$\Delta=\{(1,3),(1,2)\}$	$\Delta=\{(1,3),(1,2)\}$	$\Delta=\{(1,3),(1,2),(2,3)\}$	$\Delta=\{(1,3),(1,2),(2,3)\}$
$A=\{T_1,T_2,T_3\}$	$A=\{T_1,T_2,T_3\}$	$A=\{T_1,T_2,T_3\}$	$A=\{T_1,T_2,T_3\}$	$A=\{F_1,T_2,T_3\}$	$A=\{F_1,T_2,T_3\}$	$A=\{F_1,F_2,T_3\}$

Fig. 6. 2D example of collision

6 CUDB Performance

CUDB has been compared with OUDB in number of elements and computation time for conversion to and from EVM and CCL (using OUDB-extended version [5]). The test datasets consists of 12 objects (see Fig. 7). All datasets come from public volume repositories, where from (g) to (l) are real volume models coming from CT or MRI scanners. The corresponding programs have been written in C++ and tested on a PC Intel®Core i7-4600M CPU@2.90GHz with 7.6 GB RAM and running Linux.

Table 1 shows the attributes of test datasets. For each dataset it shows size in voxels, number of foreground voxels ($|Fvoxels|$), number of triangles of a triangular surface mesh obtained via marching cubes [36], number of extreme vertices ($|EV|$), number of boxes in its XYZ-OUDB ($|OUDB|$) and XYZ-CUDB ($|CUDB|$) representation, ratio between $|OUDB|$ and $|CUDB|$, and number of connected components ($|CC|$) allowing non-manifold configurations.

Note in this table that, although the number of boxes depends on the ABC-sorting of the original EVM, CUDB produces less elements than the other representations in all cases, regarding OUDB in some of the dataset less than 10% of elements. For instance, the XYZ-CUDB representation of the Lines dataset has only 3.8% of boxes of the corresponding XYZ-OUDB representation (see Fig. 8).

Table 2 shows the performance of CUDB regarding OUDB. For each dataset it shows the time for: EVM to OUDB ($E \rightarrow O$) conversion, OUDB-CCL (O_{CCL}), EVM to CUDB ($E \rightarrow C$) conversion and CUDB-CCL (C_{CCL}). The last columns mean $t_O = E \rightarrow O + O_{CCL}$, $t_C = E \rightarrow C + C_{CCL}$, and the ratio between t_O and t_C.

Note that, although the conversion from EVM to CUDB is slightly slower than EVM to OUDB due to the extra effort to merge the boxes, the CUDB-based CCL process is much faster due to the less number of elements and that we avoid the use of a equivalence table, in all datasets more than an order of magnitude, even the temple dataset more than two orders of magnitude faster. In any case, when computing the number of connected components starting from the EVM model, CUDB is more efficient than OUDB..

In order to show the performance of the CUDB-based exact collision detection, three scenes are presented. The first one consists of seven datasets (see Fig. 9), where the size of each is around 256^3. The second scene consists of 250 spheres and 250 objects of the Star dataset randomly placed in a volume of 1000^3 voxels (see Fig. 10). The third scene consists of 2 objects of

Table 1. Attributes of the test datasets

Dataset	size	$\|Fvoxels\|$	$\|triangles\|$	$\|EV\|$	$\|OUDB\|$	$\|CUDB\|$	% $\frac{\|CUDB\|}{\|OUDB\|}$	$\|CC\|$
(a) Cart	585×979×1000	4453852	4605468	502986	256370	100358	39.1	16
(b) Lines	500×500×500	11179011	6579736	4356	24851	954	3.8	1
(c) Cup	401×401×512	18027587	2628720	215050	236642	60429	25.5	1
(d) FanDisk	470×512×261	18706826	1336956	59290	49761	16733	33.6	1
(e) Robot	372×943×1000	29150579	4234084	126806	163281	34515	21.1	48
(f) Temple	925×1000×472	141371526	7467204	73902	260279	21084	8.1	99
(g) Aneurysm	213×215×240	69743	175064	50318	12825	10705	83.5	406
(h) Lobster	244×239×49	233509	311880	74724	27307	19322	70.8	53
(i) Engine	139×197×108	901818	663900	101114	47143	25524	54.1	9
(j) Beetle	411×371×247	1737343	567972	132184	47410	36052	76.0	17
(k) Colon	512×492×426	3995607	7772360	2142304	653717	473649	72.5	54829
(l) Mineral	376×375×206	7363953	6121828	833002	489585	232008	47.4	724

Table 2. OUDB and CUDB run time comparison in milliseconds

D.	OUDB		CUDB		t_O	t_C	% $\frac{t_C}{t_O}$
	E→O	O_{CCL}	E→C	C_{CCL}			
(a)	406	**422**	467	**16**	828	483	58
(b)	26	**15**	37	**0.3**	41	37	90
(c)	324	**366**	360	**7**	690	367	53
(d)	76	**31**	78	**1**	107	79	74
(e)	204	**251**	253	**4**	455	257	56
(f)	263	**503**	299	**2**	766	301	39
(g)	45	**10**	53	**1**	55	54	98
(h)	95	**31**	100	**2**	126	102	81
(i)	66	**66**	96	**2**	132	98	74
(j)	88	**33**	114	**3**	121	117	97
(k)	1335	**1162**	1716	**65**	2497	1781	71
(l)	611	**926**	769	**33**	1537	802	52

the Cart dataset that have interlaced parts but do not collide (see Fig. 11).

Statistics of the collision detection test are shown in Table 3. For each scene: number of objects (n), number of boxes of the object having the maximum number of boxes (m), total number of boxes in the scene (T), number of detected collisions (i.e., number of object pairs $|pairs|$) and time to detect the collisions in milliseconds.

Note that, although scene 3 has less boxes than scene 2, the required time for collision detection is bigger. This is because there is any collision, which implies that there is not any early discarding and all boxes must be evaluated.

Table 3. Statistics of the collision scenes

Scene	n	m	T	$\|pairs\|$	Time
Scene 1	6	23410	84714	4	92
Scene 2	500	1280	546750	91	67
Scene 3	2	101743	202101	0	566

7 Conclusion and Future Work

We have presented in detail, a new decomposition model for OPP, CUDB, which is an improved version of the OUDB model. Algorithms for conversion to and from EVM, for CCL and exact collision detection have also been presented.

Experimental results show that CUDB is smaller in number of elements, and so in storage size.

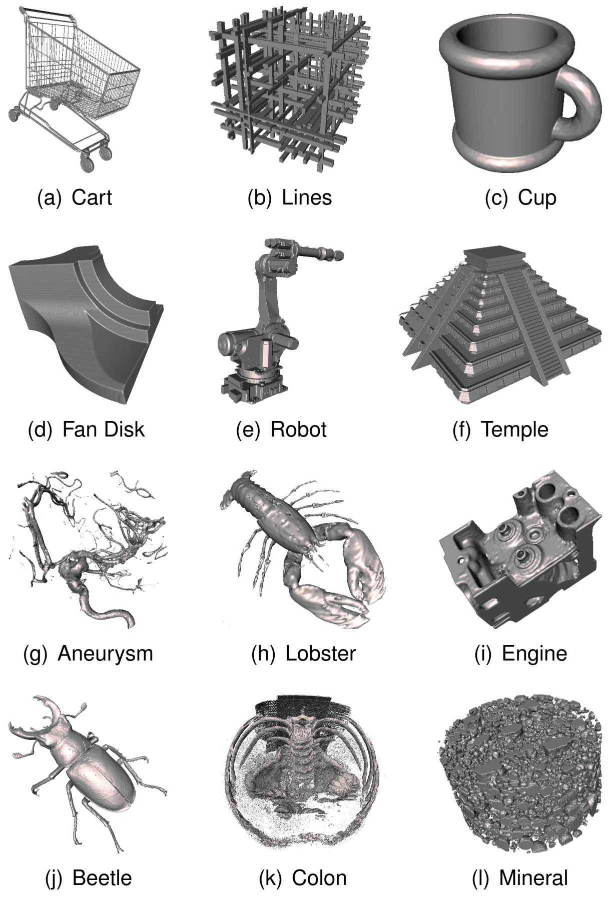


Fig. 7. Rendered images of the test datasets

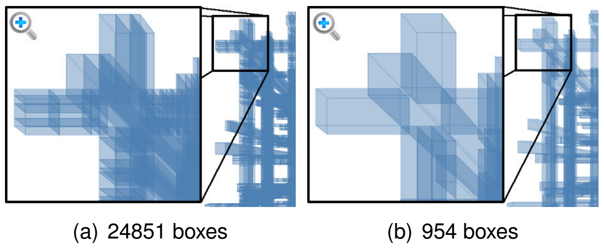


Fig. 8. Zoom in of Lines dataset. (a) XYZ-OUDB. (b) XYZ-CUDB

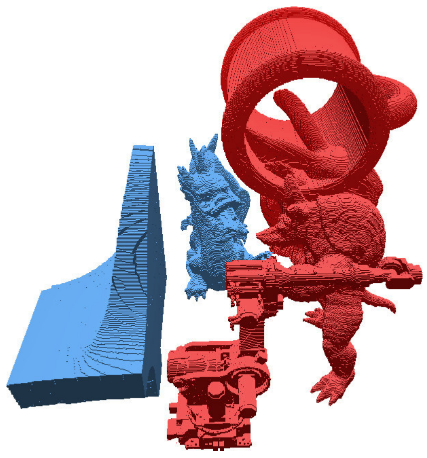


Fig. 9. Collision scene 1. Six datasets: Armadillo (18338 boxes), Bunny (23410 b.), Cup (15002 b.), Dragon (19413 b.), FanDisk (4719 b.), Robot (3832 b.). In red the objects that collide: Bunny with Cup and Armadillo, and the latter with Robot

It can be observed that computation of CCL is notably faster in CUDB than the improved version, OUDB-extended. Regarding the presented exact collision detection algorithm, although this is CPU-based, it is efficient when exact collision detection is required directly on CUDB models.

Table 4 resumes the pros and cons of CUDB with respect B-Rep and the hierarchical structures (HS) on the basis of some properties of solid representation schemes, and the CCL and collision detection performance.

The performance variability of the presented algorithms is caused by the dataset size but above all to their surface intricacy. Our method depends on the number of boxes, tightly related to the model's tortuosity (a property that represents the twist of a curve, i.e. the degree of turns or detours a model has [26], like the previous developed methods based on OPP.

Characteristics of CUDB model have been exploited in some applications as in a CUDB-based virtual porosimeter [44], which simulate mercury intrusion at increasing pressures, like the porosimeter lab. Also in a method to compute the Euler characteristic and the genus of binary volumes [14]. And a 2D version of the collision detection algorithm has been applied in a lossless simplification method to get a better shape preservation [15].

As future work, we are devising methods to compute the B-Rep model from the

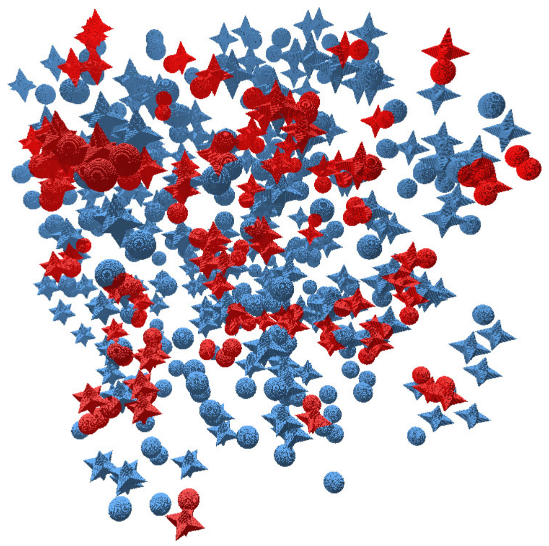


Fig. 10. Collision scene 2. 250 spheres (voxel size 50×50×50) and 250 objects of the Star dataset (70×95×70). Each sphere has 907 boxes and each Star 1280 boxes in their corresponding XYZ-CUDB representation. In red the objects that collide

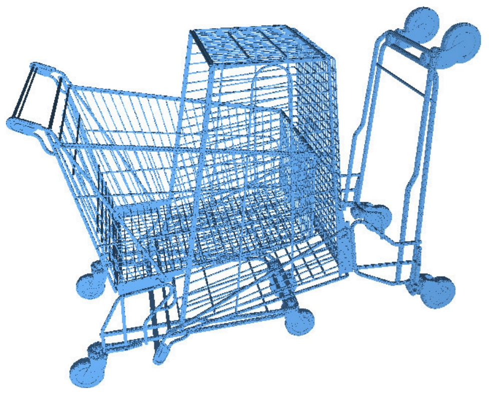


Fig. 11. Collision scene 3. 2 objects of the Cart dataset. The original Cart has 100358 boxes, the rotated one 101743 boxes. Both in its XYZ-CUDB. The objects actually do not collide

CUDB-representation and to obtain the CUDB-represented complement. Another future work is a complete analysis of whether any of the six ABC-sortings in CUDB produces an optimal number of disjoint boxes that cover the OPP, otherwise, one could think of allowing overlapping boxes in order to further reduce this number.

Table 4. Comparative of CUDB vs. other representation models

	CUDB	B-Rep & HS
Accuracy	Exact representation for OPP.	Approximate representation [21].
Domain	Represents any solid.	B-Rep represents a very wide classes of objects, HS any solid [21].
Uniqueness	Six representations depending on the ordering.	Only octrees guarantees the uniqueness [21].
Storage (# of elements)	In general, OUDB has proven to be more compact than other models [42, 43], and here we have proven that CUDB is more compact than OUDB.	
CCL	In general, OUDB has proven to be faster than other models [42, 43], and here we have proven that CUDB is faster than OUDB.	
Exact collision detection	Only for static scenes	Applications in dynamic scenes and deformable objects [49].
Boolean Operations	Not yet devised methods, but can be performed via EVM.	Known operations [21].

Finally, we think that CUDB model can be used to compute structural parameters of biomaterials as those reported in [6] and [54] and new ones emerging in the bioengineering field.

References

1. **Aguilera, A. (1998).** *Orthogonal Polyhedra: Study and Application*. Ph.D. thesis, LSI-Universitat Politècnica de Catalunya.

2. **Aguilera, A. A. & Ayala, D. (1997).** Orthogonal Polyhedra as Geometric Bounds in Constructive Solid Geometry. *Fourth ACM Symposium on Solid Modeling and Applications*, ACM, pp. 56–67.

3. **Aguilera, A. A. & Ayala, D. (2001).** *Geometric Modeling*, volume 14 of *Computing Supplement*, chapter Converting Orthogonal Polyhedra from

Extreme Vertices Model to B-Rep and to Alternating Sum of Volumes. Springer-Verlag, pp. 1–28.

4. **Ayala, D., Vergara, E., & Vergés, E. (2007).** Improved skeleton computation of an encoded volume. *Proc. of Eurographics*, volume 2007, pp. 33–36.
5. **Ayala, D. & Vergés, E. (2008).** Improved virtual porosimeter. *CASEIB'08*.
6. **Ayala, D. & Vergés, E. (2009).** Structural parameters computation of a volume using alternative representations. *Proceedings of IV Iberoamerican Symposium in Computer Graphics*, DJ Editores, C.A., pp. 73–80.
7. **Ayala, D., Vergés, E., & Cruz, I. (2012).** A polyhedral approach to compute the genus of a volume dataset. *Proc. of the Int. Conf. GRAPP 2012*, SciTePress, pp. 38–47.
8. **Biedl, T. & Genç, B. (2011).** Reconstructing orthogonal polyhedra from putative vertex sets. *Computational Geometry*, Vol. 44, No. 8, pp. 409–417.
9. **Biswas, A., Bhowmick, P., Sarkar, M., & Bhattacharya, B. B. (2012).** A linear-time combinatorial algorithm to find the orthogonal hull of an object on the digital plane. *Information Sciences*, Vol. 216, pp. 176–195.
10. **Bournez, O., Maler, O., & Pnueli, A. (1999).** Orthogonal polyhedra: Representation and computation. *Hybrid Systems: Computation and Control*, pp. 46–60. LNCS 1569, Springer.
11. **Brunet, P., Juan, R., & Navazo, M. I. (1992).** Octree representations in solid modeling. *Progress in Computer Graphics*, Vol. 1, pp. 164–215.
12. **Cohen, J. D., Lin, M. C., Manocha, D., & Ponamgi, M. (1995).** I-collide: an interactive and exact collision detection system for large-scale environments. *Proc. of the 1995 Sym. on Interactive 3D graphics*, ACM, pp. 189–ff.
13. **Coming, D. S. & Staadt, O. G. (2008).** Velocity-aligned discrete oriented polytopes for dynamic collision detection. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 14, pp. 1–12.
14. **Cruz-Matías, I. & Ayala, D. (2013).** An efficient alternative to compute the genus of binary volume models. *Proc. of the Int. Conf. GRAPP 2013*, SciTePress, pp. 18–26.
15. **Cruz-Matías, I. & Ayala, D. (2014).** A new lossless orthogonal simplification method for 3D objects based on bounding structures. *Graphical Models*, Vol. 76, No. 4, pp. 181–201.
16. **Dang, T. & Maler, O. (1998).** Reachability analysis via face lifting. In *Hybrid Systems: Computation and Control*. Springer, pp. 96–109.
17. **Dillencourt, M., Samet, H., & Tamminen, M. (1992).** A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM*, Vol. 39, No. 2, pp. 253–280.
18. **Eppstein, D. & Mumford, E. (2010).** Steinitz theorems for orthogonal polyhedra. *Proceedings of the 2010 annual symposium on Computational geometry*, ACM, pp. 429–438.
19. **Esperança, C. & Samet, H. (1998).** Vertex representations and their applications in computer graphics. *The Visual Computer*, Vol. 14, pp. 240–256.
20. **Flores, J. A. M. (1999).** *Analysis and Visualization of Complex 3D Structures: a discrete boundary-based approach*. Ph.D. thesis, Ecole Nationale Supérieure des Télécommunications.
21. **Foley, J., Damm, A. V., Feiner, S., & Hughes, J. (1997).** *Computer graphics: principles and practice*. Pearson Education.
22. **Gagvani, N. & Silver, D. (2000).** Shape-based volumetric collision detection. *Proc. of the 2000 IEEE Sym. on Volume visualization*, ACM, pp. 57–61.
23. **Geleri, F., Tosun, O., & Topcuoglu, H. (2013).** Parallelizing broad phase collision detection algorithms for sampling based path planners. *Proc. of the 21st Euromicro Int. Conference on Parallel, Distributed, and Network-Based Processing*, IEEE, pp. 384–391.
24. **Greß, A. & Klein, R. (2004).** Efficient representation and extraction of 2-manifold isosurfaces using kd-trees. *Graphical Models*, Vol. 66, pp. 370–397.
25. **Grevera, G. J., Udupa, J. K., & Odhner, D. (2000).** An Order of Magnitude Faster Isosurface Rendering in Software on a PC than Using Dedicated, General Purpose Rendering Hardware. *IEEE Transactions Visualization and Computer Graphics*, Vol. 6, No. 4, pp. 335–345.
26. **Grisan, E., Foracchia, M., & Ruggeri, A. (2003).** A novel method for the automatic evaluation of retinal vessel tortuosity. *Proc. of the 25th Annual Int. Conf. of the IEEE EMBS*, volume 1, pp. 866–869.

27. **Gupta, S., Palsetia, D., Patwary, A., Mostofa, M., Agrawal, A., & Choudhary, A. (2014).** A new parallel algorithm for two-pass connected component labeling. *IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, pp. 1355–1362.
28. **He, L.-F., Chao, Y.-Y., & Suzuki, K. (2013).** An algorithm for connected-component labeling, hole labeling and euler number computing. *Journal of Computer Science and Technology*, Vol. 28, No. 3, pp. 468–478.
29. **Hopcroft, J. & Tarjan, R. (1973).** Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, Vol. 16, No. 6, pp. 372–378.
30. **Jiménez, P., Thomas, F., & Torras, C. (2000).** 3D collision detection: A survey. *Computers & Graphics*, Vol. 25, No. 2, pp. 269–285.
31. **Kaufman, A. (1990).** *Volume Visualization*. IEEE Computer Society Press.
32. **Kaufman, A., Cohen, D., & Yagel, R. (1993).** Volume graphics. *Computer*, pp. 51–64.
33. **Lachaud, J. & Montanvert, A. (2000).** Continuous analogs of digital boundaries: A topological approach to iso-surfaces. *Graphical Models*, Vol. 62, pp. 129–164.
34. **Lee, E., Choi, Y., & Park, K. (1994).** A method of 3D object reconstruction from a series of cross-sectional images. *IEICE trans. inf and syst*, Vol. E77-D, No. 9.
35. **Lifeng, H., Xiao, Z., Bin, Y., Yun, Y., & Yuyan, C. (2014).** An efficient two-scan labeling algorithm for binary hexagonal images. *IEICE TRANSACTIONS on Information and Systems*, Vol. 97, No. 12, pp. 3244–3247.
36. **Lorensen, W. & Cline, H. (1987).** Marching cubes: A high resolution 3D surface construction algorithm. *ACM Computer Graphics*, Vol. 21, No. 4, pp. 163–169.
37. **Martínez, J., Pla, N., & Vigo, M. (2013).** Skeletal representations of orthogonal shapes. *Graphical Models*, Vol. 75, pp. 189–207.
38. **Muuss, M. J. & Butler, L. A. (1991).** *State of the Art in Computer Graphics: Visualization and Modeling*. Springer-Verlag.
39. **Park, S. C. & Choi, B. K. (2001).** Boundary extraction algorithm for cutting area detection. *Computer-Aided Design*, Vol. 33, No. 8, pp. 571–579.
40. **Quadros, W. R., Shimada, K., & Owen, S. J. (2004).** 3D discrete skeleton generation by wave propagation on PR-octree for finite element mesh sizing. *Proceedings of ACM Symposium on Solid Modeling and Applications*, pp. 327–332.
41. **Requicha, A. (1980).** Representations for rigid solids: Theory, methods and systems. *ACM Computing Surveys*, Vol. 12, No. 4, pp. 73–82.
42. **Rodríguez, J. & Ayala, D. (2003).** Fast neighborhood operations for images and volume data sets. *Computers & Graphics*, Vol. 27, pp. 931–942.
43. **Rodríguez, J., Ayala, D., & Aguilera, A. (2004).** *Geometric Modeling for Scientific Visualization*, chapter EVM: A Complete Solid Model for Surface Rendering. Springer-Verlag, pp. 259–274.
44. **Rodríguez, J., Cruz, I., Vergés, E., & Ayala, D. (2011).** A connected-component-labeling-based approach to virtual porosimetry. *Graphical Models*, Vol. 73, pp. 296–310.
45. **Rosenfeld, A. & Pfaltz, J. (1966).** Sequential operations in digital picture processing. *Journal of the ACM*, Vol. 13, No. 4, pp. 471–494.
46. **Samet, H. (1990).** *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley Longman Publishing.
47. **Samet, H. & Tamminen, M. (1985).** Bintrees, CSG trees, and time. *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '85, ACM, pp. 121–130.
48. **Seidl, T., Boden, B., & Fries, S. (2012).** Cc-mr–finding connected components in huge graphs with mapreduce. In *Machine Learning and Knowledge Discovery in Databases*. Springer, pp. 458–473.
49. **Teschner, M., Kimmerle, S., Heidelberger, B., Zachmann, G., Raghupathi, L., Fuhrmann, A., Cani, M.-P., Faure, F., Magnenat-Thalmann, N., Strasser, W., et al. (2005).** Collision detection for deformable objects. *Computer graphics forum*, volume 24, Wiley Online Library, pp. 61–81.
50. **Thibault, W. & Naylor, B. (1987).** Set operations on polyhedra using binary space partitioning trees. *SIGGRAPH Comput. Graph.*, Vol. 21, No. 4, pp. 153–162.
51. **Thurfjell, L., Bengtsson, E., & Nordin, B. (1995).** A boundary approach to fast neighborhood operations on three-dimensional binary data. *CVGIP: Graphical Models and Image Processing*, Vol. 57, No. 1, pp. 13–19.

52. **Udupa, J. K. & Odhner, D. (1991).** Fast visualization, manipulation, and analysis of binary volumetric objects. *IEEE Computer Graphics & Applications*, pp. 53–62.

53. **Udupa, J. K. & Odhner, D. (1993).** Shell rendering. *IEEE Computer Graphics & Applications*, pp. 58–67.

54. **Vergés, E. (2011).** *Modeling, Analysis and Visualization of Porous Biomaterials*. Ph.D. thesis, LSI-Universitat Politècnica de Catalunya.

55. **Vigo, M., Pla, N., Ayala, D., & Martínez, J. (2012).** Efficient algorithms for boundary extraction of 2D and 3D orthogonal pseudomanifolds. *Graphical Models*, Vol. 74, pp. 61–74.

56. **Vogiannou, A., Moustakas, K., Tzovaras, D., & Strintzis, M. G. (2010).** Enhancing bounding volumes using support plane mappings for collision detection. *Computer Graphics Forum*, Vol. 29, No. 5, pp. 1595–1604.

57. **Wilhems, J. & Gelder, A. V. (1992).** Octrees for faster isosurface generation. *ACM Transactions on Graphics*, Vol. 11, No. 3, pp. 201–227.

58. **Wong, W., Shih, F. Y., & Su, T. (2006).** Thinning algorithms based on quadtree and octree representations. *Information Sciences*, Vol. 176, pp. 1379–1394.

59. **Zachmann, G. (2002).** Minimal hierarchical collision detection. *Proc. of the ACM Sym. on Virtual reality software and technology*, VRST '02, ACM, pp. 121–128.

Article received on 07/03/2016; accepted on 07/02/2017. Corresponding author is Irving Cruz-Matías.