juliacon

My JuliaCon proceeding

Alberto Paoluzzi[1]

[1]Roma Tre University, Rome, Italy

ABSTRACT

Solid Modeling algorithms and applications usually demand very linked and pretty complex data structures, often denoted as "non-manifold representations" to remark the breadth of their domain. Conversely, we are implementing in Julia [1] topological methods making only use of sparse binary or integer arrays, and their standard algebraic operations, like product, transposition and filtering, and standard topological operators of boundary and coboundary between linear spaces of chains generated by the cells of a space partition. We show computational methods to generate the 2D/3D space partition induced by a collection of 1D/2D/3D geometric objects. Methods and language are those of basic geometric and algebraic topology. Only sparse arrays are used to compute spaces and maps (the chain complex) from dimension zero to three. In particular, we show how to build a space arrangement with general non-convex and non-contractible piecewise-linear cells, and how to compute Boolean operations between solid models generated by this approach, reducing them to standard tables of Boolean values.

Keywords

Julia, Solid modeling, Geometric programming, Computational topology, Chain Complex, Cell Complex

1. Introduction

In this paper we discuss the current status of a novel approach to geometric computing, and its implementation in Julia. Rather than using standard methods of geometric computing and solid modeling, normally built on top of complicated data structures, our work is mostly developed making advantage of sparse arrays and their algebraic operations. In particular, this approach is established over basic concepts of algebraic topology, like cellular complexes, chain and cochain spaces and operators, and chain complexes. Piecewise-linear algebraic topology allows to treat rather general complexes, with cells homeomorphic to polyhedra, *i.e.*, to triangulable spaces, and hence possibly non convex and multiply connected. We use simplicial complexes, *i.e.*, triangulations, just for graphics, where to stream triangles to the GPU is almost mandatory.
At the knowledge of the author, there are no previous approaches to geometric and solid modeling that build over a similar framework, *e.g*, to compute the *chain complex* codifying the arrangement of d-space generated by a collection of (d-1)-geometric objects. A wide comparison with previous work in this field may be found in [6, 3]. After a short synthesis of basic algebraic topological concepts, this paper discusses the arrangement of space induced by solid objects, as the basis for resolution of solid-valued expressions. A survey of functions from our packages `LinearAlgebraicRepresentation`, `Triangle`, and `ViewerGL` follows. Simple examples generating pictures conclude the paper.

2. Linear Algebraic Representation

We summarize in the following the concepts and definitions on which our `Lar` (Linear Algebraic Representation) [3] is based on.

2.1 Cell, Complex, Chain

A p-cell σ ($0 \leq p \leq d$) is a piecewise-linear, connected, but possibly non-contractible p-manifold. An r-face τ of a p-cell σ ($0 \leq r \leq p$) is a r-cell contained in the frontier of σ. A d-complex Λ is a partition of a topological d-space X in a discrete set of p-cells ($0 \leq p \leq d$) such that: (a) $\sigma \in \Lambda$ implies $\tau \in \Lambda$ for all faces τ of σ; (b) the closure intersection of every pair of cells of Λ is either in Λ or the empty set. A p-chain can be defined as a subset of p-cells. To the p-chains can be given the structure of a (graded) linear space C_p by defining (a) sums of chains with the same dimension, and (b) products times scalars in a field, with the usual properties. A basis U_p for C_p is the set of elementary chains u_p, given by single cells in Λ_p. Every chain $c \in C_p$ may be uniquely generated by a linear combination of the basis U_p. Once fixed the basis U_p, the coordinate representation of each $\sigma \in C_p$ is unique. This one is an ordered sequence of coefficients, either from $\{0, 1\}$ (said unsigned rep.) or from $\{-1, 0, +1\}$ (called signed representation).

2.2 Boundary and coboundary operators

Bondary operators are linear maps $\partial_p : C_p \to C_{p-1}$, ($1 \leq p \leq d$). Coboundary operators are linear maps $\delta_p : C_p \to C_{p+1}$, ($0 \leq p \leq d-1$). Once fixed the bases U_{p-1}, U_p, U_{p+1}, *i.e.*, once fixed an ordering among the cells in each Λ_p, their matrix representations $[\partial_p]$ and $[\delta_p]$ are uniquely determined. It is worthwhile to remark the meaning of such matrices: once fixed the bases, the matrix $[M]$ of a linear operator $M : A \to B$ contains by columns the coordinate representation (*i.e.*, the coefficients of the linear combination) of the A basis expressed as linear combination of B basis elements. This property is the conceptual key of our methods.
A *p-cycle* is a chain *without* boundary. It is a chain that the boundary operator ∂_p sends to the kernel (zero-set) of C_p. In particular, the j-th column of $[\partial_p]$ contains a basis element $u_j \in U_p$ represented as a $(p-1)$-cycle, *i.e.*, contains the coefficients of it as linear combination of elements of U_{p-1}. Such linear combinations are $(p-1)$-cycles, since $\partial_{p-1} \circ \partial_p = 0$. In other words, the matrix $[\partial_2]$ represents the 2-cells in Λ_2 as 1-cycles (closed polygons) in C_1. Analogously, $[\partial_3]$ contains by columns the 3-cells of Λ_3 as cycles (shells) made by 2-cells in Λ_2 (*e.g.*, see Figure 6d).
We have already seen the topological equations $\partial_{p-1} \circ \partial_p = 0$ ($d \leq p \leq 2$) that guarantee the boundary of a boudary being empty. Think about the 2-disk $d \in C_2$ in the plane, with support $|d| \subset \mathbb{E}^2$. The circumference $c \in C_1$, with $|c| = |\partial_2 d| \subset E^2$ is a closed curve. In fact we have $\partial(\partial c) = 0$. In our case, where cells live in Euclidean spaces, chain spaces C_p can be identified with their dual

spaces of p-cochains C^p, that are spaces of maps $\mu^p : C_p \to \mathbb{F}$ from p-chains to the field $\mathbb{F}$ of coefficients. In this case we have $C^p = C_p^\top$, and hence $[C^p] = [C_p]^t$. The matrix-vector multiplication $[\partial_p][c]$ provides the (coordinate representation of) the (p-1)-cycle on the frontier of c, whereas $[\delta^p][c]$ gives the (coordinate repre. of) the (p+1)-chain made by all (p+1)-cells incident on c.

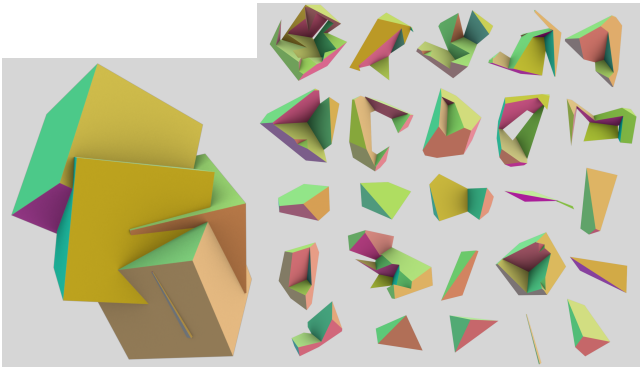

Fig. 1. The 3-cells of the arrangement of $\mathbb{E}^3$ generated by a collection of five random cubes. The 3-cells are here not in scale, and suitably rotated to better show their complex structure. Their assembly provides the union of the five cubes. Each *3-cell is given* by *a column* of the *sparse matrix* of *chain map* $\partial_3 : C_3 \to C_2$, with values in $\{-1, 0, 1\}$.

2.3 Incidence/adjacency operators

Specialized data structures are commonly used to efficiently answer the queries about the incidence and adjacecy relations between boundary elements (vertices, edges and faces), in order to implement the various geometric and topological algorithms of solid modeling. Given the three sets `V`, `E`, and `F` (vertices, edges and faces), 3×3 binary relations `VV`, `VE`, `VF`, `EV`, `EE`, `EF`, `FV`, `FE`, `FF` are used. Several different combinations of such relations have been defined as specialized data structures, with the aim of minimizing the time and space complexity of algorithms on solid models.
It is easy to see that each relation can be computed by a proper combination of one or two (co)boundary matrices, possibly transposed. Their ordered list follow: `VV` $\equiv [\partial_1][\delta_0]$, `VE` $\equiv [\partial_1]$, `VF` $\equiv [M_2]^t$, `EV` $\equiv [\partial_1]^t$, `EE` $\equiv [\delta_0][\partial_1]$, `EF` $\equiv [\partial_2]$, `FV` $\equiv [M_2]$, `FE` $\equiv [\delta_1]$, `FF` $\equiv [\delta_1][\partial_2]$, where M_p is the sparse binary *characteristic matrix* of p-cells, that holds by rows the images of characteristic functions of p-cells as subsets of 0-cells (vertices).
A typical topological query may be asked as: "what edges (elementary 1-chains) are adjacent to edge e?" The `Lar` answer is computed as the product $[\delta_0][\partial_1][e]$, where $[e]$ is the coordinate representation of $e \in C_1$ (a sparse column vector with just one non-zero element) and, of course, the matrix $[\delta_0][\partial_1] =:$ `[EE]` was computed in advance, and once for all.
The cardinality of such incidence/adjacency algebraic tools between p-cells and q-cells ($0 \le p, q \le 2$) is equivalent to that of relations itself [7], in force of their sparsity. For example, $\#$`EV` $= O(Space([\delta_1]) = 2\#$`E`), where `V` are the vertices of a solid B-rep and `EV` are binary incidences of edges with vertices. Every set of local queries about the 3×3 incidences/adjacencies between cells can be answered by multiplication, via software kernels for *sparse matrix* product and transposition, just by collecting the coordinate vectors of unit chains, "subject" of elementary queries, as *columns* of a sparse Q matrix, and by left-multiplying Q times one/two operator matrices $[\partial_1]$ and/or $[\partial_2]$, suitably ordered and/or transposed [3], to get the algebraic equivalent of multiple database queries at once.

3. Arrangement Algorithms

A *chain complex* is a short exact sequence of graded linear spaces C_p of (co)chains, with linear boundary/coboundary maps ∂_p and $\delta_p = \partial_{p+1}^\top$:

$$C_\bullet = (C_p, \partial_p) := C_3 \underset{\partial_3}{\overset{\delta_2}{\leftrightarrows}} C_2 \underset{\partial_2}{\overset{\delta_1}{\leftrightarrows}} C_1 \underset{\partial_1}{\overset{\delta_0}{\leftrightarrows}} C_0.$$

The subject of Reference [6] is the computation of the chain complex $C_\bullet = (C_p, \partial_p)$, starting from some representation[1] of an input set $\mathcal{S}$ of (d-1)-complexes. In particular, we compute the matrices of the linear maps ∂_p (and their duals δ_{p-1}) between chain spaces C_p. All definitions and more examples are given in Reference [6]. We describe here the main steps of space decomposition in reverse order, starting from the last operation, because of possible iteration on dimensions, starting from 3D.

3.1 Topological gift wrapping

The main goal is to compute, starting from (d-1)-dimensional geometric object, the d-cells of the space partition (arrangement) generated by them. Examples of input include, but are not limited to: line segments, quads, triangles, polygons, meshes, pixels, voxels, volume images, B-reps, *etc.* In mathematical terms, a geometric object is a topological space embedded in some $\mathbb{E}^d$.
The topological method introduced in [6] is reminiscent of the "gift-wrapping" algorithm [2, 5] for computing convex hulls of 2D and 3D discrete sets of points, but it works with higer-dimensional cells instead with points, and is mostly based on applications of boundary and coboundary operators. Our `TGW` (Topological Gift Wrapping) algorithm [6] takes a sparse matrix $[\partial_{d-1}]$ as input and produces in output the *unknown* sparse matrix $[\partial_d^+]$, augmented with the outer cell. A geometric embedding function $\mu : X_0 \to \mathbb{E}^d$ is used to compute the angular ordering, around some (d-2)-cells, of (d-1)-basis elements in the boundary's coboundary, while wrapping up a (d-1)-cycle, as illustrated in Figure 2. The built cycles are set up as columns of $[\partial_d]$, in the construction of a C_d basis.

3.2 Merge of 2-skeletons

The input to the computational pipeline is a collection $\mathcal{S}$ of geometric objects that do not necessarily constitute a cellular complex, since they can intersect out of cell boundaries. The first step is hence the computation of mutual interctions of their 2-skeletons, *i.e.*, of their sets of 2-cells (for a 3D problem), or their 1-skeletons (for a 2D problem). This phase is executed by intersecting independently each 2-cell $\sigma \in \mathcal{S}_2$ with all 2-cells which might intersect it. Their subset is denoted $\mathcal{I}(\sigma)$ and is computed using efficient spatial indices on the containment boxes of 2-cells. In particular, each $\mathcal{I}(\sigma)$ is given by the intersection of outputs of d independent queries over one-dimensional interval-trees built upon the input.
It may be interesting to remark that the `merge` algorithm is embarrassingly parallel, and is implemented in Julia making use of two `Channel`s to ditribute the jobs to workers and to return the output to the master node. This more efficient strategy is quite unusual in

[1] Our prototype implementation in `https://github.com/cvdlab/LinearAlgebraicRepresentation.jl`, makes use of the `Lar` representation [3], on which this approach strongly relies.

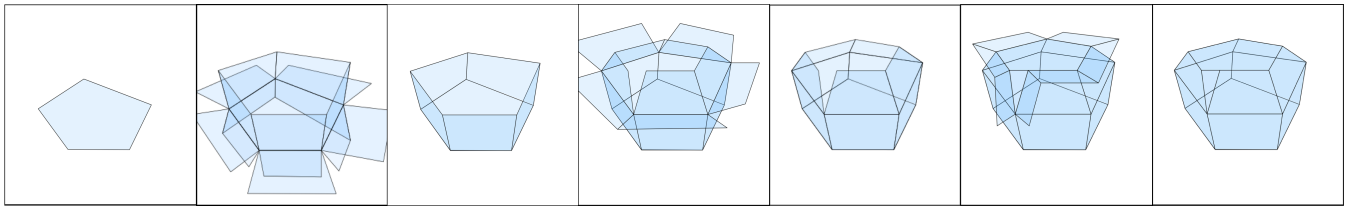

Fig. 2. Extraction of a minimal 2-cycle from $\mathcal{A}(X_2)$: (a) initial (0-th) value for $c \in C_2$; (b) cyclic subgroups on $\delta\partial c$; (c) 1-st value of c; (d) cyclic subgroups on $\delta\partial c$; (e) 2-nd value of c; (f) cyclic subgroups on $\delta\partial c$; (g) 3-rd value of c, such that $\partial c = 0$, hence stop.

Solid Modeling, where even recent variadic approaches [8] iteratively intersect a new operand against the cell complex generated by the previous operations.

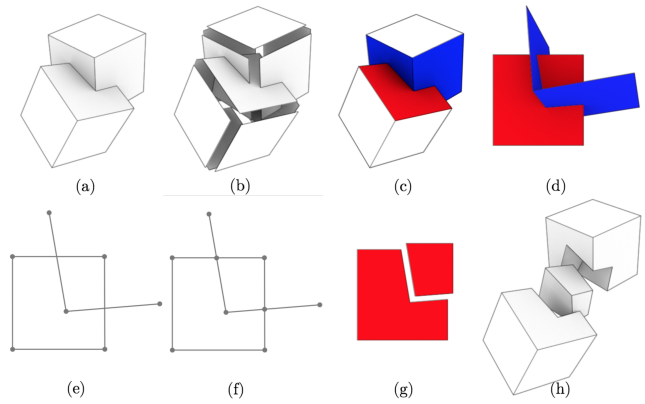

Fig. 3. Cartoon display of the computational pipeline: (a) two solids in $\mathcal{S}$; (b) the exploded input collection $\mathcal{S}_2$ in $\mathbb{E}^3$; (c) 2-cell σ (red) and the set $\Sigma(\sigma)$ (blue) of possible intersection; (d) $\sigma \cup \Sigma$ affinely mapped on $z = 0$; (e) reduction to a set of 1D segments in $\mathbb{E}^2$ via intersection with $z = 0$; (f) pairwise intersections; (g) exploded U_2 basis of C_2 generated as columns of $\partial_2 : C_2 \to C_1$; (h) exploded U_3 basis of C_3 generated as columns of operator's $\partial_3 : C_3 \to C_2$ sparse matrix, via the `TGW` algorithm in 3D.

The independent processing of 2-cells is made possible by considering and enforcing the congruence (i.e. the boundary compatibility) between adjacent fragmented cells. Hence, each input 2-cell σ is fragmented in 2D independently from each other. For this purpose, a suitable affine transformation maps each set $\{\sigma\} \cup \mathcal{I}(\sigma)$ and puts σ into the $z = 0$ subspace.

3.3 Reduction to line intersection in 2D

At this point the single 3D arrangement problem is reduced to a collection of several independent 2D arrangement problems, one-to-one with the 2-cells in the 3D input. Every 2-cell in $\mathcal{I}(\sigma)$ is independently intersected (in parallel) with $z = 0$, so providing a number n of line segment sets $\mathcal{S}_1$ for independent computation of the 2D arrangement generated by it. The computational pipeline is depicted as cartoon in Figure 3. In particular: each set of line segments is mutually intersected (using again two 1D interval-trees of containment boxes of segments for acceleration); the dangling edges and subgraphs are removed from the generated linear graph; and the `TGW` is used in 2D to build the matrix $[\partial_2]$ of the arrangement, and the corresponding basis of elementary 2-chains (2-cells) of the σ cell decomposition.

3.4 Congruence of 2D arrangements

Each 2-complex X_σ providing an independent arrangement of 2-space, is transformed back in 3D using the inverse affine transformation of σ. Here, congruent cells (e.g., edges member of the boundary of two or more fragmented faces) must be identified as belonging to the same equivalence class, and substituted by a single instance, so implementing a set quotient operation on fragmented cells. Numerical identification of vertices via dictionaries having arrays of floats as keys (permitted by Julia) are used for this purpose. Once identified congruent vertices, the identification of congruent edges and faces is done using their canonical `Lar` representation, *i.e.,* the ordered arrays of integer indices of vertices.

4. Variadic Boolean Operations

When the unknown sparse boundary matrix $[\partial_3]$ has been generated by `TGW` in 3D, the computation of any Boolean operation or solid-valued expression, including solid arguments and Boolean operators, is actually straightforward[2]. Let us note that the arrangement itself corresponds to the output of a union operation. To compute any general solid-valued expression, it suffices to suitably store the dependencies, of each possibly fragmented 2-cell of resulting 3-cells, from the input 2-cells that generated it, i.e. from its "father(s)". This storing allows to reconstruct the containment relations of arrangement's 3-cells with their parents, and hence to write truth tables having them on columns, and the original solids (or their boundaries) on the rows. It is worthwhile to note that most of the required information is already coded by the non-zero coefficients of $[\partial_3]$, and in particular by their indices and signs.

4.1 Columns and rows of 3-boundary matrix

We already know that columns of $[\partial_3]$ represent the single 3-cells of the arrangement, corresponding to cycles (closed chains) of 2-cells. First, let us consider a solid model B as a 3-chain in C_3, which is possibly made by more then one 2-cycles, corresponding to one or more columns in $[\partial_3]$. This set of columns $[\partial_3]_B$, possibly a singleton, codifies the non-empty set of shells of B, where each one may be either inner (boundary of empty holes) or outer (boundary of connected components). The number of $[\partial_3]$ rows is the number of elements of the (combinatorial) set union of fragments of original 2-cells in the input of arrangement algorithm. The rows, one-to-one

[2]Even if not yet implemented.

with fragmented 2-cells, can be subdivided in two classes, corresponding to the type of input they belong to, which can be either a B-rep or a cellular model [4]:

(1) if some shell of the "solid" chain $B \in C_3$ is represented by a 2-cycle $b \in \partial B \subseteq U_2$, then each elementary 2-chain — *i.e.*, basis element (2-cell) — $\sigma \in b$ belongs to *only one* elementary 3-chain u_3 of the B's linear combination generated by our algorithmic pipeline.[3]

(2) if a row corresponds to a fragment of 2-cell originally in the interior of B, then it will belong to *exactly two* cells $u_3 \in C_3$ that are solid fragments of B.

4.2 Mapping from output d-cells to input (d-1)-cells

According to the properties listed above, each column of $[\partial_3]$, *i.e.*, each solid 3-cell in the output, must be mapped to the input surfaces that generated it, going from fragments on the related rows to their ancestor 2-cells. This parental relationship is carefully recorded and maintained during the pipeline computation.

4.3 Boolean ops on truth tables

Two binary parental relationships between input and output 2-cells, and between output 2-cells and output 3-cells are used to compute the solid-valued results of Boolean expressions between solid models. We call Q matrix the first, whereas the second is simply the unsigned sparse matrix $[\partial_3]^o$. A third sparse binary matrix $P = [\delta_2]^i$ stores the membership of original 2-cells to solid models being operated. It is interesting to note that P is a diagonal block matrix, where the blocks are the coboundary matrices $[\delta_2]_k$ $(1 \leq k \leq n)$ of the n solid arguments of the expression to be evaluated. The analysis of results of compatible matrix product $[\delta_2]^i\, Q\, [\partial_3]^o$ finally produces the membership of the output 3-cells to the input arguments, and allows for on-off computation of solid-valued expression.

5. Julia packages

The methods and algorithms described above were implemented in the Julia packages `LinearAlgebraicRepresentation` and `Triangle`. In the following (and in our Julia software) it will be often called `Lar` for the sake of brevity. `ViewerGL` is a visual package whose development started recently. It provides OpenGL interactive visualization for `Lar` in native Julia code, on top of `GLFW`, the multi-platform library for OpenGL, OpenGL ES and Vulkan, that provides a simple API for creating windows and graphics context. The other dependency, `ModernGL`, provides OpenGL 3+ bindings for Julia. The main functionalities in our packages follow.

5.1 `LinearAlgebraicRepresentation.jl`

`Lar`, as its ancestor geometric language `PLaSM` and its father library `pyplasm`, aims to be multidimensional. Hence many functions generate geometric models of varying dimensions. Important examples are `cuboidGrid` and `simplexGrid`, whose unique parameter is the `shape` of the generated mesh, i.e. the number of 1-dimensional cells in each dimension, with `d = length(shape)`. The vertices of the mesh stay on the integer grid of suitable dimension and size.

[3] Some conceptual and notational ambiguity may arise, since a chain or cycle may be seen either as element ($\in$) of a chain space (linear combination of the basis) or as a subset ($\subset$) of the basis.

`Lar` model. A `Lar` model is a pair *geometry*, *topology*. The *geometry* is specified by the position vectors of *vertices* in a Euclidean point space $\mathbb{E}^d$ with d coordinates. The *topology* is specified by one or more bases of singleton k-chains (*i.e.*, k-cells) for $0 \leq k \leq d$. The vertex sharing between cells implicitly provides the attachment maps between cells of various dimensions. Vertex positions are represented, by columns, by a 2-array of d real coordinates. With abuse of language, we consider a finite cellular complex X as generated by a discrete partition of an Euclidean space. In computing a cellular complex as the space arrangement of a collection of geometric objects $\mathcal{S}$, *i.e.*, when $X := \mathcal{A}(\mathcal{S})$, we actually compute the whole *chain complex* $C_\bullet$ generated by X.

Multidimensional grids. `src/largrid` and `src/simplexn` allow for multidimensional *grids* of `cuboidal` and `simplicial` cell complexes, and *Cartesian product* of cellular complexes. Both kind of operators, depending on the dimension of their input, may generate either *full-dimensional* (i.e. solid) output complexes, or *lower-dimensional* complexes of dimension d embedded in Euclidean space of dimension n, with $d \leq n$. E.g., just think to a mesh of 3D cubes in three-dimensional space for the first case, and to the (non-manifold) framework of boundary polygons of such cubic meshes for the second case. In particular, both n-dimensional *solid grids* of (hyper)-cuboidal cells and their d-dimensional *skeletons* $(0 \leq d \leq n)$, embedded in $\mathbb{E}^n$, are generated by assembling the cells produced by a number n of either 0- or 1-dimensional cell complexes, that in such lowest dimensions coincide with *simplicial complexes*. Generation of *grids* works by *Cartesian product* of $0/1$-complexes; the *output complex* is generated by the product of *any number* of either 0- or 1-dimensional complexes. The product of d one-dimensional complexes generates *solid* d-cells, while the product of n zero-dimensioanl complexes and $n - d$ 1-complexes $(d < n)$ generates *non-solid* $(n - d)$-cells, properly embedded in n-space, i.e. with vertices having n coordinates.

Assemblies of cellular complexes. Hierarchical models of assemblies are generated by aggregation of cellular complexes, each one defined in a local coordinate system, and possibly relocated by affine transformations of coordinates. This operation may be repeated hierarchically, with subassemblies defined by aggregation of simpler parts, and so on, until to obtain a set of `Lar` models, which are not further decomposed. This *hierarchical model*, defined inductively as an assembly of component parts, is described by an *acyclic directed multigraph*, often called a *scene graph* or *hierarchical structure* in computer graphics and modeling. The main algorithm in `Lar` with hierarchical assemblies is the `traversal` function, which transforms every component from *local coordinates* to global coordinates, called *world coordinates*.

Hierarchical modeling. Two main advantages can be found in a hierarchical modeling approach. (a) Each component complex and each assembly, at every hierarchical level, are defined independently from each other, using a local coordinate frame, suitably chosen to make its definition easier. (b) Only one copy of each component is stored in memory, and may be instanced in different locations and orientations how many times it is needed.
A *container* of geometrical objects is defined by applying the function `Struct` to the array of contained objects. The value returned from the application is a value of `Struct` type. The coordinate system of this value is the one associated with the first object of the `Struct` parameters. Also, the resulting geometrical value is often associated with a variable name. An affine 3×3 transformation matrix, generated in homogeneous normalized coordinates by the function call `t(-0.5,-0.5)`, can be *applied* to a `Lar` object `obj`

both *explicitly* by using the function `apply(Matrix, obj)` or *implicitly* by creating a `Struct` hierarchical object.

From hierarchical models to flat models. The generation of container nodes may continue hierarchically by suitably applying `Struct`. Notice that each `Lar` object in a `Struct` container is transformed by each matrix before it *within the container*, going from right to left. The action of a transformation (tensor) extends to every object rightwise within its own container. Conversely, the action of a tensor does not extend outside its container, according to the semantics of `PHIGS` structures. The function `evalStruct`, when applied to a `Struct` value, generates an `Array` of `Lar` models, each one originally defined in a *local coordinate* system, transforming all of them in the same *world coordinate*, equal to the one of the *first* object in the `Struct` parameter sequence (see Section 6.2). Conversely, the `struct2lar` function generates a *single* `Lar` model (cellular complex), whose components are there assigned to variables `V` (coordinates of vertices), `FV` faces (2-cells), and `EV` edges (1-cells). Notice that the whole model is *embedded in 3D*, since the `V` array (coordinates by columns) has *three rows*.

Parametric objects. The `src/mapper.jl` file contains the implementation of several parametric primitives, including *curves*, *surfaces* and *solids* embedded in either 2D or 3D. A constructive approach is common to all methods. It consists in generating a simplicial or cuboidal decomposition of a simple geometrical domain in u, v or u, v, w parametric space. Then a change of coordinates, *e.g.*, from polar or cylindrical to Cartesian, is applied to vertices of the cellular complex which decomposes the domain.

Integration of monomials. A finite integration method is implemented in `src/integr.jl`, to compute monomial integrals over polyhedral solids and surfaces in 3D space. This integration can be used for the exact evaluation of domain integrals of trivariate polynomials. The evaluation of surface and volume integrals is achieved by transform into line integrals over the boundary of every 2-simplex of a boundary triangulation. The `Lar` integration formulae may also be used with models consisting of the collection of its boundary's 2-loops (polygons). Loops must be oriented counter-clockwise if external, clockwise if internal to another loop.

5.2 `ViewerGL.jl`

The work on this package started only recently, but already provides useful visualization tools, allowing for fast 3D user interaction with 2D and 3D geometric models.

Basic `ModernGL` infrastructure. Several Julia's `struct` objects are used at the basic implementation level of `ViewerGL`, and in particular for describing the current state of variables of types `Point`, `Matrix`, `Quaternion`, `Box`, `Frustum`, `Viewer`, `GLColorBuffer`, `GLMesh`, `GLPhongShader`, `GLShader`, `GLText`, `GLUtils`, `GLVertexArray`, and `GLVertexBuffer`. `Graphictext` is a cellular implementation of a native Julia's vector font, mainly used to help the visual testing and debugging of geometric codes, by visually numbering vertices, edges, faces, and solid cells.

High-level visualization primitives. At low-level, OpenGL uses only few basic primitives, whose vertices must suitably embedded into proper buffers for `points`, `normals`, and `colors`. A number of high-level, user-oriented, graphics primitives were implemented to allow direct and easy rendering of various types of cellular complexes, both 2D and 3D, each reurning an object of type `GLMesh` to direvtly feed the `Viewer`. Those primitives currently include: `GLHull`, `GLHull2d`, `GLHulls`, `GLPolygon`, `GLPolygons`, `GLLar2gl`, `GLLines`, `GLPoints`, `GLPolyhedron`, `GLPolyhedrons`, `GLGrid`, and `GLExplode`, that all accept as input a `Lar` model.

6. Examples

Both `LinearAlgebraicRepresentation` and `ViewerGL` packages contain several simple `examples/` scripts, to help the user to get acquainted. The actual APIs are not yet completely defined.

6.1 Arrangement of circles and rectangles

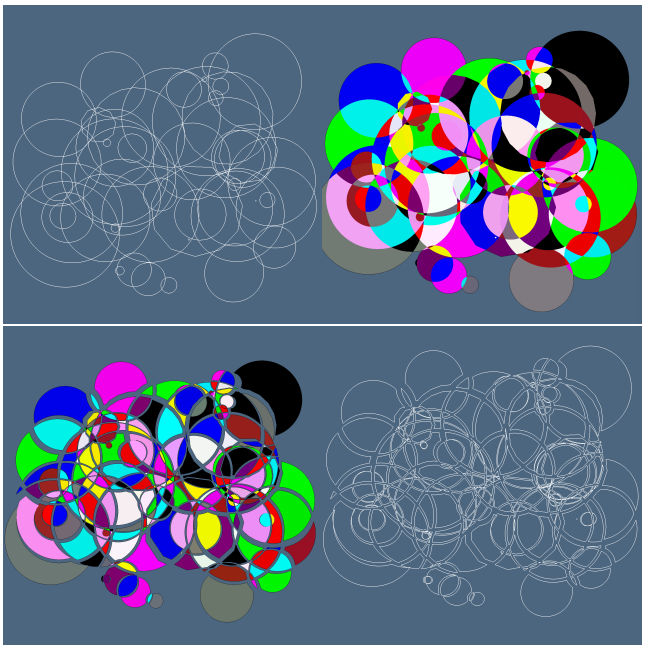

Fig. 4. Four views of the 2D arrangement generated by circles and regular polygons: (a) the input random polylines; (b) 2-space partition generated by the input; (c) exploded view of the output arrangement; (d) exploded view of boundary 1-cycles of unit 2-chains of the 2-space arrangement.

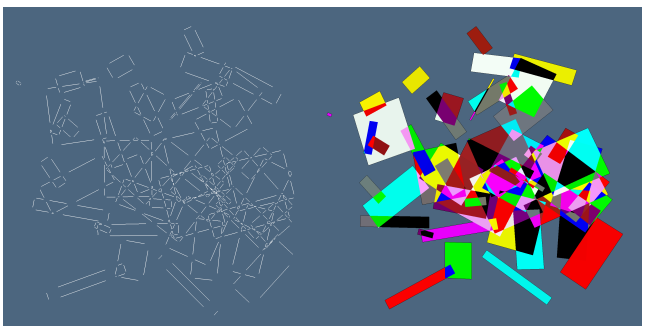

Fig. 5. Two views of the 2D arrangement generated by random rectangles: (a) exploded view of fragmented 1-chain generated by the input boundaries of rectangles; (b) the 2-space arrangement generated by the input.

6.2 Boolean 3D workflow

[4] First define cube as a single element of a grid of cubes:

```
using LinearAlgebraicRepresentation, ViewerGL, SparseArrays
Lar = LinearAlgebraicRepresentation; GL = ViewerGL
V,(VV,EV,FV,CV) = Lar.cuboidGrid([1,1,1],true)
cube = V,FV,EV
```

Then position three instances of cubes in $\mathbb{E}^3$ within a hierarchical scene graph, using the `Struct` function:

```
threecubes = 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 ]);
```

Convert `threecubes` to single (self-intersecting) model `V,FV,EV`, and visualize the collection `FV` of faces (2-cells). It is not a cellular complex, since cells intersect out of boundaries. See Figure 6a:

```
V,FV,EV = Lar.struct2lar(threecubes)
GL.VIEW([ GL.GLGrid(V,FV), GL.GLFrame ])
```

Next, prepare the input data types for computing the 3-space arrangement. We have to yet better define the IDE. `Lar.Cells` and `Lar.ChainOp` are array or sparse matrix of p-cells, respectively.

```
cop_EV = Lar.coboundary_0(EV::Lar.Cells);
cop_EW = convert(Lar.ChainOp, cop_EV);
cop_FE = Lar.coboundary_1(V,FV::Lar.Cells,EV::Lar.Cells);
W = convert(Lar.Points, V');
```

$[\delta_2],\ [\delta_1],\ [\delta_0] =$ `ChainOp[copCF, copFE, copEV]`, *Chain Complex* embedded in $\mathbb{E}^3$ by $3 \times n$ vertex matrix `V`, is finally generated:

```
V, copEV, copFE, copCF =
Lar.Arrangement.spatial_arrangement(W, cop_EW, cop_FE)
```

The 8 rows of the $\delta_2 : C_2 \to C_3$ matrix (below) or the 8 columns of $[\partial_3] = [\delta_2]^t, \partial_3 : C_3 \to C_2$, generate the eight 3-cells of Figure 6d.

```
@show Matrix(copCF);
```

```
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 -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 -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]
```

Finally, the `Lar` model is triangulated, via `Triangle.jl` package, giving CDT (constrained Delaunay Triangulation), and converted to triangle arrays per 3-cell, per 2-cell, and per polygon boundary:

```
W = convert(Lar.Points, V')
V,CVs,FVs,EVs = Lar.pols2tria(W, copEV, copFE, copCF)

GL.VIEW(GL.GLExplode(V,FVs,1.5,1.5,1.5,99,1));
GL.VIEW(GL.GLExplode(V,EVs,1.5,1.5,1.5,99,1));
meshes = GL.GLExplode(V,CVs[1:end],8,4,6,99,0.5);
GL.VIEW( push!( meshes, GL.GLFrame) );
```

Acknowledgements

The author is grateful to Antonio DiCarlo and Vadim Shapiro for sharing a lasting curiosity in the exploration of novel geometric territories, and to Giorgio Scorzelli, Francesco Furiani and Giulio Martella for help with the software.

[4] You may enjoy trying `Lar.cuboidGrid([2,2,2],true)`

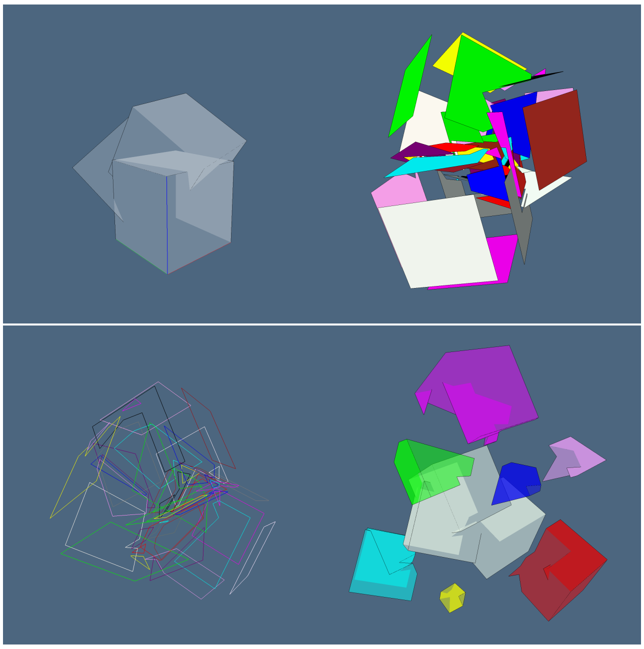

Fig. 6. Images of the 3D arrangement generated by three unit cubes: (a) transparent view of the input; (b) exploded 2-skeleton of 3D space partition; (c) exploded view of boundary 1-cycles of 2-cells; (d) exploded view of (transparent) 3-chain, which is the basis of 3-space arrangement generated by the input. The white 2-cycle is the boundary of outer space.

7. Conclusion

In this paper a very simple and general approach to geometric and topological computing using only Julia's [1] sparse arrays was presented. Sparse arrays should fit well with the fast diffusion of hybrid architectures and their advanced applications, using the best-in-class numerical language. The future applications may cover disparate fields of visual calculus, including geomapping, solid and geometric computer-aided design, virtual reality, computer vision, and medical imaging. We are just at the beginning of this journey. A lot of work remains, and this author has strong hope to share the enjoyment of discovery with the Julia's community.

8. References

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] Antonio Dicarlo, Alberto Paoluzzi, and Vadim Shapiro. Linear algebraic representation for topological structures. *Comput. Aided Des.*, 46:269–274, 2014.

[4] Christoph M. Hoffmann and Vadim Shapiro. Solid modeling. In J.E. Goodman, J. O'Rourke, and C. D. Tóth, editors, *Handbook of Discrete and Computational Geometry*. Chapman and Hall/CRC, Boca Raton, FL, 3rd edition edition, 2017.

[5] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21, March 1973.

[6] A. Paoluzzi, V. Shapiro, A. DiCarlo, F. Furiani, G. Martella, and G. Scorzelli. Topological computing of arrangements with (co)chains. *ACM Transactions on Spatial Algorithms and Systems*, August 2017. Submitted.

[7] T. Woo. A combinatorial analysis of boundary data structure schemata. *Computer Graphics & Applications, IEEE*, 5(3):19–27, March 1985.

[8] Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. Mesh arrangements for solid geometry. *ACM Trans. Graph.*, 35(4):39:1–39:15, July 2016.