Finite Boolean Algebra for Solid Geometry with Julia's Sparse Arrays

Alberto Paoluzzi

Workshop on Computer Modeling for Research on Liver Perfusion - Pilsen 2019

Summary of talk

1. Roman Research on Solid Modeling
2. Motivation for a new start
3. Linear Algebraic Representation
4. LAR and Imaging
5. Novel Approach to Solid Geometry Algebra
6. Julia as Language for Numerical Computations

Roman Research on Solid Modeling

Modeling a large architectural complex (CAD&A, 2008)

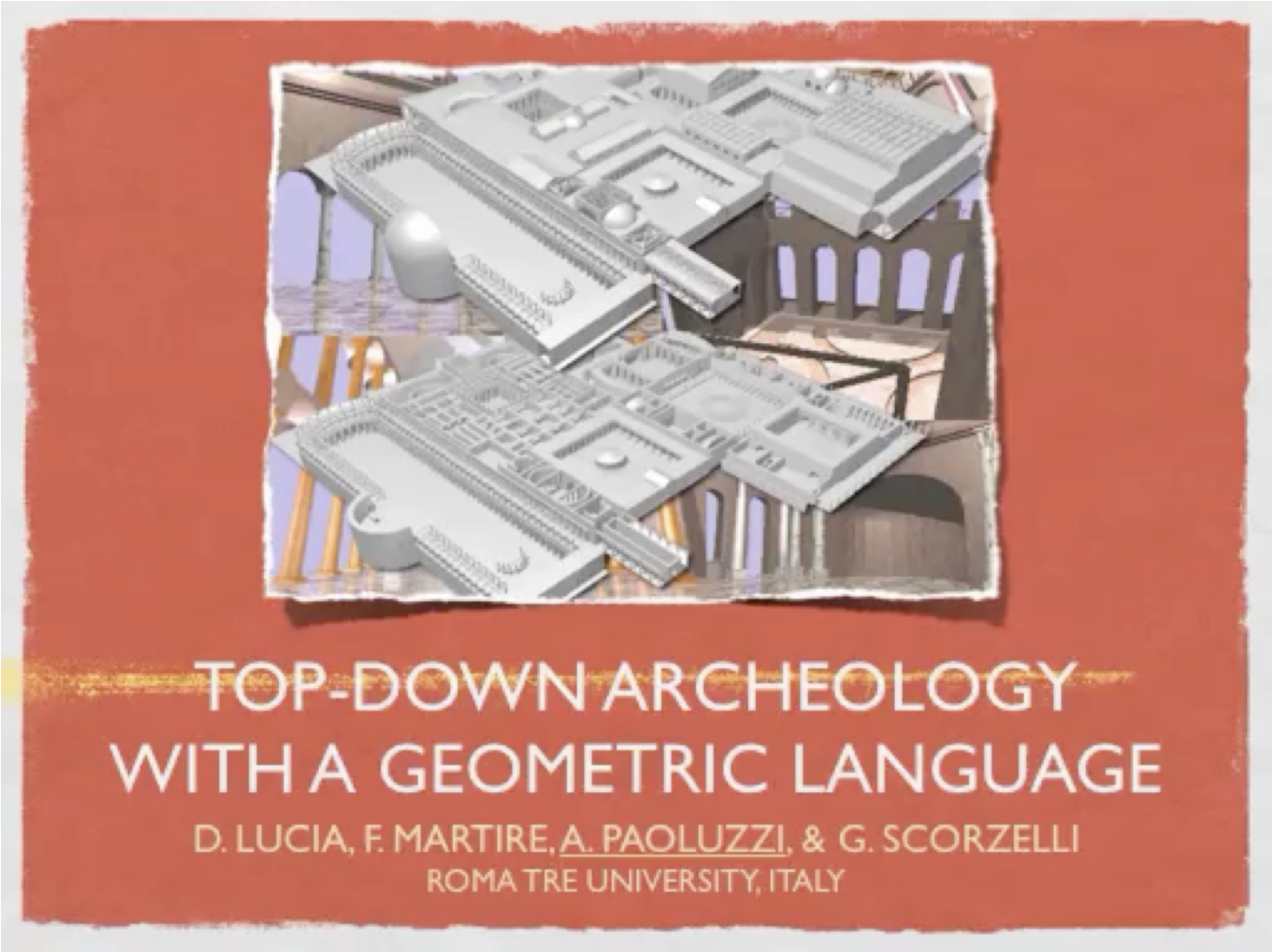


Figure 1: Apple. The emperor's palaces of Palatin hill

The context of our start in Solid Modeling: 1984-1988

solid modeling R&D was at the time on Lisp-machines!

Figure 2: Symbolics LISP Machine, from Google NY office computer museum

Figure 3: IBM XT personal computer

Figure 4: Apple Macintosh SE

we started on IBM first PCs and Apple Macintosh writing Solid Modeling software in Pascal

Minerva: the first solid modeler on a PC (1985–)

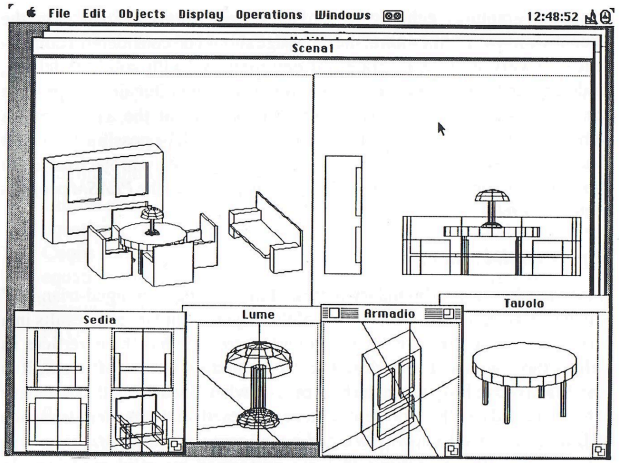


Figure 1: The window-based user interface of *Minerva*, with a structure definition

Figure 5: Apple GUI

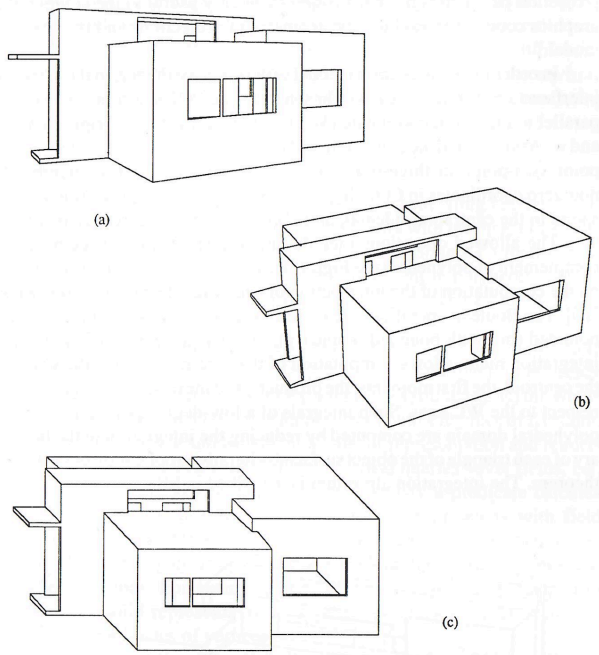


Figure 3: Solid model of the Oud's Mathenesse (Rotterdam, 1923) generated by *Minerva*. (a) Two-point perspective; (b) three-point perspective; (c) dimetric Cavalier axonometric view

Figure 6: Standard views of Mathenesse building

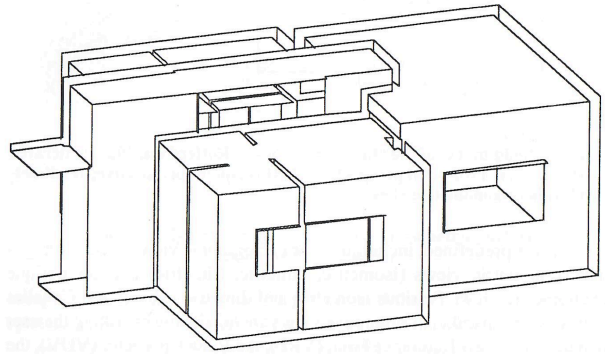


Figure 4: Hidden surface removed view of the complemented Mathenesse building

Figure 7: Boolean Algebra over Linear Polyhedra, CAD 1990

Integrating hierarchical assemblies, Boolean ops, parametric surfaces, integration of polynomials, HSR algorithms, full-fledged Apple GUI

PLaSM: FL-based Language for Solid Modeling (1992-)

Programming language for solid variational geometry (CAD 92, TOG 95)

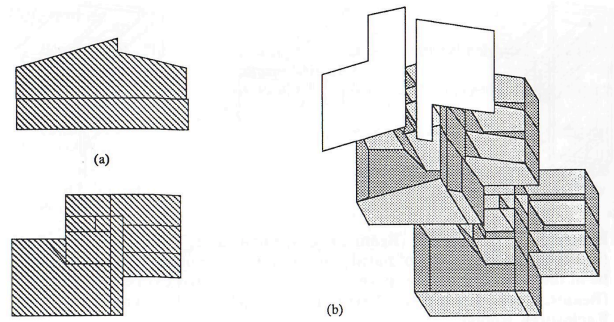


Figure 6: (a) The arguments *Plan* and *Section* of an operation && (intersection cell-by-cell of extrusions) in the *PLASM* language; (b) The two-dimensional polyhedral complex obtained by the evaluation of the expression @2:(*Plan* && *Section*), represented as an exploded drawing

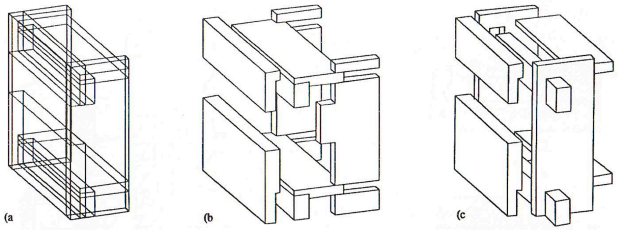


Figure 7: (a) A sequence (Beams, Roofs, Enclosures, Partitions) of isothetic (interpenetrating) pairs of parallelopipeds, for sake of simplicity assumed to be in the same coordinate space; (b) *PLASM* evaluated expression *STRUCT*: (Beams, Roofs, Enclosures, Partitions); (c) STRUCT: (Beams, Partitions, Enclosures, Roofs)

Example 1.5.6 (Building facade)
An example is given in Script 1.5.6, and shown in Figure 1.6a, where a 2D complex is generated by Cartesian product of 1D complexes produced by the `QUOTE` operator. Several other examples of the `QUOTE` operator are given in Section 1.6.2.

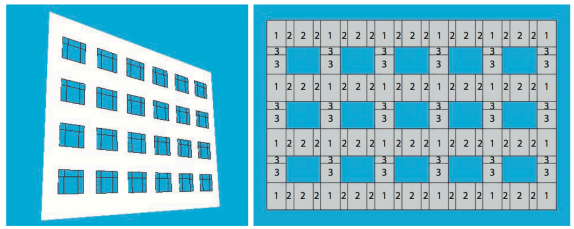


Figure 1.6 (a) the 2D complex generated as `facade:<6,4>` by combining `QUOTE`, product and 1-skeleton operators (b) the generating scheme of `facade` panels

INTRODUCTION TO FL AND PLASM 31

The `facade` generating function works by assembling three 2D Cartesian products of alternating 1D complexes, produced by `Q:xRithm`, `Q:xVoid` and by `Q:yRithm`, `Q:yVoid`, respectively. In particular, the `xRithm` sequence contains the numeric series used in the x direction; analogously `yRithm` for the y direction. Conversely, `xVoid` and `yVoid` host the series with opposite signs of elements. So, the first three Cartesian products in the `STRUCT` sequence produce a sort of checkboard covering that follows the scheme given in Figure 1.6b.

Script 1.5.6 (Building facade)

```
DEF facade (n,m::IsIntPos) = STRUCT:<
   Q:xRithm * Q:yRithm,
   Q:xVoid * Q:yRithm,
   Q:xRithm * Q:yVoid ,
   @1:(Q:xVoid * Q:yVoid) >
WHERE
   xRithm = ##:n:<5,-2,-5,-2> AR 5,
   yRithm = ##:m:<7,-5,-2> AR 7,
   xVoid = AA:-:xRithm,
   yVoid = AA:-:yRithm
END;
```

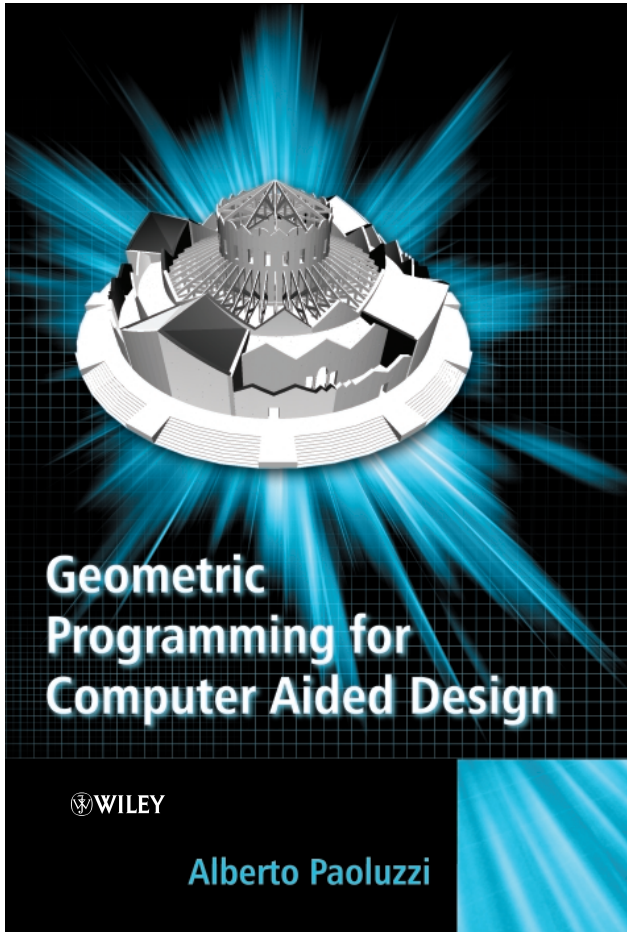


Figure 8: GP4SM (2003)

Multidimensional representations and algorithms

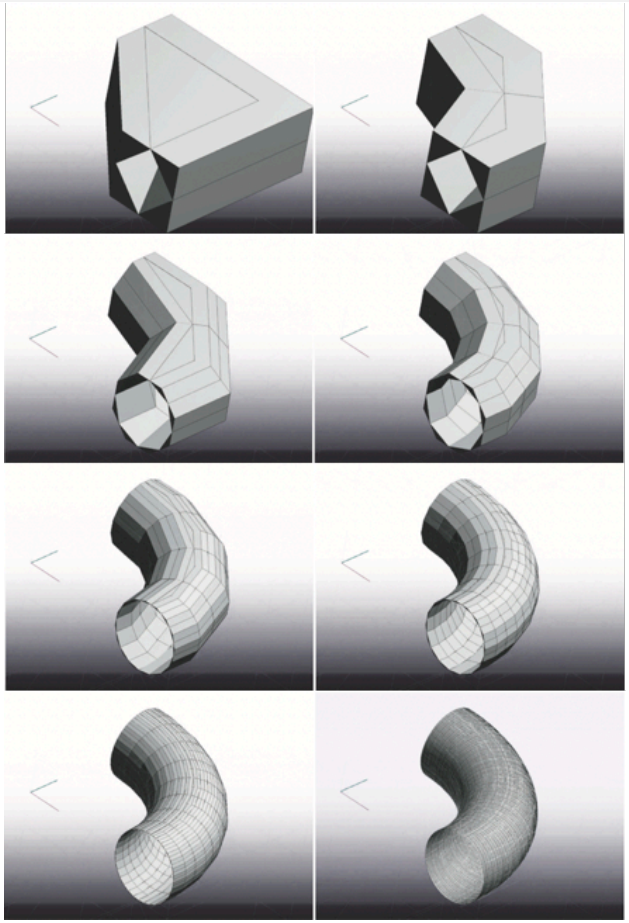


Figure 9: Dataflow refinement of rational B-spline with BSP nodes (SM&A, 2004)

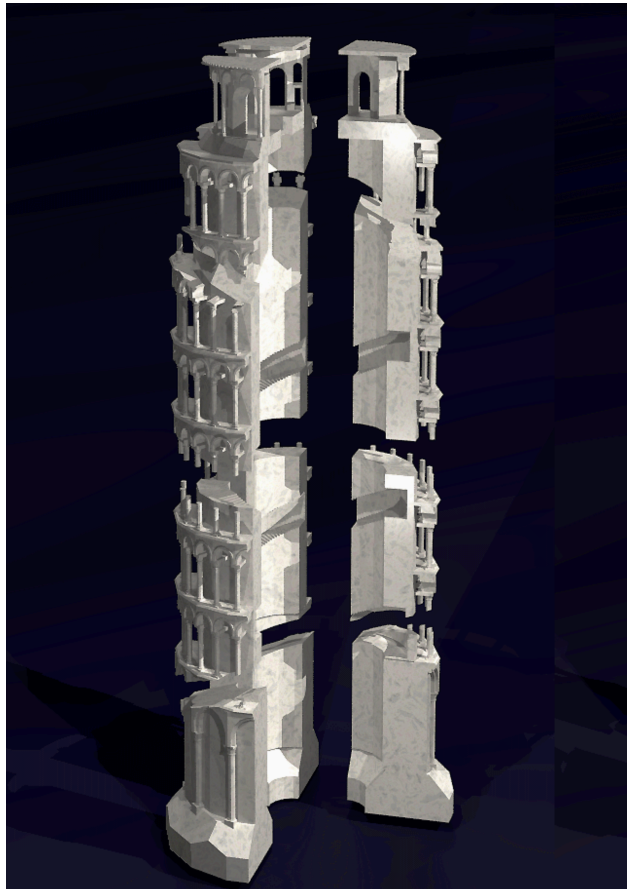


Figure 10: Parallel & distributed computing ((IJCGA, 2008)

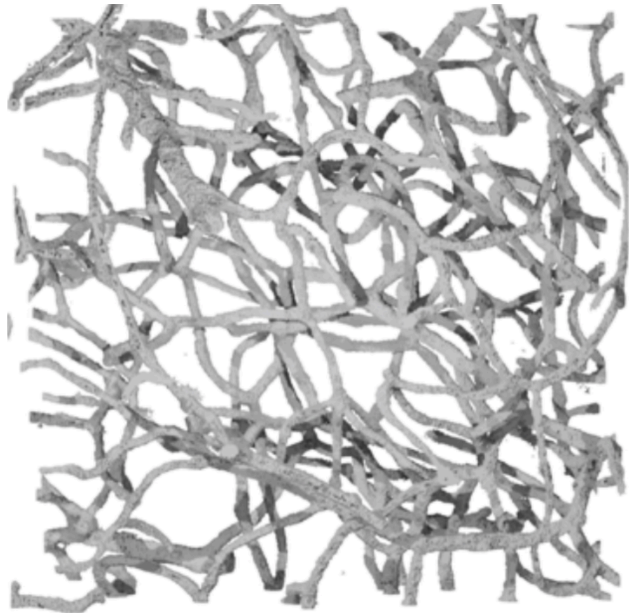


Figure 11: Topologically exact LAR models of microvessels between neurons (CAD&A, 2016)

LAR: Linear Algebraic Representation

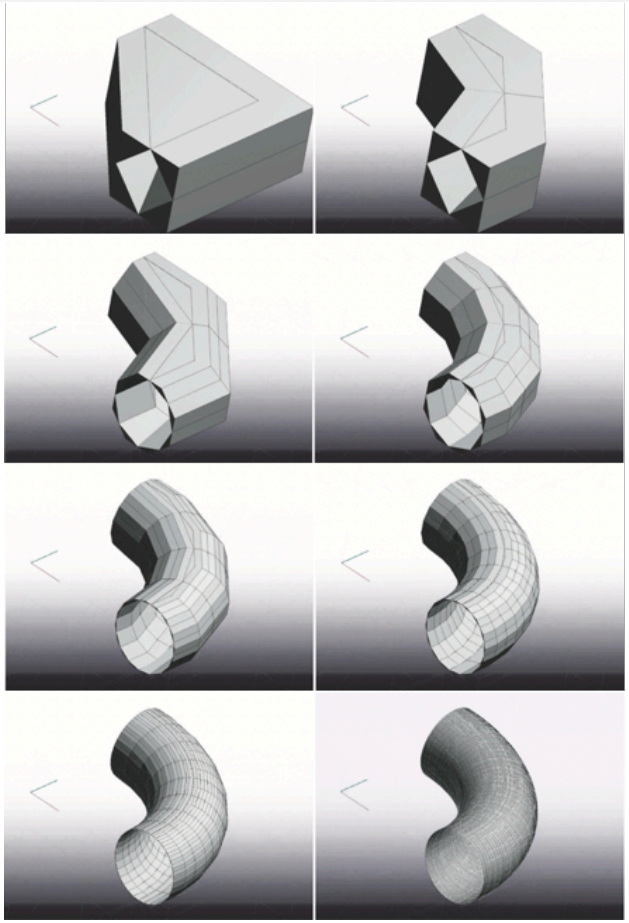


Figure 12: Dataflow refinement of rational B-spline with BSP nodes (SM&A, 2004)

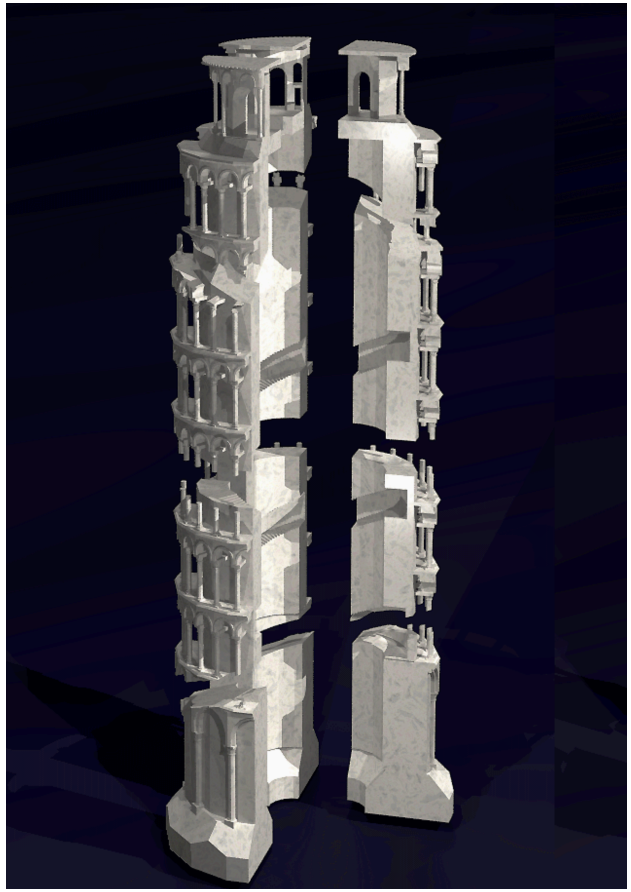


Figure 13: Parallel & distributed computing ((IJCGA, 2008)

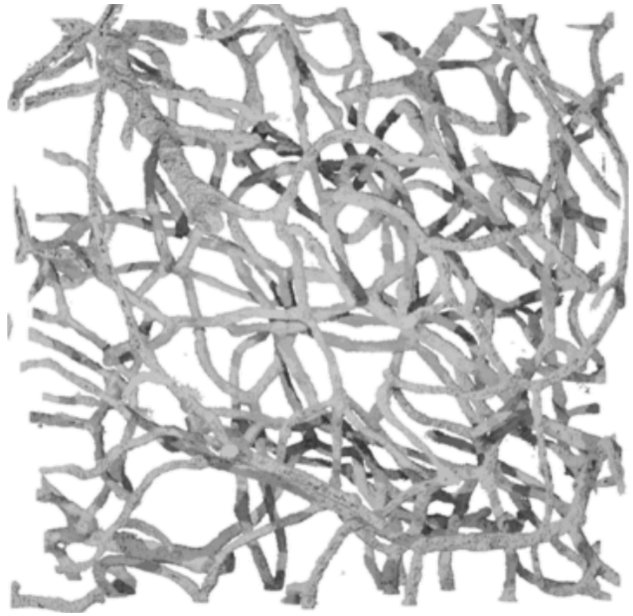
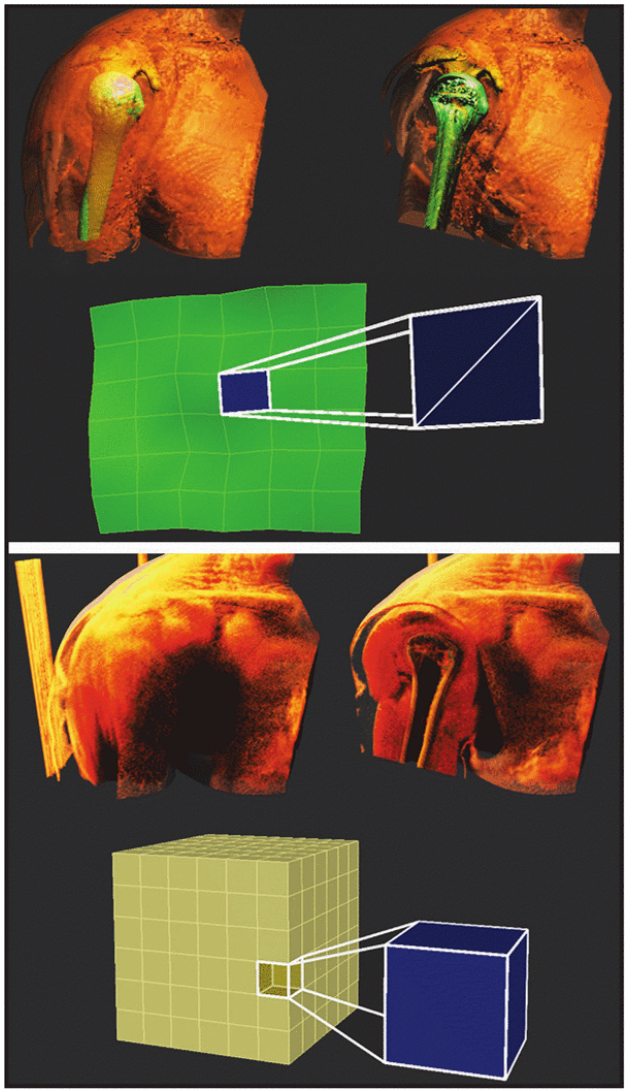
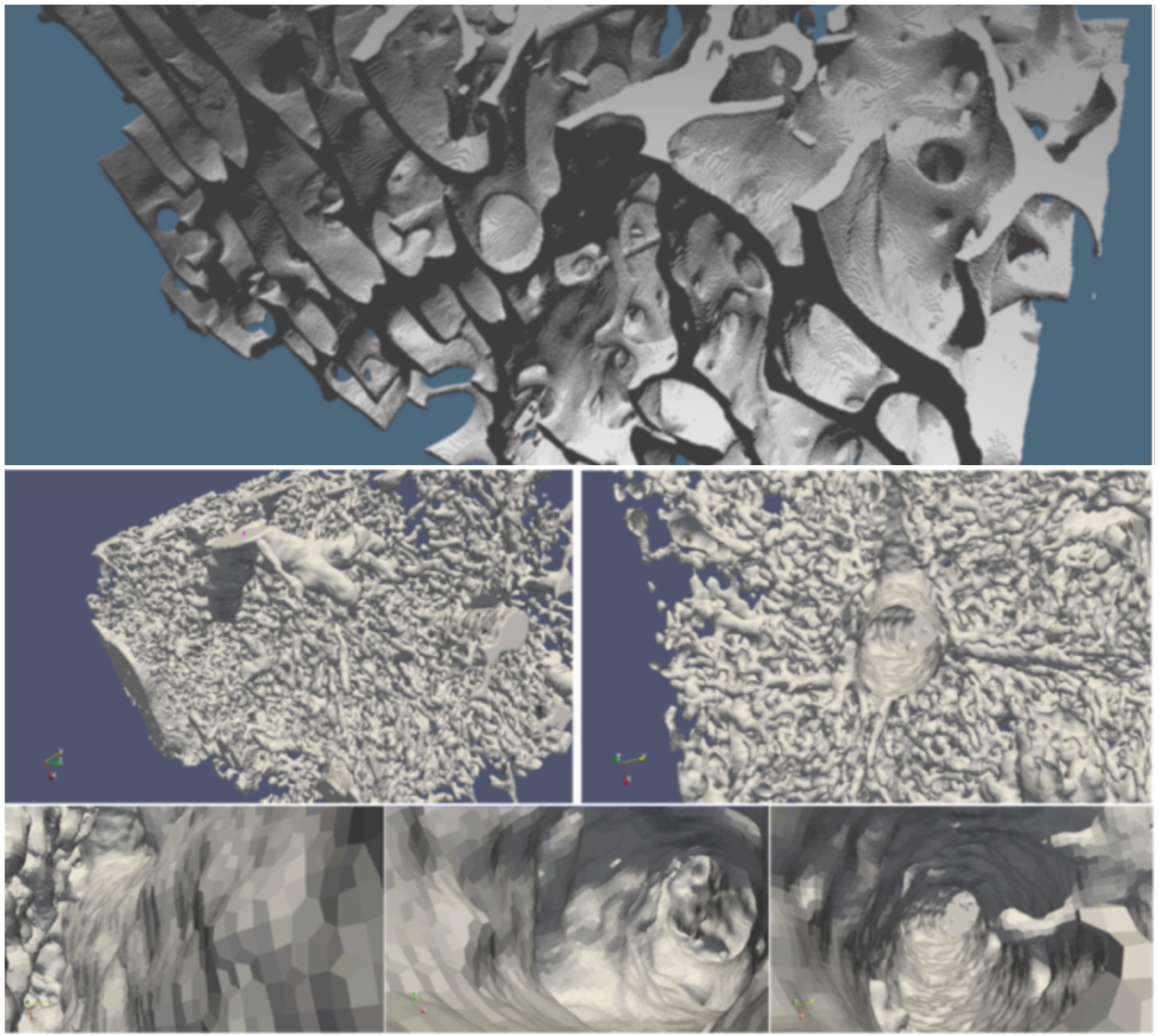


Figure 14: Topologically exact LAR models of microvessels between neurons (CAD&A, 2016)

Challenges: persolized biomedicine & virtual surgery

SMA should enter the IEEE-SA P.3333.2 WG !!s





Motivation for a new start

Geometric and topological computing

rethinking some foundations

Complexity of geometric information stems from dramatic increase in size, diversity, and complexity of geometric data:

- point clouds,
- boundary meshes,
- NURBs representations,
- finite element meshes,
- CT scans,
- and so on

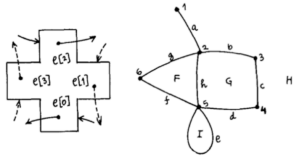
Emerging applications (e.g. medical 3D) require the convergence of data structures from:

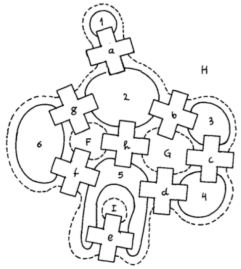
- 3d computer imaging
- computer graphics
- solid modeling
- computer-aided geometric design
- discrete meshing of domains
- physical simulations

The goals of unification, scalability, and distributed computing call for rethinking some foundations of geometric and topological computing

Quad-Edge data structure

(Guibas & Stolfi, ACM Transactions on Graphics, 1985)





(a) Edge record showing Next links.
(b) A subdivision of the sphere.
(c) Data structure for the subdivision

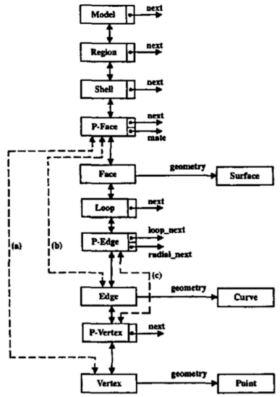
Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams

largely used in computational geometry algorithms and in geometric libraries

Partial-Entity data structure

(Sang Hun Lee & Kunwoo Lee, ACM Solid Modeling, 2001)

Compact Non-Manifold Boundary Representation Based on Partial Topological Entities



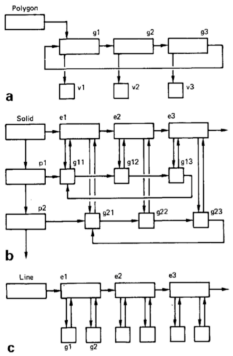

```
class Entity {
  int        _id;
  Attribute  *_attribute;
};
class Model : public Entity {
  Model      *_next;        // next model
  Region     *_region;      // list of regions
};
class Region : public Entity {
  Region     *_next;        // link field of the region list of a model
  Model      *_model;       // parent model
  Shell      *_shell;       // peripheral shell
};
class Shell : public Entity {
  Shell      *_next;        // next void shell
  Region     *_region;      // parent region
  Pface      *_pface;       // partial face
};
class Pface : public Entity {  // partial face (p-face) class
  Pface      *_next;        // next p-face
  Shell      *_shell;       // parent shell
  Entity     *_child;       // child entity: a face, an edge, or a vert
  Orient     _orient;       // orientation flag w.r.t. the face normal
  Pface      *_mate;        // mate p-face
};
class Face : public Entity {
  Pface      *_pface;       // one of two incident p-faces
  Loop       *_loop;        // peripheral loop
  Surface    *_geometry;    // surface
};
```

```
class Loop : public Entity {
  Loop       *_next;        // next hole loop
  Face       *_face;        // parent face
  Pedge      *_pedge;       // a p-edge in a loop
};
class Pedge : public Entity {  // partial edge (p-edge) class
  Loop       *_loop;        // parent loop
  Entity     *_child;       // child entity: an edge or a p-vertex
  Orient     *_orient;      // orientation flag w.r.t. the edge direction
  Pvertex    *_vertex;      // start p-vertex
  Pedge      *_looped_prev; // previous p-edge in the loop cycle
  Pedge      *_looped_next; // next p-edge in the loop cycle
  Pedge      *_radial_prev; // previous p-edge in the radial cycle
  Pedge      *_radial_next; // next p-edge in the radial cycle
};
class Edge : public Entity {
  Entity     *_parent;      // parent entity: a p-edge or a p-face
  Pvertex    *_pvertex[2];  // two end p-vertices
  Curve      *_geometry;    // curve
};
class Pvertex : public Entity {  // partial vertex (p-vertex) class
  Pvertex    *_next;        // another p-vertex associated with _vertex
  Entity     *_parent;      // parent entity: an edge or a p-edge
  Vertex     *_vertex;      // mother vertex
};
class Vertex : public Entity {
  Entity     *_parent;      // parent entity: a p-vertex or a p-face
  Point      *_geometry;    // position
};
```


Hybrid Edge data structure

(Kalay, Computer-Aided Design, 1989)

a topological data structure for vertically integrated geometric modelling

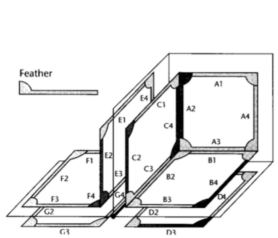



```
type   vptr   = ^vertex;
       gptr   = ^segment;
       pptr   = ^polygon;
       eptr   = ^edge;
       sptr   = ^solid;
       lptr   = ^lines;

solid  = record
              snext     : sptr;
              elist     : eptr;
              plist     : pptr;
              case cavity : boolean
                 false : (child  : sptr)
                 true  : (parent : sptr)
           end;

lines  = record
              lnext     : lptr;
              edge      : eptr;
           end;
```

```
vertex  = record
              form      : point
           end;

segment = record
              gnext     : gptr;
              form      : line;
              gpoly     : pptr;
              gedge     : eptr;
              gvert     : vptr
           end;

polygon = record
              pnext     : pptr;
              glist     : gptr;
              form      : plane;
              case hole : boolean of
                 false  : (child  : pptr);
                 true   : (parent : pptr)
           end;

edge    = record
              enext     : eptr;
              eseg      : array [1..2] of gptr;
              esolid    : sptr
           end;
```


Coupling Entities data structure

(Yamaguchi & Kimura, IEEE Computer Graphics and Applications, 1995)



Fan
B4 & D3 :D3 = FM(B4), B4 = FM(D3)

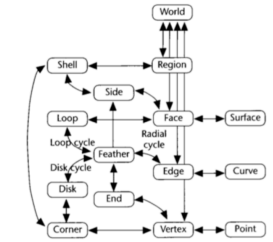
Blade
C2 & E2 :E2 = BM(C2), C2 = BM(E2)

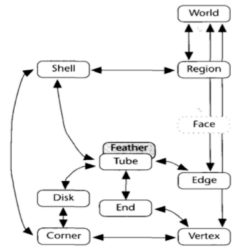
Wedge
A2 & C4 :C4 = WM(A2), A2 = WM(C4)

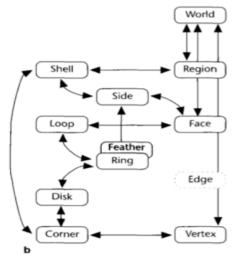
Loop cycle
A1>>A2>>A3>>A4 :A2 = CCL(A1), A3 = CCL(A2)
A4>>A3>>A2>>A1 :A1 = CL(A2), A2 = CL(A3)

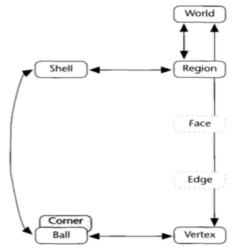
Radial cycle
D2>>C3>>F4 :C3 = CCR(D2), F4 = CCR(C3)
F4>>C3>>D2 :D2 = CR(C3), C3 = CR(F4)

Disk cycle
A3>>C4>>B2 :C4 = CCD(A3), B2 = CCD(C4)
B2>>C4>>A3 :A3 = CD(C4), C4 = CD(B2)









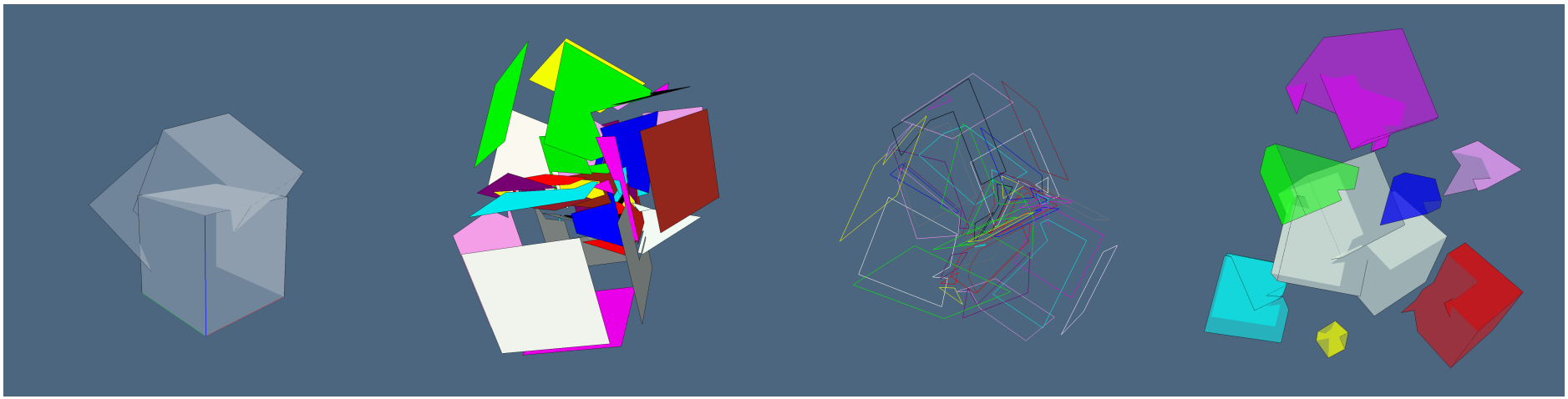
Linear Algebraic Representation

Solid modeling with sparse arrays

Chain Complex

$$C_\bullet = (C_p, \partial_p) := C_3 \underset{\partial_3}{\overset{\delta^2}{\longleftrightarrow}} C_2 \underset{\partial_2}{\overset{\delta^1}{\longleftrightarrow}} C_1 \underset{\partial_1}{\overset{\delta^0}{\longleftrightarrow}} C_0 \qquad \partial_p^\top = \delta^{p-1}$$

```
Int8[
-1 0 1 0 0 0 1 0 -1 0 -1 0 0 0 -1 1 0 1 0 0 -1 1 0 1 -1 0 0 0 -1 -1 0 1 0 0 -1 0 1 0 0 0 1 -1 0 -1 0 0 1;
 0 -1 0 0 0 1 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 -1 0 0 0 0 0 0 1 0 -1 0 0 0 -1 1 0 1 0 0 -1;
 0 0 0 -1 0 0 0 -1 0 0 0 1 1 0 0 0 -1 0 0 1 0 0 -1 0 0 0 1 0 0 0 0 0 0 -1 0 0 0 0 1 0 0 0 0 0 1 0 0;
 0 0 0 0 -1 0 0 0 0 0 0 0 0 1 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 -1 0 0 0 0 0 -1 0 0;
 1 0 -1 0 0 0 -1 0 1 0 1 0 0 0 1 -1 0 -1 0 -1 0 0 1 0 0 0 -1 0 0 0 0 0 -1 0 0 0 0 1 0 0 0 0 -1 0 0 0 0;
 0 0 0 1 0 0 0 1 0 0 0 -1 -1 0 0 0 1 0 0 0 1 -1 0 -1 1 0 0 0 1 1 0 -1 0 0 0 -1 0 0 0 1 0 0 0 0 0 1 0;
 0 1 0 0 0 -1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 1 0 0 0 0 -1 0 0 0 0 1 0 0 0 0;
 0 0 0 0 1 0 0 0 0 0 0 0 0 -1 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 -1 0 0 0 0 0 -1 0]
```



Chain complex (the topology of space partition)

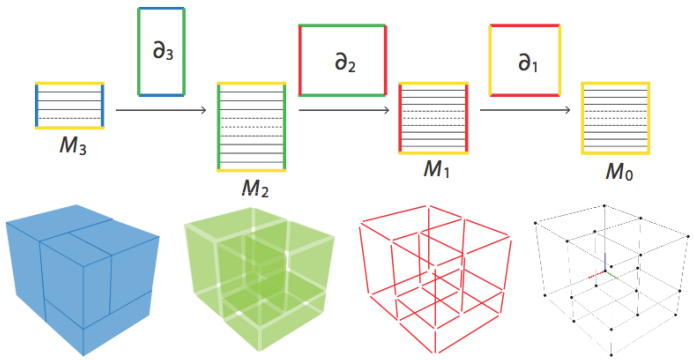
A sequence of linear spaces and linear operators

$$C_\bullet = (C_p, \partial_p) := C_3 \underset{\partial_3}{\overset{\delta^2}{\leftrightarrows}} C_2 \underset{\partial_2}{\overset{\delta^1}{\leftrightarrows}} C_1 \underset{\partial_1}{\overset{\delta^0}{\leftrightarrows}} C_0 \qquad \partial_p^\top = \delta^{p-1}$$

A **complex** C is a sequence $\cdots \longrightarrow C_{d+1} \xrightarrow{\partial_{d+1}} C_d \xrightarrow{\partial_d} C_{d-1} \longrightarrow \cdots$ of **linear** spaces C_d and **linear** boundary maps ∂_d, where $\partial_{d+1} \circ \partial_d = 0$, for all d

Chain and cochain complex

A chain complex C is a complex of chain spaces and boundary maps:



Unit d-chains (single d-cell subsets), are the standard bases (M_d rows) of d-chain spaces

Characteristic matrices in CSR matrix form

$M_3 =$

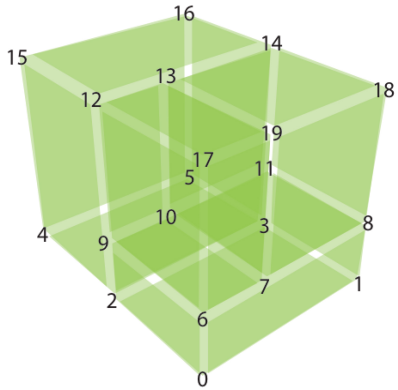
```
0 0 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0
1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 1 1 1 0 1 1 0 0 0 1 0 1
0 0 0 0 0 0 0 1 1 0 1 1 0 1 1 1 0 0 0 1 1
```

```
2 3 4 5 9 10 11 12 13 14 15 16
0 1 2 3 6 7 8 9 10 11
6 7 9 10 12 13 17 19
7 8 10 11 13 14 18 19
```

$M_2 =$

```
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
1 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0
0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0
0 0 1 0 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0
0 0 0 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0
0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0
0 0 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1
```

```
0 1 2 3
0 1 6 7 8
0 2 6 9
1 3 8 11
2 3 4 5
2 3 9 10 11
2 4 9 12 15
3 5 11 14 16
4 5 15 16
6 7 9 10
6 7 17 19
6 9 12 17
7 8 10 11
7 8 18 19
7 10 13 19
8 11 14 18
9 10 12 13
10 11 13 14
12 13 14 15 16
12 13 17 19
13 14 18 19
```



d-cells as subsets of vertices

CSR form of a binary matrix

Compressed Sparse Row (CSR) matrix storage

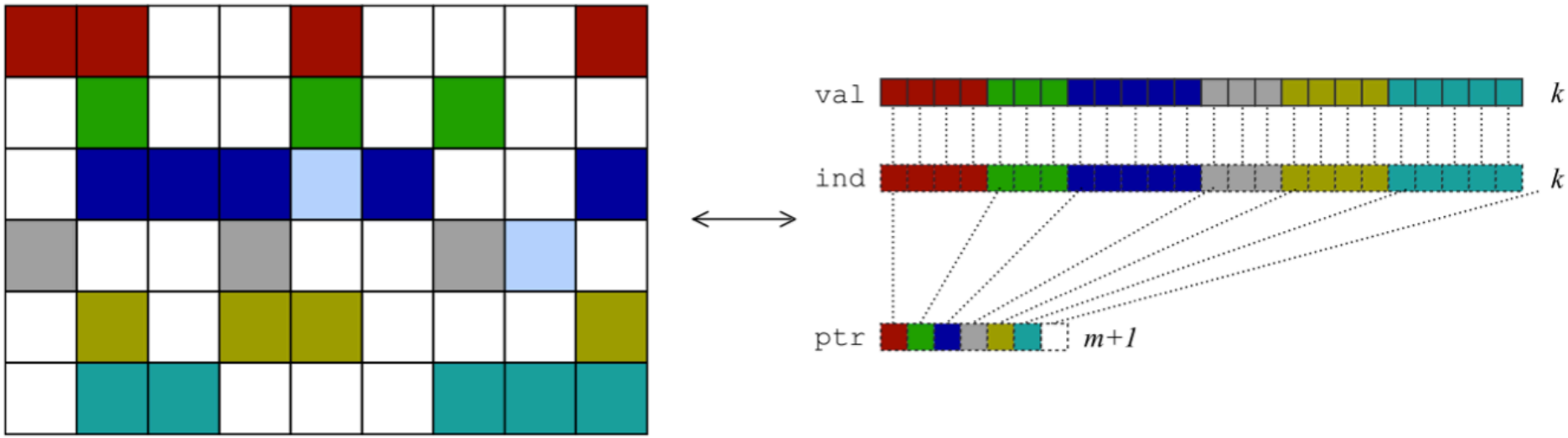


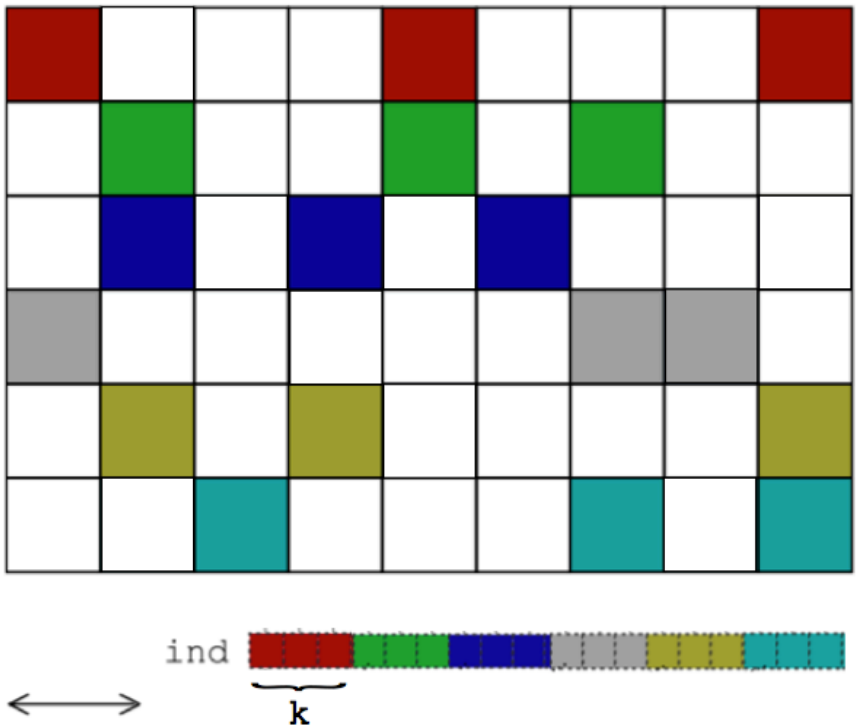
image from

Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel, Optimization of sparse matrix-vector multiplication on emerging multicore platforms, Proceedings of the 2007 ACM/IEEE conference on Supercomputing (New York, NY, USA), SC '07, ACM, 2007, pp. 38:1–38:12.

Storage of LAR(X) $\equiv$ CSR(M_d) matrix

Amazingly compact storage of a **solid** model

REDUCED LAR



simplicial d-complexes: $k = d + 1$
cuboidal d-complexes: $k = 2^d$

Remark (Input and **long-term storage**)

$$space(\text{LAR}) = |FV| = 2|E| \quad !!!$$

Remark (**Full** topology representation)

$$|VE| + |VF| = 4|E|$$

Remark (**Any** topological queries)

single SpMV multiplication

Remark (Sparse Matrix-Vector Multiplication)

is one of the most important computational kernels, for very effective iterative solution methods

LAR and Imaging

LAR: topology extraction from *d*-images

Fast algebraic extraction (GPGPU) of spongy bone's **exact** topology

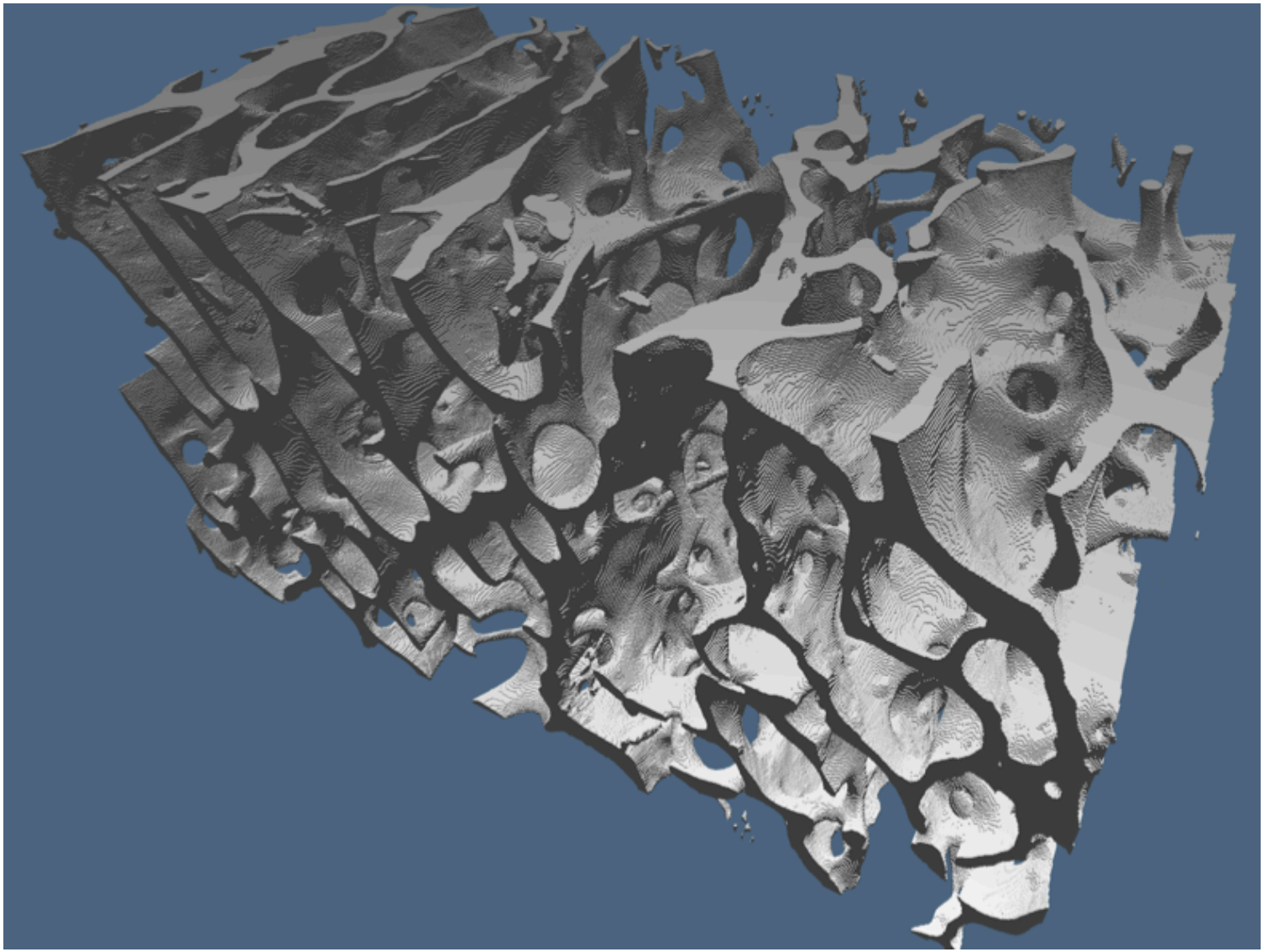
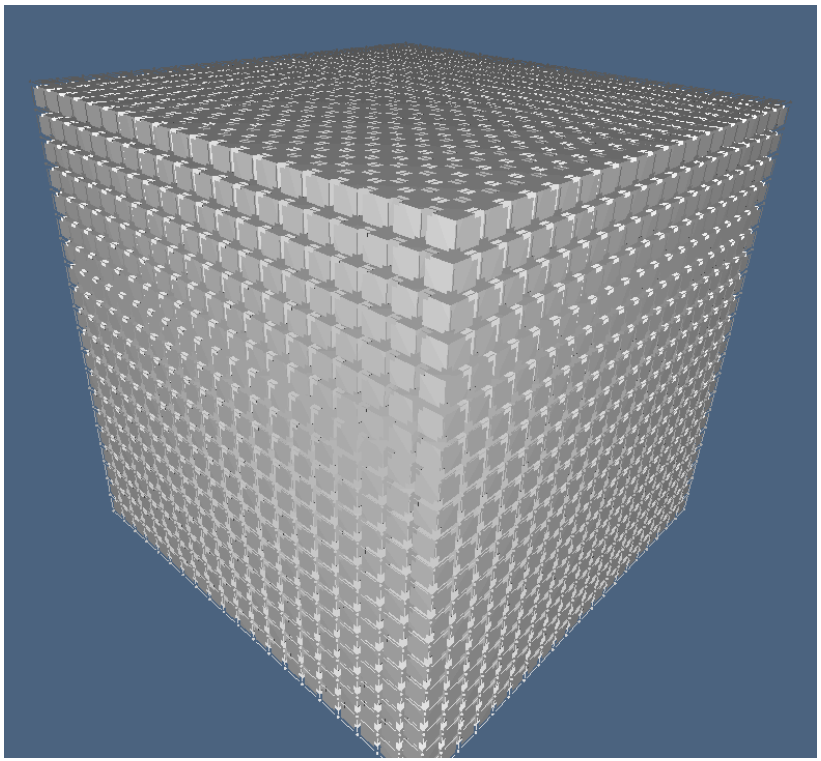


Image complex = implicit LAR of voxels

cell description is made by 4 or 8 indices

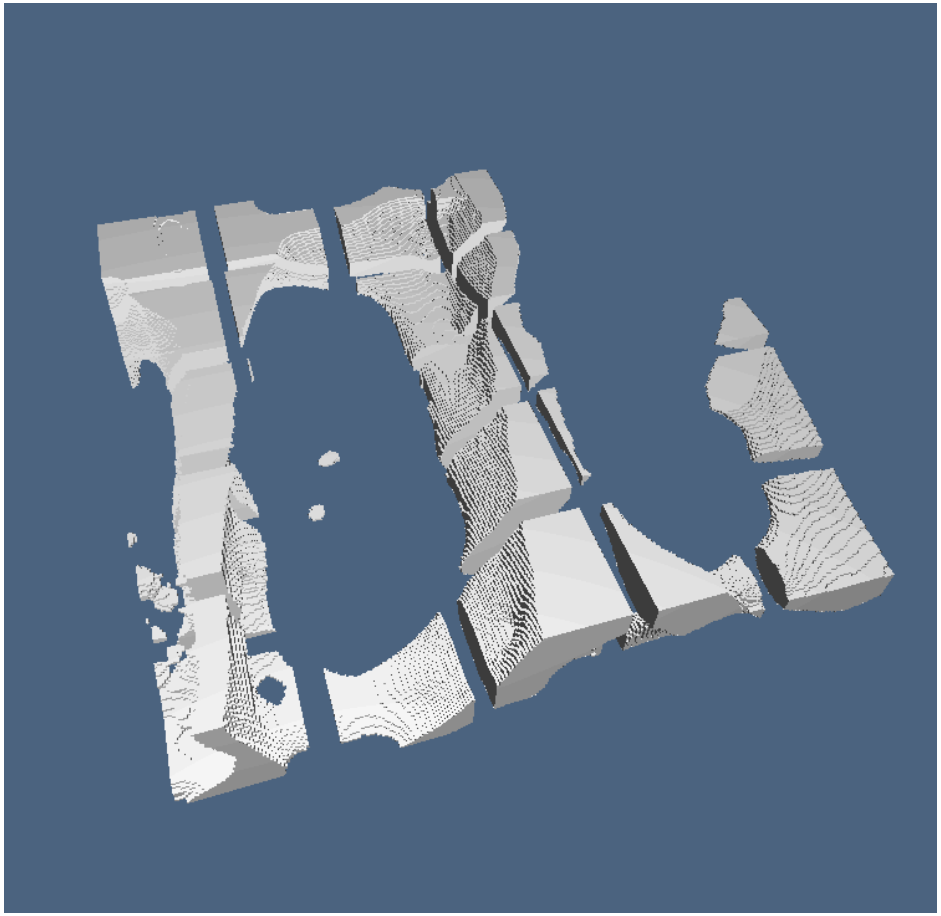
`cell[i,j,k] = [` $(i,j,k),(i+1,j,k),(i,j+1,k),(i,j,k+1),(i+1,j+1,k),(i+1,j,k+1),(i,j+1,k+1),(i+1,j+1,k+1)$ `]`

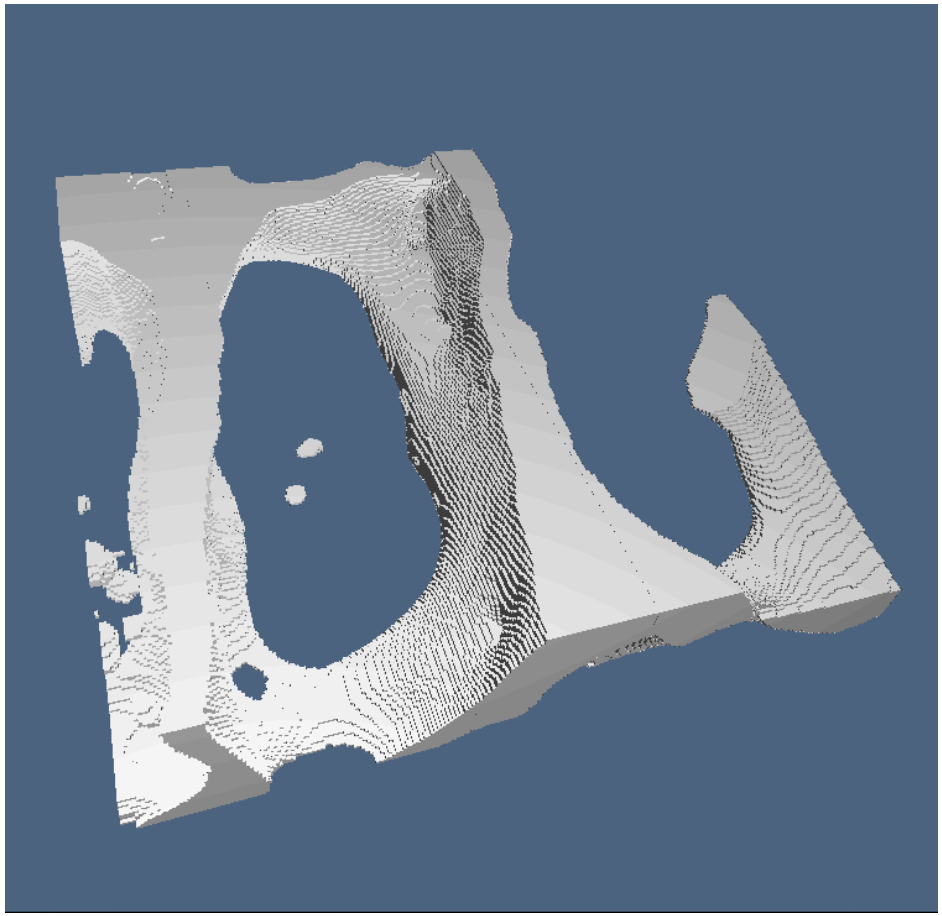


the boundary matrix may be computed only one time (depending on the image size), and stored on disk

Paradigm: Divide et Impera

Bottleneck of GPGPU: moving data from global to local memory





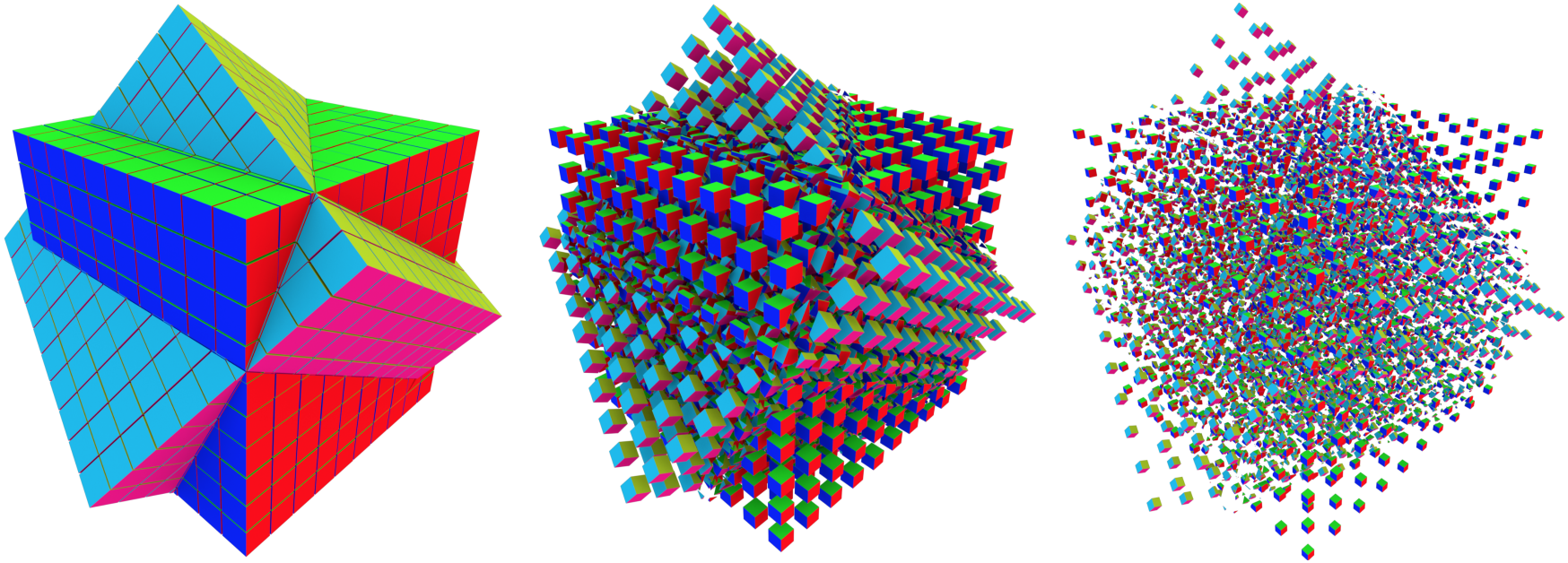
Solution: store the (sparse) $[\partial_3]$ of n^3 voxels in device Constant Memory, and move the (binary) coordinate vectors of chains in Private Memory

Novel Approach to Solid Geometry Algebra

Space Arrangement

Partition of $\mathbb{E}^3$ space induced by the input complexes

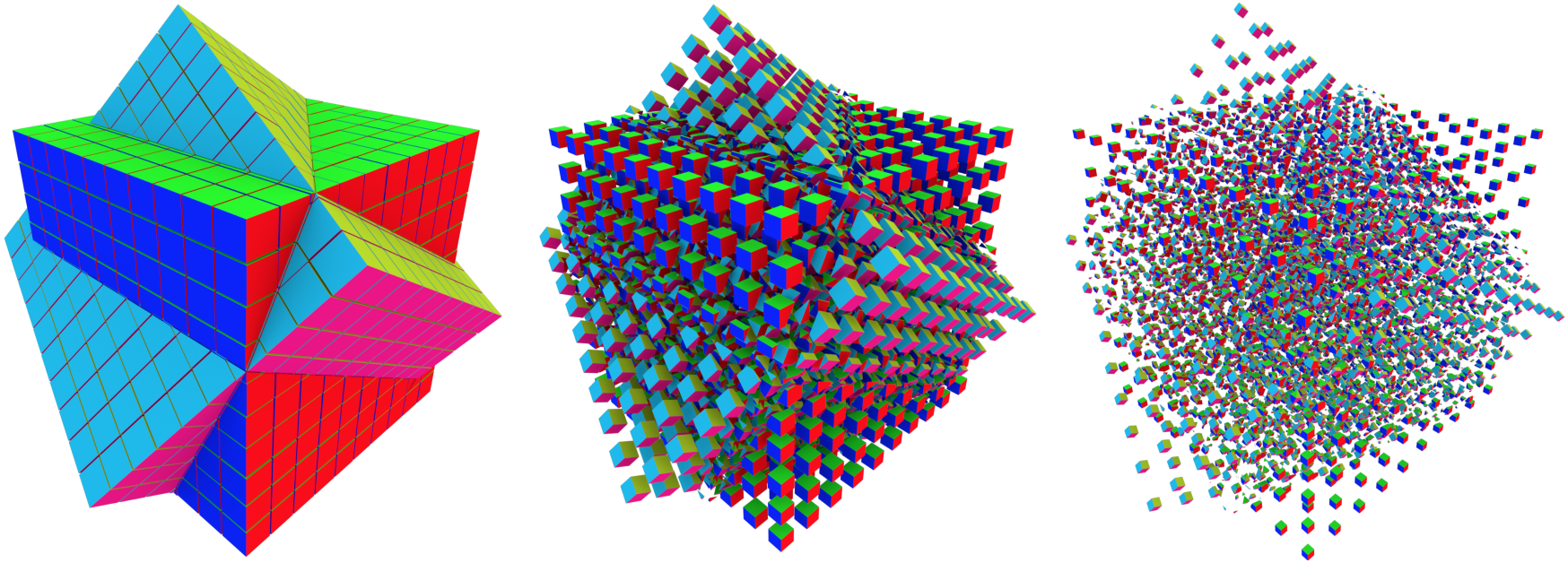
3D cells are the columns of boundary operator matrix $[\partial_3]$



Space Arrangement

Partition of $\mathbb{E}^3$ space induced by the input complexes

3D cells are the columns of boundary operator matrix $[\partial_3]$

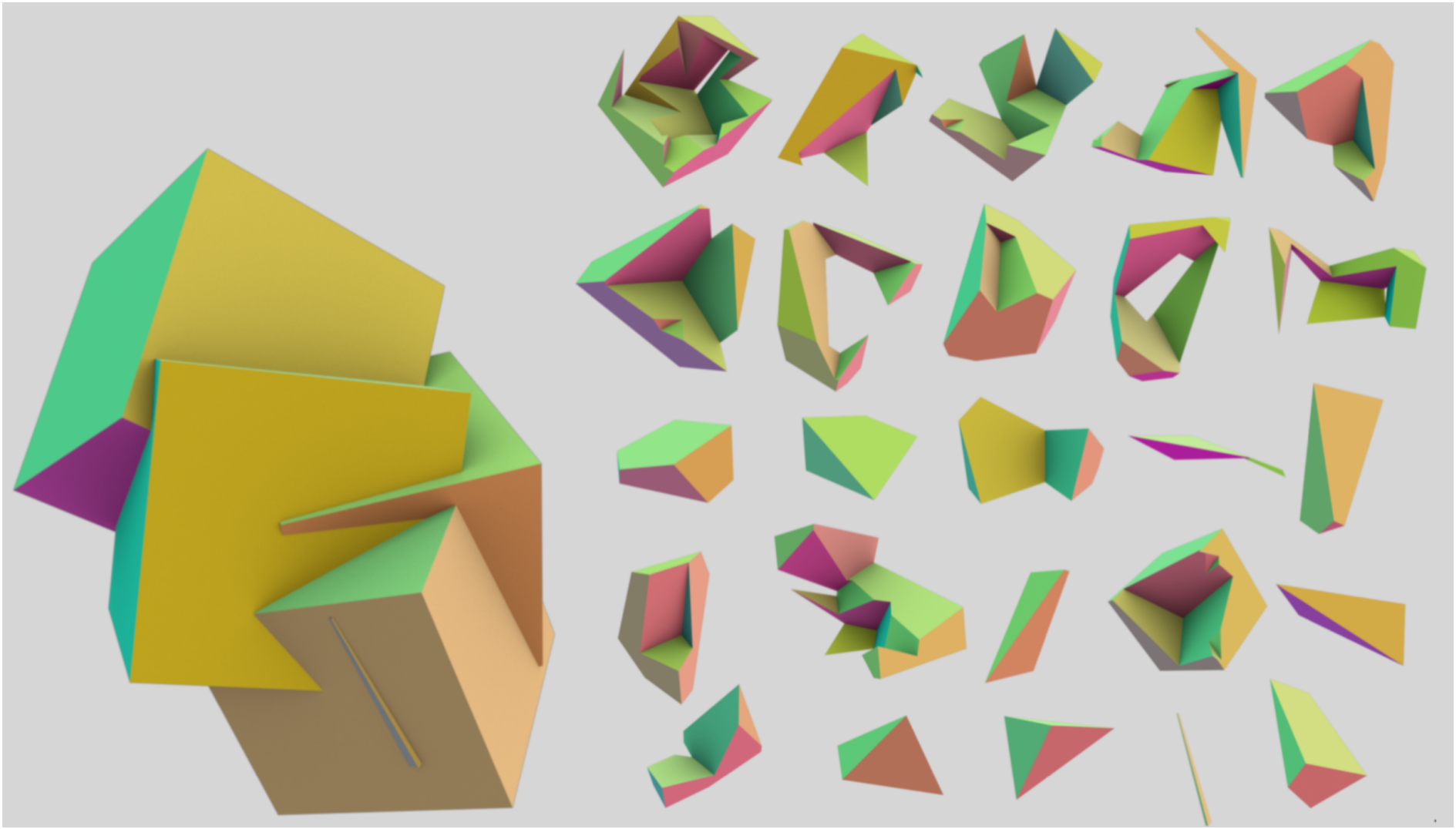


but, also:

3D cells are the atoms of a finite Boolean algebra

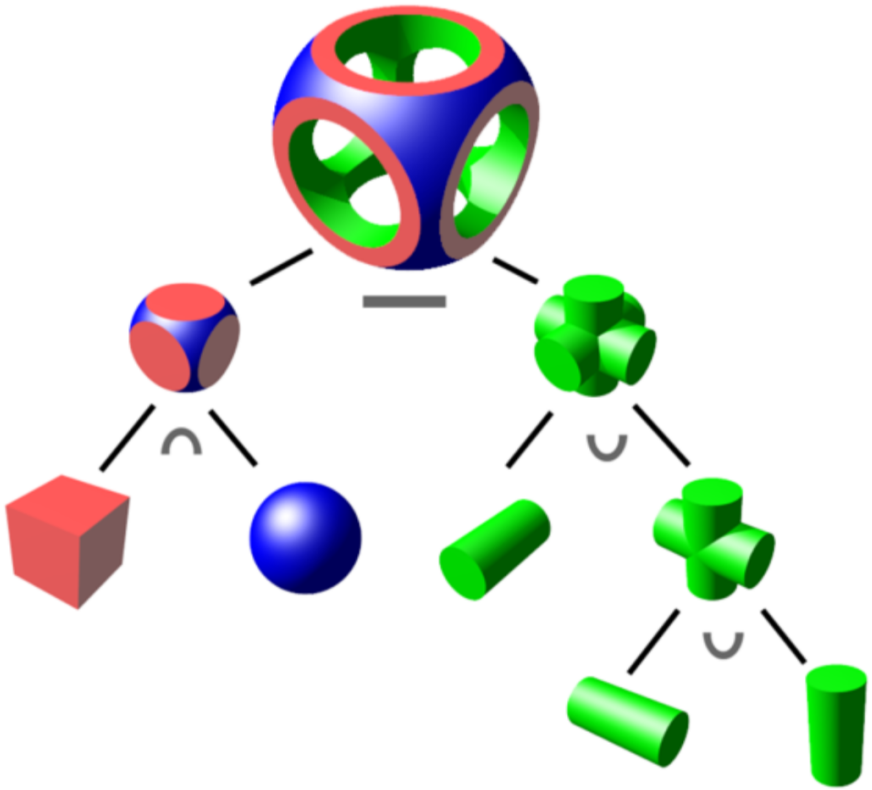
Space Arrangement and Algebras

Like LEGO block, atoms allow to build ANY solid expression with $\cup, \cap$ and $-$ operators



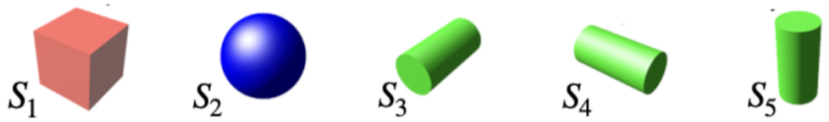
Boolean Algebras of SOLIDS

Alternate approach to ConStructive Solid Geometry (CSG)



CSG-tree

$\mathbb{E}^3$ arrangement induced by $\{S_i\}$



$$C_3 \underset{\partial_3}{\overset{\delta^2}{\leftrightarrows}} C_2 \underset{\partial_2}{\overset{\delta^1}{\leftrightarrows}} C_1 \underset{\partial_1}{\overset{\delta^0}{\leftrightarrows}} C_0$$

$$\chi : \{S_1, \ldots, S_5\} \to C_3$$

$$X := [\chi(S_1); \ldots; \chi(S_4); \chi(S_5)]$$

$$X_j = X[:, j]$$

$$((X_1) \cap (X_2)) - ((X_3) \cup ((X_4) \cup (X_5)))$$

binary formula evaluation

Binary representation of Boolean terms

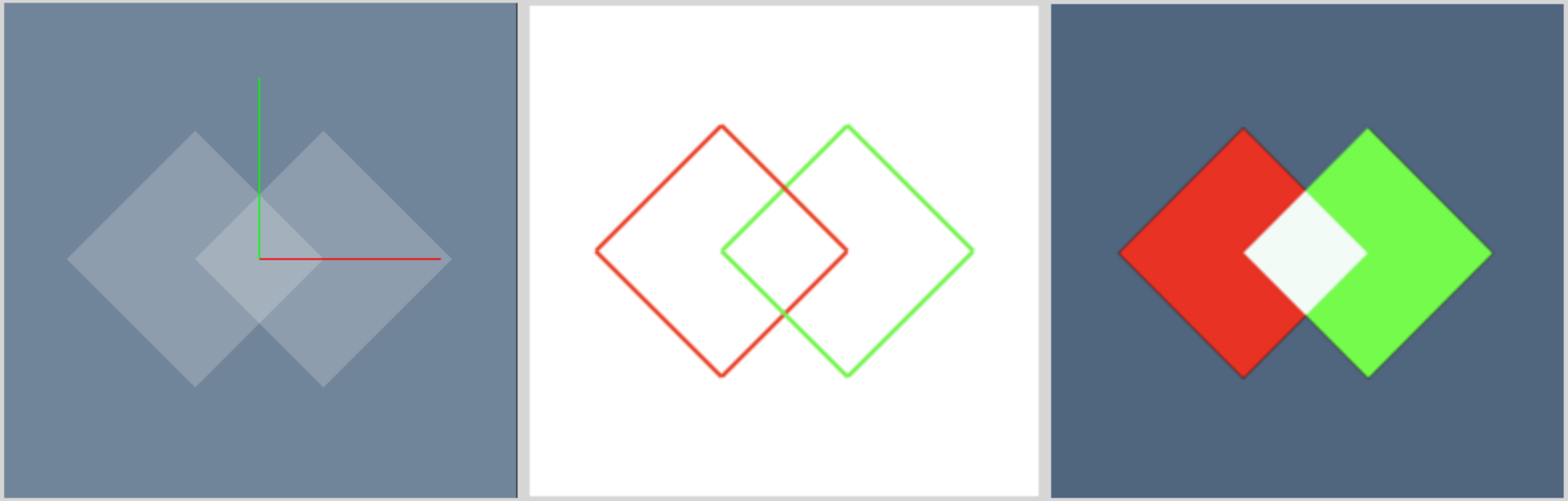


Figure 1: *Space arrangement* generated in $\mathbb{E}^2$ by: (a) two overlapping 2-cells A and B; (b) the 4+4 line segments (1-cells) in their skeletons, generating the *four* 2-cells in (c): blue (c_1), red (c_2), white (c_3), and green (c_4). The first (c_1) is the *outer* or *exterior* cell Ω, which is the complement of the sum of the others.

Figure 16: Some examples

Binary representation of Boolean terms

Table 1
Truth table associating 2-cells c_i (i=1,...,4) in U_2 (rows) and solid variables A, B and $\Omega = \overline{A \cup B}$ (columns). See Figure 1.

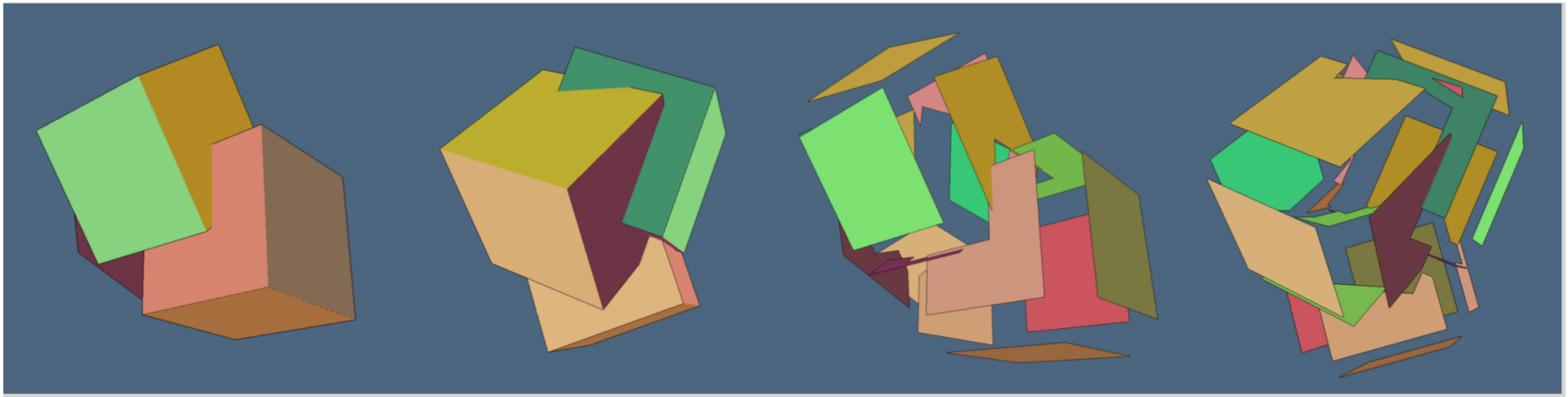
U_2	Ω	A	B
c_1	1	0	0
c_2	0	1	0
c_3	0	1	1
c_4	0	0	1

Table 2
Truth table providing the complete enumeration of elements S of the finite Boolean algebra $\mathcal{A}$ generated by two solid objects A, B and by four atoms c_i in the basis U_2 of chain space C_2 associated to the arrangement $\mathcal{A}(S)$, with $S = \{\partial(A), \partial(B)\}$.

U_2	$X = \mathbb{E}^2$	A	B	$\overline{A \cup B}$	$A \cup B$	$A \cap B$	$A \setminus B$	$B \setminus A$	$A \oplus B$	$\overline{A \setminus B}$	$\overline{B}$	$\overline{B \setminus A}$	$\overline{A}$	$\overline{A \oplus B}$	$\overline{A \cap B}$	$\varnothing$
c_1	1	0	0	1	0	0	0	0	0	1	1	1	1	1	1	0
c_2	1	1	0	0	1	0	1	0	1	0	1	1	0	0	1	0
c_3	1	1	1	0	1	1	0	0	0	1	0	1	0	1	0	0
c_4	1	0	1	0	1	0	0	1	1	1	0	0	1	0	1	0

Figure 17: Some examples

Generation of space arrangement of CSG terms



Boolean union $A \cup B \cup C$ of three cubes, with 2-cells in different colors: (a) view from the front; (b) view from the back; (c) front with exploded 2-cells; (d) back with exploded 2-cells.

Space arrangement generated by assembly

assembly object done by three instances of unit cube, suitably rotated and translated.

The semantics of Lar.Struct() is that of PHIGS structures

```
julia> Lar = LinearAlgebraicRepresentation;
    V,(VV,EV,FV,CV) = Lar.cuboidGrid([1,1,1],true);
    cube = V,FV,EV;

julia> assembly = Lar.Struct([ cube,
    Lar.t(.3,.4,.25), Lar.r(pi/5,0,0), Lar.r(0,0,pi/12), cube,
    Lar.t(-.2,.4,-.2), Lar.r(0,pi/5,0), Lar.r(0,pi/12,0), cube ]);

julia> W, EV, FE, CF, boolmatrix = Lar.bool3d(assembly);
```

The application of Lar.bool3d() to assembly returns the (geometry,topology) of 3D space partition generated by assembly.

1. geometry $\equiv$ W, the matrix of vertices (0-cells),
2. topology $\equiv$ CF, FE, EV, i.e., $[\delta_3], [\delta_2], [\delta_1]$, the sparse matrices of chain complex of $\mathbb{E}^3$ arrangement $\mathcal{A}$(assembly)

From space arrangement to CSG terms

The array of type `Bool` returned in `boolmatrix` contains by column the results of efficient point-solid containment tests (SMC) for atomic 3-cells (rows), w.r.t.~the terms of $\mathbb{E}^3$ partition: the outer 3-cell Ω and each `cube` (columns)

```
julia> Matrix(boolmatrix)
8x4 Array{Bool,2}:
 true  false  false  false
false  false  false   true
false   true   true  false
false   true   true   true
false   true  false  false
false  false   true  false
false   true  false   true
false  false   true   true
```

```
julia> A,B,C = boolmatrix[:,2],boolmatrix[:,3],boolmatrix[:,4]
\end{verbatim}
```

Variables Ω, A, B, C extracted from `boolmatrix` columns, describe how each term is partitioned by 3-cells in U_3.

The whole space is $X = \Omega \cup A \cup B \cup C$:

Prefix and Variadic Union of solid terms

- get the $\mathbb{E}^3$ space partition generated by `assembly` through the function `Lar.bool3d`;
- `**combine the arrays \texttt{A}, \texttt{B}, and \texttt{C}**`, building the value of `union`, that stores the logical representation of the solid 3-chain.

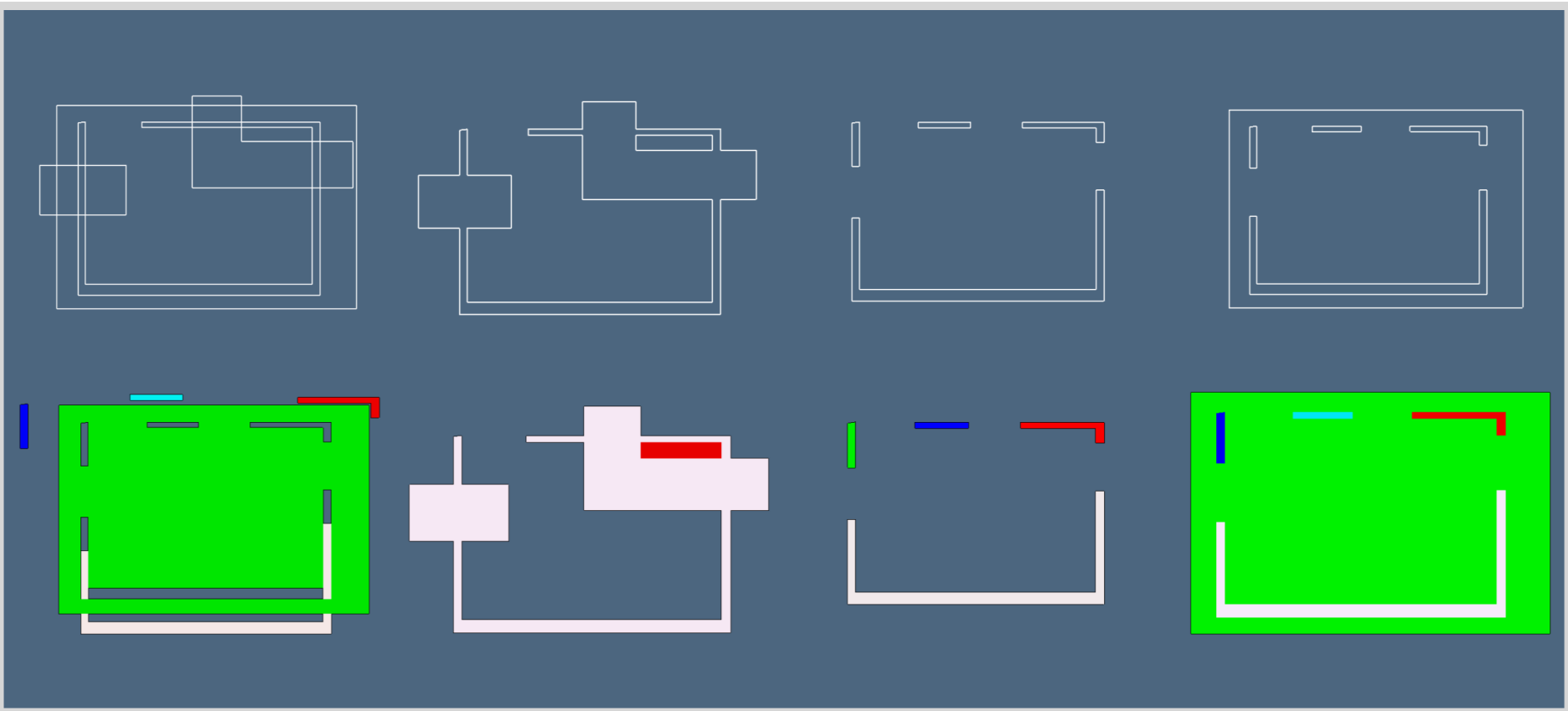
```
julia> W, (EV, FE, CF), boolmatrix = Lar.bool3d(assembly);
julia> A,B,C = boolmatrix[:,2],boolmatrix[:,3],boolmatrix[:,4]
julia> union = .|(A, B, C);
julia> @show union;
union = Bool[false, true, true, true, true, true, true, true]
```

- the boundary 2-cycle `faces` is generated by multiplication of the (sparse) matrix $[\partial_3]$ (`CF'`), times the binary `union`.

```
julia> faces = CF' * Int8.(union);
julia> @show faces;
faces = [1, 0, -1, 0, 0, 0, -1, 0, 1, 0, 1, 0, 0, 0, 1, -1, 0, -1, 0, 0, 1, -1, 0,
0, 0, 0, 1, 1, 0, -1, 0, 0, 1, 0, -1, 0, 0, 0, -1, 1, 0, 1, 0, 0, -1]
```

$$\texttt{faces} \mapsto f_{A\cup B\cup C} = f_1 - f_3 - f_7 + f_9 + f_{11} + f_{15} - f_{16} - f_{18} + f_{21} - f_{22} - f_{24} + f_{25} + f_{29} + f_{30} - f_{32} + f_{35} - f_{37} - f_{41} + f_{42} + f_{44} - f_{47}$$

Example 2D



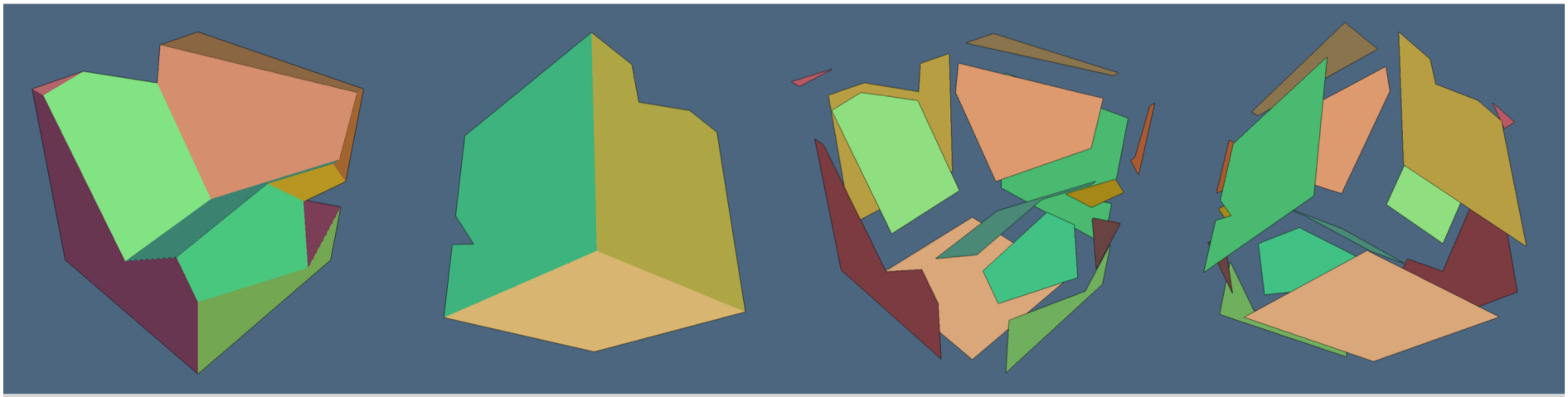
(a,b) (c,d) (e,f) (h,g) .

Figure 5: Some examples of variadic Booleans, obtained by applying the Boolean operators "`.|`", "`.-`", "`.!`", and "`.&`" to the `assembly` variable, with 1-cells (B-reps) imported from `SVG` files. (a,b) start-to-end: from polygon input via `SVG` to (exploded) difference of outer box and interior walls; (c,d) boundary of union and union of some shapes; (e,f) difference 2-complex and its boundary; (g,h) boundary of outer box minus the previous 2-complex, and the corresponding 2-complex.

Figure 19: Some examples

Example 3D: ternary difference of cubes

Boolean difference $(A \setminus B) \setminus C$ of three cubes



(a) (b) (c) (d)

Boolean difference $(A \setminus B) \setminus C$ of three cubes, with 2-cells in different colors: (a) view from the front; (b) view from the back; (c) front with exploded 2-cells; (d) back with exploded 2-cells.

Note that 2-cells of the resulting boundary may be non-convex.

Summary

1. Basic tools of linear algebra and algebraic topology are used, namely, (sparse) matrices of linear operators and matrix multiplication or filtering. Interval-trees and *kd*-trees introduced only for acceleration.
2. Accuracy of topological computations is guaranteed, since operator matrices satisfy by construction the (graded) constraints $\partial^2 = 0$ and $\delta^2 = 0$.
3. The evaluation of CSG expressions of arbitrary complexity is done in a novel way. All input B-reps are accumulated in a single collection of 2-cells, each of which is operated independently, so generating a collection of local topologies, that are merged by boundary congruence.
4. Once the global space partition is generated, and 3-cells are classified w.r.t. all component solids, via a single point-set containment test, all CSG expressions—of any complexity—can be evaluated simply by bitwise vectorized logical operations.
5. This approach can be extended to general dimensions and/or implemented on highly parallel computational engines, also using standard computing kernels.

Julia as Language for Numerical Computations

About Julia


Alan Edelman of MIT Recognized with Prestigious 2019 IEEE Computer Society Sidney Fernbach Award
julia
Download
Documentation
Blog
Community
Learning
Research
JSoC
Donate
The Julia Programming Language
Download v1.2
Documentation
Star
24,207

About Julia

Julia in a Nutshell

Julia is fast!

Julia was designed from the beginning for high performance. Julia programs compile to efficient native code for multiple platforms via LLVM.

Dynamic

Julia is dynamically-typed, feels like a scripting language, and has good support for interactive use.

Optionally typed

Julia has a rich language of descriptive datatypes, and type declarations can be used to clarify and solidify programs.

General

Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns. The standard library provides asynchronous I/O, process control, logging, profiling, a package manager, and more.

Easy to use

Julia has high level syntax, making it an accessible language for programmers from any background or experience level.

Open source

Julia is free for everyone to use, and all source code is publicly viewable on GitHub.

About Julia

Packages

Julia has been downloaded over 10 million times and the Julia community has registered over 2,000 Julia packages for community use. These include various mathematical libraries, data manipulation tools, and packages for general purpose computing. In addition to these, you can easily use libraries from Python, R, C/Fortran, C++, and Java. If you do not find what you are looking for, ask on Discourse, or even better, contribute!

Package Docs | Julia Observer

About Julia

Editors and IDEs

Juno

Atom Plugin

Visual Studio Code

VS Code Extension

Jupyter

Jupyter kernel

JetBrains

IntelliJ IDEA Plugin

Vim

Vim plugin

Emacs

Emacs plugin

SublimeText

Sublime Text

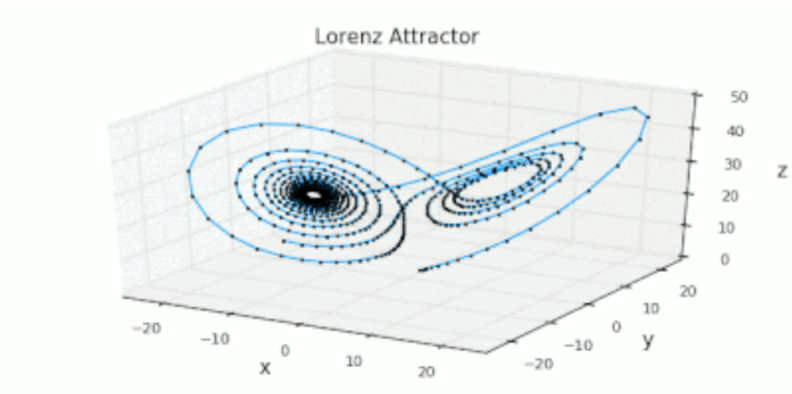
Revise

Revise.jl

About Julia

Ecosystem

Visualization | General Purpose | Data Science | Machine Learning | Scientific Domains | Parallel Computing



Rich Ecosystem for Scientific Computing

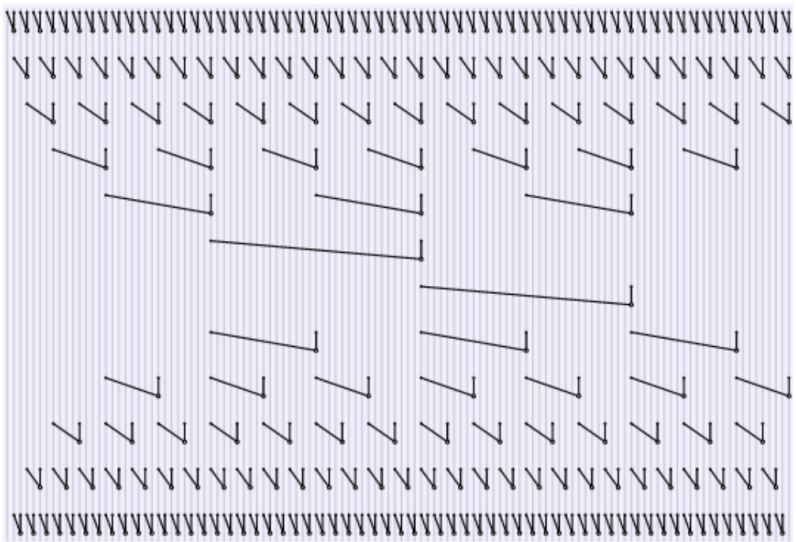
Julia is designed from the ground up to be very good at numerical and scientific computing. This can be seen in the abundance of scientific tooling written in Julia, such as the state-of-the-art differential equations ecosystem (DifferentialEquations.jl), optimization tools (JuMP.jl and Optim.jl), iterative linear solvers (IterativeSolvers.jl), a robust framework for Fourier transforms (AbstractFFTs.jl), a general purpose quantum simulation framework (Yao.jl), and many more, that can drive all your simulations.

Julia also offers a number of domain-specific ecosystems, such as in biology (BioJulia), operations research (JuliaOpt), image processing (JuliaImages), quantum physics (QuantumBFS, QuantumOptics), nonlinear dynamics (JuliaDynamics), quantitative economics (QuantEcon), astronomy (JuliaAstro) and ecology (EcoJulia). With a set of highly enthusiastic developers and maintainers from various parts of the scientific community, this ecosystem will only continue to get bigger and bigger.

About Julia

Ecosystem

Visualization | General Purpose | Data Science | Machine Learning | Scientific Domains | Parallel Computing



Parallel and Heterogeneous Computing

Julia is designed for parallelism, and provides built-in primitives for parallel computing at every level: **instruction level parallelism**, **multi-threading** and **distributed computing**. The Celeste.jl project achieved 1.5 PetaFLOP/s on the Cori supercomputer at NERSC using 650,000 cores.

The Julia compiler can also generate native code for various hardware accelerators, such as GPUs and Xeon Phis. Packages such as DistributedArrays.jl and Dagger.jl provide higher levels of abstraction for parallelism.