CUDB: An Improved Decomposition Model for Orthogonal Pseudo-Polyhedra

Irving Cruz-Matías and Dolors Ayala
Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics

Abstract

We present a new decomposition model for Orthogonal Pseudo-Polyhedra (OPP): the Compact Union of Disjoint Boxes. This model is an improved version of the Ordered Union of Disjoint Boxes model. Our model has many desirable features versus the OUDB, such as less storage size and a better efficiency in the connected-component labeling (CCL) process. CCL is a very important operation for manipulating volume data where multiple disconnected components that compose a volume need to be identify. We present the algorithms for conversion to and from the Extreme Vertices Model, which is closely related to the OUDB, and for CCL. The performance of the CUDB is experimentally analyzed with 2D and 3D datasets.

1 Introduction

In the Ordered Union of Disjoint Boxes (OUDB) model a spatial partition is made in a non-hierarchical, sweep-based way, where an Orthogonal Pseudo-Polyhedra (OPP) is divided by planes perpendicular to two different main-axis. These planes split the full geometry of the OPP, however, unnecessary divisions at certain local regions of the OPP are made many times due to the necessity of OUDB to keep the resulting boxes sorted to preserve the adjacency information.

We present the Compact Union of Disjoint Boxes (CUDB) model, which also partitions the geometry in a non-hierarchical, sweep-based way, but, all the boxes that have been unnecessary divided in the OUDB are merged. Although the implicit order of the boxes is lost, it is easy to preserve the adjacency information with a tiny storage effort.

On the other hand, in most of the reported bibliography, the operations to study binary models are performed directly on the classical voxel model. However, in the field of volume analysis and visualization, several alternative models have been devised for specific purposes.

Hierarchical decomposition models, such as octrees and kd-trees, have been used for Boolean operations [13], connected-component labeling (CCL) [4], and thinning [8] [16]. Octrees are used as a means of compacting regions and getting rid of the large amount of empty space in the extraction of isosurfaces [15]. Kd-trees have been used to extract 2-manifold isosurfaces [5].

There are 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 [6]. Certain methods of erosion, dilation and CCL use this representation [14]. The slice-based binary shell representation stores only surface voxels and is used to render binary volumes [7].

CCL is a very important operation for manipulating volume data where multiple disconnected components that compose a volume need to be identify. A traditional Voxel-based method is presented in [12].

The OUDB model [1] has been proved to be efficient for CCL [10, 9]. 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 [9].

An improvement of the OUDB-based CCL is presented by Ayala and Vergés [3], where they compute the so-called OUDB-extended that allows to jump directly to the required box that needs to be tested, instead of querying and skipping all intermediate boxes. In this direction, we also devise an algorithm for CCL in the CUDB model.

This paper is organized as follows. Section 2 presents a short introduction to the EVM and OUDB models. Section 3 introduces the characteristics and algorithms of the CUDB model. Section 4 discusses the CUDB performance. Finally, Section 5 presents the conclusions and trends for future work.

2 EVM an OUDB

2.1 The Extreme Vertices Model

The Extreme Vertices Model (EVM) [2] is a very concise representation scheme in which any orthogonal polyhedron (OP) or orthogonal pseudo-polyhedron (OPP) can be described using only a subset of its vertices: the extreme vertices. The EVM is actually a complete (non-ambiguous) solid model: it is an implicit boundary representation (BRep) model, i.e., all the geometry and topological relations concerning faces, edges and vertices of the represented OPP can be obtained from the EVM.

The EVM is very fast performing Boolean operations between OP and the OUDB conversion to and from the EVM is straightforward. Therefore, we have devised the corresponding algorithms for conversion in the CUDB model.

Next, some formal definitions and properties of the EVM are reviewed, which are used in the rest of the present work.

Definitions Let P be an OPP:

- *ABC-sorted*. Let Q be a finite set of points in E^3. The ABC-sorted set of Q is the set resulting from sorting Q according to coordinate A, then to coordinate B, and then to coordinate C. Thus, six possible ABC-sorted sets can be defined in 3D: XYZ, XZY, YXZ, YZX, ZXY, and ZYX.

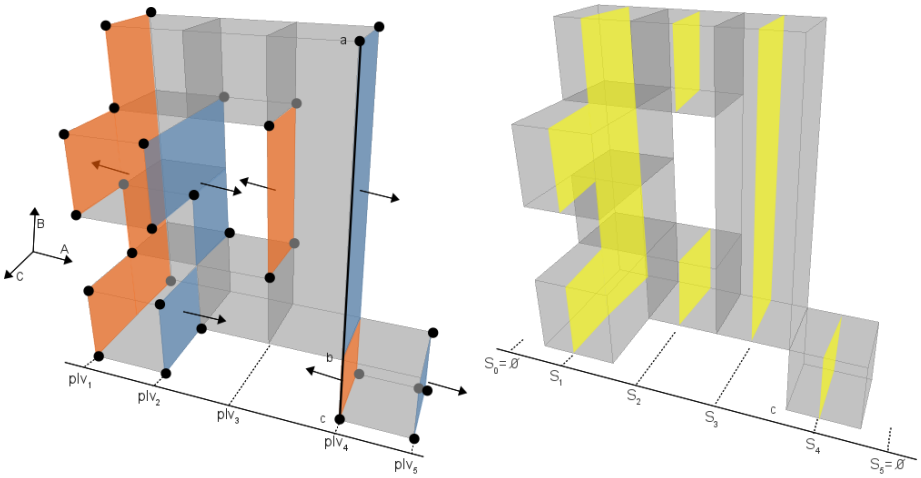
- A *plane of vertices* (plv) of P is the set of vertices lying on a plane perpendicular to a main axis of P (usually the A axis). Similarly *line of vertices* (lov) is the set of brinks lying on a line parallel to a coordinate

axis (usually the C axis). Planes and lines of vertices are axis-prefixable and both of them are hereafter referred as plv.

- A *slice* is the region between two consecutive planes (or lines) of vertices. P can be expressed as the union of all its slices in a certain orthogonal direction. Hence, $P = \bigcup_{k}^{np-1} slice_k(P)$, where np is the number of different A-coordinates in P.

- A *section* (S) is the resulting polygon (or set of polygons) from the intersection between a slice of P and an orthogonal plane (or line) perpendicular to a certain orthogonal direction. $S_i(P)$ will refer to the i^{th} section of P between $plv_i(P)$ and $plv_{i+1}(P)$. A slice from a 2D-OPP is a set of one or more disjoint rectangles, while a slice from a 3D-OPP is a set of one or more disjoint prisms, whose base is the slice's section. Each slice has its representing section.

Figure 1(a) shows an OPP with all its EV's and plv's, also a brink from vertex a to vertex c (both EV's) where vertex b is non-EV, is depicted. Figure 1(b) shows the sections (in yellow).



(a) (b)

Figure 1: Example of an OPP encoded with EVM for an ABC-sorted. (a) All EV's, plv's, and a brink from point a to c (both EV's), where b is non-EVM. (b) The four sections of the object highlighted in yellow.

Sections can be computed from planes of vertices and vice versa. For $i = 1...n$:

$$\overline{S_i(P)} = \overline{S_{i-1}(P)} \otimes^* \overline{plv_i(P)} \tag{1}$$

$$\overline{plv_i(P)} = \overline{S_{i-1}(P)} \otimes^* \overline{S_i(P)} \tag{2}$$

where $\overline{plv_i(P)}$ and $\overline{S_i(P)}$ denote the projections of $plv_i(P)$ and $S_i(P)$ onto a main plane parallel to P, and $\otimes^*$ denotes the regularized XOR operation. Note that in order to operate with the projections we need not consider the coordinate of the extreme vertices that corresponds to the projecting plane.

2.2 The OUDB model

An OPP can be represented as a list of disjoint boxes. Whit this model, algorithms for OPPs with an even simpler complexity can be obtained. From an ABC-sorted EVM we can obtain its ABC-ordered OUDB for an OPP P . This model is the set of boxes obtained by:

- Splitting P at every internal plane of vertices of P perpendicular to the A-axis $plv_k(P)$, $k = 2, \ldots, np-1$. Thus, obtaining an ordered sequence A-slices (A-sections extruded in the A-direction).

$$P = \bigcup_{k}^{np-1} slice_k(P)$$

- Splitting each A-slice at every one of its internal plane of vertices perpendicular to the B-axis, thus obtaining a sorted sequence of Disjoint Boxes.

$$slice_k(P) = \bigcup_{j=1}^{npk-1} Box_{k,j}(P) \quad \text{Therefore,} \quad P = \bigcup_{k}^{np-1} \bigcup_{j=1}^{npk-1} Box_{k,j}(P)$$

There are six different ABC-OUDB models for a given OPP, which correspond to each of the ABC-sorted EVM. Their corresponding sets of disjoint boxes are generally different. Thus, the number of obtained boxes depends of the ABC-sorting of the original EVM. Figure 2 shows an OPP and three possible OUDB decompositions.

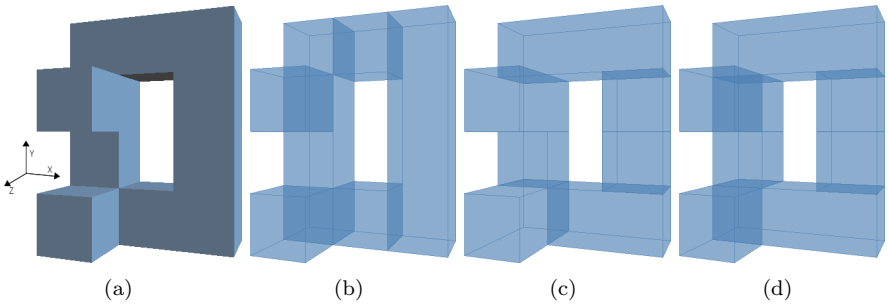


Figure 2: (a) An OPP. (b) XZY-OUDB (6 boxes). (c) YXZ-OUDB (7 boxes). (d) YZX-OUDB (8 boxes).

For more details concerning EVM and OUDB, see [1] and [2].

3 The CUDB model

The CUDB model 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, for instance, the YZX-OUDB partitioning of P, P is subdivided by planes perpendicular to the Y-axis first, and then by planes perpendicular to the Z-axis, at every Cut 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 has actually no relationship, are further divided unnecessarily. Figure 3(b) illustrates this situation where some pieces forces unnecessarily divisions. In OUDB this constraint is mandatory to maintain 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.

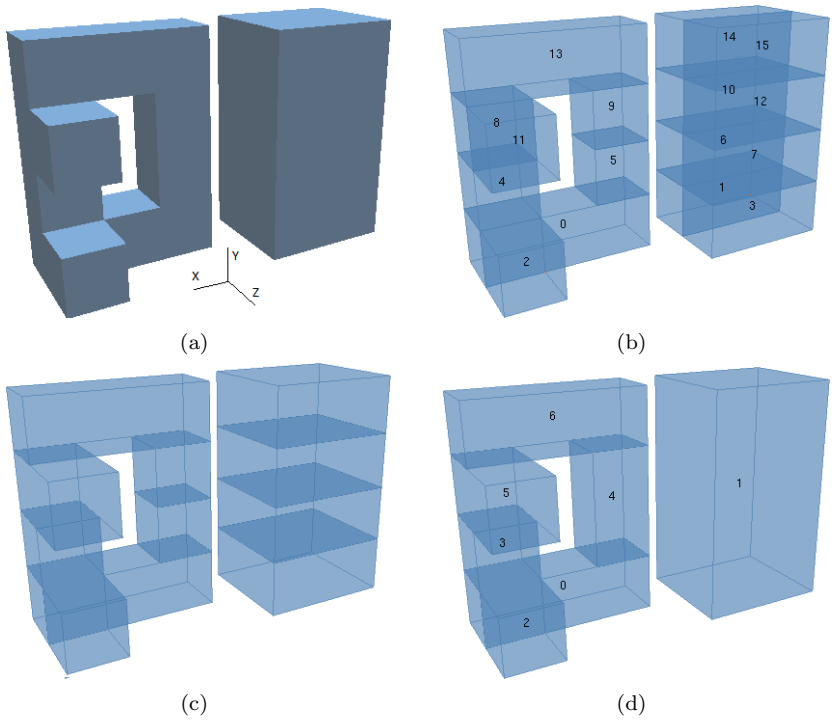


Figure 3: (a) An OPP. (b) YZX-OUDB with 16 boxes. (c) Result after first compacting in Z-coordinate. (d) YZX-CUDB with 7 boxes.

Formally, let b_i and b_j be two adjacent boxes of the $OUDB(P)$ in a ABC-Sorted, and let BP_i and BP_j be their projections respectively over the plane perpendicular to B-axis, then b_i and b_j can be compacted as a single box if $BP_i = BP_j$. Figure 3(c) shows that boxes (1,3),(6,7), (8,11),(10,12) and (14,15) can be compacted following this property. But the model still can be more compacted. Now let b_i and b_j two adjacent boxes and let AP_i and AP_j their projections respectively over the plane perpendicular to A-axis, then b_i and b_j can be compacted as a single box if $AP_i = AP_j$. Figure 3(d) shows the resultant CUDB model.

Although the implicit order among the boxes of the OUDB model is lost, preserving the adjacency information in the CUDB model is easy with a tiny storage effort.

Because of the splitting way, each box will have neighboring boxes in only two orthogonal directions: A and B-axis, and for each direction we have two opposite senses, so four arrays (two for each direction) of pointers to the neighboring boxes is enough to preserve the adjacency information that is required for future operations. This arrays are called: $BackwardNeighbors$ and $ForwardNeighbors$.

It is important to point out that compacting is done on the fly when we pass from EVM to CUDB model, i.e., we do not need to get the OUDB model before. Next we describe the class of the CUDB for object-oriented programming and the algorithms for conversion to and form EVM, and for CCL.

3.1 CUDB class

This construct is very similar to the OUDB class, but, it has the aforementioned four arrays of pointers in the *Box* class.

```
Class Box
{
  public:
    point3D vertices[2];
    vector<Box *> _ABackwardNeighbors;
    vector<Box *> _AForwardNeighbors;
    vector<Box *> _BBackwardNeighbors;
    vector<Box *> _BForwardNeighbors;
    void SetLabel(int value);
    int GetLabel();
    int GetId();
    void SetId_(int id);
  protected:
    int Label;
    int ID;
}

Class CUDB
{
  public:
    vector<Box *> _LastBoxes;   //Boxes in the last 2D Slice
    vector<Box *> _BoxesTmp;   //Temporal vector for 2D compacting
    vector<Box *> _SliceLastBoxes;   //Boxes in the last 1D Slice
    vector<Box *> _SliceBoxesTmp;   //Temporal vector for 3D compacting
    Box *GetBoxRefPosition(int id);
    void SetCurrentBoxRef(const Box *ptr);
    int GetId(void);
    DimType GetDimension(void);
    void SetDimension(DimType Dim);
    void SetSorting(SortingType s);
    void AddBox(MyPoint3D &Vb,MyPoint3D &Ve, int label);
    CUDB(void);
    ~CUDB(void);
  protected:
    int Number_of_Boxes;
    DimType dimension;
    SortingType sorting;
    vector<Box *> vector;   //Vector with all the boxes.
}
```

3.2 Converting EVM to CUDB model

The corresponding conversion algorithm is obtained by reading each A-slice. In this algorithm, a slice of a slice (a box) is represented with the variables: P_pre, P_post, L_pre, L_post), where P_pre and P_post are the initial and final A-coordinate respectively of the 3D slice (given by the plane of vertices), and similarly L_pre, L_post are the initial and final B-coordinate for the 2D slice.

So, the *EVMToCUDB* procedure (see Algorithm 1) with dim =Dim3 produces the sequence of 2D sections (3D slices) of a 3D OPP P. Each 2D section is recursively processed with dim =Dim2 to produce the sequence of 1D sections (one or more boxes). In turn, each 1D section is recursively processed by the base case with dim =Dim1 which calls an *AppendBox()* procedure (see Algorithm 2) at a time to create the CUDB model q.

In these two procedures, the adjacency relationship among the boxes can be set according to the convenience of the user. When two adjacent boxes share a face or part of a face, then they will have a two-manifold connection. But if they share an edge, a part of an edge or a vertex, then they have a non-manifold configuration. These cases are related to the well known 6-adjacency and 26-adjacency respectively. In 2D the equivalent cases are 4-adjacency and 8-adjacency respectively.

Algorithm 1 EVM to CUDB

Input: $q \leftarrow$ CUDB Model
Input: $p \leftarrow$ EVM Model
Input: $Ppre \leftarrow$ initial A-coordinate
Input: $Ppost \leftarrow$ final A-coordinate
Input: $Lpre \leftarrow$ initial B-coordinate
Input: $Lpost \leftarrow$ final B-coordinate
Input: $dim \leftarrow$ Model dimension
Input: $inputSort \leftarrow$ Model sorting
  if dim =Dim1 **then**
    for all brink br **do**
      AppendBox($q, br, Ppre, Lpre, Ppost, Lpost, inputSort, dim$)
    end for
    $q.LastBoxes \leftrightarrow q.BoxesTmp$
    Clear $q.BoxesTmp$
  else
    $dim \leftarrow dim - 1$
    EVM $S0 \leftarrow \emptyset$
    EVM $plv0 \leftarrow P.GetPlv(\&coordIni)$
    EVM $S1 \leftarrow S0 \otimes plv0$
    EVM $plv \leftarrow P.GetPlv(\&coordFin)$
    if dim =Dim2 **then**
      $Ppre \leftarrow coordIni$
      $Ppost \leftarrow coordFin$
    else
      $Lpre \leftarrow coord_ini$
      $Lpost \leftarrow coord_fin$
    end if
    while plv not NULL **do**
      $EVMtoCUDB(q, S1, Ppre, Lpre, Ppost, Lpost, dim, inputSort)$

```
      EVM S2 ← S1 ⊗ plv
      S1 ← S2
      EVM plv ← P.GetPlv(&coordIni)
      if dim =Dim2 then
        Ppre ← P_post
        Ppost ← coordIni
        for all box1 in q.SliceBoxesTmp do
          for all box2 in q.SliceLastBoxes do
            if box1 = box2 in B and C coordinates then
              Compact box2 ← box1 ∪ box2
              Update BBackwardNeighbors and BForwardNeighbors from box1 to
              box2
              Update ABackwardNeighbors and AForwardNeighbors from box1 to
              box2
              Add box to Aux
            else if box1 is adjacent to box2 in B and C coordinates then
              Add box2 to box1.ABackwardNeighbors
            end if
          end for
          if box1 was not compacted then
            Add box1 to q
            Add box1 to Aux;
            Add box1 as forward neighbor for each ABackwardNeighbors of box1
          end if
        end for
        q.SliceLastBoxes ↔ Aux
        Clear q.SliceBoxesTmp
        Clear q.LastBoxes
      end if
    end while
  end if
```

Observe the Algorithm 2, where if a new box is created with *dim*=Dim3, it is not added directly to CUDB model, but stored in a temporal vector in order to be compared with future boxes and possibly compacted.

3.3 Converting CUDB to EVM model

Same that in OUDB model, all the boxes in CUDB are disjoint, so a simple operation XOR of all the boxes in EVM format is necessary to get the whole EVM model.

3.4 Area and Volume Computation

Computing the Volume (3D) or Area (2D) is straightforward if we just do a traverse in all the boxes in CUDB model, and get its volume or area depend of the dimension, like in OUDB model.

3.5 Connected component labeling

Many approaches [12, 10, 9, 3] apply the classical two-pass strategy of first labeling and then renumbering a set of equivalences. As we have already computed

Algorithm 2 AppendBox

Input: $q \leftarrow$ CUDB Model
Input: $br \leftarrow$ Brink
Input: $Ppre \leftarrow$ initial A-coordinate
Input: $Ppost \leftarrow$ final A-coordinate
Input: $Lpre \leftarrow$ initial B-coordinate
Input: $Lpost \leftarrow$ final B-coordinate
Input: $dim \leftarrow$ Model dimension
Input: $inputSort \leftarrow$ Model sorting

```
if dim =Dim2 then
    ComparedIndex ← B-coordinate
    CompactedIndex ← A-coordinate
else if dim =Dim3 then
    ComparedIndex ← C-coordinate
    CompactedIndex ← B-coordinate
end if
Create box1 defined by br, Ppre, Post, Lpre, Lpost
for all box2 in q.LastBoxes do
    if box1 = box2 in ComparedIndex coordinate then
        Compact box2 ← box1 ∪ box2
        Add box2 to q.BoxesTmp
    else if box1 is adjacent to box2 in ComparedIndex coordinate then
        Add box2 to BackwardN
    end if
end for
if box1 was not compacted then
    if dim =Dim2 then
        Add box1 to q
        Add box1 to q.BoxesTmp
        box1.ABackwardNeighbors ← BackwardN
        Add box1 as forward neighbor for each ABackwardNeighbors of box1
    else if dim =Dim3 then
        Add box1 to q.SliceBoxesTmp
        Add box1 to q.BoxesTmp
        box1.BBackwardNeighbors ← BackwardN
        Add box1 as forward neighbor for each BBackwardNeighbors of box1
    end if
end if
```

the neighborhood of all boxes, we can apply the same strategy and achieve a faster computation due to we avoid to test the boxes.

In our labeling process (see Algorithm 3), the CUDB model is traversed and, on the fly, the current box is labeled with the minimum value of the already labeled neighbor boxes or with a new label, when no neighbor has been labeled yet. When a box has two or more neighbors labeled with different values, a label equivalence is recorded.

Algorithm 3 CCL

Input: $q \leftarrow$ CUDB Model
Output: $s \leftarrow$ Number of connected components
  $equivalences \leftarrow$new map<integer,integer>
  $currentLabel \leftarrow 1$
  for all $box1$ in q **do**
    $label \leftarrow$ Minimum label from all already labeled backward neighbors
    if No labeled backward neighbors **then**
      $label \leftarrow currentLabel$
      $currentLabel \leftarrow currentLabel + 1$
    end if
    for all $box2$ in $box1.ABackwardNneighbors$ **do**
      makeEquivalence($equivalences, box2.GetLabel(), label$)
    end for
    for all $box2$ in $box1.BBackwardNneighbors$ **do**
      if $box2$ is labeled **then**
        makeEquivalence($equivalences, box2.label$,$label$)
      else
        $box2.label \leftarrow label$
      end if
    end for
  end for
  $s \leftarrow$ relabeling($q, equivalences$)
  return s

The label equivalences are saved in an dynamic map. Maps are a kind of associative container that stores elements formed by the combination of a *key* value and a *mapped* value, so, the *key* corresponds to the region number and the *mapped* to its label. Unlike the approaches [9] and [3] where they save values for all regions (even if they do not have equivalence), we only save the equivalences, which means less memory requirement. The procedure $MakeEquivalence$ updates the map in the way shown in the Algorithm 4.

All the equivalences are solved in a second traversal of the model, the *relabeling* pass. This procedure (see Algorithm 5) first sorts out all the equivalences and propagates them correctly. At the end, the boxes are re-labeled with the correct label and in a consecutive order.

Algorithm 4 makeEquivalence

Input: $equivalences \leftarrow$ Map of equivalences
Input: $a \leftarrow$ base label
Input: $b \leftarrow$ equivalence label
  if $a = b$ **then**
    return
  else if No $equivalences[a]$ exists **then**
    $equivalence[a] \leftarrow b$
  else if $b < equivalences[a]$ **then**
    $makeEquivalence(equivalences, equivalences[a], b)$
    $equivalences[a] \leftarrow b$
  else
    $makeEquivalence(equivalences, b, equivalences[a])$
  end if

Algorithm 5 relabeling

Input: $q \leftarrow$ CUDB Model
Input: $equivalences \leftarrow$ Map of equivalences
Output: $s \leftarrow$ Number of connected components
  for all $baseLabel$ in $equivalences$ **do**
    $index \leftarrow baseLabel$
    while $equivalences[index]$ exists **do**
      $index \leftarrow equivalences[index]$
    end while
    $equivalences[baseLabel] \leftarrow index$
  end for
  $CC \leftarrow$new map<integer,integer> {For CC counting}
  $counter \leftarrow 1$
  for all box in q **do**
    $label \leftarrow box.GetLabel()$
    if exist $equivalence[label]$ **then**
      $box.SetLabel(equivalence[label])$
    end if
    $label \leftarrow box.GetLabel()$
    if No $CC[label]$ exists **then**
      $CC[label] \leftarrow counter$
      $counter \leftarrow counter + 1$
    end if
    $box.SetLabel(CC[label])$ {Consecutive label}
  end for
  return Size of CC

4 CUDB Performance

The new CUDB model has been compared with the OUDB-extended in storage size, conversion (to an from EVM) and CCL speed. Below some experimental results are presented. The testing set consists of one 2D image (Logo) and three volume datasets (Phantom, Aneurysm, Stone) See figure 4. The algorithms have been developed in C++ and run on a PC Intel® Pentium® CPU 3.2 GHz with 3.2 GB of RAM.

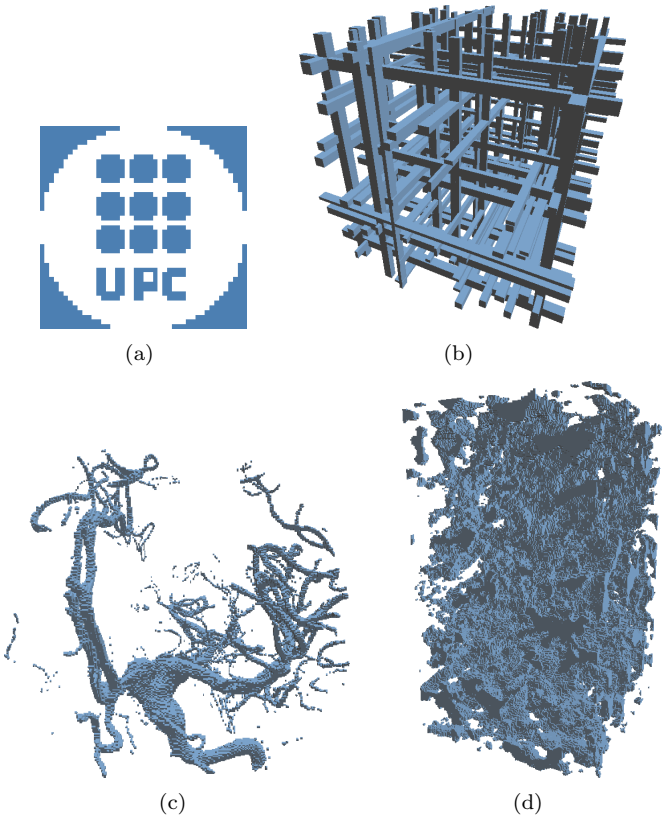
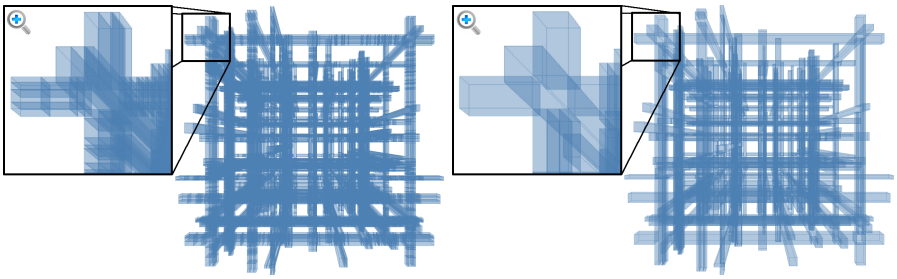


Figure 4: (a) Logo. (b) Phantom. (c) Aneurysm. (d) Stone

Table 1 show the compilation of storage size comparative between EVM, OUDB and CUDB model (the last two in XYZ-sorted). We note that CUDB model is smaller than the others, mainly due to the reduced number of elements. For illustration the Figure 5 shows that the CUDB-XYZ of Phantom has less than 3.9% of boxes of the corresponding OUDB-XYZ decomposition.

Table 2 in turn shows a compilation of performance comparative, here we note that, although the conversion from EVM to CUDB is slower than EVM to OUDB due to the extra effort to compact the boxes, both the inverse conversion and the CCL process are faster.



(a) (b)

Figure 5: Decompositions of a phantom sample . (a) XYZ-OUDB with 24851 boxes. (b) XYZ-CUDB with 954 boxes.

Table 1: Comparative EVM,OUDB,CUDB size in bytes

Dataset	Resolution	EVM	OUDB	CUDB	#boxes OUDB	#boxes CUDB
Logo	50x50	2002	3,590	1,263	145	92
Phantom	500x500x500	57,111	787,349	23,156	24,851	954
Aneurysm	213x215x240	666,501	404,117	275,347	12,825	1,0705
Stone	159x271x179	6,739,767	4,767,472	2,837,183	161,751	124,717

5 Conclusions and future work

We have presented a new decomposition model for orthogonal pseudo-polyhedra, the CUDB, which is a compact version of the OUDB model. We have shown that this new model is smaller in storage size and has a better performance for CCL than OUDB model. Moreover, this model has been satisfactorily applied in our virtual porosimetry approach for computing the narrow throats [11]. We now plan to use CUDB in some other applications such as the pore graph construction and another pore space partition method.

Table 2: Comparative EVM,OUDB,CUDB performance in seconds.

Dataset	EVM to OUDB	EVM to CUDB	OUDB to EVM	CUDB to EVM	OUDB-ext CCL	CUDB CCL
Logo	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Phantom	0.182	0.163	0.205	0.012	0.088	<0.001
Aneurysm	0.351	0.429	0.189	0.139	0.050	0.012
Stone	4.280	13.233	2.163	1.953	2.004	0.162

References

[1] A. Aguilera. *Orthogonal Polyhedra: Study and Application*. PhD thesis, LSI-Universitat Politècnica de Catalunya, 1998.

[2] A. Aguilera and D. Ayala. *Geometric Modeling*, volume 14 of *Computing Supplement*, chapter Converting Orthogonal Polyhedra from Extreme Vertices Model to B-Rep and to Alternating Sum of Volumes, pages 1 – 28. Springer, 2001.

[3] D. Ayala and E. Vergés. Improved virtual porosimeter. In *CASEIB'08*, 2008.

[4] M. Dillencourt, H. Samet, and M. Tamminen. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM*, 39(2):253 – 280, 1992.

[5] A. Greßand R. Klein. Efficient representation and extraction of 2-manifold isosurfaces using kd-trees. *Graphical Models*, 66:370 – 397, 2004.

[6] G. J. Grevera, J. K. Udupa, and D. Odhner. 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*, 6(4):335–345, 2000.

[7] B. Kim, J. Seo, and Y. Shin. Binary volume rendering using Slice-based Binary Shell. *The Visual Computer*, 17:243 – 257, 2001.

[8] W. R. Quadros, K. Shimada, and S. J. Owen. 3d discrete skeleton generation by wave propagation on pr-octree for finite element mesh sizing. In *Proc. ACM Symposium on Solid Modeling and Applications*, pages 327 – 332, 2004.

[9] J. Rodríguez and D. Ayala. Fast neighborhood operations for images and volume data sets. *Computers & Graphics*, 27:931–942, 2003.

[10] J. Rodríguez, D. Ayala, and A. Aguilera. *Geometric Modeling for Scientific Visualization*, chapter EVM: A Complete Solid Model for Surface Rendering, pages 259–274. Springer Verlag, 2004. ISBN: 3-540-40116-4.

[11] J. Rodríguez, I. Cruz, E. Vergés, and D. Ayala. Skeletonless porosimeter simulation. In M. Choverand and M. O. Eds., editors, *Proceedings of CEIG 2010*, pages 49–56. Ed. Ibergarceta, 2010.

[12] A. Rosenfeld and J. Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM*, 13(4):471–494, 1966.

[13] H. Samet. *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[14] L. Thurfjell, E. Bengtsson, and B. Nordin. A boundary approach to fast neighborhood operations on three-dimensional binary data. *CVGIP: Graphical Models and Image Processing*, 57(1):13 – 19, 1995.

[15] J. Wilhems and A. V. Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[16] W. Wong, F. Y. Shih, and T. Su. Thinning algorithms based on quadtree and octree representations. *Information Sciences*, 176:1379 – 1394, 2006.