Computer-Aided Design 134 (2021) 103003

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad

PolyFrame, Efficient Computation for 3D Graphic Statics☆

Andrei Nejur [a], Masoud Akbarzadeh [b,*]

[a] Université de Montréal Faculté de l'Aménagement École d'Architecture, Montréal, Canada
[b] Polyhedral Structures Laboratory, School of Design, University of Pennsylvania, Philadelphia, USA

ARTICLE INFO

Article history:
Received 27 April 2020
Received in revised form 7 January 2021
Accepted 12 January 2021

Keywords:
Three-dimensional graphic statics
Polyhedral reciprocal diagrams
Polyhedral graphic statics
Parallel manipulation of polyhedral diagrams

ABSTRACT

In this paper, we introduce a structural form finding plugin called PolyFrame for the Rhinoceros software. This plugin is developed based on the methods of 3D Graphic Statics and Polyhedral Reciprocal Diagrams. The computational framework of this plugin uses new robust and efficient algorithms for the creation and modification of complex funicular, compression-only structural forms and is freely available for students, designers, researchers, and practitioners in the fields of architecture, structural engineering, mechanical engineering, and material science. The geometry-based structural design methods are one of the most intuitive yet powerful structural design methods that have recently been extended to 3D based on the Principles of the Equilibrium of Polyhedral Frames. Still, the increased geometrical complexities of the polyhedral diagrams hinder more in-depth practical applications and the research in this field. The framework proposed in this paper can manage, in near real-time, the creation and transformation of reciprocal polyhedral diagrams with a large number of elements as form and force diagrams for structural design purposes. The paper also introduces a hybrid object-oriented data structure that extends and generalizes the previously proposed approaches and thus allows the users to incorporate a variety of different geometric constraints, including edge lengths and the location of the supports from the initial stages of design. Additionally, a new parallel manipulation algorithm is introduced that is capable of transforming polyhedral diagrams while preserving the edge directions and face normal. As a result, a designer can effectively manipulate both structural form and its force distribution without breaking their reciprocity.

1. Introduction

Funicular structural forms carry the applied loads in the form of pure tensile or compressive axial forces since the form or geometry of the structures precisely matches the direction of its internal flow of forces. Such structural forms are considered as highly efficient systems if loaded based on the initial design loading case and excluding buckling performance. This is due to the fact their geometry maximizes the structural performance and minimizes the use of materials.

Physical form finding techniques

Since the 17th century, many scholars, researchers, and practitioners have worked on this topic and suggested a variety of methods to find the geometry of compression-only or tension-only structural forms for a given system of applied loads. The most well-known instance of these methods is the *hanging chain* model proposed by Robert Hooke in 1675 [1]. He used a simple chain and showed that the chain forms a funicular tension-only geometry under its weight. The inverted geometry then works as a compression-only form for the same given applied loads if buckling is not of concern [2–4]. The Sagrada Familia church is an excellent example of using such techniques in design and engineering where Antoni Gaudí spent years in building tedious hanging chain models to find the 3D funicular forms for his breathtaking structures (Fig. 1) [5–7].

Numerical form finding techniques

Advances in computer science and engineering have allowed the development of techniques such as physics simulation engines [8], particle–spring systems [9,10], force density methods [11,12], and dynamic relaxation [13] to simulate the physical transformation of materials and find the funicular forms for the given loading conditions and substitute tedious physical form-finding techniques [14].

In all these techniques, a designer starts with an arbitrary network as an input and receives a tension/compression-only

☆ This paper has been recommended for acceptance by C. Christopher Williams.

* Corresponding author.
E-mail addresses: andrei.nejur@umontreal.ca (A. Nejur), masoud@upenn.edu (M. Akbarzadeh).

https://doi.org/10.1016/j.cad.2021.103003
0010-4485/

Fig. 1. Left: Physical form finding by Antoni Gaúdi; and Right: The Basílica de la Sagrada Família's funicular structures designed by Gaúdi (photo credit: authors, 2019).

result as an output. In fact, the final geometry is the result of a *black box* computational processes where the contribution of the designer is limited to framing the design question, similar to the physical form finding process. Note that the word *black box* in this context only refers to the lack of an intuitive relationship between the derived form and the internal and external forces. Although numerical methods are quite powerful in finding efficient equilibrium solutions, it is still challenging for some designers to recognize the effective parameters in the form finding process. Moreover, these methods are harder to be used as an intuitive pedagogical tool to explain the structural concepts for educational purposes.

Geometry-based form finding methods

In contrast to these black-box numerical methods, there is a powerful and intuitive method of structural design called *Graphic Statics* (GS) which is based on pure geometry. Graphic Statics (GS) methods originated in the pre-digital era and continue to be used and developed even today [15–27].

In this method, the geometry of the structure is represented by a geometric diagram called *form*, and the magnitude and equilibrium of forces is represented by another geometric diagram called *force*. These diagrams are reciprocal i.e. *geometrically dependent* and *topologically dual*. Some of the structures designed by GS-based methods are among the best examples of innovative use of material and efficiency, and many eminent engineers and designers such as Guastavino, Maillart, Koechlin, Nervi constantly used graphic statics in the design of their masterpieces [4,21,28,29]. Despite its clear strength and advantages, traditional graphical statics were based on 2D diagrams, and therefore, a designer can only design 2D abstraction of three-dimensional structures. Moreover, the lack of computational and representational tools in the 19th century limited the use and progress of graphic statics. It encouraged many researchers to shift to numerical methods at the end of the 19th century.

These methods have been extended to 3D on the basis of several approaches. One approach is based on "the Principle of the Equilibrium of Polyhedral Frames", a 150-year-old proposition by Rankine in Philosophical Magazine [16,17,30]. This approach is

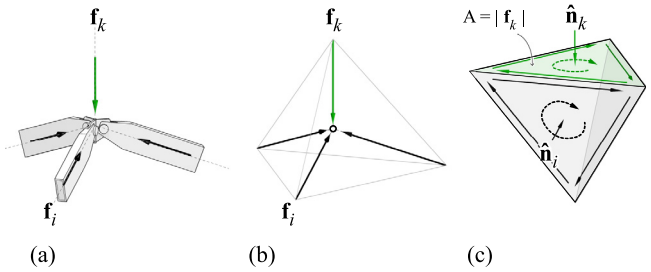


Fig. 2. (a) A 3D structural joint with an applied force and internal forces in its members; (b) the form diagram/bar-node representation of the same joint in the context of 3DGS; and (c) the force diagram/polyhedron representing the equilibrium of the same node in 3DGS [43].

used for form finding of high-performance lattice structural systems with polyhedral geometries [25,31–41]. The other approach is based on vector-based reciprocal diagrams in 3D suggested by Maxwell [16] in his groundbreaking paper "On Reciprocal Figures, Frames, and Diagrams of Forces", which is beyond the scope of this paper.

The scope of this paper is limited to the method based on Rankine's proposition which is called *3D Graphical Statics using Reciprocal Polyhedral Diagrams*. In this method, the equilibrium of the forces in a single node is represented by a closed *polyhedron* or a polyhedral *cell* with planar faces (Fig. 2.c). Each face of the force polyhedron is perpendicular to an edge in the form diagram (Fig. 2.b), and the magnitude of the force in the corresponding edge is equal to the area of the face in the force polyhedron. The sum of all area-weighted normals of the cell must equal zero. This can be proved using the divergence theorem [31,34,37,42].

In this method, the term *polyhedral frame* is used for a configuration of bars and nodes, establishing a three-dimensional thrust network of forces with no moment resistance at the joints. The term frame should not be mistaken with the structural engineering literature's frame structures with fixed joints and bending moment capacities.

The limitations of polyhedral graphic statics

The 3D/polyhedral graphic statics has some limitations as pronounced by Maxwell [16] in 1864. As mentioned by Maxwell, there are specific equilibrium problems that can be solved using vector-based graphic statics, but might not be easily solved using polyhedral graphic statics.

Vector-based 3D graphic statics

Graphic statics has also been extended to 3D based on *polygonal* reciprocal diagrams as suggested by Maxwell [16] and later by Cremona [20]. In this approach, each edge in the form diagram is reciprocal to an edge in the force diagram, and the equilibrium of forces is represented by closed non-planar polygons. This method is often referred to as the *vector-based 3D graphic statics* [26,44–47].

Current implementation limitations

Using the reciprocal polyhedral diagrams for form finding requires a robust computational framework able to handle in near-real-time the creation and modification of those diagrams. The extraction of a reciprocal funicular form from a given group of faces that define closed polyhedral cells was addressed in [32] and [48]. However, the algorithms presented in the former cannot handle a large number (300+) of faces as input in a reasonable amount of time (e.g., less than 10 s) that would permit user feedback or without crashing the host application.

Furthermore, the user did not have direct control in manipulating the diagrams in the former. Lee [48] addressed this problem and released a set of tools for 3d graphic statics as a Python package part of COMPAS [49]. The extension is named COMPAS-3gs [50]. Nevertheless, the capability of the proposed methods to handle interactive geometric manipulations for a large number of faces should be attested. Moreover, the direct manipulation of the polyhedral geometry that does not break the diagrams' reciprocity was not addressed. All these algorithms are based on iterative geometric methods. i.e., the reciprocity is induced by making the edges of the constructed diagram perpendicular to the faces of the input diagram in multiple iterative steps. The slow processing time for the convergence of the solutions and the self-intersection of polyhedral faces during the computation process are among the main challenges of working with iterative methods.

Recently, a new tool that brings the research developed at Block Research Group by Akbarzadeh et al. [31] was released by an independent developer, Graovac [51], as an add-on for Grasshopper. Although the add-on is considerably faster and able to handle a large number of (10k+) elements, it still does not address the diagram manipulation and subsequent transformation of the reciprocal polyhedral diagrams. Moreover, it cannot construct reciprocal diagrams for a given set of geometric constraints and thus is quite limited in design applications.

In another recent development, Hablicsek et al. [43] provided an algebraic method for a single-step construction of the reciprocal polyhedral diagrams of 3D graphic statics. Although the algebraic formulation is quite robust in defining the reciprocal diagrams' mathematical relationships, the accumulation of the numerical errors might cause problems for a very large number (10k+) of elements (See Section 3.2). As a result, even though, in some cases, the algebraic methods cannot find any solution, the iterative methods can still find some within a certain tolerance defined by the user. This is an essential advantage of using iterative methods over algebraic techniques.

1.1. Contribution

This paper presents a hybrid data structure which can reduce the computational time of working with such polyhedral geometry. Consequently, it can increase the maximum number of polyhedral faces that can be processed within a reasonable amount of time (less than 10 s) that allows for user feedback. It also propose a new and more general computational algorithm capable of handling multiple geometric constraints applied simultaneously to polyhedral elements. Further, a new damping function is introduced for the iterative process that greatly reduces the self-intersecting instances in the compression-only form finding method. Although self-intersection is not desirable for compression-only form finding, it is necessary for combined tension and compression systems which is not discussed in the context of this paper (see for example, [48,52,53]). Moreover, a new, graph-based parallel transformation algorithm is presented capable of direct manipulating polyhedral diagrams while keeping the alignment of the edges (reciprocity of the diagrams) intact. The outcome of this research is available as a free plugin called *PolyFrame* [54] for Rhinoceros software [55]. The plugin implements via separate Rhino commands the workflow presented in Fig. 3 for iterative, compression-only, form finding of structural form. Section 5 presents an overview of the implemented software solution in Rhinoceros. All algorithms and methods presented in this paper are accessible on the PolyFrame repository on GitHub [56].

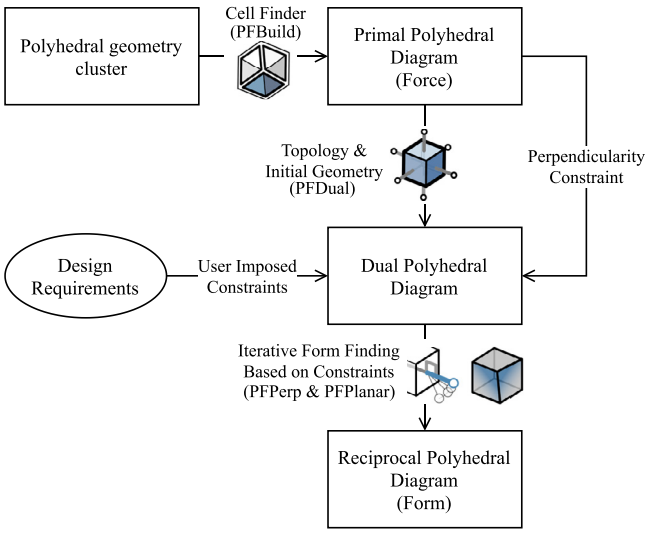


Fig. 3. A general overview of the workflow using the iterative transformation algorithm. The flow chart includes the icons and the names for the corresponding commands (in parenthesis) implemented in PolyFrame the Rhino3d plug-in.

1.2. Nomenclatures

Table 1 presents a synthetic view of all the symbols used in the paper and their associated notations.

2. Developing a new hybrid data structure

The first step in making possible a faster and more robust interaction with 3D graphic statics (3DGS) using reciprocal polyhedral diagrams was to create a new data structure. A more intuitive, less memory intensive, and an arguably more robust data representation of the polyhedral diagram will be introduced in the next paragraphs.

In order to fully explain this concept, we will first describe the components of the data graph and their relation to the parts of the polyhedral diagram. We call this a hybrid data structure because its objects encode the polyhedral properties (geometry) as well as each part's links to relevant other polyhedral parts (topology).

2.1. 3D reciprocal diagrams

In the context of graphic statics, we have two reciprocal diagrams that are geometrically dependent and topologically dual, where the change of one affects the properties of the other. The *force* diagram consists of closed polyhedral cells. The closeness of each cell represents the equilibrium of a node in the dual diagram, and the face areas of the force represent the magnitude of internal/external forces in the form diagram. In our polyhedral approach, the *form* diagram consists of both open and closed polyhedral cells with planar faces where the edges of the open cells represent the applied loads or reaction forces in the system [31]. The inclusion of open cells in the form diagram is the main difference of topological relationship between the primal and dual as presented in this paper and the relationship proposed by Maxwell in 1864 [16]. In Maxwell's proposition, both form and force diagrams consist of closed polyhedral cells where both diagrams can be called form and force interchangeably. However, the form diagram in Maxwell's case was a self-stressed system, which does not allow for the inclusion of the external forces.

Let us call the starting (input) diagram the *primal*, Γ, and the reciprocal polyhedron the perpendicular *dual*, $\Gamma^{\dagger}$ (Fig. 4). Both

Table 1
Nomenclature for the symbols used in this paper and their corresponding descriptions.

Topology	Description
Γ	Primal diagram
$\Gamma^\dagger$	Dual, reciprocal diagram
v_i	Vertex i in Γ
v_i^t	Transformed vertex i from diagram Γ
$e_{(i,j)}$	Edge between vertices i and j in Γ
$e^t_{(i_t,j)}$	Partially transformed edge with transformed vertex i
$f_{(i,j,k,..)}$	Face in Γ defined by i,j,k,.. vertex set
$f^t(i_t, j, k_t)$	Partially transformed face with transformed vertices i,k
$c_{(i,j,k,..)}$	Cell in Γ defined by i,j,k,.. vertex set
$v_i^\dagger$	Vertex i in the dual diagram $\Gamma^\dagger$
$e^\dagger_{(i,j)}$	Edge between vertices i and j in dual diagram $\Gamma^\dagger$
$f^\dagger_{(i,j,k,..)}$	Face in $\Gamma^\dagger$ defined by i,j,k,.. vertex set
$c^\dagger_{(i,j,k,..)}$	Cell in $\Gamma^\dagger$ defined by i,j,k,.. vertex set
V	Set of vertices of Γ
V^t	Set of transformed vertices of Γ
E	Set of edges of Γ
F	Set of faces of Γ
F^t	Set of partially transformed faces in Γ
F^a	Subset of F^t with active in-face intersections
F^p	Subset of F^t with passive in-face intersections
F^o	Subset of F^t with out-of-face intersections
Geometry	
1	Original position in 3d space for vertex 1
1_p	Perpendicular transformation for vertex 1 position
1_l	Edge length transformation for vertex 1 position
1_s	Face area scale transformation for vertex 1 position
1_t	Transformed position for vertex 1
p	Position for a generic vertex
e_x	Generic edge with one vertex in p
p_x	Transformed position for p by rotation of e_x
p_{pc}	Position constraint for vertex in p
p_{ae}	Average position of multiple edge-related transforms
p_{af}	Final average position of vertex
p_{an}	Normalized average position of vertex
Vectors	
$\mathbf{v}_i$	Computed constraint vector acting on v_i
$\mathbf{v}_t$	Averaged transform vector acting on v_i
Parameters	
t_{v_i}	Constraint function for v_i
$t_{e_{(i,j)}}$	Constraint function for $e_{(i,j)}$
$t_{f_{(i,j,k..)}}$	Constraint function for $f_{(i,j,k..)}$
m_{v_i}	Multiplication factor for t_{v_i}
$m_{e_{(i,j)}}$	Multiplication factor for $t_{e_{(i,j)}}$
$m_{f_{(i,j,k..)}}$	Multiplication factor for $t_{f_{(i,j,k..)}}$
s	Iteration number
s_{max}	Maximum iteration number
δ	Maximum recorded vertex translation per iteration
δ_{min}	Minimum allowed vertex translation
Other	
V_v	Dictionary of sets of vectors vertices in Γ
V_f	Dictionary of sets of scale factors for vertices in Γ

form and force diagrams can be considered the primal (input), and thus the other diagram will be called the dual (output). The vertices, edges, faces, and cells of the primal are denoted by v, e, f, and c, respectively, and the ones of the dual are super-scripted with a dagger (†) symbol (Fig. 4). Each face of one diagram is perpendicular to the edges of the other.

2.2. Primal as force diagram

The force diagram consists of closed polyhedral cells and if regarded as the primal, its topological dual diagram will be called

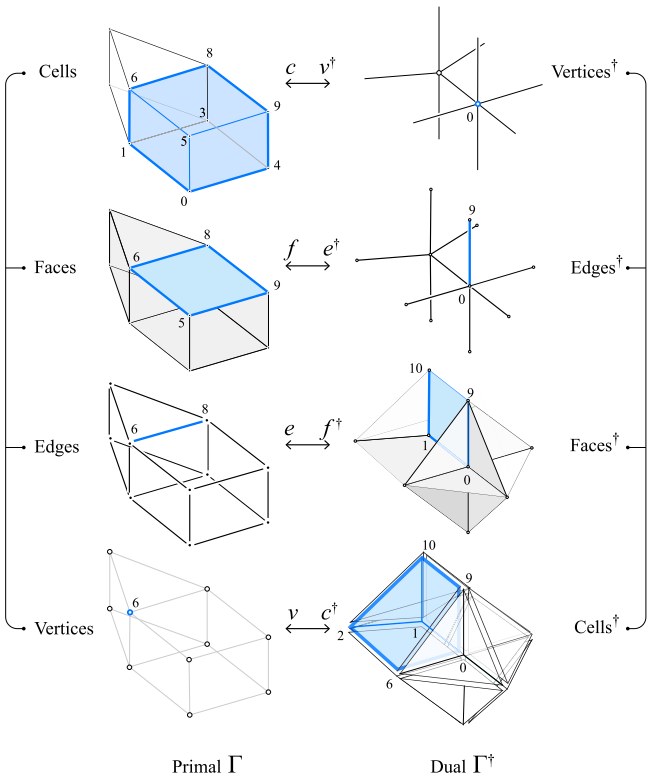


Fig. 4. Data structure connections in two dual diagrams.

the form diagram as shown in Fig. 4. In this case, each polyhedral cell in the primal is reciprocal to the *internal* vertices of the dual, e.g. cell $c_{(0,1,3,4,6,5,9,8)}$ is reciprocal to $v_0^\dagger$ in $\Gamma^\dagger$ (Fig. 4).

In addition, the faces f of the primal correspond to the *internal* edges $e^\dagger$ of the dual. The internal edges are those that are shared by more than two faces. In the case of open cells in the form diagram, an edge that belongs to a single face may be called an open edge, and their corresponding faces may be called an open face. For instance, the edge connecting vertices 9 and 10, in the form, is an open edge and does not have any reciprocal counterpart in the primal. Its vertices 9 and 10 also excluded from the topological relations of the two diagrams. Nevertheless, the faces with an open edge in the dual are included in the reciprocal diagrams and correspond to the primal's external edges. Besides, the primal's external faces, in this case, are perpendicular to the internal edges of the open cells in the form diagram. The areas of the primal's external faces represent the magnitude of the applied loads or the reaction forces in the form. Note that if the primal has an open/missing face, then it will be considered as a form diagram with open cells and not as the force diagram.

2.3. Primal as form diagram

The form diagram can also be used as an input and will be referred to as the primal. The figure starting from the form is not included to avoid redundancy in this paper. Fig. 4 may still represent the primal as a form if the geometric data on the right and left are switched. Note that starting from a funicular form is not always ideal because form finding aims to find the geometry of the force flow for a given internal and external force distribution. However, for highly complex models, the resulting form may have edges that are not fully perpendicular to the dual faces. In such cases, considering the form diagram as the primal may let us adjust the force diagram and reach a higher level of precision in establishing the reciprocity between the two.

2.4. Polyhedral elements as data

The *data objects* follow the general rules of a half-face data system described Open Volume Mesh by Kremer et al. [57] with object types created for the following polyhedral components: vertex, edge, face, and cell. OpenVolumeMesh focuses on the smallest possible memory footprint for each mesh element to facilitate working with a high number of elements. In contrast, our approach focuses on more feature-packed elements that store considerably more connections to topologically related elements. The main reason is simplicity in the traversal of the graph of interconnections and following an object-oriented design paradigm. This approach includes not only the data at the element level but also the functionality. Each element is a quasi-self-contained entity with properties and methods accessible directly from itself, able to reference all its topological relations via pointers directly. COMPAS [49] by Block Research Group, through its VolMesh class, implements a similar data structure to OpenVolumeMesh that uses key–value pairs to maintain topological relations between polyhedral elements. Reeves et al. [58] present a fabrication oriented implementation of the same data structure but focuses on geometrical construction aspects instead of presenting the encoding of topological relationships.

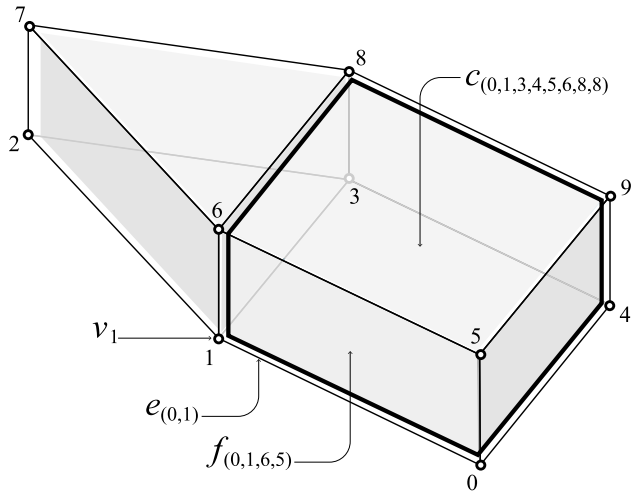
The most basic element of a polyhedral diagram is a *vertex* representing a polyhedral node. The polyhedral *edge* or half-edge is represented as a data container with two ordered vertices. Each half-edge maintains an active connection to its pair. The polyhedral *face* or half-face is an object containing a set of ordered edges, which include ordered sets of vertices. Polyhedral faces are also linked to their respective pair as the base elements of a half-face data system. The polyhedral *cell* is stored as an object containing sets of faces, edges, and vertices. The *polyhedral diagram* is an encompassing data object storing sets of cells, faces, edges, and vertices.

2.5. Polyhedral elements interconnections

The object interconnections are designed to describe the topological relations of 3d reciprocal polyhedral diagrams closely. The proposed data structure consists of a set of layers on which resides a bidirectional graph representing the connections of a single type of data objects like cells, faces, edges, or vertices. Every element on every layer connects to elements on other layers following the topological hierarchy of the polyhedral diagram. For instance, cells are linked to their constitutive faces, faces are linked to contour edges, and edges to their end vertices. Each connection mentioned above is bi-directional in the sense that all elements that are part of more complex ones maintain an active link to those. Besides those, there are also a number of shortcut connections that skip the hierarchy and connect the most complex elements. For example, cells connect directly to vertices.

Moreover, each element on a layer is connected to all its neighboring elements of the same type using the inter-layer connections. For instance, two vertices sharing an edge will be connected as neighbors on the same layer through the edge connection. The same connections can be traced for all the other element types such as edges, faces, and cells.

Fig. 5 illustrates a part of the data connections between the elements of a simple polyhedral diagram, made up of only ten vertices. In order to show the connections between the elements, the picture contains tables with comprehensive data descriptions for the main types that make up the polyhedral diagram described above. Each data object (vertex, edge, face, and cell) has a number of slots that contain pointers or collections of pointers to other data entities with whom the object has topological



Vertex v_1	
Id	1
Pos	(x:5.0, y:0.0, z:0.0)
Edges	$\{e_{(1,0)}, e_{(1,6)}, e_{(1,2)}, e_{(1,3)}, e_{(0,1)}, e_{(6,1)}, e_{(2,1)}, e_{(3,1)}\}$
Faces	$\{f_{(0,1,6,5)}, f_{(0,1,3,4)}, f_{(1,3,8,6)}, f_{(1,3,2)}, f_{(1,6,7,2)}, f_{(3,1,6,8)}\}$
Cells	$\{c_{(0,1,3,4,5,6,8,9)}, c_{(1,2,3,6,7,8)}\}$
Dual	cell: $c^{\dagger}_{(0,1,2,6,7,8)}$

Edge $e_{(0,1)}$	
Id	1
Verts	(v_1, v_0)
Faces	$\{f_{(0,1,6,5)}, f_{(0,1,3,4)}\}$
Cells	$\{c_{(0,1,3,4,5,6,8,9)}\}$
Pair	$e_{(1,0)}$
Dual	face: $f^{\dagger}_{(6,0,7)}$

Face $f_{(0,1,6,5)}$	
Id	1
Verts	$\{v_0, v_1, v_6, v_5\}$
Edges	$\{e_{(0,1)}, e_{(1,6)}, e_{(6,5)}, e_{(5,0)}\}$
Cell	$c_{(0,1,3,4,5,6,8,9)}$
Pair	$f_{(0,5,6,1)}$
Dual	edge: $e^{\dagger}_{(0,6)}$

Cell $c_{(0,1,3,4,5,6,8,9)}$	
Id	1
Verts	$\{v_0, v_1, v_3, v_4, v_5, v_6, v_8, v_9\}$
Edges	$\{e_{(0,1)}, e_{(1,6)}, e_{(6,5)}, e_{(5,0)}, e_{(5,9)}, e_{(9,8)}, e_{(8,6)}, e_{(1,3)}, e_{(3,8)}, e_{(1,6)}, e_{(9,4)}, e_{(4,3)}, e_{(4,0)}\}$
Faces	$\{f_{(0,1,6,5)}, f_{(0,1,3,4)}, f_{(5,0,4,9)}, f_{(6,5,9,8)}, f_{(1,6,8,3)}, f_{(3,4,9,8)}\}$
Dual	vertex: $v^{\dagger}_0$

Diagram Γ	
Id	1
Verts	$\{v_0, v_1, v_3, v_4, v_5, v_6, v_8, v_9\}$
Edges	$\{e_{(0,1)}, e_{(1,0)}, e_{(1,6)}, e_{(6,1)}, e_{(6,5)}, e_{(5,6)}, e_{(5,0)}, e_{(0,5)}, e_{(5,9)}, e_{(9,5)}, e_{(9,8)}, e_{(8,9)}, e_{(8,6)}, e_{(6,8)}, e_{(1,3)}, e_{(3,1)}, e_{(3,8)}, e_{(8,3)}, e_{(9,4)}, e_{(4,9)}, e_{(4,3)}, e_{(3,4)}, e_{(4,0)}, e_{(0,4)}\}$
Faces	$\{f_{(0,1,6,5)}, f_{(0,1,3,4)}, f_{(5,0,4,9)}, f_{(6,5,9,8)}, f_{(1,6,8,3)}, f_{(3,4,9,8)}, f_{(1,2,7,6)}, f_{(7,2,3,8)}, f_{(7,8,6)}, f_{(2,1,3)}, f_{(3,8,6,1)}, f_{(5,6,1,0)}, f_{(4,3,1,0)}, f_{(9,4,0,5)}, f_{(8,9,5,6)}, f_{(8,9,4,3)}, f_{(6,7,2,1)}, f_{(8,3,7,2)}, f_{(6,8,7)}, f_{(3,1,2)}\}$
Cells	$\{c_{(0,1,3,4,5,6,8,9)}, c_{(1,2,3,6,7,8)}, c_{(0,1,2,3,4,5,6,7,8,9)}\}$
Dual	Diagram: $\Gamma^{\dagger}$

Fig. 5. Structure of the data inside a polyhedral diagram.

connections. Each table exemplifies the connections for one type of element for this connection. Each component is part of the polyhedral diagram shown in the top part of the figure, and the connections for the examples can be traced to it.

2.6. Connections between dual elements

The dual/reciprocal relations of a polyhedral diagram with its dual diagram are stored at multiple levels presented in Fig. 5. The connections between dual components are the geometric and topological relationships of the polyhedral reciprocity. Vertices in the primal diagram point towards corresponding cells in the dual diagram. Edges in the primal diagram, reference faces in the dual diagram. Faces in the primal diagram reference edges in the dual and finally cells in the primal reference vertices in the dual. Fig. 4 shows a graphical representation of those described relationships.

2.7. Advantages of a hybrid data structure

A hybrid data structure provides a series of advantages over a traditional hierarchical one with exclusive top-down component

connections where more complex elements maintain a list of their parts. Some of the most important advantages include a decentralized and redundant way of storing topological dependencies between the components of the two diagrams; simple access, with multiple available routes, from any element in one diagram to its corresponding element in the dual diagrams; and the reconstruction of the dual diagram from partial data.

All those objectives are accomplished by storing pointers inside each object to the relevant connected parts. A high memory footprint is avoided since only the number of connections (pointers) is increased to resolve all necessary connections. As depicted in Fig. 5 each component maintains a number of direct connections to other data objects via pointers. Each entry in the table under the element name, written with subscript or superscript, represents a connection to another related data object. As we are showing in the algorithm implementation in Section 3 and especially in Section 3.4, by storing object references to connected sub-parts as properties of any polyhedral diagram element, we inherently have access to those objects and their methods directly from the starting object (embedded functions see the concluding parts of Section 2.7). In the case of Lee et al. [41] more steps are required to achieve the same goal. One must first extract a key or index from the current object then access the connected object from a central collection using a lookup created at start and finally run a static algorithm with the values read from the collection. In our case, since the memory pointer is already present in the object, moving to the next, related object is a one step process. In the case of dictionaries and the same is true for stored indexes, the similar process takes at least two steps. Our method simplifies the traversal of the data structure compared to the other methods when it comes to data structure explorations and algorithm development. Any developer using a modern coding environment with code hinting (like InteliSense from Microsoft's Visual Studio) can interactively explore the functionalities embedded in the objects. Furthermore, this approach enables the direct navigation of the data structure as is stored in memory at run-time. This is useful for debugging or exploration reasons. The navigation is similar with the navigation of the geometrical polyhedral diagram itself.

Simpler component access through redundant interconnections.

Redundantly-interconnected components establish a robust data structure. The system provides for an efficient search allowing multiple paths from one object to the other parts of the data structure and their duals. In Fig. 4 for instance, in order to get all the applied loads adjacent to vertex v_0 from the primal (force) diagram, one can list all the faces connected to v_0 or list all the non-exterior edges of cell $c^{\dagger}_{(0,6,5,9)}$ in the dual diagram.

All topological connections are created at the beginning, and the connectivity is cached for later reuse. Since all connections between elements are established when the structure is built, any subsequent specific traversal operation can be effectuated much simpler. This connection helps users access the data and manipulate the diagrams instantly in an interactive environment.

Additional data is stored in the structure, and each object in the data structure can store additional information, such as geometric or structural properties. These include coordinates, constraints, target positions, length and area, as well as buckling performance, material details, etc.

Partial diagrams reconstruction allowed

The entire geometry and topology of both primal and dual can be reconstructed even if (due to user errors) some geometric information is lost or topological elements get disconnected. This property is especially helpful for recording and propagating partial transformations of the diagrams while enforcing the reciprocal relationship. See Section 4 for an implementation of the parallel transformation algorithm that makes use of this propriety.

Primal and dual diagrams maintain connections at the element level. Once the dual relations are established, they can be instantly reconstructed from their dual if their geometry is lost since the primal and the dual diagrams are referencing each other. Furthermore, the reference to the dual elements ensures the inclusion of all further modifications to any parts of the diagrams in the connected dual reference.

A hybrid data structure offers intuitive connections between elements with fewer steps for subsequent operations. All topological connections exist as connections between data objects. Traversing the graph representing the polyhedral diagram can follow the user's geometrical and visual understanding of the diagram. Each connection between objects is accessible via a specific property of the start object using simple dot notation in a scripting environment. For example *startCell*.*Faces*[*x*].*Pair*.*Cell* gets a certain neighbor of a polyhedral cell by picking a specific face of the *startCell*, selecting the element with index of *x* from its *Faces* collection, getting the *Pair* face of the selected face, and finally getting the *Cell* of that *Pair* face.

In contrast, in a predominantly hierarchical data structure, only a minimum number of connections between the topologically connected elements is stored. This makes the diagram's initial creation faster but slows down any structure traversal routine that needs connections that are not yet computed. Many specific operations of establishing and exploring reciprocity of polyhedral diagrams require multiple graph traversal operations. Having all required connections between elements in the graph cached and ready, rather than compute them for each step or iteration, will increase the speed of the algorithms that make use of the data structure.

The data structure is implemented in the .Net Framework 4.5, specifically in *C#* with individual classes for each element type like vertex, edge, face, cell, and the polyhedral diagram. Each class implements several properties (wrapped fields with accessors) that hold the collections of pointers (reference types) to the relevant connected objects, presented in Fig. 5. Besides constructors, each class implements several dynamic functions (methods) accessed directly from created class instances. These provide functionality that is specific to each type and its run-time context (like its creation, the population of its data fields, linking and un-linking to other members, transformation etc.)

3. A robust iterative algorithm for the construction of the reciprocal polyhedral diagrams

This part of the paper discusses contributions to methods and algorithms employed for creating, maintaining and exploring the reciprocal relationship between polyhedral diagrams, using iterative algorithms.

The present research implements most of the processes described in [31] using the new hybrid data diagram, thus allowing for increased speed and diagram complexity. Our contribution, presented below, addresses some of the limitations of the perpendicularization process described in the same paper and extends the process' functionality through a new, more general algorithm. All implemented algorithms are restricted to polyhedral diagrams with planar and convex faces. A high-level overview of the workflow making use of the iterative algorithm is presented in Fig. 3. From simple geometry, the primal diagram is created. This is usually the force diagram. The dual diagram is topologically derived from the primal. The dual is bound by several constraints, most notably the edges' perpendicularity to the primal faces. Other constraints like edge length or vertex positions, can be set too at this step. Finally, the iterative algorithm transforms the dual diagram's geometry to minimize the deviation from all the imposed constraints.

3.1. Solution space

An algebraic formulation for the reciprocal diagrams of 3DGS clarifies the number of solutions and the parameters to control the properties of the solutions. This information can be quite useful, even if we use iterative algorithms to find solutions. Constructing the dual from a given primal is possible by developing algebraic constraints between the two diagrams [43,59]. Consider an edge $e_{(i,j)}$ of the primal diagram Γ and all its attached faces $f_1, f_2, \ldots, f_k$. The edge $e_{(i,j)}$ is reciprocal to a face $f_k^\dagger$ in the dual. By keeping the thumb of the right hand parallel to the edge, the fingers curl around the edge going through the attached faces f_{1-k} establishing a consistent orientation of the edges $e^\dagger_{(m,n)}$ of the face $f_k{}^\dagger$ of the dual. We will denote these directed edges of the dual by $\mathbf{e}^\dagger_{(m,n)}$.

Since the face $f_k{}^\dagger$ is a closed polygon, the sum of the edge vectors $\mathbf{e}_{(m,n)}{}^\dagger$ should be zero. Hence, we obtain a vector equation

$$\sum_{f_k^\dagger} \mathbf{e}^\dagger_{(m,n)} q_{(m,n)} = \mathbf{0}$$

where the sum runs over the attached faces f_m of the edge $e_{(i,j)}$ of the primal Γ, and $q_{(m,n)}$ denotes the edge length of $e^\dagger_{(m,n)}$ in the dual $\Gamma^\dagger$. We can rewrite the above equation in terms of the chosen unit normal vectors $\mathbf{n}_{1-k}$ of faces f_{1-k} as

$$\pm \mathbf{n}_1 q_{(0,1)} \pm \mathbf{n}_2 q_{(1,3)} \pm \ldots \pm \mathbf{n}_k q_{(m,n)} = \mathbf{0} \tag{1}$$

where we have

$$\begin{cases} +\mathbf{n}_i & \text{if matches the curl direction around } e_i \\ -\mathbf{n}_i & \text{otherwise.} \end{cases}$$

Similarly, as before, each vector equation yields three linear equations for the edge lengths, and we obtain a linear equation diagram for the edge length vector $\mathbf{q}$ which can be described by a $[3e \times f]$ matrix that we call the equilibrium matrix $\mathbf{A}$:

$$\mathbf{Aq} = \mathbf{0}. \tag{2}$$

3.2. Number of solutions

In the equilibrium matrix $\mathbf{A}$, e denotes the number of edges of the primal diagram, and f denotes the number of faces of the primal diagram [43,59]. The dimension of the solution space of Eq. (2) equals

$$GDoF = f - r \tag{3}$$

where r is the rank of the equilibrium matrix $\mathbf{A}$, the GDoF is called the Geometric Degrees of Freedom of the dual diagram which is the dimension of the solutions space. If GDoF is zero, then the only possible solution of the equilibrium equation system is the zero vector. In that case, the dual collapses into a point, which we do not consider a solution. If the GDoF equals one, then there is a unique solution, which provides a unique dual diagram (up to scaling). Finally, if the GDoF is bigger than one, then there are multiple significantly different solutions. The inaccuracy or the accumulation of the numerical errors in the calculation of the rank of the equilibrium matrix might result in GDoF to be zero. The specific relation between the geometry of the primal which results in zero GDoF is not entirely evident to the authors.

If the GDoF is zero, which rarely happens, the algebraic formulation of 3DGS cannot help finding a solution. Nevertheless, constructing reciprocal diagrams using the iterative approach allows finding the closest solution within a specific tolerance/deviation defined by the user, and thus, is more forgiving than the algebraic methods. Simply put, the equilibrium is achieved if the deviation between the angle of an edge in the dual and the normal of its corresponding face in the primal is quite small — depending on the precision of the model space. A qualitative observation over a variety of form and force diagrams suggests that usually the form diagram with triangular faces might have a larger deviation compared to the configuration with no triangular face.

3.3. The algorithm development

The process proposed in this paper uses a new, more general computational engine that works in a similar manner to a particle–spring system operated by forces and stiffness [14]. In mathematical terms, the cumulative effect of those forces and spring stiffness on the individual points of the systems (the particles) is expressed as the average of a set of vectors anchored in each point. In our implementation, all geometrical transformations, acting iteratively on a diagram, are expressed as vectors computed individually for each vertex. Each transformation is defined with a scaling factor that is applied to the respective resulting vector acting on the vertex when computing the average. Even though the scaling of the vectors makes this a weighted average, for the clarity of the explanations and of the figures, we will present it as a simple average. The exception is the description of the software algorithm where exact description of the implemented solution is more important.

The approach presented below allows for the inclusion of additional user imposed rules in the form-finding process. It also allows for the exploration of the solution space when the dual diagram has Geometric Degrees of Freedom (GDoF).

The proposed algorithm can work in a reverse approach if the dual has very few GDoF, and simple linear iteration cannot establish reciprocity. This effectively stops the transformation of the dual and starts transforming the primal until the reciprocity and thus the equilibrium is achieved.

The iterative process works by moving the vertices of the dual diagram in multiple steps according to the average of the set of vectors computed for each vertex. For example, in Fig. 6, two vectors are acting simultaneously on vertex v_0 and v_1. 0_p and 1_p are the target positions for v_0 and v_1 based on the rotation that would make edge $e_{(0,1)}$ parallel to the normal of its dual.

Once the edge rotates around its midpoint R, the vertices 0_l and 1_l are the target positions for the edge $e_{(0,1)}$ with the specified length. We can consider all those simultaneous target positions for the vertices as vectors anchored on the vertices. At each step of the iteration, each vertex moves to a position determined by the average of the vectors anchored in it. The vertices move at every step until the length of the resultant average is zero. This final transformed position is marked as 0_f and 1_f respectively in Fig. 6. This can be when all the constraints are satisfied and all individual vectors are zero in length, or when the constraints are in equilibrium and the individual vector average in every vertex is zero in magnitude.

3.4. Constraint-based manipulations

Since the criteria are stored in the pertinent data objects, not as variables but rather as functions and are computed dynamically during each iteration, there is no restriction regarding the number and type of simultaneous constraints applied on a transformed diagram. Together with the reciprocity specific constraints, like *perpendicularity* and *planarity*, other user imposed criteria like *edge length*, *face area* or *vertex position in space* can be added. The only requirement is that any applied constraints should be expressed as sets of vectors applied at the vertex level.

In the following paragraphs we will describe how the constraints for reciprocity at diagram level translate into multiple divergent vectors at the vertex level, for two specific operations: Perpendicularization and Planarization. Additionally we will introduce a set of design specific constraints and their translation into vector based actions at vertex level.

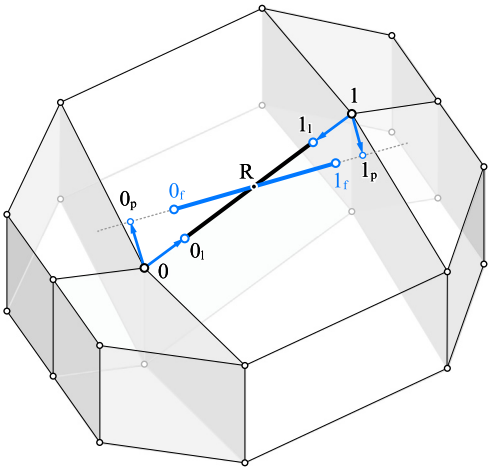


Fig. 6. Example of vector decomposition for two separate constraints: perpendicularization and edge length.

3.5. Perpendicularization

Perpendicularization is the core operation of polyhedral diagram transformation to achieve reciprocity between two dual diagrams. It ensures that the edges of the dual polyhedral diagram become perpendicular to the corresponding faces in the primal diagram. The most important transformation in the case of perpendicularization is the rotation of every edge around its mid-point. This operation can be expressed as two translations of the edge endpoints and thus can be represented by one vector anchored in each of the edges vertices. Fig. 6 already details this operation.

One of improvements introduced with the framework we proposed is that every vertex regardless of the applied constraints, has a movement limit per each step of the iteration. This can be expressed as the damping function (magnitude limitation) for the vector average acting on every vertex.

This feature greatly reduces the chance that large individual vertex movements cause disturbance in the diagram through edge flipping.

This way, transformations have the chance to be absorbed by the polyhedral diagram before any edges reverse their direction. This is especially useful to mitigate the potential disruptions caused by rotating long edges connected to relatively short edges. In such situations the short edges can be accidentally reversed by the large movements forced by the long edges causing other edges to intersect and produce complex faces. Fig. 8b shows how this feature works. For an explanation of the process depicted in Fig. 8 see Section 3.7

3.6. Planarization

Planarization viewed as a singular operation is a common technique for making the faces of a polyhedral diagram planar. For every vertex of the planarized face this implies a translation into the best-fit plane computed for the position of the face vertices. In this operation the damping function facilitates the absorption of large movements into the diagram without significant disturbance.

User-imposed constraints like a target vertex position can be expressed as a vector connecting a polyhedral node to the prescribed position at every step of the process. More complex goals like a desired edge length, or a desired face area can be decomposed into individual vertex movements that, in turn, can be expressed as vector acting on polyhedral nodes.

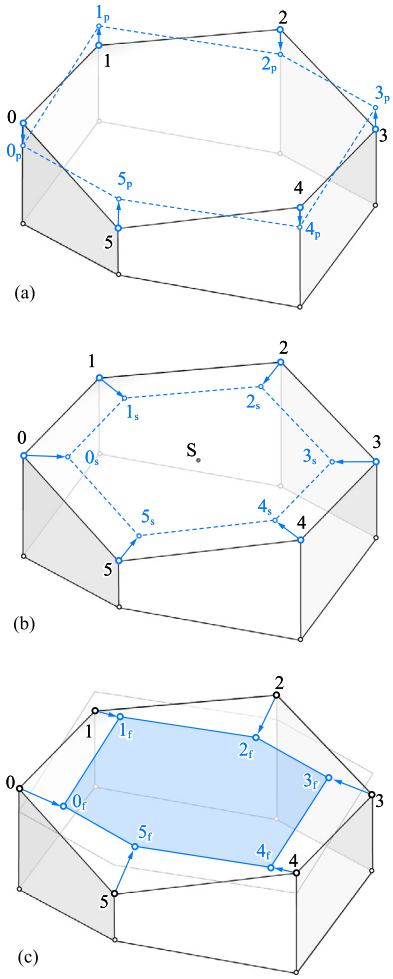


Fig. 7. Example of vector decomposition for two separate constraints: face planarization and scale to area.

Fig. 7 shows two different constraints decomposed as vectors at vertex level, planarity in Fig. 7a, and face area Fig. 7b. Face $f_{(0,1,2,3,4,5)}$ is not planar. During the iterative process, at each step, the best fitting plane is computed for the face and the vertices are projected on that plane. Fig. 7a illustrates this part of the process. In the Figure, the projected vertex positions are in blue with *p* subscript. For each vertex the vector denoting the position transformation associated with the planarity is shown as a blue arrow in the same Figure.

This vector is added to the vertex position. For the second constraints, in order to achieve a certain area, the face is scaled based on its centroid *S*. The operation is depicted in Fig. 7b. This translates in moving vertices along the vertex-centroid axis. The points denoted with *s* subscript show the vertex position that resolves this particular constraint. The blue arrows in the same Figure show the vectors that describe the transformation associated with the area scale.

After each constraint is computed the average of the vector actions for each vertex is added to the vertex position thus moving the vertex to a new position, closer to satisfying the constraints. The transformed version of the face is presented in blue in Fig. 7c as face $f_{(0_t,1_t,2_t,3_t,4_t,5_t)}$ and is defined by the final positions of the vertices after a number of iterations.

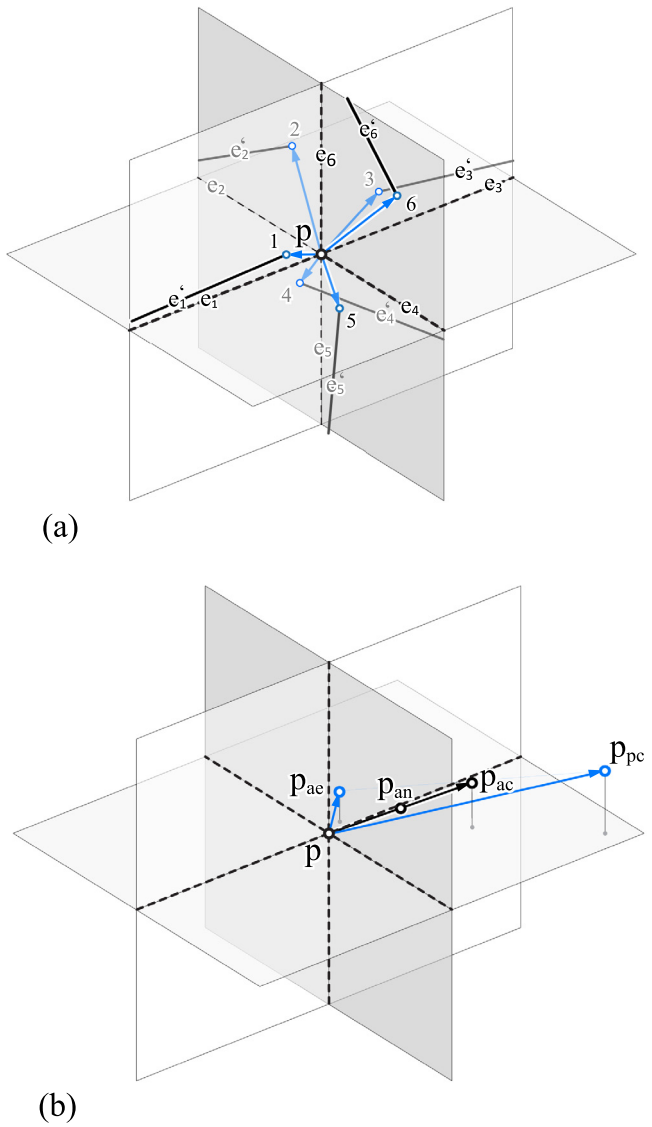


Fig. 8. Weighted average of concurrent constraints (perpendicularization of 6 edges and a target position) on one vertex during one iteration and damping function. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.7. Implementation

The algorithm operates on a simplified graph representation of the polyhedral diagram presented in Section 2.4 and Fig. 5. The steps of the routine are presented in algorithm 1. Besides the Γ graph input, the algorithm requires as inputs the constraints applicable to individual vertices t_{v_i}, edges $t_{e_{(i,j)}}$ and faces $t_{f_{(i,j,k..)}}$. Each constraint needs to be accompanied by a multiplication factor denoted m_x. The subscript x next to the multiplication factor links it with the corresponding constraint. If they are linked the multiplication factor and the constraint will have the same subscript notation. The constraints are functions stored as variables in sets connected to each graph element. All constraints need to be defined prior to the transformation but they can adapt to the current state of the graph (polyhedral diagram) at each iteration. s_{max} defines the maximum number of iterations for the main loop and δ_{min} sets the minimum distance for vertex translation in order to consider the vertex as moved. This value is useful to determine if the graph changes from one iteration to the next.

The algorithm has one main while loop with two parts: Part 1 (lines 3–25) deals with computing individual constraints for each vertex and part 2 (lines 26–36) averages individual transformations into resultant positions for each vertex at each step of the iteration.

Part 1. The routine works by computing each constraint function for vertices, edges and faces if they are present in the input. Each constraint function produces a vector denoting a vertex translation. V_v is a dictionary storing vector sets for each vertex and V_f is a similar dictionary storing scale factors for each vector in the vector set. Lines 7–11 process and store individual vertex constraints. Edges constraints are processed in lines 12 to 18. Here the constraint function produces individual vectors for each edge-end vertex. The vectors are stored in V_v while the unique scaling factor for the edge is stored for each vertex in V_f. Face constraints are processed between lines 19–25 with a similar approach to the edges but for each individual face vertex.

Part 2. The second part of the loop computes the weighted average of the set of vectors for each vertex using the stored scale factors. The damping function is enforced next between lines 28–30. If the magnitude of the weighted average vector $\hat{v_t}$ is larger than θ the vector is scaled down to θ.

Next, between lines 31–33 the maximum magnitude per step recorded so far δ becomes the current average vector magnitude $\hat{v_t}$ if this is larger than its current value. The while loop restarts if maximum number of steps is not reached $s < s_{max}$ and maximum vertex movement per iteration is larger than an imposed minimum $\delta > \delta_{min}$.

Fig. 8 illustrates how the algorithm works on a single node part of a larger polyhedral diagram during one iteration of the iterative transformation of polyhedral diagrams (ITPD) algorithm. In the Figure, node p is the common node for a number of 6 edges e_1 to e_6. In Fig. 8a each edge connected to p is rotated due to perpendicularization, thus producing 6 divergent translation vectors anchored in p and pointing towards different positions in space marked 1 to 6 and shown as blue arrows. For the sake of simplicity the other end of the edges is not shown and neither is the center of rotation for each edge. In the second step shown in Fig. 8b the vectors produced by each edge are averaged into a single translation vector denoted p_{ae}. This vector is in turn averaged with a fixed position for vertex p stipulated by the user and denoted with p_{pc}. The positional constraint represented by p_{pc} requires p to move as close as possible to a fixed position in space. The average between p_{ae} and p_{pc} produces the median result of the imposed constraints for vertex p. This is shown by notation p_{ac} in the same Figure. This image showcases also the damping function that reduces the magnitude of the constraint average to the maximum allowed value (θ in the algorithm) and produces the final position for vertex p in p_{an}.

3.8. Initial benchmark

Even though the method has been in use through a work-in-progress version of the Rhino extension Polyframe [54] for a more than a year at the time of writing this paper, no in depth benchmark of its speed and capabilities was undertaken so far. In the following paragraphs we will present a number of initial speed and capabilities tests for the application and the embedded algorithms. For this initial benchmark we have included two different tests. The first is a test (Fig. 9) where a number of three design examples with increasing complexity, taken from our work at PSL are used. The second is an exponential stress test based on a polyhedral module with a construction method detailed in Fig. 12. The resulting unit, shown in Fig. 11a and b, is repeated in space along in the x, y and z directions in

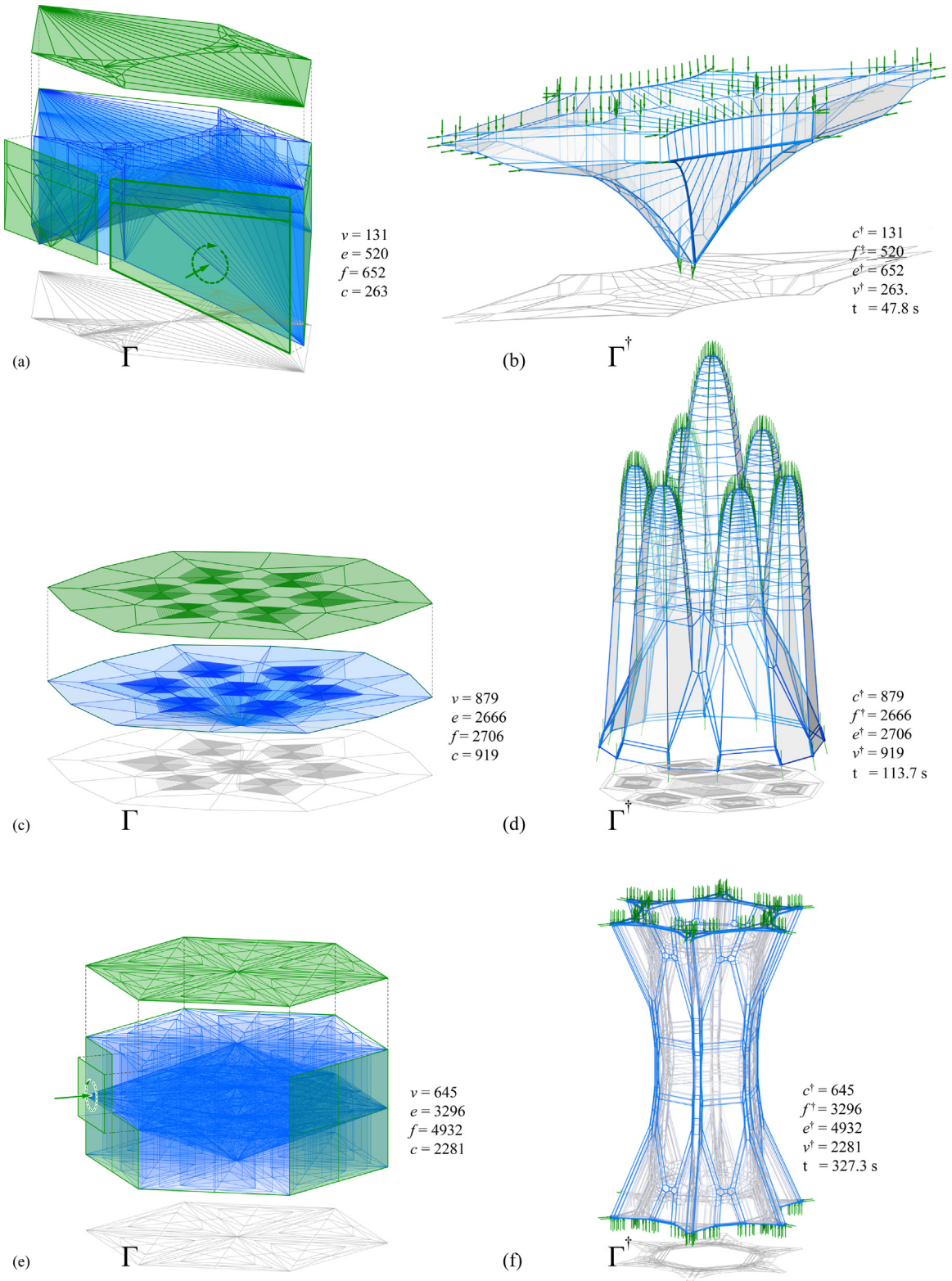


Fig. 9. Reciprocal force and form diagrams form-found using the proposed algorithm.

multiple steps, doubling the number of modules in each direction at every step. Fig. 11c and d shows how a first multiplication creates diagrams with an exponentially increased complexity. Two Tables detail the complexity of the diagrams based on the number of vertices, edges, faces and cells and show perpendicularization time in seconds for each benchmark test. In each case the reciprocity between the dual diagrams (form and force) is achieved up to a maximum deviation of 1 degree. Table 2 for the design examples and Table 3 for the repeated modules test. The data from the Tables is visualized in the graph shown in Fig. 10.

The graph shows that for both tests the complexity of the form diagrams increases from first to last and the same is true for the perpendicularization time. The rate of growth however, is

Algorithm 1: Constraint based iterative transformation of polyhedral diagrams.

Data:

1. $\gamma = (V, E, F)$ graph of the polyhedral diagram where $v_i \in V$, $e_{(i,j)} \in E$ and $f_{(i,j,k..)}$ in F;
2. T_v, T_e, T_f sets of constraint functions where $t_{v_i} \in T_v$, $t_{e_{(i,j)}} \in T_e$, $t_{f_{(i,j,k..)}} \in T_f$ constraint functions for v_i, $e_{(i,j)}$, $f_{(i,j,k..)}$ and m_{v_i}, $m_{e_{(i,j)}}$, $m_{f_{(i,j,k..)}}$ multiplication factors for each function;
3. s_{max} maximum iterations; δ_{min} minimum translation; θ maximum vertex movement limit;

Result: $\Gamma_t = (V, E, F)$ graph with vertex positions closest to satisfying all constraints

begin
  $s \longleftarrow 0$
  while $\delta > \delta_{min}$ & $s < s_{max}$ **do**
    $\delta \longleftarrow 0$
    V_v # store (vertex: vector set)
    V_f # store (vertex: scale set)
    for $m_{t_i} \in T_v$ **do**
      $\hat{v_i} \longleftarrow t_{v_i}$ # compute vector from constraint
      $V_v[v_i] \longleftarrow \hat{v_i}$
      $V_f[v_i] \longleftarrow m_{v_i}$
    end
    for $t_{e_{i,j}} \in T_e$ **do**
      $\hat{v_i}, \hat{v_j} \longleftarrow t_{e_{i,j}}$ # compute end vertex vectors from constraint
      $V_v[v_i] \longleftarrow \hat{v_i}$
      $V_f[v_i] \longleftarrow m_{e_{(i,j)}}$
      $V_v[v_j] \longleftarrow \hat{v_j}$
      $V_f[v_j] \longleftarrow m_{e_{(i,j)}}$
    end
    for $t_{e_{(i,j,k..)}} \in T_f$ **do**
      $\hat{v_i}, \hat{v_j}, \hat{v_k}, \ldots \longleftarrow t_{e_{(i,j,k..)}}$ # compute corner vertex vectors from constraint
      for $x \in (i, j, k, \ldots)$ **do**
        $V_v[v_x] \longleftarrow \hat{v_x}$
        $V_f[v_x] \longleftarrow m_{f_{(i,j,k..)}}$
      end
    end
    for $v_i \in V$ **do**
      $\hat{v_t} \longleftarrow \frac{\sum_{x=0}^{n}(V_v[v_i][x] * V_f[v_i][x])}{\sum_{x=0}^{n} V_f[v_i][x]}$
      if $|\hat{v_t}| > \theta$ **then**
        $\hat{v_t} \longleftarrow \hat{v_t} * \frac{\theta}{|\hat{v_t}|}$
      end
      if $\delta < |\hat{v_t}|$ **then**
        $\delta \longleftarrow |\hat{v_t}|$
      end
    end
    $s \longleftarrow s + 1$
  end
end

different. For the design examples test the complexity increases somewhat linearly but the time grows exponentially. For the repeated modules test the rates of growth are reversed. The complexity grows exponentially but the increase in the perpendicularization time is only linear.

Beyond testing the capacity and the limits of our proposed 3DGS framework the benchmarks show that the topology of

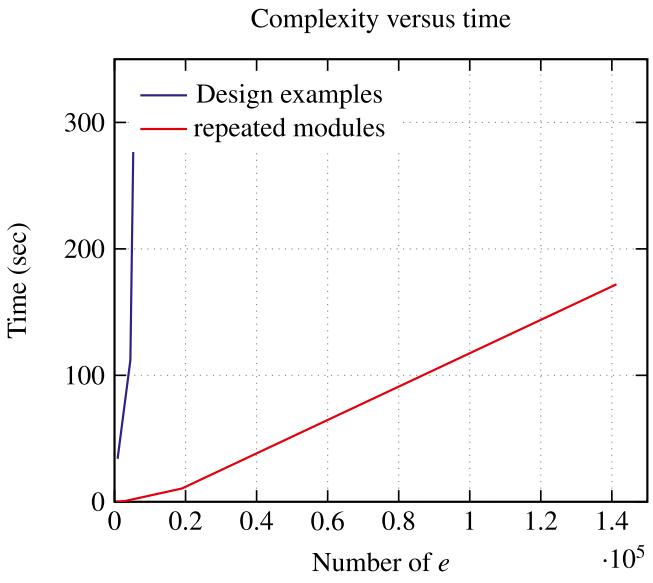


Fig. 10. The change in perpedicularization time as a factor of diagram complexity expressed by the number of edges of the form diagram.

Table 2
Table showing the complexity and perpendicularization times for the example models in Fig. 9.

Fig.	Model	$v^{(\dagger)}$	$e^{(\dagger)}$	$f^{(\dagger)}$	$c^{(\dagger)}$	Time (s)
9a	Bridge force	131	520	652	263	
9b	Bridge form (†)	263	652	520	131	34
9c	Shell force	879	2666	2706	919	
9d	Shell form (†)	919	2706	2666	879	113.7
9e	Tube force	645	3296	4932	2281	
9f	Tube form (†)	2281	4932	3296	645	327.3

Table 3
Table showing the complexity and perpendicularization times for different aggregations of the benchmark model presented in Figs. 11 and 12.

Fig.	Model	$v^{(\dagger)}$	$e^{(\dagger)}$	$f^{(\dagger)}$	$c^{(\dagger)}$	Time (s)
11a	$1 \times M$	43	210	296	129	
11b	$1 \times M^{\dagger}$	129	296	210	43	0.13
11c	$2^3 \times M$	221	2404	2208	1025	
11d	$2^3 \times M^{\dagger}$	1025	2208	2404	221	0.44
–	$4^3 \times M$	1369	10 200	17 024	8193	
–	$4^3 \times M^{\dagger}$	8193	17 024	10 200	1369	10.5
–	$8^3 \times M$	9521	77 616	133 632	65 537	
–	$8^3 \times M^{\dagger}$	65 537	133 632	77 616	9521	172

the diagrams has a far greater impact on the time required to achieve reciprocity than the number of edges of the form diagram. Furthermore, we can see that for repeating force patterns the framework scales remarkably well being able to handle 8-fold increases in the number of elements without a similar increase in time. It is also worth noting that for the repeating modules test benchmark, the number of steps was the same (15) for all diagrams. In order to provide context for our benchmarking we have attempted to test the examples presented in Tables 2 and 3 using the previous framework presented in [31] and implemented as a series of python scripts in the Rhino environment. The framework was not able to handle the test examples presented in Fig. 9 and it crashed so a perpendicularization times could not be extracted. For the benchmark model presented in Table 3 the previous framework was able to successfully perpendicularize the first model $1 \times M^{\dagger}$ in 12.9 s and the second model $2^3 \times M^{\dagger}$ in 76.87 s. For the subsequent models starting with $4^3 \times M^{\dagger}$ the creation of dual diagrams was not possible as the framework could not handle the large number of input faces

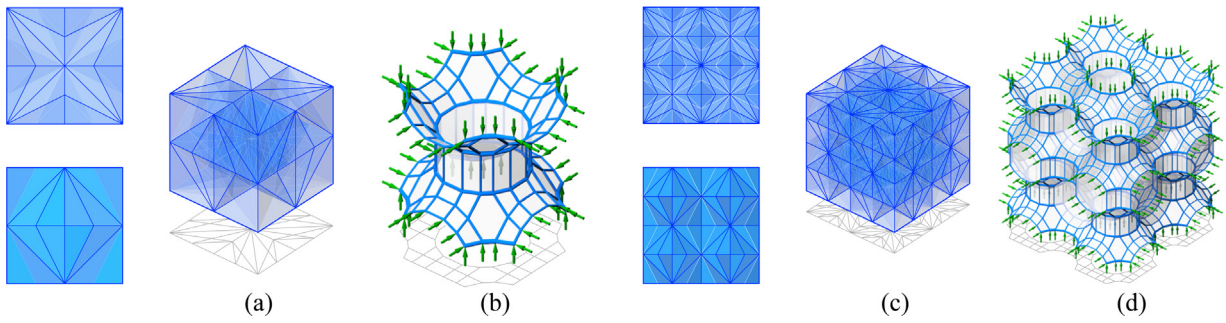


Fig. 11. (a) A unit force module ($M^{\dagger}$) and its reciprocal form (M) and (b) used as a benchmark repeated eight times, two times in $-x$, $-y$, and $-z$ directions ($2^3 \times M^{\dagger}$).

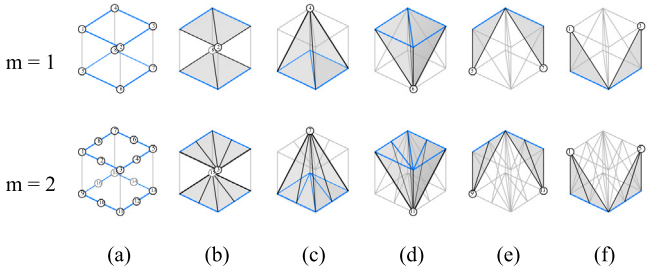


Fig. 12. (a) to (f) illustrates the development process of a quarter of the benchmark force polyhedron and its construction process using the edges of a cube divided by m.

The same testing was attempted with the 3D Graphics Statics for Grasshopper tool by Graovac [51]. The tool was able to perpedincularize the benchmark models in similar times when it could load the model. Exact times for perpendicularization cannot be presented as the tool does not present the total compute time for a diagram. Furthermore since the tool runs using both internal cycles and Grasshopper cycles, an exact run time could not be measured. The only exact metric we could compare was the number of iterations which was 49 for the first three models $1 \times M^{\dagger}$, $2^3 \times M^{\dagger}$ and $4^3 \times M^{\dagger}$ compared with 15 iteration for PolyFrame. $8^3 \times M^{\dagger}$ could not be obtained as the perpendicularization process froze the application. 3D Graphics Statics for Grasshopper requires pre-processed data in order to construct any primal diagram. The geometry needs to be provided as ready-made polyhedral cells in order to use the tool. This makes the tool cumbersome to use and less robust as the user needs to take care of finding the polyhedral cells and needs to make sure no errors (like disjoint or overlapped cells) are present. For very large diagrams with several hundreds of cells this is very impractical.

For reference all computations have been performed single threaded on a laptop with a Core i7 8750H CPU and 32 GB of RAM.

4. Parallel transformations of polyhedral diagrams

The parallel transformation of a polyhedral diagram or PTPD is an operation that changes the geometry of a polyhedral diagram while keeping the direction of its edges intact. The geometric exploration of reciprocity or GER is the use of PTPD in order to transform a polyhedral diagram while conserving the reciprocity to its dual.

The method of PTPD is a sequence of geometrical transformations that reconstruct the geometry of the polyhedral diagram starting from a sub-part modified by the user. The transformations are governed by a set of rules that dictate how and in what order the remainder of the polyhedral diagram is reconstructed to keep its edge directions unchanged. The initial exploration of this method was presented in [60]. A different implementation of a similar technique is presented in [48] pp. 92–93 and an implementation is provided in [50] In this section we will provide a more in-depth explanation of our method and we will present the algorithm and its implementation.

PTPD is especially useful for form diagrams where changes can be operated in order to adapt the diagram to transformed boundary conditions. The method allows for the change of edge lengths (whether internal or external) or even for the transformation of some edges from compression to tension members, within the limits of reciprocity with the same force diagram. The last feature needs further exploration and it is not in the scope of the present paper that deals with compression only polyhedral diagrams. PTPD is a technique universally applicable to clusters of polyhedral cells. However, in this paper, beyond explaining the method, we focus only on a narrow subset of its applications in the context of graphic statics compression-only polyhedral form diagrams. The presented examples are meant only to demonstrate the technique. A more in-depth investigation of PTPD in the context of graphic statics and a broader survey of its capabilities and limitations (including self-intersecting faces, force diagrams, and the impact of triangular faces) will be presented in a forthcoming publication. Additional use case examples within the application limits presented in the paper can be viewed in [61].

The introduced method allows the user to take one or more vertices, an edge, or a face of a polyhedral diagram and move them in space. The movement is generating transformation solutions for the whole polyhedral diagram, so that the newly transformed edges or faces stay parallel to their original geometry. For instance, in Fig. 13, the moving faces $f_{(0,1,2,3,4,5)}$, $f_{(0,1,10,9,8,7)}$ and $f_{(0,7,6,5)}$ are the result of the translation of vertex v_0 from position 0 to position 0_t. The transformation solution for a whole polyhedral diagram generated by PTPD starting from the user input is in fact the process of finding a transformation path that connects all transformed elements in the polyhedral diagram. The transformation path is a sequential chaining of vertex movements where each movement is determined by a specific movement before it and determines one or more movements after it. This chain of determinations allows for PTPD to be described as a tree graph spanning the base graph of edge interconnected vertices in the polyhedral diagram.

In the following sections, we will expand on how PTPD works and what are the methods and rules for the construction of the transformation path.

4.1. Single versus multiple vertex inputs

We can consider the root of this tree graph, the first moved vertex in the polyhedral cluster. If the transformation originates

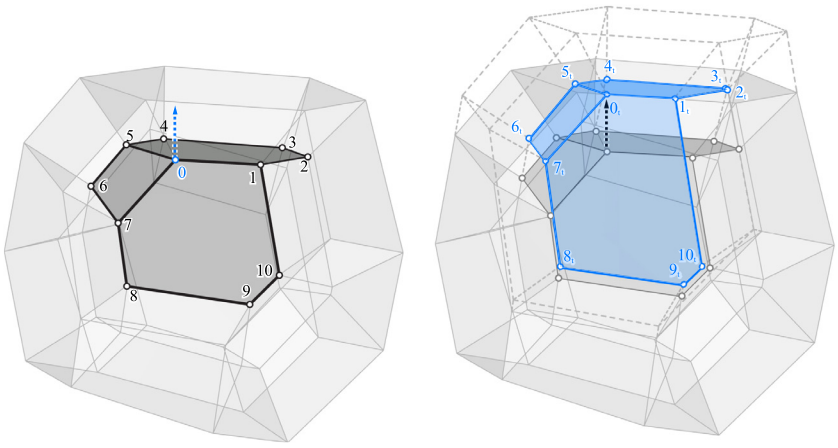


Fig. 13. Parallel transformation of polyhedral diagrams (PTPD).

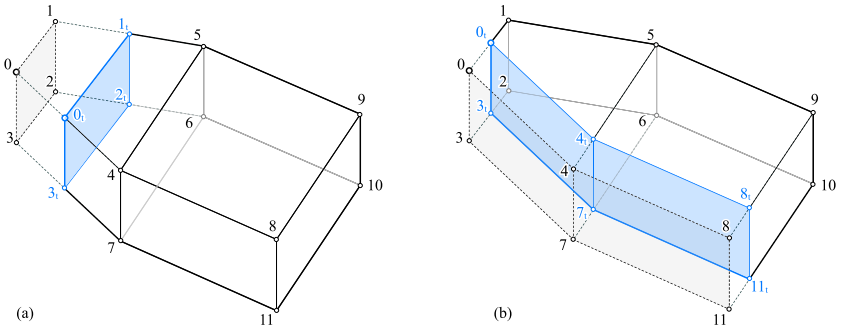


Fig. 14. Single versus multiple point or face input for PTPD.

in multiple vertices, which is also the case when edges or faces are moved by the user, any vertex from the moved set can be considered the root. If we are transforming the polyhedral diagram based on the translation of a single vertex, that vertex, the root of the tree can be placed anywhere.

The direction of the translation together with the original topology of the diagram will dictate what other elements from the diagram will be reconstructed. The magnitude of the translation together with the original geometry (orientation of edges compared to each other) and topology (complexity of vertex interconnection) will determine if some edges and faces will flip their direction.

Fig. 14 showcases this using a very simple example involving only two polyhedral cells, a cube and a prism. This is an important mechanism to understand because it underlines all aspects of PTPD and because it is the base of regarding multiple vertex input transformations as single input vertex transformations with added constraints for the direction and magnitude of the initial vertex translation. Fig. 14a shows how moving vertex v_0 along edge $e_{(0,4)}$ is the equivalent of moving face $f_{(0,1,2,3)}$ in the direction of its normal. But, if we consider further the movement of face $f_{(0,1,2,3)}$, we realize that if we want to move any of the vertices v_1, v_2 or v_3 along vectors that are not parallel with the edges $e_{(1,5)}$, $e_{(2,6)}$ or $e_{(3,7)}$ or with magnitudes different from that of vector $|0, 0_t|$, we end up with positions for them that are incompatible with the base requirement of PTPD, constant orientation for all edges and faces in the diagram. With this, we can state that using any continuous part of a polyhedral diagram (like a face or a set of edges) as the start of the transformation, is the equivalent of moving any of the vertices contained in that part, with specific constraints for direction and amplitude and running the transformation algorithm.

To illustrate this, we can look at Fig. 14b where the same vertex v_0 is moved, this time along the edge $e_{(0,1)}$ from position 0 to position 0_t. The movement is the equivalent of simultaneously moving faces $f_{(0,3,7,4)}$ and $f_{(4,8,11,7)}$ while keeping their normal direction constant. Thus we can regard the movement of a face, an edge or more generally of certain sets of interconnected vertices as a particular case of the more general free translation of a single vertex of the polyhedral diagram.

The scope of this part of the paper is to discuss the mechanisms of this general latter case.

4.2. The main steps of PTPD

According to the user input, the root vertex is moved to a new location and from it the transformation is followed outwards, in steps, from vertex to vertex via the edges until all vertices in the graph have been parsed. After each vertex is processed, a new one is picked from the un-parsed group from the ones that conform to the rules of **propagation** and **prioritization**. We will explain and exemplify those rules in the following sections.

Multiple transformation paths exist for any given input on the same polyhedral diagram. For any un-directed graph there are multiple spanning trees starting from the same root [62]. In our case, since at certain steps in the transformation, there are multiple available vertices that fit the rules mentioned above, there are multiple transformation paths for any given root. Some of those transformation paths, depending on the choices made along the way, can lead to different transformation results. It is also true that there are cases when path altering choices lead to the same transformed result. This aspect of PTPD will be the focus of a forthcoming study and publication.

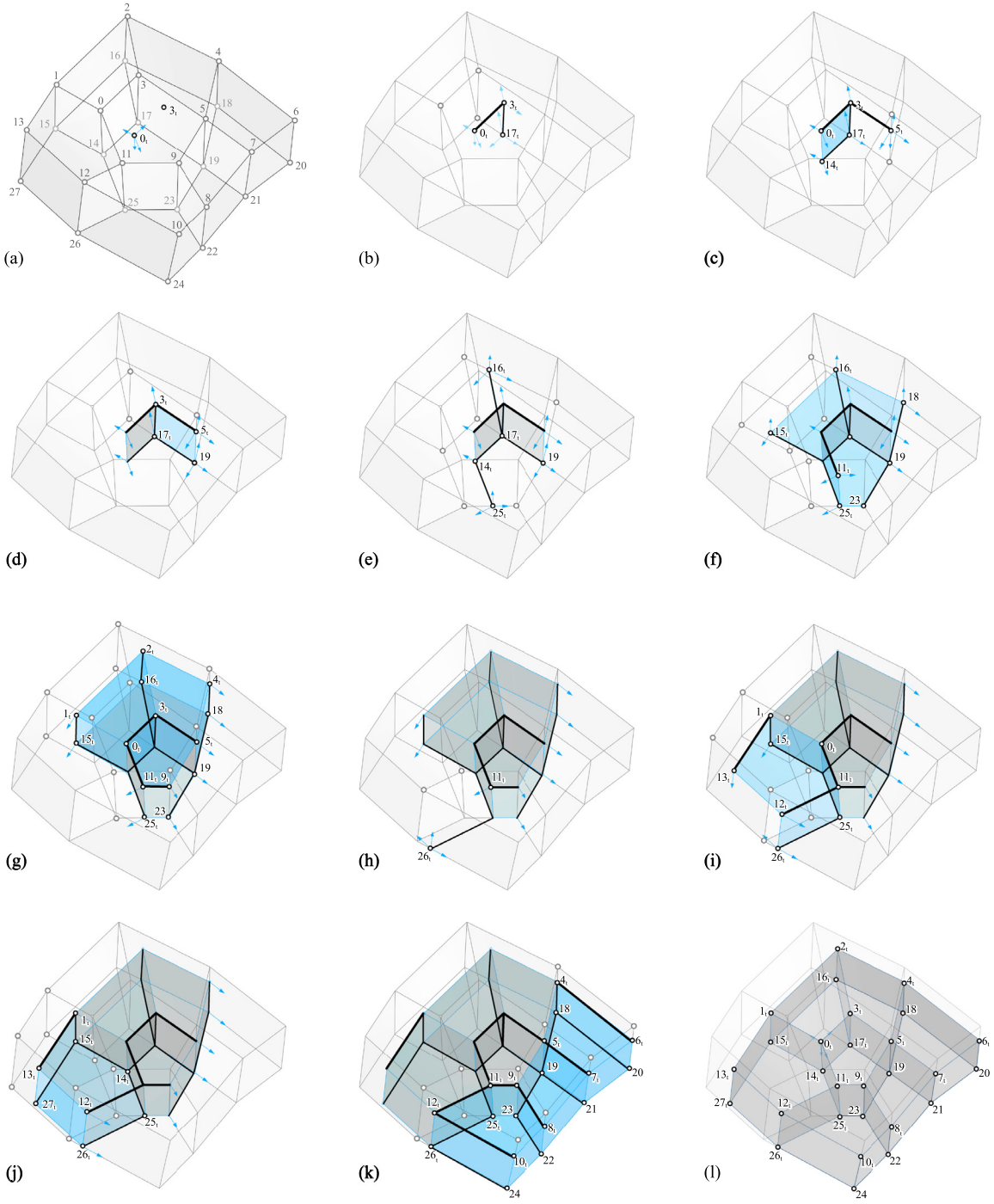


Fig. 15. A typical example for a simple parallel geometric manipulation. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

In graph theory terms the process described above is similar to a *Breadth First Search* or BFS, with customized rules that prioritize the available graph edges at every step.

4.3. The transformation rules and a simple example

In order to explain the main mechanisms of PTPD we will use the example depicted in Fig. 15a through l.

The figure shows the step by step process of transformation for a typical simple polyhedral diagram consisting of a cluster of 7 cells. The cells are grouped in one single layer for reasons pertaining to process visibility. The gray line background plus the gray points and gray notations, refer to the original geometry of the polyhedral cluster. The black overlay, including the black dots and the black notations with underscore, refers to the transformed geometry. The blue fill tracks the geometry completed in the current step. The gray fill refers to the fully transformed faces completed in the previous steps. At each step the blue arrows show the available expansion choices for the transformation path. The thick black line overlay depicts the transformation path walked so far by the process. The thick black lines display at every step the extent of the spanning tree constructed on top of the base graph of the polyhedral diagram.

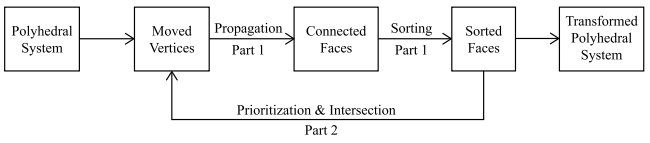


Fig. 16. Algorithm loop during PTPD.

Before proceeding with the example we must briefly explain two key concepts (transformation propagation and transformation prioritization) of the process and define a few supporting notions. We will refer back to those concepts and notions as we make our way through the example.

Transformation propagation and *transformation prioritization* are in fact two sets of rules. The rules ensure that regardless of the polyhedral diagram, the starting vertex and the ordering choices made in spanning the transformation path, the resulting transformed edges will always stay parallel to their original direction. The first rule set addresses how graph edges are selected for the expansion of the spanning tree and the second addresses how they are categorized and prioritized to be included into the tree.

Transformation propagation uses the connectivity of the original polyhedron topology to extend the transformation from the end vertices of the transformation path to all the polyhedral edges that connect to them, and from there, to all the faces that contain those edges. By attempting to fully determine those transformed edges, other yet un-transformed vertices are reconstructed and thus the transformation propagates further. Moved vertices create *partial edges* (directions anchored in space or rays) and those edges create *partial faces* (partially bounded planes in 3d space). Using ray-ray or ray-plane intersections and a strict set of priorities, the missing vertices from the partial edges are determined step by step. The creation of those new vertices produces complete (fully determined) edges and faces and produces new partial edges and partial faces. This is the mechanism that is propagating the transformation into the polyhedral graph. Fig. 16 offers a high level overview of the looping mechanism that propagates the change in the polyhedral diagram.

Transformation prioritization deals with the strict set of rules involved in the intersection operations that yield new positions for transformed vertices. The concept essentially dictates the order in which the intersections available at some point in the transformation can be processed. Three types of intersections are identified, each with its own priority. *In-face active* intersections refer to line-line intersections between two partial edges. *In-face passive* intersections describe intersections between one partial edge and one original (not transformed) edge. *Out-of-face* intersections are intersections between a partial edge and a face containing the vertex that has yet to be determined. The order of processing is: in-face intersections, first active and then passive and finally out-of-face intersections. This ensures that all transformed edges stay parallel to their original direction regardless of when in the process their vertex positions are determined.

In the example, the transformation starts with the translation of vertex v_0 from 0 to 0_t we will call the transformed vertex v_{0_t} (see Fig. 15a). This is the root of the transformation path. Since the point at 0_t is placed inside of a cell and not in any particular position on a face-plane or along an edge, the only available intersections for the first step are out-of-face intersections. The first out-of-face intersection is computed by intersecting partial edge $e_{(0_t,3_t)}$ with the face plane of $f_{(3,17,19,5)}$ to get the location for the translation of v_3 that is point 3_t.

In the next step, pictured in Fig. 15b, the transformation of v_3 in v_{3_t} propagates to all edges topologically connected to v_3. From all the potential intersections added, the ones towards v_5 and v_{17} can be classified as in-face passive intersections. In the same Fig. 15b transformed vertex v_{17_t} has been added to the transformation path.

In the next step we have an active in-face intersection. The intersection occurs between two partial edges pointing towards a new position for original vertex v_{14}. Both edges are in the face-plane of partial face $f_{(0_t,3_t,17_t)}$. Since this is the only active in-face intersection in the wait-list at the moment, we can operate it and thus get the position for v_{14_t}. In the same Fig. 15c another in-face-passive intersection yields v_{5_t}.

In the next images similar operations are shown and the full transformation of the diagram can be followed using the highlighted transformation path in thick black line. As the process progresses, the number of steps shown in each image increases but the principles stated above remain the same.

The final steps of the transformation are shown in Fig. 15k. Here, the last vertices are computed and if necessary translated. Fig. 15l depicts the final form of the polyhedral diagram after all its vertices have been parsed.

4.3.1. PTPD implementation

The next paragraph presents a description of the actual algorithm for PTPD and a pseudo-code implementation. The algorithm is written as a modified Breadth First Search (BFS), with multiple inner loops and an expansion behavior based on the transformation prioritization rules described above. A high level overview of the typical workflow is presented in Fig. 16. The essential steps of the routine are presented in Algorithm 2.

The algorithm works on a graph representation of the polyhedral diagram $\Gamma = (V, E, F)$ with interconnected vertices, edges and faces. The other significant input is V^t a set of already transformed vertices that will determine the transformation in the graph. The set of pre-translated vertices can also represent an edge or a face in the polyhedral diagram. The algorithm is a large while loop containing two functional sections: Section 1 (lines 5 to 23) deals with gathering and sorting partially transformed faces from transformed vertices and Section 2 (lines 24 to 42) with processing each partially transformed face through appropriate edge intersections to yield more transformed vertices. Special notations: Superscript t denotes a transformed vertex, or a partially transformed edge or face. Subscript t used on the subscript notation of an edge or face, like $e^t_{(i_t,j)}$ denotes that a particular vertex in the edge or face is transformed.

The first section works by establishing all the connected faces to the transformed vertices in V^t. Those faces are all stored in F^t. Subsequently all faces in F^t are parsed and based on the internal configuration of un/transformed vertices and edges the appropriate collection is chosen for the face: F^a active in-face intersections, F^p passive in-face intersections, F^o out-of-face intersection or F^c for the fully transformed faces with no available intersection. The choice is made according to the rules of transformation prioritization presented in the previous paragraph.

After the first section, a check is performed to see if faces with intersections are still available to be processed in any of the specialized collections. If that is not the case the Boolean variable controlling the main while loop is set to false. The second section contains specialized parts for working with each of the three collections of transformable faces. The processing order is the actual transformation prioritization.

First, the active intersection faces are processed (line 24–31). Since active intersections have the highest priority, all faces in the collections can be parsed. If the face has an active intersection then it should contain two partially transformed consecutive edges $e^t_{(i_t,j)}$ and $e^t_{(j,k_t)}$ where v_j is the yet un-transformed common vertex. The new position for v_j is found through the intersection of the two edges. The resulting transformed vertex is added to V^t.

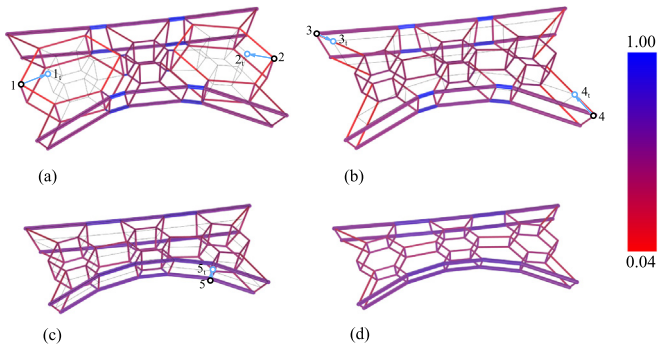


Fig. 17. PTPD example of a transformation of a form diagram.

Once all active intersection faces are processed F^a is emptied and the rest of the while loop is skipped. This makes sure that if v_j^t

Second, passive in-face intersection faces are processed (lines 32–37). This happens only if there are not any active intersection faces left. Only one intersection is computed and the current while cycle is skipped. This accounts for any new active intersection faces that might appear. A passive intersection is computed between a partially transformed edge $e^t_{(i_t,j)}$ and an untransformed one $e_{(j,k)}$ sharing a vertex v_j. The new position for the common vertex is added to V^t.

Out-of-plane intersection faces (lines 38–42) are processed last in this section and just like passive in-face intersections, at most one intersection is computed per while loop. The out-of-plane intersection requires one partially transformed edge $e_{(i_t,j)}$. Since no sufficient data exists in the face to compute the other vertex of the edge, v_j, the intersection is computed between the edge and another face containing v_j in the graph G. The resulting transformed vertex v_j^t is included in V^t. The loop ends when all the faces connected to vertices in V^t are processed and the newly transformed vertices already belong to completed faces thus producing no other partially transformed faces.

Another example of PTPD used on a bridge-like form diagram, where it is transformed using parallel transformation tool in several steps as shown in Fig. 17a-d. In Fig. 17a points 1 and 2 are moved to the transformed positions 1_t and 2_t. Based on the mentioned rules of PTPD (see Section 4.2), the target geometry including the transformed diagram is visualized in light gray before the completion of the operation as shown in Fig. 17a,b.

Similarly in Figs. 17b and c, the movement of points 3, 4 and 5 further transforms the diagram in to its final configuration of Fig. 17d. One of the advantages of using this method is the direct sculpting the polyhedral forms for design purposes. Besides, this parallel transformation can be paired with some analytical approaches to inform the user of certain performative aspects of the changes made into the form by using this method. For instance, the buckling capacity of the members may change and can be tuned using this function.

The colors of the members in all instances correspond to the relative buckling capacity of the members calculated using Euler's critical load formula for a filled circular cross section [63]. The cross sections are sized according to the area of their corresponding faces in the force diagram. The values for all configurations are normalized ($\leq$1). As presented in the figure change in the geometry of the form affects its buckling performance. The user can manually change the geometry to adjust long members to sculpt the diagram while observing the buckling performance of the system. This property can be paired with volume calculation for load-path optimization of the structural forms.

Algorithm 2: Geometric graph transformation

Data: $\Gamma = (V, E, F)$ graph of the polyhedral diagram where $v_i \in V$, $e_{i,j} \in E$ and $f_{i,j,k..} \in F$; V^t a set of translated vertices; F^a, F^p, F^o, F^c empty sets of active, passive, out-of-face and complete faces

Result: $\Gamma^c = (V^c, E^c, F^c)$ completely transformed graph

1 **begin**
2 **while** $|F^a|+|F^c|+|F^o|>0$ **do**
3 F^t # store transformed faces
4 **for** $v_i^t \in V^t$ **do**
5 $F^t \longleftarrow F^t_{v_i}$ # set of faces connected to v_i
6 **end**
7 **for** $f^t \in F^t$ **do**
8 **if** $\exists\ v_x \in f^t$ # *f has transformable v* **then**
9 **if** $v_i^t \in P_f$ # v_i^t *in original face plane* P_f
10 **then**
11 **if** $\exists\ e^t_{(i_t,j)}$ & $\exists\ e^t_{(k_t,j)} \in f^t_{(i_t,j,k_t..)}$ **then**
12 $F^a \longleftarrow f^t_{(i_t,j,k_t..)}$
13 **end**
14 **if** $\exists\ e^t_{(i_t,j)}$ & $\exists\ e_{k,j} \in f^t_{(i_t,j,k..)}$ **then**
15 $F^p \longleftarrow f^t_{(i_t,j,k..)}$
16 **end**
17 **else**
18 $F^o \longleftarrow f^t_{(i_t,j,k..)}$
19 **end**
20 **else**
21 $F^c \longleftarrow f^t_{(i_t,j_t,k_t..)}$
22 **end**
23 **end**
24 **if** $|F^a|>0$ **then**
25 **for** $f^t_{(i_t,j,k_t..)} \in F^a$ **do**
26 $e^t_{(i_t,j)} \in f^t_{(i_t,j,k_t..)}$ & $e^t_{(k_t,j)} \in f^t_{(i_t,j,k_t..)}$
27 $V^t \longleftarrow e^t_{(i_t,j)} \cap e^t_{(k_t,j)}$
28 **end**
29 $F^a \longleftarrow \varnothing$ # empty F^a
30 continue # skip rest of loop
31 **end**
32 **if** $|F^p|>0$ **then**
33 $f^t_{(i_t,j,k..)} \longleftarrow F^p[0]$ # extract from set
34 $e^t_{(i_t,j)} \in f^t_{(i_t,j,k..)}$ & $e_{(k,j)} \in f^t_{(i_t,j,k..)}$
35 $V^t \longleftarrow e^t_{(i_t,j)} \cap e_{(k,j)}$
36 continue # skip rest of loop
37 **end**
38 **if** $|F^o|>0$ **then**
39 $f^t_{(i_t,j,k..)} \longleftarrow F^o[0]$ # extract from set
40 $e^t_{(i_t,j)} \in f^t_{(i_t,j,k)}$ & $f_{(j,v,w..)} \in F$
41 $V^t \longleftarrow e^t_{(i_t,j)} \cap f^t_{(j,v,w..)}$
42 **end**
43 **end**
44 **end**

5. Implementation of the algorithms as a Rhino3d plug-in

The algorithms presented in this paper have been implemented as an add-on for the Windows version of the Rhinoceros3d popular CAD platform. The plug-in is called PolyFrame and introduces a number of 7 Rhino commands that cover the workflows described in Fig. 3 and in Fig. 16. Fig. 18 shows the toolbar that our plug-in introduces in the Rhino work space. For a more in-depth description of the available

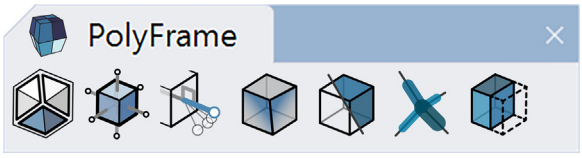


Fig. 18. An image of the toolbar introduced by PolyFrame in the workspace of Rhino.

commands, options and workflows, please refer to the PolyFrame Manual [64]. The PolyFrame plug-in contains the PolyFramework API which is the core library of the add-on as a separate DLL file. The API can be referenced in any scripting environment of Rhino or in its visual scripting environment Grasshopper. A fully-fledged PolyFrame add-on for Grasshopper has not yet been released.

The first four commands implement the iterative workflow for the construction of reciprocal polyhedral diagrams detailed in Section 3.

The first command in the toolbar is *PFBuild*. The command can create a polyhedral diagram (referred to as a *polyframe* in the context of the plug-in) from native Rhino geometry presented as surfaces. The command interprets the raw geometry, removes duplicates, repairs overlaps, disconnects and finally creates the data structure. The data is saved back in the Rhinoceros document together with the cleaned geometry to be used by the other tools. The *polyframe* data is saved with the Rhino geometry.

The second command *PFDual* constructs the dual polyhedral diagram for an existing diagram in the Rhino viewport. The dual is a *polyframe* that has a topological connection to the primal diagram as described in Section 2.6, but it is not reciprocal to it yet.

Reciprocity between dual diagrams can be achieved using the third command in the toolbar *PFPerp*. Through the command-line interface the constraint-based manipulations presented in Section 3.4 can be imposed. The specifics of the algorithms of the command are presented in Section 3.5.

PFPlanar is the fourth tool in the toolbar and it used to enforce face planarity in a polyhedral diagram. Similar to the *PFPerp* it can attempt to planarize the provided diagram while enforcing a number of user specified constraints, introduced through options present in the command-line interface of the tool. The specifics of the algorithms of the command are presented in Section 3.6.

The next two commands *PFSwitch* and *PFPipe* are utility tools used mainly for visualization purposes. *PFSwitch* can change the Rhino geometry used to visualize any polyhedral diagram. Each diagram can be viewed and stored in the Rhino document as a group of edges, face or cells. Faces and cells can be either Rhino surfaces or Rhino meshes. *PFPipe* is a special visualization command dedicated to form diagrams. For each internal edge of the form diagram a pipe is created. The length of the pipe is equal to the edge's length and its width is proportional with the area of the correspondent dual face present in the force diagram.

The last command in the toolbar *PFTransform* implements the algorithm for parallel transformation presented in Section 4. Starting from any polyhedral diagram the user can pick a vertex and move it, thus transforming the diagram according to the rules of PTPD. For now this tool is implemented only as a proof of concept and the full possibilities of the algorithm will be made available in a future release.

6. Limitations and future work

This paper introduced a new and improved framework for the efficient computation of 3D graphic statics based on reciprocal polyhedral diagrams applicable to compression-only spatial structural form finding. The framework is limited to working with planar convex faces. The framework is based on a new data structure capable of handling large diagrams with thousands of cells and tens of thousands of faces and edges. The reciprocal construction algorithms can speed up the constraint-based form finding process compared to the methods introduced by Akbarzadeh et al. [31]. We also compared our implementation with the 3D graphic Statics add-on for Grasshopper by Graovac [51] and found that raw perpendicularization speed is similar. However we the add-on has no cell finder and needs perfect polyhedral input to produce a dual/reciprocal diagram. Additionally the perpendicularization process does not accept any constraints so the tool cannot be employed for any practical use. Comparisons with another solution proposed by Lee [48] were not possible as a working version for compas_3gs was not available on the compas repository at the time of writing this paper.

The robustness and the efficiency of the algorithms provided in this paper extend the ability to explore the limits of reciprocity between polyhedral diagrams with multiple geometric degrees of freedom. The framework is especially useful for scenarios with low geometric degrees of freedom where partial solution can still be found.

Additionally, the paper has introduced a new method of parallel transformation for the polyhedral diagrams without reconstructing the reciprocal diagrams. This method facilitates the transformation of any cluster of polyhedral cells, based on the user input, while preserving the edge directions of the diagram. This allows for almost instantaneous, near real-time transformations of a polyhedral diagram within the limits of its geometric degrees of freedom. As mentioned, the presented framework has been released as an extension for the modeling software Rhinoceros [55] and is called PolyFrame. This plugin can be freely downloaded [54]. Since its release in 2018, the plugin has been used in extensively in research and education [39,40,65–68]. Fig. 19 shows a selection of design research works produced by using this plugin at the Weitzman School of Design, University of Pennsylvania. What is presented in the images exceeds the strict context of the paper and the cantilever structures shown in the models are in fact the section from an original design and should not be mistaken as a compression-only structural form.

Note that in all examples, the external loading scenario was used as a dominant loading condition. Besides, loading on the structure's internal nodes is not possible using the methods explained in this paper. Moreover, some cases contain lateral loading in their design stage, where later in the real structure was removed. This removal of the external loads will result in changing the direction of the internal force in some peripheral members from compression into tension, which is an existing limitation of the current tool. In sum, the main point of presenting these examples is to emphasize the level of articulation a designer may achieve using this paper's proposed tool.

The introduced research and the presented framework are opening new avenues of research into 3DGS through reciprocal polyhedral diagrams. This framework can be extended for form finding with combined compression and tension forces by either including the self-intersecting faces in the geometry of the force and form diagrams or by flipping the direction of the edges in the form to represent tensile members [43,69].

At the moment, no reliable method exists for the determination of a solution space defined by a reciprocal relationship between polyhedral diagrams. As a result it is impossible to determine if an iteratively found reciprocal solution is a global optimal solution. We believe that the research introduced in this paper is a step towards this goal. Section 3.8 delineates the connection between the complexity of the diagram and the time required to find the solution. It also shows that the number of

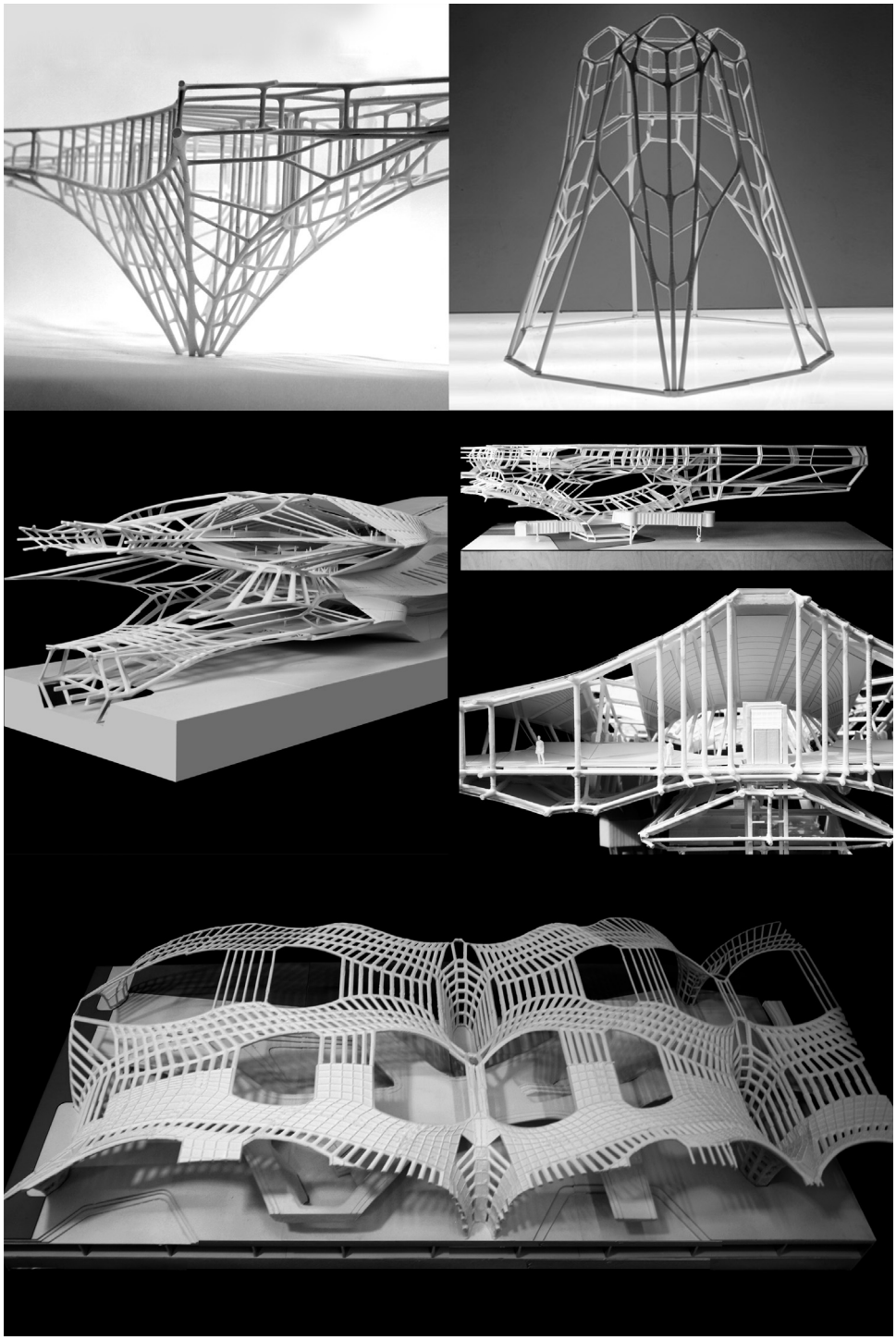


Fig. 19. A collection of architectural structures designed by students. Works developed by using the PolyFrame plugin and the algorithms presented in this paper (Photo credits: PSL). Some of these design exercises show a section of the structure which should not be interpreted as a cantilever system.

elements (edges, faces) in a diagram has less of an impact on its perpendicularization times (time to find the solution) than its type of topological interconnections and specific geometry. In an upcoming research we will investigate this connection that we believe is very well described by the number of Geometric Degrees of Freedom (GDoF) a diagram has.

A more thorough set of tests and a comparison with other software packages that tackle similar tasks such as 3DGS extension presented in [48] and [49], will help to gather relevant data towards this goal.

We speculate that the number of elements in a diagram defines the search space for a reciprocal solution while the GDoF of the diagram is a measure of the solution space where the same reciprocal diagram can be found. Once a better understanding of this relationship is developed, a machine learning algorithm or a genetic optimization algorithm could help predict better solutions with smaller deviations within the limits of any given connected diagrams.

Moreover, further investigation is needed to enhance the user control, particularly for the parallel transformation algorithm.

The parallel transformation algorithm can also be improved by considering multiple transformation paths simultaneously as well as considerations that could result in topological changes in the primal or dual diagram.

Finally a skin generating algorithms based on implicit and explicit modeling will be implemented for the visualization and fabrication of form diagrams with member sections proportional to the face areas in the force diagram.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The funding for the development of this projects was provided by the Weitzman School of Design, University of Pennsylvania, USA. Spacial thanks to Chair Winka Dubbeldam of the architecture department and Dean Frederick Steiner of the Weitzman School. This research is partially funded by the National Science Foundation CAREER Award NSF, USA CAREER-1944691. The works presented in Fig. 19 are produced by Yamba Moise Tshilono, Kim Gwan Sook, Zehua Qi, Qi Liu, Tian Ouyang, Jasmine Gao, Mingxin He, Yangchao Ni, Ali Tabatabaei Ghomi, and Ayodh Kamath in Architectural Design Research Studio taught by Masoud Akbarzadeh at the Weitzman School of Design, University of Pennsylvania.

References

[1] Hooke R. A description of helioscopes, and some other instruments. London: John Martyn; 1675.
[2] Poleni G. Memorie Istoriche della gran cupola del tempio vaticano, Vol. 33. Padova: Nella Stamperia del Seminario; 1748.
[3] Otto F, Rasch B. Finding form: Towards an architecture of the minimal. Edition Axel Menges; 1996.
[4] Collins GR. Antonio gaudi: Structure and form. Perspecta 8: Yale Archit J 1963;63–90.
[5] Addis B. History of using models to inform the design and construction of structures. In *Proceedings of IASS symposium 2004, shell and spatial structures: from models to realization*, Montpelier, France, 2004.
[6] Huerta Santiago. Structural design in the work of gaudi. Archit Sci Rev 2006;49(4):324–39.
[7] Collins George R. Antonio gaudi: structure and form. Perspecta 1963;63–90.
[8] Piker D. Kangaroo physics. 2013, URL http://www.food4rhino.com/project/kangaroo.
[9] Kilian A, Ochsendorf J. Particle-spring systems for structural form finding. J Int Assoc Shell Spat Struct 2005;46(2):77–85.
[10] Kilian A. Cadenary tool v.1 [computer software]. 2004, http://www.designexplorer.net/projectpages/cadenary.html (Accessed March 1, 2009).
[11] Linkwitz K, Schek HJ. Einige bemerkung von vorsgepannten seilnetzkonstruktionen. Ing-Arch 1971;40:145–58.
[12] Schek HJ. The force density method for form finding and computation of general networks. Comput Methods Appl Mech Engrg 1974;3(1):115–34.
[13] Barnes MR. Form finding and analysis of tension structures by dynamic relaxation. Int J Space Struct 1999;14(2):89–104.
[14] Adriaenssens S, Block P, Veenendaal D, Williams C, editors. Shell structures for architecture: Form finding and optimization. London: Taylor & Francis - Routledge; 2014.
[15] Varignon P. Nouvelle mecanique ou statique. Paris: at Claude Jombert; 1725.
[16] Maxwell JC. On reciprocal figures and diagrams of forces. Philos Mag J Ser 1864;4(27):250–61.
[17] Rankine M. Principle of the equilibrium of polyhedral frames. Phil Mag 1864;27(180):92.
[18] Wolfe WS. Graphical analysis: A text book on graphic statics. McGraw-Hill Book Company, Inc.; 1921.
[19] Culmann K. Die graphische statik. Zürich: Verlag Meyer & Zeller; 1864.
[20] Cremona L. Graphical statics: Two treatises on the graphical calculus and reciprocal figures in graphical statics. Oxford: Clarendon Press; 1890.
[21] Allen Edward, Zalewski Waclaw. Form and forces: designing efficient, expressive structures. John Wiley & Sons; 2009.
[22] Van Mele T, Rippmann M, Lachauer L, Block P. Geometry-based understanding of structures. J Int Assoc Shell Spat Struct 2012;53(2).
[23] Block P. Thrust network analysis: Exploring three-dimensional equilibrium (Ph.D. thesis), Cambridge, MA, USA: MIT; 2009.
[24] Baker W F, Beghini L L, Mazurek A, Carrion J, Beghini A. Maxwell's reciprocal diagrams and discrete michell frames. Struct Multidiscip Optim 2013;48:267–77.
[25] Lee J, Van Mele T, Block P. Form-finding explorations through geometric manipulations of force polyhedrons. In *Proceedings of the international association for shell and spatial structures (IASS) symposium 2016*, Tokyo, Japan, September 2016.
[26] Ohlbrock PO, D'Acunto P, Jasienski JP, Fivet C. Vector-based 3d graphic statics (part III): Designing with combinatorial equilibrium modelling. In: Kawaguchi K, Ohsaki M, Takeuchi T, editors. Proceedings of the IASS annual symposium 2016 "spatial structures in the 21st century". IASS; 2016.
[27] Fivet C, Zastavni D. Constraint-based graphic statics: New paradigms of computer-aided structural equilibrium design. J Int Assoc Shell Spat Struct 2013;54(4):271–80.
[28] Zalewski W, Allen E. Shaping structures: statics. New York: Wiley; 1998.
[29] Ochsendorf J, Freeman M. Guastavino vaulting: The art of structural tile. Princeton Architectural Press; 2013, URL https://books.google.com/books?id=-q_MngEACAAJ.
[30] Maxwell JC. On reciprocal figures, frames and diagrams of forces. Trans R Soc Edinb 1870;26(1):1–40.
[31] Akbarzadeh M, Van Mele T, Block P. On the equilibrium of funicular polyhedral frames and convex polyhedral force diagrams. Comput Aided Des 2015;63:118–28. http://dx.doi.org/10.1016/j.cad.2015.01.006.
[32] Akbarzadeh M. 3d graphic statics using polyhedral reciprocal diagrams (Ph.D. thesis), Zürich, Switzerland: ETH Zürich; 2016.
[33] Beghini A, Beghini L L, Schultz J A, Carrion J, Baker W F. Rankine's theorem for the design of cable structures. Struct Multidiscip Optim 2013.
[34] McRobie A. Maxwell and rankine reciprocal diagrams via Minkowski sums for 2d and 3d trusses under load. Int J Space Struct 2016;31:115–34.
[35] McRobie A. Rankine reciprocals with zero bars. 2017, preprint.
[36] McRobie Allan, Baker William, Mitchell Toby, Konstantatou Marina. Mechanisms and states of self-stress of planar trusses using graphic statics, part II: Applications and extensions. Int J Space Struct 2016;31(2–4):102–11.
[37] Konstantatou Marina, Mcrobie Allan. Reciprocal constructions using conic sections & poncelet duality. In: Proceedings of IASS annual symposia, Vol. 2016. International Association for Shell and Spatial Structures (IASS); 2016, p. 1–10.
[38] Konstantatou Marina, D'Acunto Pierluigi, McRobie Allan. Polarities in structural analysis and design: n-dimensional graphic statics and structural transformations. Int J Solids Struct 2018;152:272–93.
[39] Bolhassani M, Akbarzadeh M, Mahnia M, Taherian R. On structural behavior of the first funicular polyhedral frame designed by 3d graphic statics. Structures 2017;56–68.
[40] Bolhassani M, Tabatabaei Ghomi A, Nejur A, Akbarzadeh M. Structural behavior of a cast-in-place funicular polyhedral concrete: applied 3d graphic statics. In: Proceedings of the IASS symposium 2018, creativity in structural design. Boston, USA: MIT; 2018.
[41] Lee J, Van Mele T, Block P. Disjointed force polyhedra. Comput Aided Des 2018;99:11–28. http://dx.doi.org/10.1016/j.cad.2018.02.004.
[42] Stokes GG. Mathematical and physical papers. Cambridge: Cambridge University Press; 1905.
[43] Hablicsek Márton, Akbarzadeh Masoud, Guo Yi. Algebraic 3d graphic statics: Reciprocal constructions. Comput Aided Des 2019;108:30–41. http://dx.doi.org/10.1016/j.cad.2018.08.003, URL http://www.sciencedirect.com/science/article/pii/S0010448518303026.
[44] Föppl A. Das fachwerk im raume. Leipzig: Verlag von B.G., Teubner; 1892.
[45] Schrems MJ, Kotnik T. Statically motivated form-finding based on extended graphical statics (EGS). In: Stouffs R, Janssen P, Roudavski S, Tuner B, editors. Open systems: Proceedings of the 18th international conference on computer-aided architectural design research in Asia (CAADRIA 2013). 2013.
[46] Kotnik Toni, D'Acunto Pierluigi. Operative diagramatology: structural folding for architectural design. In: Rethinking prototyping: Proceedings of the design modelling symposium Berlin 2013. Universität der Künste; 2013, p. 193–203.
[47] Ohlbrock Patrick Ole, D'Acunto Pierluigi. A computer-aided approach to equilibrium design based on graphic statics and combinatorial variations. Comput Aided Des 2020;121:102802.
[48] Lee J. Computational design framework for 3d graphic statics (Ph.D. thesis), Zurich: ETH Zurich, Department of Architecture; 2018.
[49] Van Mele T, Liew A, Mendéz Echenagucia T, Rippmann M, et al. Compas: A framework for computational research in architecture and structures.. 2017, http://compas-dev.github.io/compas/.
[50] Lee J. Compas 3gs: A 3d graphic statics add-on package for the compas framework. 2019, https://github.com/BlockResearchGroup/compas_3gs.

[51] Graovac O. 3d graphic statics. 2019, https://www.food4rhino.com/app/3d-graphic-statics.
[52] McRobie A. The geometry of structural equilibrium. R Soc Open Sci 2017;4(160759). http://dx.doi.org/10.1098/rsos.160759.
[53] Akbarzadeh Masoud, Hablicsek Marton. Algebraic 3d graphic statics: Constrained areas. 2020.
[54] PSL. Polyframe: Structural form finding tool using 3d graphic statics. 2017–2020, https://www.food4rhino.com/app/polyframe.
[55] McNeel R. Rhinoceros: NURBS modeling for windows. Comput Softw 2014. URL http://www.rhino3d.com/.
[56] Nejur A, Akbarzadeh M. Polyframe: Structural form finding tool using 3d graphic statics. 2020, https://github.com/PolyhedralStructuresLaboratory/PolyFrame.
[57] Kremer Michael, Bommes David, Kobbelt Leif. Openvolumemesh–a versatile index-based data structure for 3d polytopal complexes. In: Proceedings of the 21st international meshing roundtable. Springer; 2013, p. 531–48.
[58] Reeves David, Bhooshan Vishu, Bhooshan Shajay. Freeform developable spatial structures. In: Proceedings of IASS annual symposia, Vol. 2016. International Association for Shell and Spatial Structures (IASS); 2016, p. 1–10.
[59] Akbarzadeh M, Hablicsek M, Guo Y. Developing algebraic constraints for reciprocal polyhedral diagrams of 3d graphic statics. In: Proceedings of the IASS symposium 2018, creativity in structural design. Boston, USA: MIT; 2018.
[60] Nejur A, Akbarzadeh M. Constrained manipulation of polyhedral systems. In: Proceedings of the IASS symposium 2018, creativity in structural design. Boston, USA: MIT; 2018.
[61] PSL. Polyframe demoreel 3. 2018, URL https://vimeo.com/282934830 (accessed Aug 16, 2020).
[62] Skiena Steven S. The algorithm design manual: Text, Vol. 1. Springer Science & Business Media; 1998.
[63] Timoshenko Stephen P, Gere James M. Theory of elastic stability. Courier Corporation; 2009.
[64] Nejur A. Manual for polyframe ver. 0.1 for rhino. 2018, https://psl.design.upenn.edu/wp-content/uploads/2018/08/PolyFrame-Beta-01.pdf.
[65] Tabatabaei Ghomi A, Bolhassani M, Nejur A, Akbarzadeh M. Effect of subdivision of force diagrams on the local buckling, load-path and material use of founded forms. In: Proceedings of the IASS symposium 2018, creativity in structural design. Boston, USA: MIT; 2018.
[66] Akbarzadeh M, Bolhassani M, Nejur A, Yost J R, Byrnes C, Schneider J, Knaack U, Borg Costanzi C. The design of an ultra-transparent funicular glass structure. In *Structures congress 2019*, Orlando, Florida, 2019.
[67] Akbari M, Bolhassani M, Akbarzadeh M. From polyhedral to minimal surface funicular spatial structures. In *Proceedings of IASS symposium 2019 and structural membranes 2019, FORM and FORCE*, Barcelona, Spain, October 7-10 2019.
[68] Zheng Hao. Form finding and evaluating through machine learning: The prediction of personal design preference in polyhedral structures. In: The international conference on computational design and robotic fabrication. Springer; 2019, p. 169–78.
[69] Akbarzadeh M, Hablicsek M. Geometric degrees of freedom and non-conventional spatial structural forms. In *Impact: Design with all senses*, Berlin, Germany, September 23-252020.