EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Two geometry conformal methods for the use in a multi-disciplinary non-orthogonal building spatial design optimisation framework

Ezendam, T.

Award date:
2021

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

MSc. THESIS

For the master 'Architecture, Building and Planning' with specialization 'Structural Engineering and Design' at Eindhoven university of Technology

Two geometry conformal methods for the use in a multi-disciplinary non-orthogonal building spatial design optimisation framework

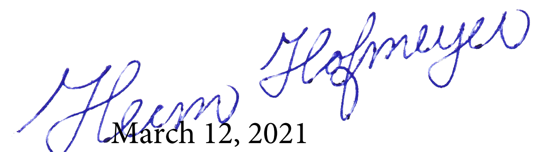
Tessa Ezendam

Student information

Name:	T. (Tessa) Ezendam
Student number:	0891861
Graduation date:	10-03-2021

Graduation Committee

Chairman:	dr. H. (Herm) Hofmeyer
Second supervisor:	dr. S. (Sjonnie) Boonstra
Third supervisor:	dr. P. (Pieter) Pauwels



March 12, 2021

Contents

1 Introduction **2**
- 1.1 Problem definition . . . 2
- 1.2 Related work . . . 4
  - 1.2.1 Building spatial design optimisation . . . 4
  - 1.2.2 Conformal model . . . 5
- 1.3 Background and motivation . . . 5
- 1.4 Paper content . . . 6

2 Design and optimisation framework **6**
- 2.1 Building spatial design model . . . 6
- 2.2 Conformal model . . . 7
- 2.3 Discipline specific models . . . 7

3 Two geometry conformal methods **7**
- 3.1 Requirements and preferences . . . 8
- 3.2 Concepts . . . 8
- 3.3 Quad-hexahedron geometry conformal method . . . 11
  - 3.3.1 General overview . . . 11
  - 3.3.2 Step 1 . . . 16
  - 3.3.3 Step 2 . . . 18
  - 3.3.4 Step 3 . . . 19
- 3.4 Tetrahedron geometry conformal method method . . . 21

4 Demonstration and analysis **25**
- 4.1 Quad-hexahedron method . . . 25
- 4.2 Tetrahedron method . . . 27
- 4.3 non-orthogonal BSD example . . . 30
- 4.4 Comparison . . . 30

5 Discussion **31**

6 Conclusions **33**

Acknowledgements **34**

References **34**

A C++ code **37**
- A.1 main . . . 38
- A.2 ms_building . . . 39
- A.3 cf_building_model . . . 60
- A.4 cf_geometry_model . . . 73
- A.5 cf_entity . . . 78
- A.6 cf_space . . . 83
- A.7 cf_cuboid . . . 91
- A.8 cf_tetrahedron . . . 132
- A.9 cf_rectangle . . . 142
- A.10 cf_triangle . . . 148
- A.11 cf_line . . . 151
- A.12 cf_building . . . 155

B Designs tested **160**

C Geometry check errors **177**

Two geometry conformal methods for the use in a multi-disciplinary non-orthogonal building spatial design optimisation framework

Tessa Ezendam

Eindhoven University of Technology, The Netherlands

Abstract

Two geometry conformal methods will be presented to generate a conformal model for the use in a design and optimisation framework, which transforms a quad-hexahedron non-orthogonal building spatial design (NOBSD) to multiple discipline-specific models. This conformal model has multiple use-cases regarding discipline-specific model definition, such as element connection and intersection problems within a finite element mesh, definition of properties and loads, and it allows for the grouping of sub-parts of a spaces into zones for spatial layouts that are more logical from a disciplines point of view. The first method, the quad-hexahedron method, generates a conformal model by splitting a quad-hexahedron NOBSD in quad-hexahedron geometry entities. The second method, the tetrahedron method, generates a conformal model by splitting the quad-hexahedron NOBSD in tetrahedron geometry entities. Both methods are demonstrated, analysed and compared. As a result, the following conclusions can be made. Future work on the geometry conformal methods is needed, since both methods are not able the obtain a conformal model for all NOBSD. However, the quad-hexahedron method is already able to generate a conformal model for a certain scope of NOBSD and is observed to be suitable for the mentioned use-cases. In addition, the tetrahedron method might be able to make a conformal model for NOBSDs consisting of tetrahedrons. This is an application area for which the quad-hexahedron method would not be applicable to, without making a hybrid between the two. As a result, future research may result in two geometry conformal methods each applicable in their own field of strength.

Keywords: Geometry conformal method, Conformal model, Non-orthogonal building Spatial Design, multidisciplinary optimisation, Automatic partitioning

1 Introduction

1.1 Problem definition

The built environment consumes about 40% to 60% of all energy and material resources (Anderson et al., 2015; European Commission, 2005), and reductions of this amount have to be reached in the upcoming years (*Paris Agreement*, 2015; *Klimaatakkoord*, 2019). In addition, the conceptual phase of a design is highly interdisciplinary, involving collaboration between customers, designers, and multiple engineers from different specializations. In this early stage of the design, it is difficult to oversee the complex relationships between the different disciplines. This may lead to sub-optimal decisions and these will determine to a large extent the final performance of the building. Decisions made later in the design process often have less impact (Wang et al., 2002). Therefore better support with relevant tools at the beginning of the design process will give a great improvement. These relevant tools should consider a large extend of the building spatial design scope currently used in practice. So not only the rectangular spatial designs but also the non-orthogonal buildings spatial designs (NOBSD). Furthermore, such tools should consider multiple disciplines since the combination of multiple disciplines would support the highly interdisciplinary decision making seen in the conceptual phase of the design process. For these reasons, there is a need for a multidisciplinary optimisation tool, which in addition to the orthogonal rectangular shapes, also take into account the non-orthogonal building spatial design.

Within such a multidisciplinary optimisation tool, the framework of transforming a building spatial design model to multiple discipline-specific models may consist of the following three sub-transformations: (1) from an input to a building spatial design model, (2) from an initial spatial design model to a conformal model (in the following referred to as "geometry conformal method"), and (3) from a conformal model to (multiple) discipline-specific models. In this case, the conformal model described the building spatial design (BSD) in their decomposed geometry entities for discipline-specific model definition in two ways, namely the building conformal model and the geometry conformal model. The building conformal model described the original BSD, whereas the geometry conformal model described the BSD as a conformal geometry. A conformal geometry is characterized by the fact that it contains no non-corner vertex intersections, as seen in figure 2. In addition, The lines of the adjacent geometry entities should match (in the following referred to 'compliant lines'), as can be seen in figure 1.

Such a distinction between a geometry conformal model and a building conformal model has multiple use-cases regarding discipline-specific models. In the first use-case, the geometry conformal model is used to solve

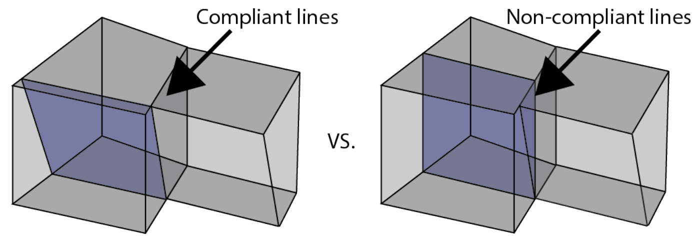


Figure 1: Comparison of compliant and non-compliant lines

Building spatial design

Space 2

Space 1

Building spatial design; cross section

Conformal model

Cell 2

Cell 1a

Cell 1b

Geometry conformal model; cross section

Dicipline-specific models

Finite element model: mesh (plate elements) with non coinciding nodes

R_{ex} R_{ex} R_{ex} $\mathbf{R_{ex}/R_{in}}$ R_{ex} R_{ex} R_{ex}

Building physic model: assign R-value R_{ex}=exterior wall, R_{in} =interior wall with single R-value wrongly assigned to full wall

q life load

Zone 1

q life load

Zone 2

Structural optimisation via zones: zones based on spaces of building spatial design

Dicipline-specific models

Finite element model: mesh (plate elements) with coincided nodes

R_{ex} R_{ex} R_{ex} $\mathbf{R_{in}}$ R_{ex} R_{ex} R_{ex} R_{ex} R_{ex}

Building physic model: assign R-value R_{ex}=exterior wall, R_{in} =interior wall with correct separate R-value zones for full length wall

q life load

Zone 1

q life load

Zone 1

Zone 2

Structural optimisation via zones: zones based on cells of the geometry conformal model

Figure 2: Generation of discipline-specific models from a building spatial design vs a conformal model

element connection and intersection problems in a mesh of the finite element model (see figure 2), as was previously presented as the multi-block method for the meshing of complex geometry (Frey and George, 2000). The second use-case considers both conformal models to define the properties and loads of the components in the discipline-specific models. For example, the building conformal model can be used to define the live load or temperature in a space, since this is a constant within the space. However, the definition of the heat transfer coefficients for a building physic model should be done according to the geometry conformal model. This will allow for the required division of a single 'wall' in two separate heat transfer coefficient sections, as is shown in figure 2. In the third use-case, the geometry conformal model allows for the grouping of sub-parts (cells) of spaces into zones for spatial layouts that are more logical from a disciplines point of view, as previously done in the research of Claessens, Boonstra, en Hofmeyer (2020). In the case of the structural discipline, the structural optimisation via zones result in the preferred direct load path from the wall to the ground, as presented in figure 2. For these reasons, it is beneficial to have a conformal model as part of a multidisciplinary optimisation framework. The problem under investigation in this paper is how to obtain and represent this conformal model for non-orthogonal building spatial designs NOBSD. However, the NOBSD scope is large. therefore, as an initial start, the focus within this research is on obtaining and representing a conformal model for NOBSD consisting of quad-hexahedron. Some examples of these designs as given in figure 3. In addition, special attention will be given for the possibility of extending the quad-hexahedron NOBSD scope in the future. To solve the defined problem, two geometry conformal methods will be presented as well as the framework (of the three sub-transformation) in which these geometry conformal methods will function.

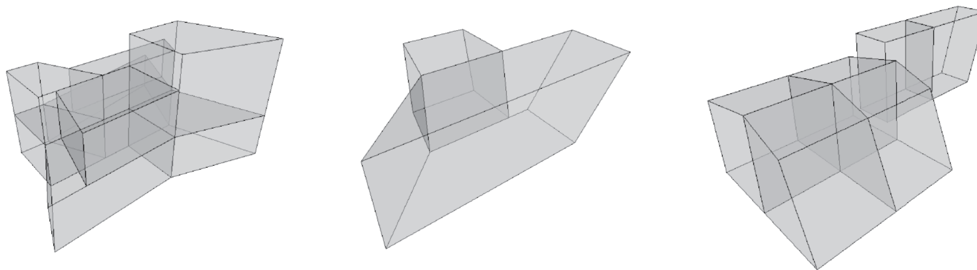


Figure 3: Examples of the Quad-hexahedron NOBSD scope included in this research

1.2 Related work

This section will first discuss the related work on building spatial design optimisation with special attention to spatial representations in optimisation. Furthermore, the goal within this research is to obtain a conformal model, therefore related research on conformal models will be discussed. And finally, the background and the motivation for the presented work is presented.

1.2.1 Building spatial design optimisation

The building spatial design optimisation representation determines to a large extent the design space of a building spatial design (BSD) problem. For example, a representation which is limited to orthogonal shapes cannot represent non-orthogonal shapes in a building. Both the optimisation efficiency and success are dependent on this design space. Therefore, it is important to consider the used representation for building spatial design optimisation.

In literature multiple full building scale design representation can be found, both considering orthogonal rectangular BSD and NOBSD. At first the orthogonal rectangular BSD examples. The first and second example consist of a rectangular layout optimisation with the same floor scheme on each elevation of the building, the first considers structural design (Sharafi et al., 2015) and the second building energy consumption (Li et al., 2018). The third and fourth examples consist of rectangular layout optimisation with different floor scheme per elevation of the building, considering building energy consumption and daylight performances (Dino and Üçoluk, 2017; Granadeiro et al., 2013). The firth example consists of plane definitions, which represents walls and slabs, to form a building spatial design (Geyer, 2008). These planes are then replaced with a beam or frame construction depending on the grammar. The examples considering NOBSD representations are as follows. The first example is a building spatial design representation on the hand of five planes controlled by agent points, which controls the position of the individual child point which defines and morphs the plane (Yi and Malkawi, 2009). The second example optimizes a parametric spatial representation, according to the parameters: building

height, number of levels, tapering or twisting factors, canyon width, and top set back (Lin and Gerber, 2014). The third example is the optimisation program GENE_ARCH, which optimises a complete building design in term of geometry, spatial layout and room characteristics, such as construction materials, internal finishes, types and characteristics of window and glazing systems, and even mechanical and electrical installations (Caldas, 2008). This spatial representation is based on rectangular layout with different floor scheme per elevation, however, the top floor is allowed to have a roof tilt.

Most of these full building size spatial design optimisation examples consider the building physic discipline. Considering the structural discipline are there also a lot of example found which consider parts of a buildings. Some of these examples are in the structural typology optimisation field, considering a 2D bracing system for multistory steel frames (Liang et al., 2000; Baldock and Shea, 2006) or beam optimisations (Sigmund, 2001; Tugilimana et al., 2017). The later uses a modular system representation, or a voxel representation. In the voxel representation case, a highly variable non-orthogonal shape is obtained. However, this voxel representation should then be interpreted for creating a structure.

Given this background, the conclusion is that, to the writers knowledge, no such representation as is proposed in this research has been developed for the use in a multidisciplinary optimisation tool. Therefore, this research will provide an addition to the current scope of representation available in research.

1.2.2 Conformal model

As mentioned earlier, the conformal model has multiple use-cases regarding discipline-specific models. From where one such use-case is the original origin of such a conformal model, namely to generate a mesh for finite element methods. In this content, the conformal model allows for a simple method to be applied to the conformal partition as the union of the meshes of two conformal partition automatically results in a conformal mesh, as is also demonstrated in the 'finite element model' example of figure 2. This process of first partitioning a geometry (either in a conformal partition or non-conformal partition), then meshing the partitioned is called the multi-block method (Frey and George, 2000).

In general, the multi-block method is seen as a semi-automatic method (Frey and George, 2000; Smith and Eriksson, 1987) since the partition process is mainly carried out manually. Some research considering manual partitioned geometry can be found consisting of the automatic generated boundary layer mesh using an input surface mesh (Lu et al., 2018), a hybrid dynamic mesh generation method for a multi-block structured grid (Chen et al., 2017), a multi-block mesh extrusion drive by a globally elliptic system (Vassberg, 2000), and an integrated CAD system to assist the boundary definition within a numerical software package for mesh generation (Huang, 1997). However, some research is also performed on automating this partition process. Specifically, the automatic generation of multi-block decomposition of surfaces (Fogg et al., 2015) or the automatic domain partitioning for quadrilateral meshing with line constrains (Kowalski et al., 2015). These examples consider both 2D and 3D surface partitioning. Furthermore, less prominent in research is the automatic partitioning of volume elements, as is regarded within this research. Some examples of these considering the automatic partitioning of orthogonal rectangular cuboid BSD (Boonstra et al., 2018; Hofmeyer et al., 2011), and the automatic partitioning of complex vascular trees (Bols et al., 2016). However, no automatic partitioning of a similar representation as is regarded in this reseach has been found.

Furthermore, some related research of interest are the conformal finite elements meshing techniques since these are, in essence, a finer partitioned conformal geometry. Within this context, the related research are mesh refinement techniques (Oh and Lim, 1996; Nicolas and Fouquet, 2013), a automatic generation of 3D-network structures with tetrahedron elements (Nguyen-Van-Phai, 1982), automatic mesh generation over intersection surfaces (Lo, 1995), and converting a non-conforming hexahedral-to-hexahedral interfaces into conforming interfaces (Staten et al., 2010).

Given this background, the conclusion is that, to the writers knowledge, no such conformal model generator for NOBSD as defined in this work exists. Therefore, within this research, multiple new automatic partitioning methods, named geometry conformal method, will be developed.

1.3 Background and motivation

The presented research in this paper is performed within a larger research scope that focuses on multi-disciplinary design and optimisation of BSDs for support of the conceptual phase of the design process. Within this scope, a toolbox has been developed, from which the C++ code is available (Boonstra and Hofmeyer, 2020). The toolbox contains among others the presented framework of the three sub-transformation with the conformal model, however, only for orthogonal rectangular BSDs (Hofmeyer et al., 2011; Boonstra et al., 2018).

This conformal model has already been used in the orthogonal rectangular BSD cases to solve among others the mentioned use-cases in the 'problem definition'(Hofmeyer et al., 2011; Boonstra et al., 2018; Claessens

et al., 2020). Furthermore the toolbox has been used to research: (1) discipline-specific design grammars for automated design in spatial-structural design (Smulders and Hofmeyer, 2012; Claessens et al., 2020) and building physics design (Boonstra et al., 2018), (2) analysing models for structural design (Hofmeyer and Russell, 2009) and building physics (Boonstra et al., 2018), (3) a co-evolutionary approach for optimized building spatial and building structural design (Hofmeyer and Davila Delgado, 2015), (4) an application and configuration of evolutionary algorithms to building spatial design optimisation (van der Blom et al., 2017; van der Blom et al., 2016), and (5) a super-structured and super-structure free multi-disciplinary building spatial design optimisation method (Boonstra et al., 2018).

The choice has been made to research if this framework using the conformal model is also applicable to NOBSD since it has been proven to be useful. In addition, to the writters knowlage, no such conformal model generator for NOBSD exist (as has been seen in section 1.2.1). Therefore, a new 'geometry conformal method' is needed, which can generate this conformal model for NOBSD.

1.4 Paper content

This paper is structured as follows. At first the design and optimisation framework in which the geometry conformal method will function will be explained in section 2. Thereafter, in section 3, the requirements, concepts, and workings for the two geometry conformal method will be explained in detail. These methods will be demonstrated and analysed in section 4. Whereafter the discussion and conclusion is presented in section 5 and 6.

2 Design and optimisation framework

As explained in section 1.3, this research is performed within a toolbox, which among others contained the design and optimisation framework of the geometry conformal method presented in this research, however, only for orthogonal rectangular BSD (Hofmeyer et al., 2011). The general overview of this framework consist of three main kinds of models, which are obtained via three kinds of sub-transformations, as is presented in figure 4. In this section, the NOBSD variants of the first two models and sub-transformations of this framework will be explained in section 2.1, and 2.2. The third kind of model is the discipline-specific model, which represent the use-cases of the conformal model regarding discipline specific model definition. In general, several discipline specific models can be defined with the help of the conformal model. However, some discipline-specific models were already developed for the orthogonal rectangular BSD toolbox, for which special attention will be taken during the development of the NOBSD geometry conformal method. Therefore, within section 2.3, a brief summary of these will be given.

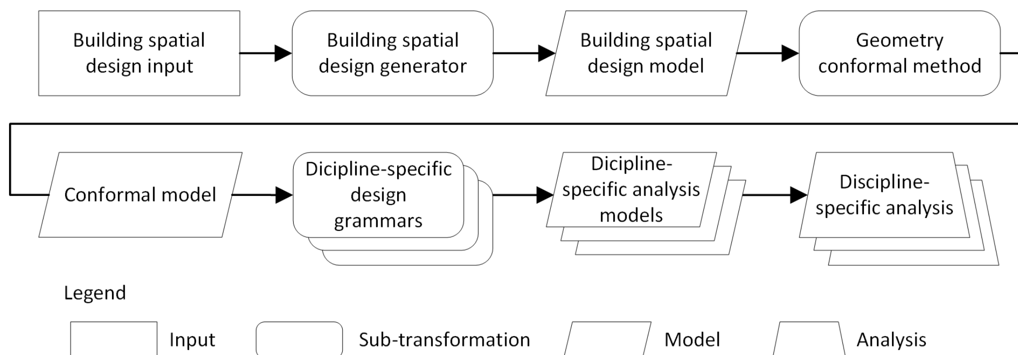


Figure 4: Design and optimisation framework for transforming a building spatial design into multiple discipline-specific models with the help of the geometry conformal method and model

2.1 Building spatial design model

The building spatial design model stores all the different instances of the NOBSD during the optimisation process, including the initial BSD. The initial BSD functions as a starting point for optimisation and is defined as follows. A BSD is defined with the help of the 'building spatial design input'. The input contains one or more spaces (corresponding to rooms in a building) that are each specified by an ID and eight corner-vertices of the quad-hexahedron. Additionally, in all cases, z=0 represents the ground surface and values below zero

(z<0) are underground. The input is interpreted by the 'building spatial design generator' that generates the quad-hexahedron entities. The vertices within this quad-hexahedron will be rearranged and sorted before being stored in the 'building spatial design model'.

2.2 Conformal model

The conformal model is represented by a geometry conformal model and a building conformal model, as was previously explained in the 'problem definition' (section 1.1). The building conformal model represents the NOBSD, but stores the inserted spaces in the NOBSD in the decomposed geometry entities: spaces, surfaces of-, edges of-, and points of spaces. The geometry conformal model stores the NOBSD as a conformal geometry. This conformal geometry is stored in the following geometry entities: cell, polygons, line segments, and vertices. These entities are linked to the building conformal model, as is presented in the UML diagram in figure 5. Thereby can, for example, a space corresponds to multiple cells and a surface to multiple polygons.

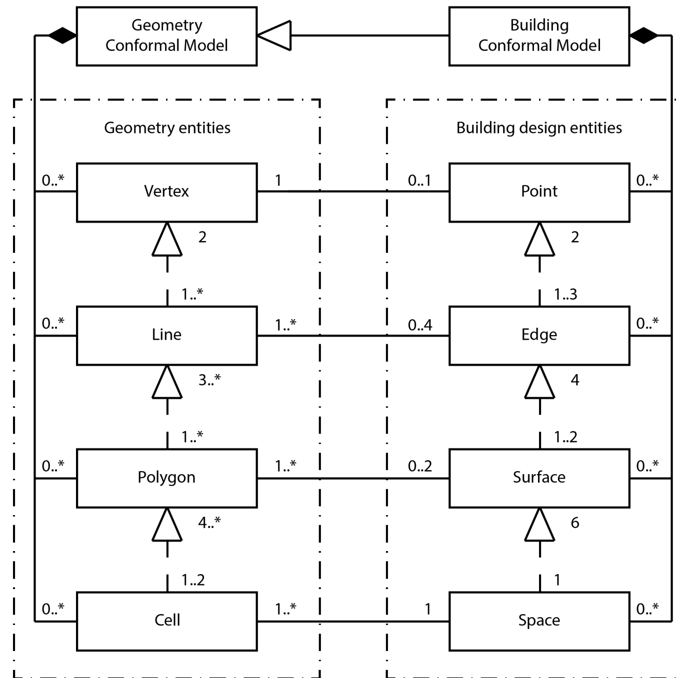


Figure 5: UML class diagram of the conformal model consisting of the geometry conformal model and the building conformal model

2.3 Discipline specific models

Some discipline-specific models which were already developed for the orthogonal rectangular BSD toolbox consist of the structural model and a building physic model. The structural model is defined with the help of a design response grammar and evolutionary algorithm (Boonstra et al., 2020), which assigns structural components to the surfaces of the geometry conformal model. This definition is done according to the zoned spatial layouts that allow for a more logical spatial layout from a disciplines point of view (Claessens et al., 2020). These structural elements are then meshed, before being analysed with the finite element method (Hofmeyer and Russell, 2009). The Building physics analysis in the toolbox is currently limited to an evaluation of thermal building behaviour (Boonstra et al., 2018). For this a building physic model is defined with a grammar which assigns wall/floor materials and properties to each polygons that belong to a surface in the building conformal model. This model is then analysed with the help of an RC-network approach.

3 Two geometry conformal methods

The two geometry conformal methods used to define a conformal model will be presented within this section, for which the used C++ code can be found in appendix A. The requirements and preferences regarding the geometry conformal methods are defined in section 3.1. Thereafter, the concepts of the two geometry conformal

methods are described in section 3.2. Finally, in section 3.3 and 3.4, both geometry conformal methods are explained in detail.

3.1 Requirements and preferences

A conformal method should result in a conformal model. Meaning that there are no non-corner vertex intersections allowed in the resulting conformal model, and all the lines should be compliant, as was previously defined in the 'problem definition'. Considering the building spatial design scope, a conformal model should be obtained for as many, if not all, quad-hexahedron building spatial designs. Also, the conformal method should at least apply to other building spatial design shapes, consisting of non-orthogonal triangular prism and a non-orthogonal pentagonal prism for future research. Lastly, the replicability of the research performed with the Toolbox is of importance, so that the inserted building spatial design, with the same method, always results in the same conformal model.

Besides these requirements, there are also some general preferences. Namely, the conformal method: (1) should be optimized for short process time since the conformal framework is going to be performed numerous times during the optimisation process. And (2) the resulted conformal model is preferable clear and structured for the reviewing of results and building discipline-specific models.

From a discipline-specific model point of view there are, beside the requirement of a conformal model, the following preferences defined. In general, the geometry conformal method should result in building like partitions. Building like partitions, in this case, are defined as favorable partitions for a structural design of a building. This concise of a geometry conformal model needing to represent a regular build-up grid for the placement of structural elements, as is suggested as the third used-case of the geometry conformal model in the 'problem definition' (see figure 2). Furthermore, for the structural discipline-specific model, there are some additional preferences concerning the mesh generation as a result of the geometry conformal model (Frey and George, 2000). for which it is preferred to have resulting polygon partitions in the geometry conformal model that are suitable for the local meshing process. A polygon partition is suitable when: (1) it has a similar partition size compared to the surrounding partitions, thereby resulting in a more consistent mesh grid size, (2) the faces of the partition are as close as possible to a convex region.

3.2 Concepts

Within this research, two main methods are chosen to obtain a conformal model, namely the 'quad-hexahedron' and the 'tetrahedron' geometry conformal method, which will be explained in the following. Where after the reasoning for choosing these methods will be explained.

At first, the general setup of both methods is explained, which consist of an 'initialize' phase and an 'adapt' phase. The initiate phase generates the final building conformal model and an initial geometry conformal model, as is depicted in figure 6. This initial geometry conformal model contains the direct decomposition of the NOBSD in geometry entities, from which the cell entities need not to be of conformal geometry. However, this will be assured in the iterative adapt phase, where the initial geometry conformal model is step wise adapted through multiple iterations to obtain a conformal geometry. This conformal geometry will then be the final geometry conformal model.

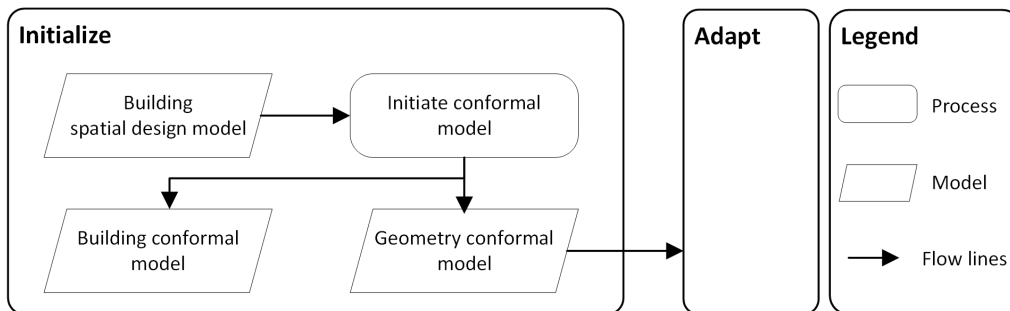


Figure 6: General setup of both geometry conformal methods

The difference between both methods is the considered cell entity, in the geometry conformal model, to go through the iterative adapt phase. In case of the quad-hexahedron method, the NOBSD is decomposed in cells of quad-hexahedron geometry entities, as presented in figure 7. This process results in the UML diagram of the conformal model, as presented in figure 8. In the adapt phase, in case of intersection, the quad-hexahedron is

split in the according amount of quad-hexahedrons depending on the intersection case considered (either a line or a polygon intersection case). The way the quad-hexahedron is split can vary in multiple ways as shown in figure 9. Therefore, two sub-methods will be developed, which describe how the new quad-hexahedron cells are defined. These sub-methods are called the 'perpendicular to opposite' and 'perpendicular to intersected' sub-method. Either one of these sub-methods will be applied to the whole adapt process of the geometry conformal method and is determent by the user. In case of the tetrahedron method, the NOBSD is decomposed in cells of tetrahedron geometry entities, as presented in figure 10. This process results in the UML diagram of the conformal model, as presented in figure 11. In the adapt phase, in the case of an intersection, the tetrahedron cell is split in the according amount of tetrahedron cells depending on the intersection case considered (either a line or polygon intersection case). The tetrahedron cell is split so that only new tetrahedrons are generated. In this case there are only 2 possible methods available due to its geometry, as can be seen in figure 9.

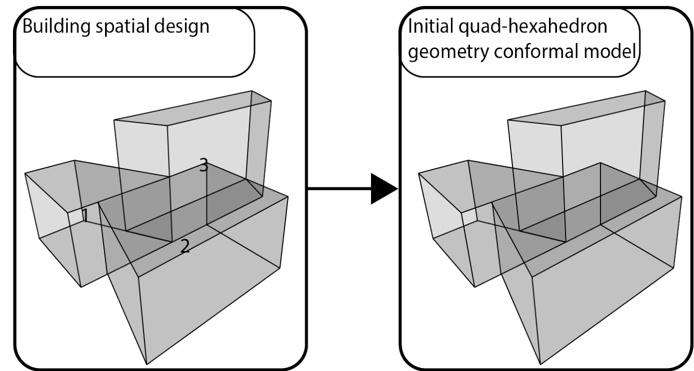


Figure 7: Decomposition of quad-hexahedron building spatial design into the initial quad-hexahedron geometry conformal model

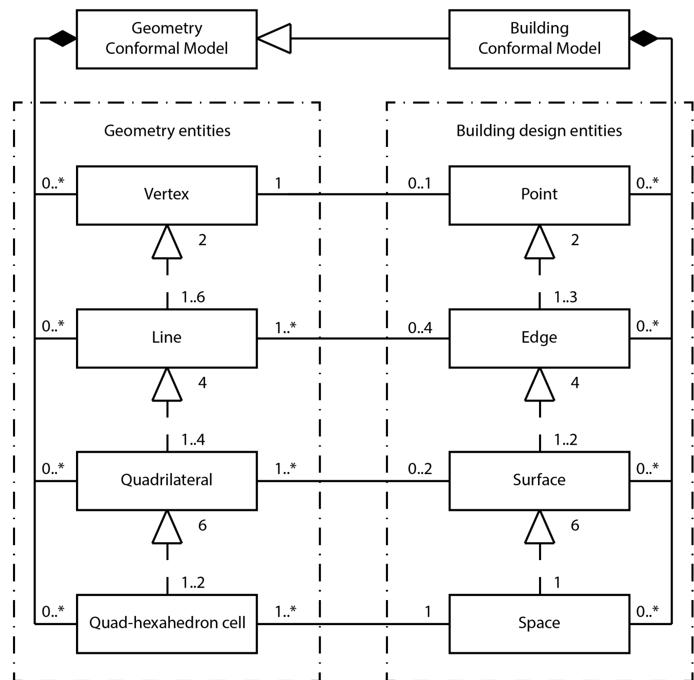


Figure 8: UML class diagram of the building conformal model in case of the hexahedron method

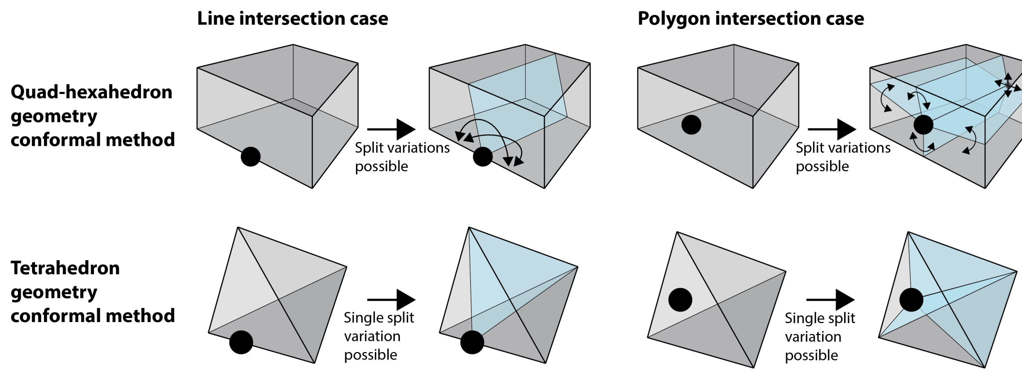


Figure 9: Cell split variation comparison between the quad-hexahedron method and tetrahedron method on the two intersection cases line intersection and polygon intersection

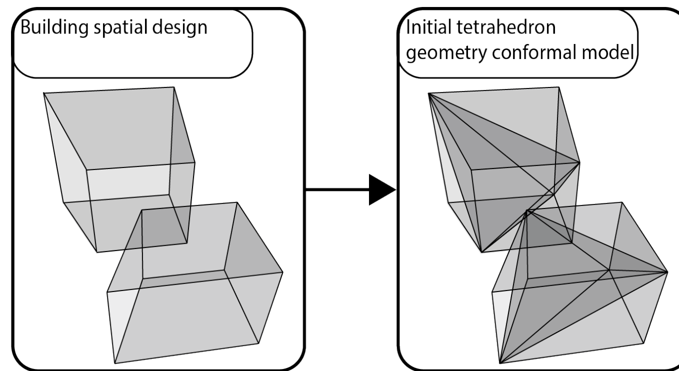


Figure 10: Decomposition of quad-hexahedron building spatial design into the initial tetrahedron geometry conformal model

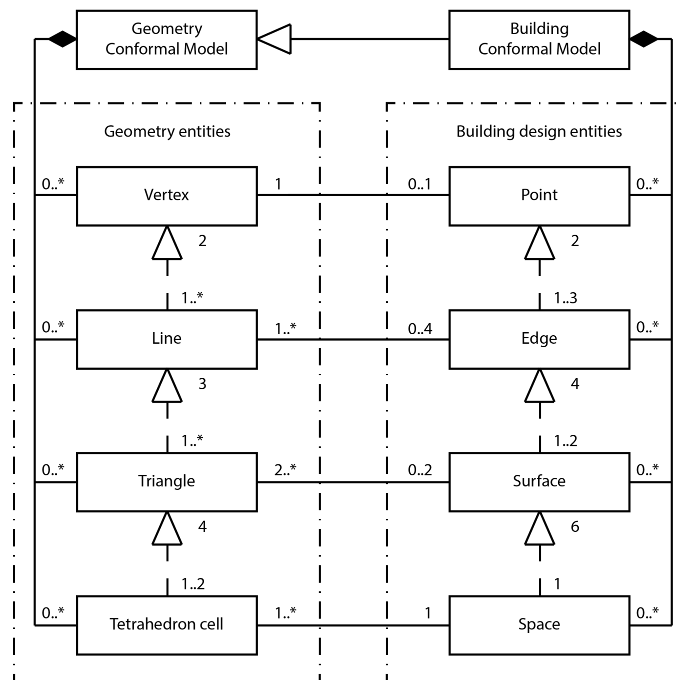


Figure 11: UML class diagram of the building conformal model in case of the tetrahedron method

The first method, the quad-hexahedron method, is the extension/adaptation from the orthogonal rectangular conformal method (Hofmeyer et al., 2011) and is chosen due to the following reasons. It has the potentials to result in building like partitions, since the geometry entity used is often seen in building design. Furthermore, less partitions are needed to make a conformal model in comparison to the tetrahedron method, resulting in a comprehensive conformal model. Also, if the inserted BSD is setup with consistent sizes and intersection positions, then the partition will also be of consistent size. Consequently resulting in a consistent mesh, the latter is preferable for the mesh generation of the structural discipline-specific model. The challenge with this method is the multiple ways the quad-hexahedron could be split, as was previously presented in figure 9. Consequently resulting in other types of partition geometry entities than quad-hexahedron partitions. The other partition geometry entities will then need to be supported with its own split function in the adapt phase, therefore complicating and enlarging the needed scope considered in this method. To make sure the quad-hexahedron method will result in quad-hexahedrons the previously mentioned sub-methods are developed, each with its own quality and applicability field.

The second method, the tetrahedron method, is chosen to overcome the splitting challenge of the quad-hexahedron method. The main advantage comes from the tetrahedron geometry itself, which is always able to result in a tetrahedron partition, as was previously presented in figure 9. Furthermore, there is only one single vertex required as input to perform the partition of the cell, meaning that it is a simple method to implement and later on scale to other NOBSD, which can be build-up from tetrahedrons. The challenge of this method is that it does not result in building like partitions, since the tetrahedron geometry entity is rarely seen in building practices. Furthermore, there are some concerns with this method on how applicable the partition will be for the meshing of the polygons, as well as the clarity of the conformal model and the process time needed to obtain the conformal model.

3.3 Quad-hexahedron geometry conformal method

The adapt phase of the quad-hexahedron geometry conformal method will consist of three main steps, which are going to be discussed in detail in the upcoming sections. At first, a general overview of all steps is given in section 3.3.1, followed by an explanation of each step in detail in section 3.3.2, 3.3.3, and 3.3.4.

3.3.1 General overview

In general, the quad-hexahedron adapt phase of the geometry conformal method consists of three main steps, which are illustrated for a single cell in figure 12. In simple terms, step one checks, on a conformal model level, for intersections and collects each unique intersection vertex as well as each unique intersection line (of the intersecting cells) for a space combination or cell combination and sends them to step two. Step two is performed on a space level and needs these intersection lines of the surrounding cells since this information can not be collected on a space level. Step two then makes for each unsolved intersection a selection of vertices, with the help of the intersection lines, for a single cell split action performed by step three. This step two is needed since the split cell function (step three) is chosen to split according to either a single line intersection vertex or a single polygon intersection vertex, both accompanied by guide vertices. These guide vertices will overrule the standard splitting direction defined by the sub-method (of step three). See figure 13 for an example of compliant lines and non-compliant lines due to the overruling of the standard splitting direction by the guide vertices. The guide vertices are determined from the intersection lines found in step one, since compliant lines in the case of an intersected cell i by possibly numerous cells j consist of cell i replicating the lines from the cells j on the shared surface. Furthermore, it is chosen that step three splits the cell according to the intersection found on a single polygon of the cell. This in combination with the consideration of a single intersection vertex accompanied by multiple guide vertices will prevent the need for multiple specialized split functions for each individual amount of line- or polygon-intersections found for an intersected cell i on each individual polygon. See figure 14 for some examples of the numerous variations of cases existing to consider for a single intersected polygon of cell i. Step three then performs the cell split action, on a cell geometry level, according to either one of the two sub-methods. Ones the cell is split, then the geometry conformal model is directly adapted accordingly and used in the continuation of the considered step and following steps.

These main three steps however are performed in a nested sequence to solve all intersections, for which the process is depicted in the 'adapt' section of figure 15. This nested process will be explained in more detail in the following section 3.3.2, 3.3.3, and 3.3.4. Furthermore, see figure 16 for a visual overview of these steps applied on a single space of a NOBSD.

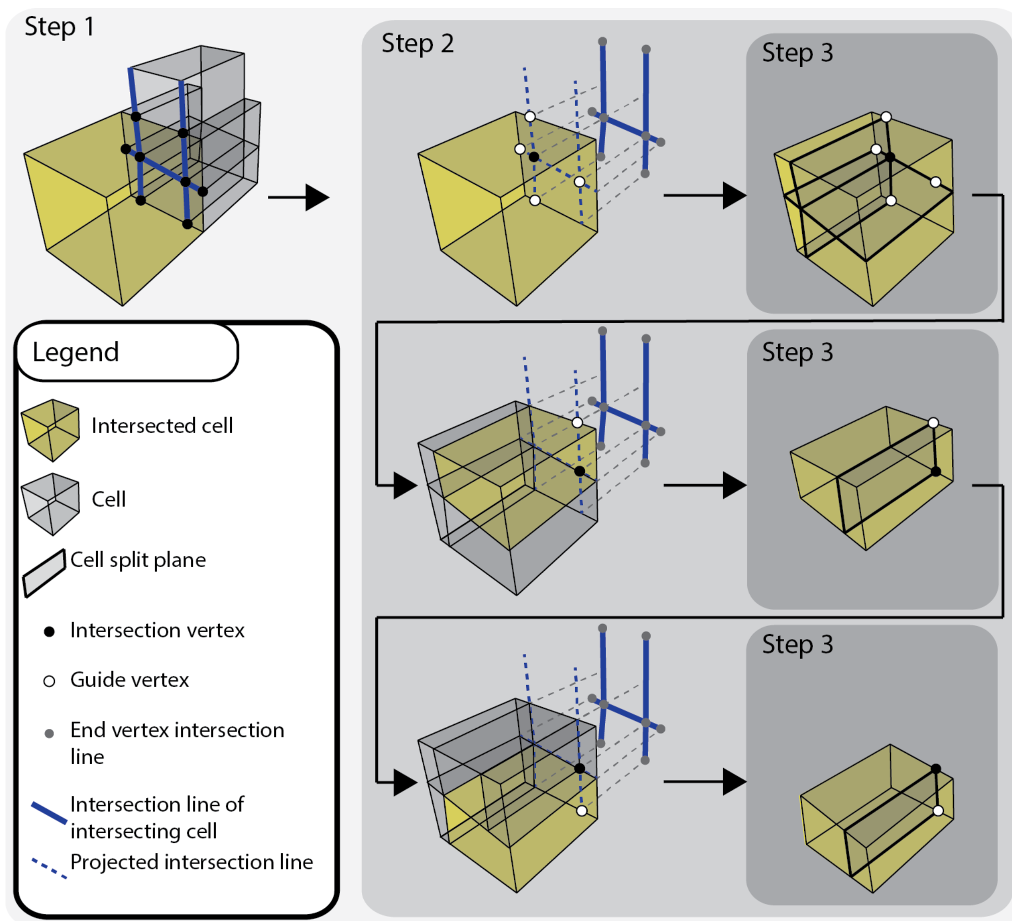


Figure 12: Example of the three steps performed on a single cell of the geometry conformal model, resulting in a conformal cell; note: the intersection lines not coinciding with the intersecting plane are not drawn for the clarity of image

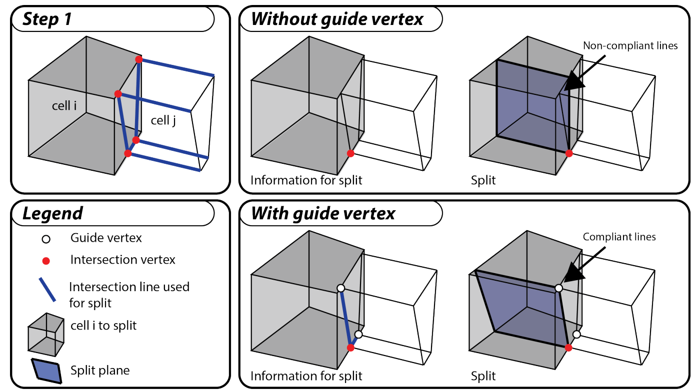


Figure 13: Comparison of with and without the use of guide vertices for an individual split action

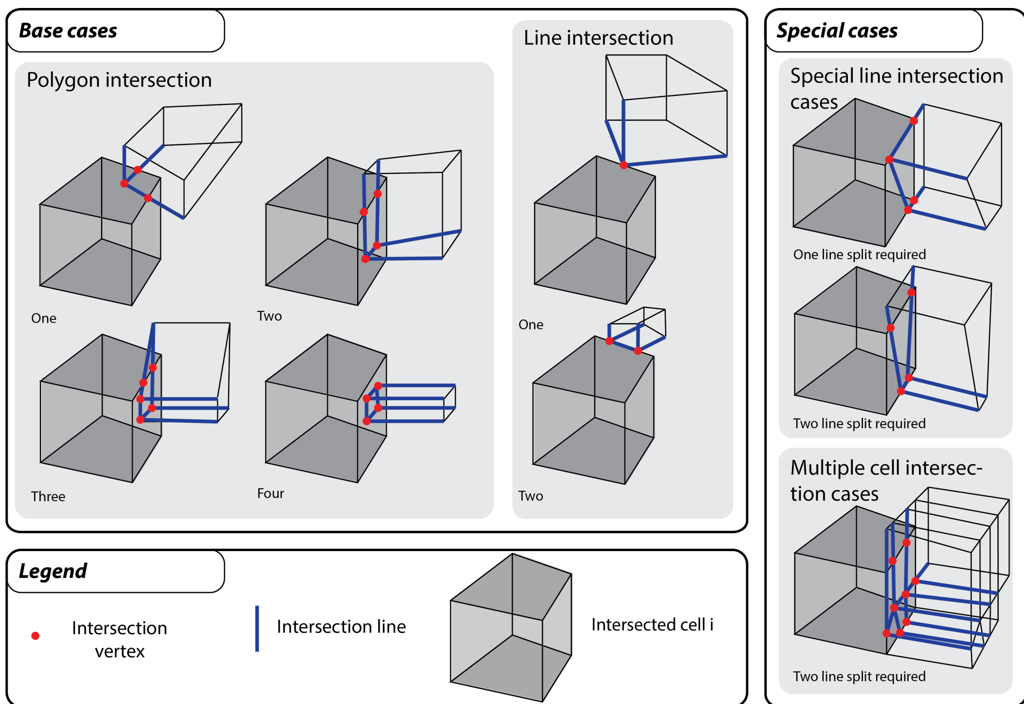


Figure 14: Examples of the numerous variation of cases existing to consider for a single intersected polygon of the intersected cell i

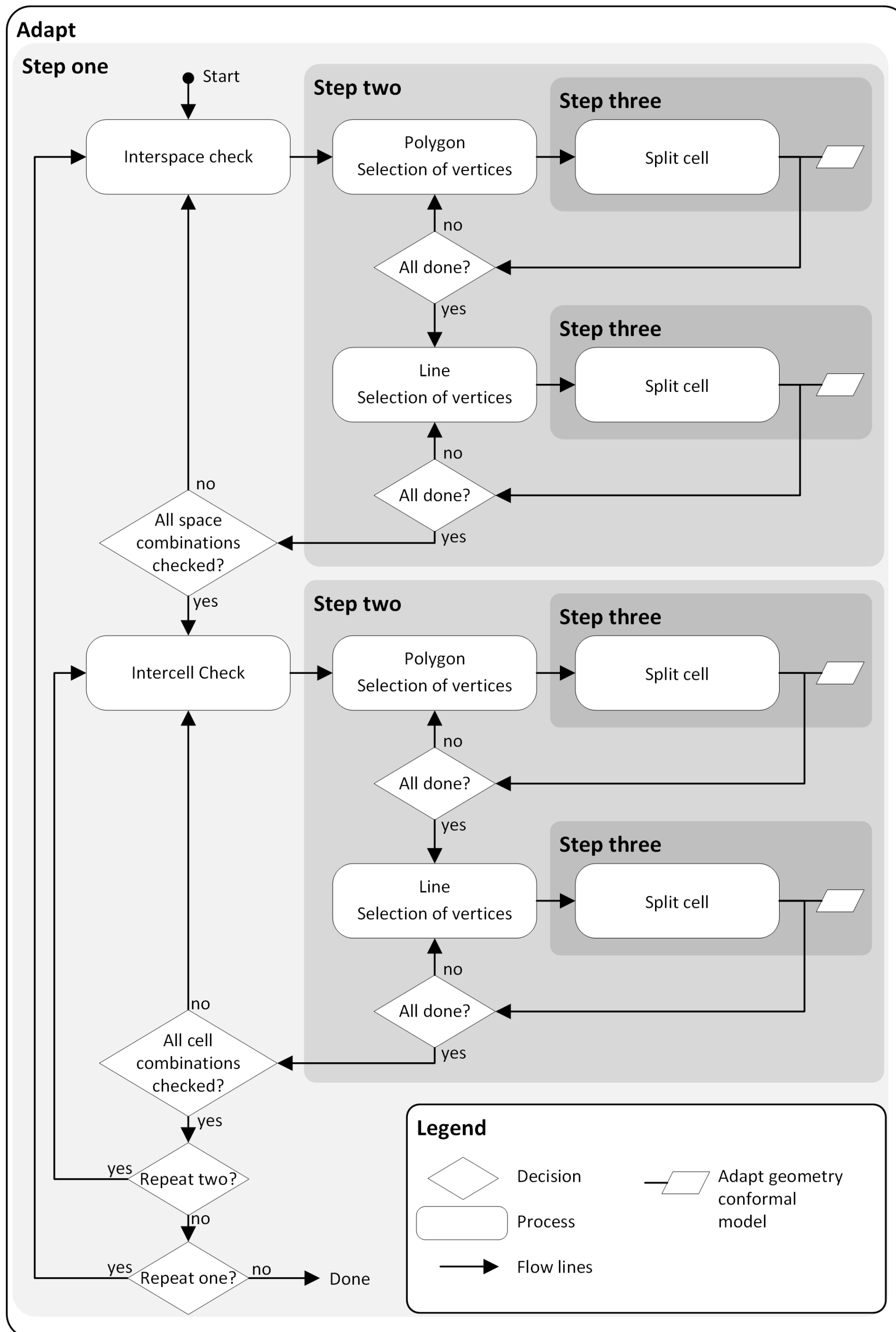


Figure 15: Process diagram of the quad-hexahedron geometry conformal methods adapt phase

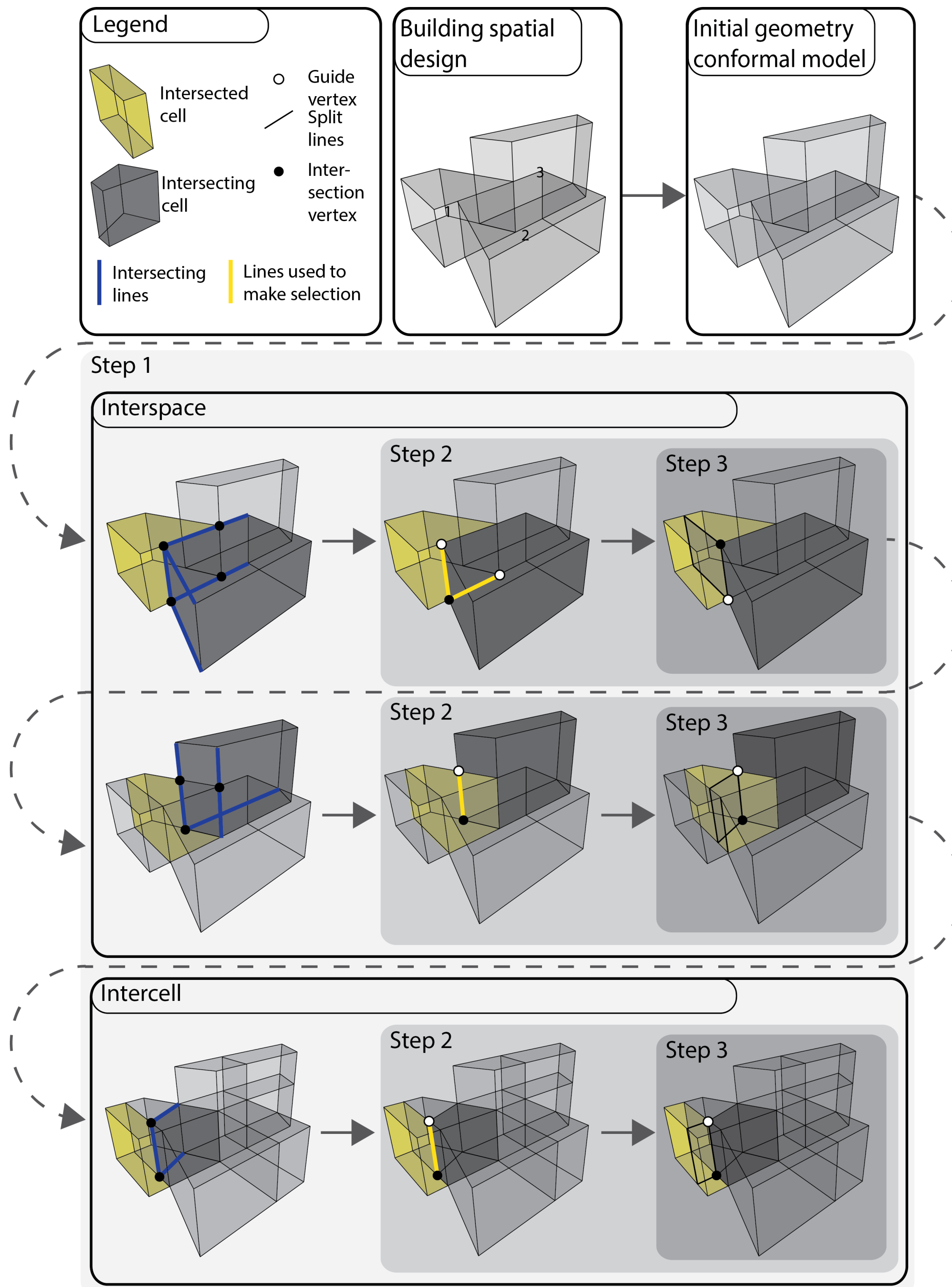


Figure 16: Quad-hexahedron split process explained on the hand of the full split of all cells within space 1 according to the intersecting cells of space 2 and 3

3.3.2 Step 1

In general, step one checks, on a conformal model level, for intersections and collects each unique intersection vertex as well as each unique intersection line (of the intersecting cells) for a space combination or cell combination and sends them to step two. For this there are two main check processes defined, as well as two repeat processes. These processes combined will assure obtaining a conformal geometry as will be explained in the following, where after the principles behind the checking for intersections are explained, which is done on the basis of the line-line and line-polygon checks. See algorithm 1 for a full overview of the algorithm used by this step one.

Algorithm 1 Quad_hexahedron step 1

```
1: while a single intersection is found during the interspace check process do      ▷ Repeat one

2:     for each space k do                                          ▷ Start interspace check process
3:         for each space l do
4:             Skip space l if it is the same as space k
5:             for each cell i of space k do
6:                 for each cell j of space l do
7:                     Perform all line-line and line-polygon checks and store the found vertices in IntersectionV
8:                     Store the intersecting lines from cell j in IntersectionL
9:                 end for
10:            end for
11:            Remove duplicates lines and vertices from IntersectionL and IntersectionV
12:            if size of IntersectionV ≥ 1 then
13:                Execute step 2(IntersectionV, IntersectionL) for space k
14:            end if
15:            Clear content IntersectionV and IntersectionL
16:        end for
17:    end for

18:    for each space k do                                          ▷ Start intercell check process
19:        while a single intersection is found during the line-line and line-polygon check do   ▷ Repeat two
20:            for each cell i of space k do
21:                for each cell j of space k do
22:                    Perform all line-line and line-polygon checks and store the found vertices in IntersectionV
23:                    Store the intersecting lines from cell j in IntersectionL
24:                    Remove duplicates lines and vertices from IntersectionL and IntersectionV
25:                    if size of IntersectionV ≥ 1 then
26:                        Execute step 2(IntersectionV, IntersectionL) for space k
27:                    end if
28:                    clear content IntersectionV and IntersectionL
29:                end for
30:            end for
31:        end while
32:    end for

33: end while
```

At first, the two main processes named the interspace check and the intercell check process are explained. The interspace check process checks all cells i of space k with all cells j of space l for unique intersection vertices and lines and send them to step two per space combination. The intercell check process checks cell i of space k with cell j of space k for unique intersection vertices and lines and send them to step two per cell combination. In both check processes, in case of intersection(s), all the found intersection vertices and intersection lines are stored and send to step two for further processing. Both check processes follow the order in which the cells where initially generated (in figure 6 the 'initiate conformal model' step), which is according to the space order in the BSD input. Resulting in two problems, which will be demonstrated with the help of the '*follow-up cell between different spaces*' and the '*follow-up cells within a space*' cases in figure 17. Specifically, in the '*follow-up cell between different spaces*' case, the first full iteration of the interspace check is as follows. First, cell 1 with cell 2 is checked, thereby finding no intersection. Afterwards, when cell 2 is checked with cell 3 intersections

are generated between cell 1 and 2. Therefore, this first iteration will not convert to a conformal model since cell 1 is not checked again. However, an additional full iteration of interspace would, in this case, assure a conformal model. In the '*follow-up cell between different spaces*' case, the first iteration of interspace generates a non-conformal model since the cells within a single space are not checked with each other. To solve these cases the intercell check process is added to step one. However, the first iteration of intercell is not sufficient to obtain a conformal model either due to the multiple cells in a row to be split. This could be solved, in this case, by an additional iteration of intercell. In conclusion, in these cases, the single full iteration of the interspace or intercell check process is insufficient to obtain a conformal model and multiple iteration of these check processes are required. To overcome these kinds of cases the repeat one and repeat two processes are added. Repeat two encapsulates the intercell check process, thereby ensuring that the '*follow-up cells within a space*' cases are solved. Repeat one encapsulates both check processes (intercell and interspace) and works as a final check for intersections, as well as solving the '*follow-up cell between different spaces*' cases.

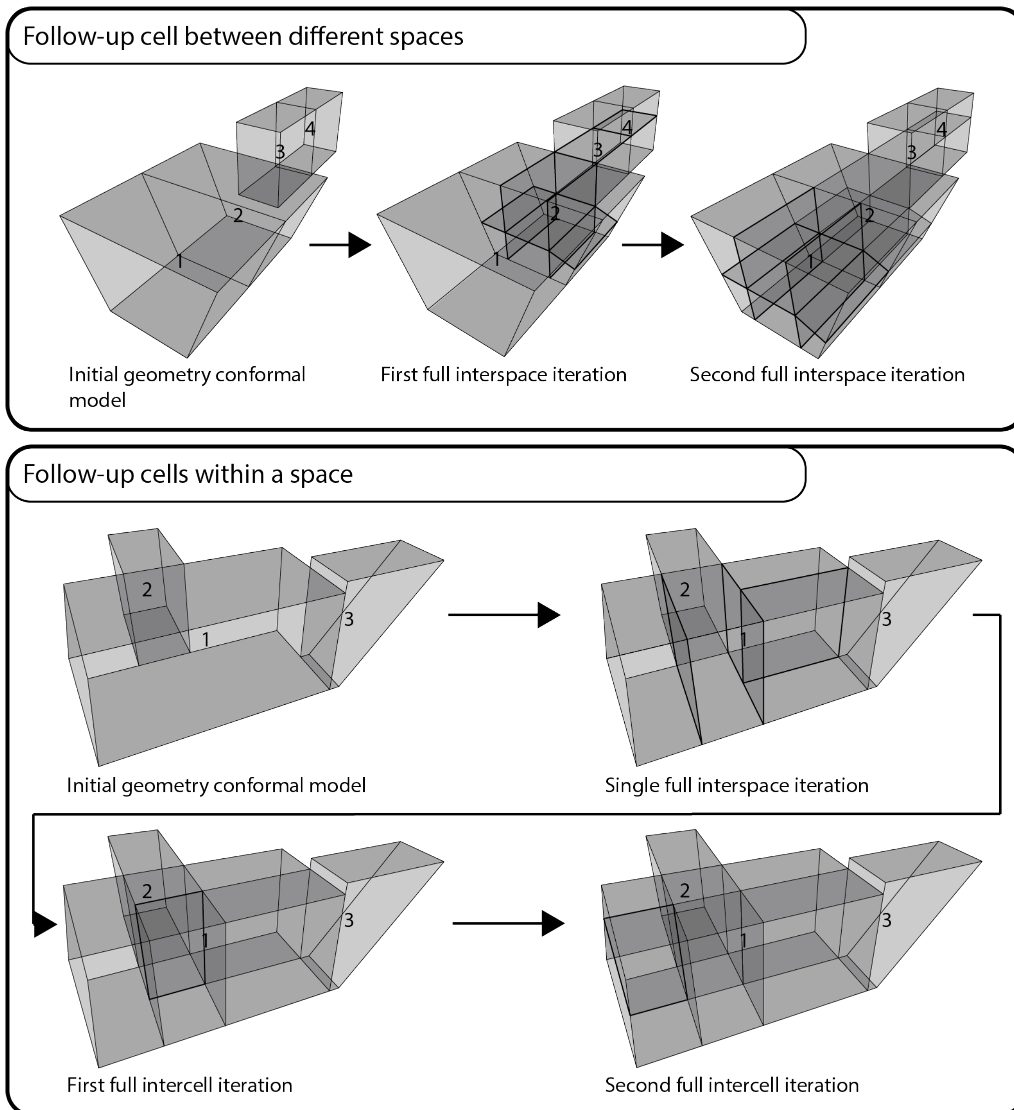


Figure 17: 'Follow-up cell between different spaces' and 'follow-up cells within space' problem cases and how they are solved by the interspace and intercell check

Secondly, the principles behind checking for intersection, which work on the basis of a line-line and a line-polygon intersection check are explained. The line-line intersection check performs a check with a line n of cell i and an intersecting line m of cell j. Every intersection found between the end vertices of line n is considered an intersection vertex, so excluding the end vertices of line n. If the end vertex of the intersecting line m is between the end vertices of line n, then this is considered an intersection vertex. If the lines are parallel or partly or fully coincided then there is no intersection vertex between line n and the intersection line m. For the line-polygon intersection check an intersection check is performed between a polygon of cell i and a line of cell j.

The line can only intersect if the intersection vertex is found on or between the end vertices of the line. For the polygon, only the intersection within the borderlines of the polygon are considered to be an intersection vertex, so excluding the borderlines of the polygon. If the line is parallel or coinciding with the plane, then there are no intersection vertices between the two entities. However, the fully coinciding line with the plane is still stored as an intersection line for step two since this line describes the line to replicate during the split in step three to reach compliant lines. Furthermore, the also relevant non fully coinciding line with the plane is stored as an intersection line for step two during the line-line check, thereby assuring that all relevant intersection lines are stored for step two.

3.3.3 Step 2

The second step in the conformal method makes a selection of vertices for each unsolved intersection vertex found in step one to guide a single splitting action in step three. See algorithm 2 for the full algorithm used for step two. At first, it will be discussed how such a selection is made, where after the order of selection making is discussed.

Algorithm 2 Quad_hexahedron step 2

```
1: Split all lines in IntersectionL at the intersections vertices of IntersectionV

2: for each vertex m in IntersectionV do                  ▷ Start polygon intersection selections
3:     Set bool onGeometry to false
4:     for each cell i in space k do
5:         for each polygon n of cell i do
6:             Break for loop if onGeometry is true
7:             if vertex m is on polygon n then                          ▷ define guide vertices
8:                 onGeometry is true
9:                 Add the lines of IntersectionL connected to vertex m to linesToAccount
10:                Add vertex m to splitVertixen
11:                for each line in linesToAccount do
12:                    Add the non vertex m vertex to splitVertices
13:                end for
14:                Remove duplicates from splitVertices
15:                Execute step 3(splitVertices) for cell i
16:            end if
17:        end for
18:    end for
19: end for

20: for each each vertex m in IntersectionV do               ▷ Start line intersection selections
21:     for each cell i in space k do
22:         for each line l of cell i do
23:             if vertex m is on line l then                            ▷ define guide vertices
24:                 Add the connected lines from IntersectionL to vertex m to linesToAccount
25:                 Add vertex m to splitVertices
26:                 for each linesToAccount n do
27:                     Add the non vertex m vertex from n to splitVertices
28:                 end for
29:                 Execute step 3 with input splitVertices for cell i
30:             end if
31:         end for
32:     end for
33: end for
```

In general, a selection of vertices for a single splitting action will consist of the single intersection vertex considered, and if needed one or multiple guide vertices. These guide vertices are determined on the basis of the intersection lines found in step one. This is done by looking at the intersection lines connected to the single intersection vertex considered, from which the 'non-intersection end vertex' will be taken as a guide vertex (see the previous figure 12 for an example). However, before this guide vertex definition may start the intersection lines are split at the intersection vertices, to make sure that if possible the 'non-intersection end vertex' of the

intersection lines are on the geometry of the cell to split. This is needed since only the guide vertices on the intersection cell geometry are taken into account in step three. Such a case where the intersection lines extend beyond the cell to split geometry has already been observed in figure 12. Furthermore, by taking only the guide vertices on the intersection cell geometry into account, the found guide vertices from the non-coincided intersection lines are not considered in the split of step three, as is required. Then before sending these guide vertices in combination with the considered intersection vertex, the duplicates vertices are removed to assure that step three does not define duplicated cell split planes, but only the required split planes. Once all guide vertices are found for the considered intersection vertices these are sent to step three for splitting.

The order in which the selection of vertices is made is as follows. At first, step two starts by generating a selection for each unsolved polygon intersection vertex, where after each line intersection vertex is solved. By first solving the polygon intersection vertices, in most cases, the line intersection vertices are automatically solved as well, resulting in less split actions. Such a process can be seen in the previous figure 12, where in the first polygon intersection split also solves three line intersections. Furthermore noticeable in this case is that after the first time performing the step three split action, the split cell is directly taken into account in the continuation of step two. Therefore, the in step one determined polygon intersection is transformed to a line intersection for the continuation of step two.

3.3.4 Step 3

Step three consists of two cell splitting sub-methods, namely the 'perpendicular to opposite' and 'perpendicular to intersected' which both split the cell with the help of defining a splitting plane. Each of these methods has to solve two types of intersection cases, namely the line and polygon intersection case. These will be explained after an explanation of the general setup of both sub-methods.

Algorithm 3 Quad_hexahedron step 3

```
 1: Bool split = false
 2: if more then one vertex in IntersectionV intersect with polygon of cell i and the first vertex in container
    IntersectionV is intersecting with a polygon then                                        ▷ Polygon split
 3:     Set bool split to true
 4:     Find the opposite polygon to the intersected polygon
 5:     for each corner vertex of the intersection polygon do
 6:         Add the polygon intersection vertex to container cornervertices
 7:         Add the corner vertex of the intersection polygon to cornervertices
 8:         Find the lines to split and store them in container lineToSplit
 9:         Find the corresponding corner vertex to the intersection corner vertex on the opposite polygon and
    add to cornervertices
10:         Add the corresponding corner vertex to cornervertices
11:         Find rest corner vertices with corresponding split sub-method and add to cornervertices
12:         Initiate cell with vertices in cornervertices and add the new cell to container newCells
13:     end for
14: end if
15: if split = false then
16:     if more then one vertex in IntersectionV intersect with line of cell i and the first vertex in container
    IntersectionV is intersecting with a line then                                              ▷ Line split
17:         Set bool split to true
18:         for each corner vertex of intersected line do
19:             Add intersection vertex to container cornervertices
20:             Find the polyInCell
21:             Find the lines To split and store in container lineToSplit
22:             Find rest corner vertices with corresponding split sub-method and add to cornervertices
23:             Initiate cell with vertices in cornervertices and add to container newCells
24:         end for
25:     end if
26: end if
27: if split = true then                                                ▷ Adapt geometry conformal model
28:     Remove the original cell, rectangles, lines and vertices from the geometry conformal model
29:     Add the new cells, rectangles, lines and vertices to the building conformal model
30: end if
```

The general setup of both splitting sub-methods (see algorithm 3) consist of the following stages: an identification, a search for the relevant geometric entities, a new cell definition, and adding and removing cells. The identification stage identifies which split intersection case should be performed based on the first intersection vertex stored in the vertex selection (splitVertices input) from step 2 and dispatches all guide intersection vertices in the according container for later use. The relevant geometric entities stage searches for the geometric entities needed for the split, see figure 18 and 19 for the named entities. In the polygon case, the geometric entities needed for the cell split into four new cells are the 'intersection polygon' and the 'opposite polygon', furthermore, per cell the geometric entities 'lines to split' are searched. In the line intersection case, the geometric entities needed are the 'intersected line', 'polygon in cell' and the 'lines to split', whereas the later two are defined per cell. The cell definition stage finds all the corner vertices of each new cell with the help of one or multiple intersection plane(s), according to the 'perpendicular to intersected' or 'perpendicular to opposite' sub-method. The final stage is the adding and removing of cells, in which the adaptation of the geometry conformal model and the accompanied links between the building conformal model is done. Specifically, the deletion of the cell which was split and the addition of the newly defined cells, including their geometric entities: polygon, line, and vertices.

As previously mentioned, defining new cells is done with the help of their corner vertices. The first set of corner vertices is found on the basis of the original cell geometry. These vertices are, in the case of line intersection, the corner vertices of the 'polygon in cell' and the intersection vertex. In the case of polygon intersection, these are the polygon intersection vertex, a corner vertex of the 'intersection polygon', and the 'corresponding corner vertex' on the opposite polygon to this corner vertex. The rest of the corner vertices are found with the help of an intersection plane. In general, these corner vertices are found by defining either one or two intersection planes on the hand of 2 vectors or the normal of the plane. The plane is then intersected with the lines to split. The way these two vectors or a normal of a plane are defined differs per intersection case and sub-method, as will be explained in the following paragraphs.

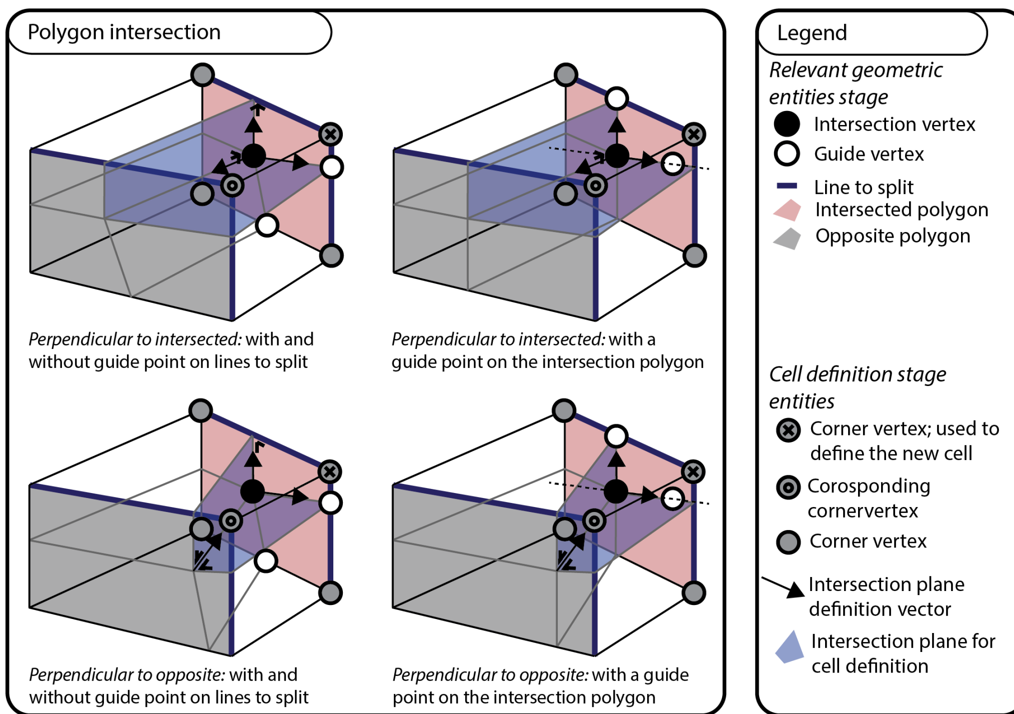


Figure 18: Overview of the two split methods, splitting the space by defining the intersection plane and finding the intersection vertices with the lines to split

At first, the polygon intersection cases, which exist of an intersected polygon definition stage and an opposite polygon definition stage. The intersected polygon definition stage is the same for both sub-methods. Namely, it first checks if there is a guide vertex on the lines to split or on the intersected polygon. If there is a guide vertex on the line to split, it will add this vertex to the corner vertex container. If there is a guide vertex on the intersected polygon, it will draw an infinite line thought the intersection vertex and the guide vertex. This infinite line will then be intersected with the lines to split and only added to the corner vertex container if the guide vertex is in-between the intersection vertex and the intersection vertex on the line to split vertex. If there are no guide vertices, then the closest vertex to the intersection vertex from the line to split is added

to the corner vertex container. In this case, the newly generated line is perpendicular to the lines to split of the intersected polygon. The definition of the corner vertex on the opposite polygon side is carried out by two intersection planes, each defined by two vectors. The first vector for all sub-methods is defined from the polygon intersection vertex considered to the the vertex found on the lines to split of the intersected polygon. For the perpendicular to intersected the second vector is defined as the perpendicular line on both intersected polygon vectors. Lastly, for the perpendicular to opposite the normal of the opposite polygon is taken as second vector.

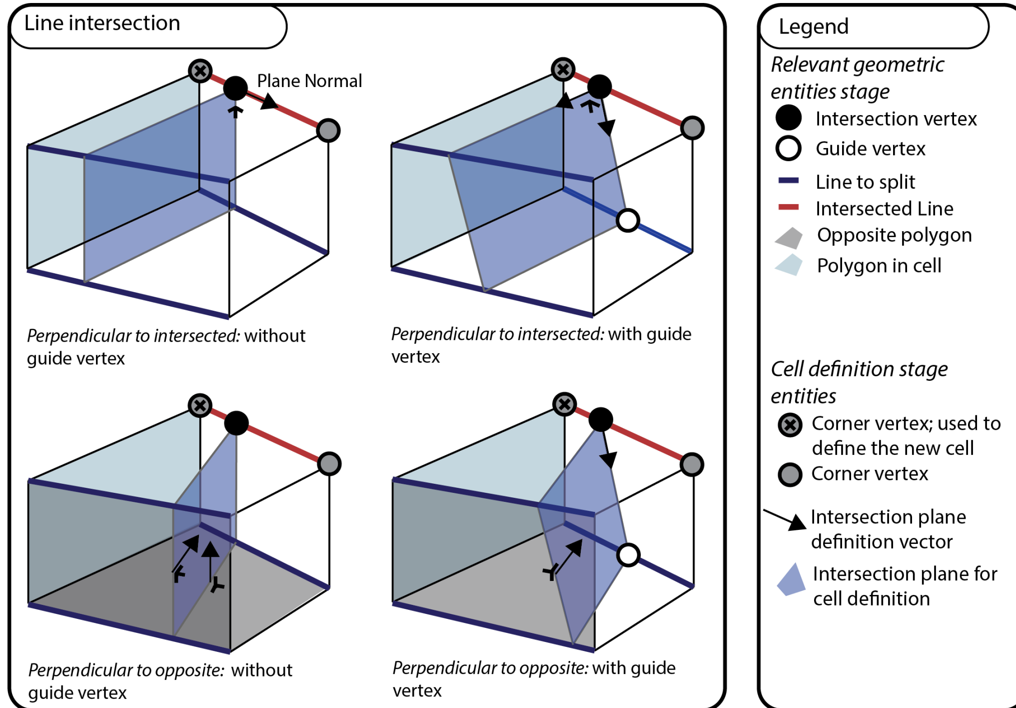


Figure 19: Overview of the two split methods, splitting the space by defining the intersection plane and finding the intersection vertices with the lines to split

Second, the line intersection cases. In the case of a guide vertex, the first vector is for all methods defined the same, namely from the intersection vertex to the guide vertex. The guide vertices taken into account during the line split should be on the line to split since otherwise the resulting cells would not be a quad-hexahedron cell. The second vector, in the perpendicular to intersected method, will be defined by taking the cross product of the guideline vector and the intersected line vector, resulting in a vector perpendicular to the two mentioned vectors. The second vector, in the perpendicular to opposite method, will be defined by taking the normal of the opposite polygon, which in this case is the opposite polygon to the first vector containing polygon. In the case of no guide vertex, during the perpendicular to intersected method, the intersection plane is defined by the intersection line vector, which will serve as the normal of the intersection plane. In the case of no guide vertex, both vectors, during the perpendicular to opposite method, are defined as the normal of both the opposite planes to the intersection vertex.

As previously described, after the definition stage, the adding and removing cells stage is performed, thereby adapting the geometry conformal method. This brings the adapt phase of the geometry conformal method closer to the end and thereby final geometry conformal model.

3.4 Tetrahedron geometry conformal method method

The adapt phase of the Tetrahedron geometry conformal method will consist of three main steps, which are performed in a nested sequence to solve all intersections. This process is depicted in the 'adapt' section of figure 20 and is based on the setup of the orthogonal rectangular geometry conformal method. Furthermore, see figure 21 for a visual overview of these steps applied for a single vertex on one single space.

The three main steps will be described in detail in the following. Step one identifies the intersection vertices between tetrahedrons and adds them to the vertex container of the geometry conformal model (see algorithm 4). The identification of intersection vertices is done according to the line-line and line-polygon checks previously explained in section 3.3.2. These checks are performed on all the lines and all polygons within the geometry conformal model. Thereby generating a vertex container within the geometry conformal model containing all

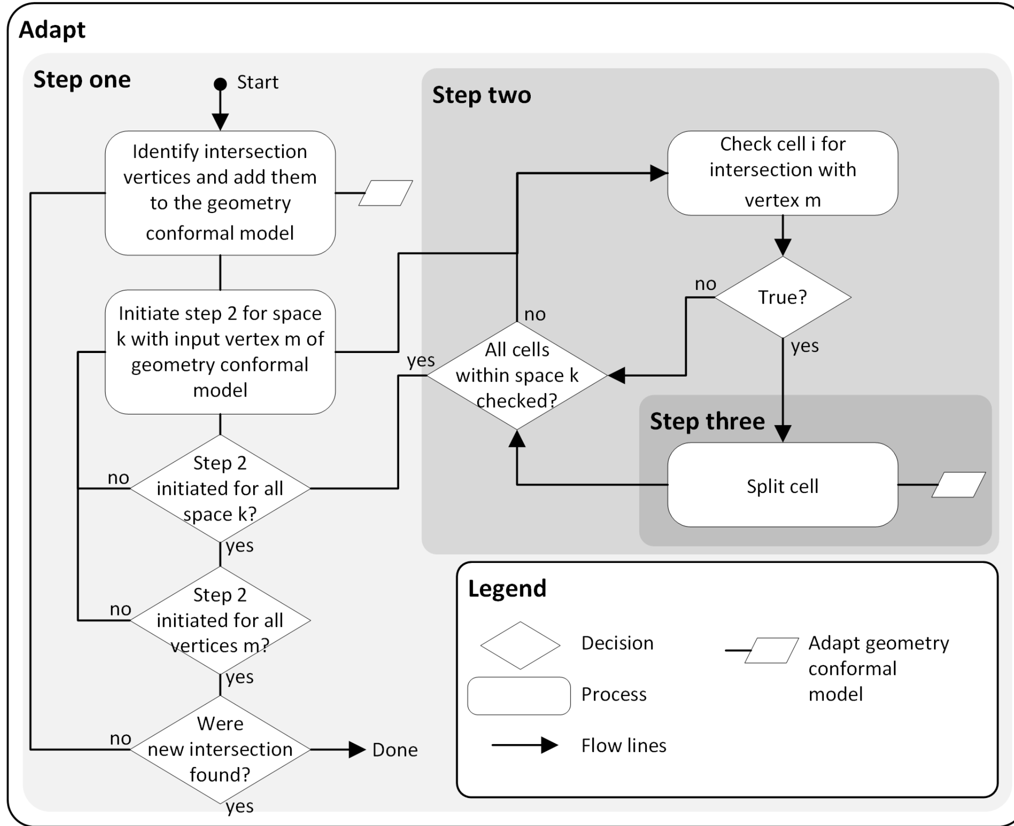


Figure 20: Process diagram of the tetrahedron geometry conformal method adapt phase

Algorithm 4 Tetrahedron step 1

1: int *repeat*=0
2: **for** $repeat < mCfVertices$.size() **do** ▷ Repeat
3: $repeat = mCFVertices$.size()
4: Perform line-line check, and in case of intersection, add the intersection vertex to the geometry conformal model *vertex* container
5: Perform line-polygon check, and in case of intersection, add the intersection vertex to the geometry conformal model *vertex* container
6: **for** each *vertex m* in the *vertex* container of geometry conformal model **do**
7: **for** each *space k* **do**
8: Perform step 2(*vertex m*) for *space k*
9: **end for**
10: **end for**
11: **end for**

Algorithm 5 Tetrahedron step 2

1: Bool *onGeometry* = false
2: **for** each polygon in *space i* **do**
3: Check if *vertex k* is on polygon, if so then set *onGeometry* = true
4: **end for**
5: **for** each line in *space i* **do**
6: Check if *vertex k* is on line, if so then set *onGeometry* = true
7: **end for**
8: **if** *onGeometry* = true **then**
9: **for** each *cell l* in *space i* **do**
10: initiate step 3(vertex k) for *cell l*
11: **end for**
12: **end if**

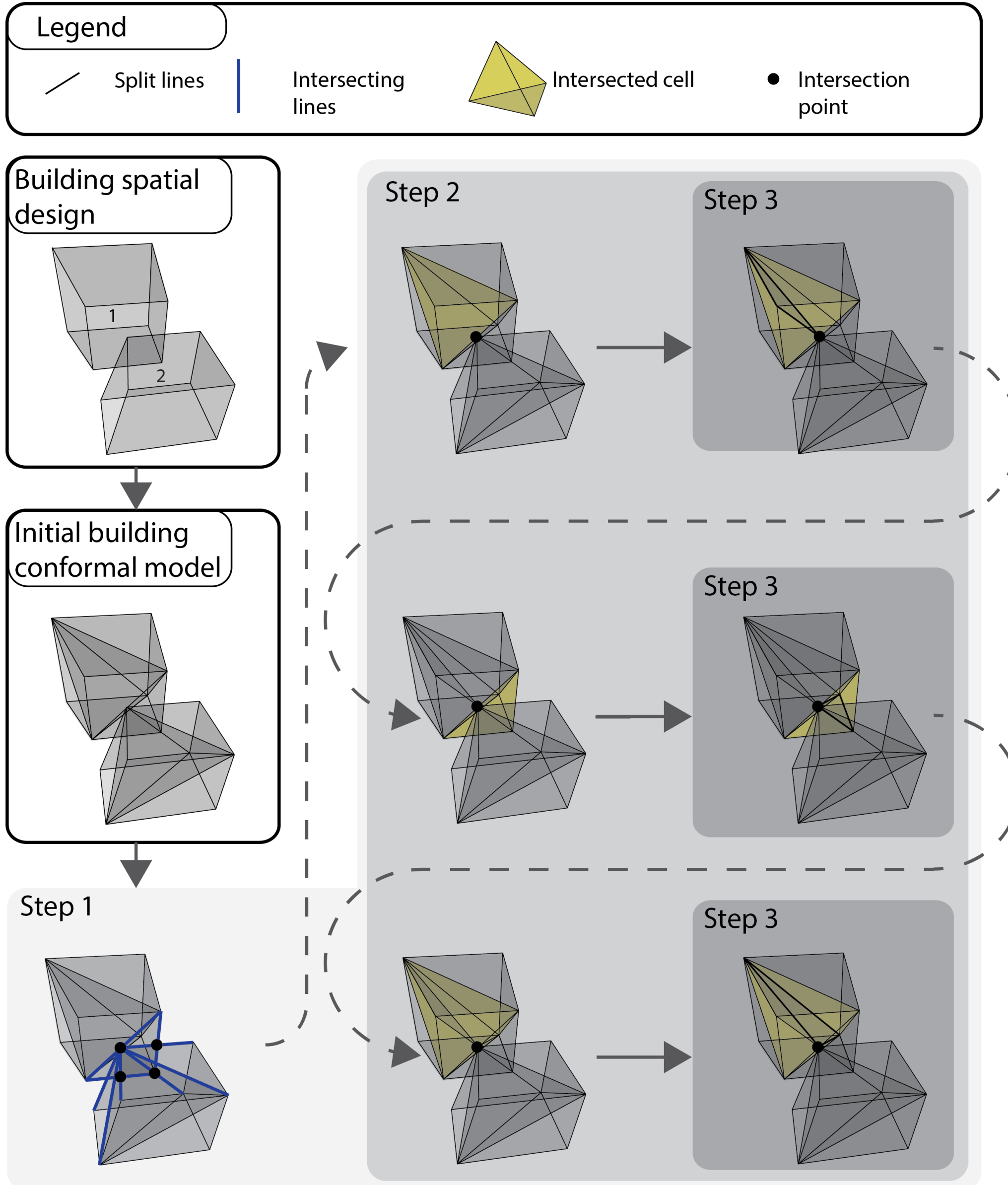


Figure 21: Tetrahedron split process explained on the basis of all cell splits performed for space one according to one intersection vertex

the intersection vertices as well as all the vertices of the already defined cells. Then for each vertex m within the geometry conformal model is step two initiated for each space k. Step two first checks if vertex m is intersecting with any of the cell(s) within space k (see algorithm 5) since this information is not identified within step one and is needed to initiate step three. If vertex m intersects, then step three (the tetrahedron split step) is initiated for each tetrahedron in space k, whereby the input of the single intersection vertex m is considered. This initiation for each tetrahedron in space k is performed since the considered vertex m may intersect multiple tetrahedrons within the same space, as can be seen in figure 21 where three individual tetrahedron are split according to a singular vertex m. Step three is the split function of the tetrahedron cell and adjusts the geometry conformal model including the links with the building conformal model accordingly. This step three will further be elaborated in detail the following paragraph. This third split step may generate some new intersection vertices between the different spaces, which are unaccounted for in the first time performing the step one process. Therefore, all the steps are repeated, if needed multiple times. When all the intersection vertices are solved, the repeat process is stopped and the final geometry conformal model is obtained.

As previously mentioned, step three is the split function of the tetrahedron cell. This split function splits the tetrahedron cell according to either two of the intersection cases, namely, the line intersection or the polygon intersection case as is shown in figure 22. In general, the cell is split by defining the new cells with the help

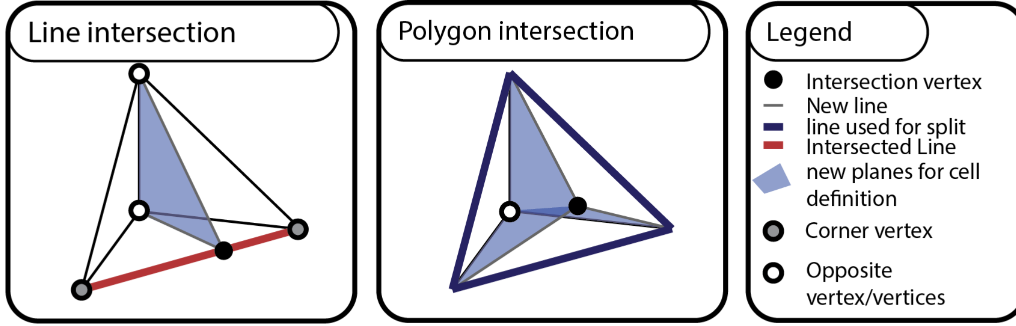


Figure 22: Tetrahedron split cases

of its original corner vertices and the intersection vertex, for which the process is described in algorithm 6. In the line intersection case, the two new cells are defined according to the following corner vertices: the opposite vertices to the intersection line, the intersection vertex, and one of the corner vertices of the intersection line. In the polygon intersection case, the three new cells are defined according to the following corner vertices: the opposite vertex to the intersected polygon, the intersection vertex, and the lines end vertices of one 'line used for split'. After the new cells are defined, the original cell is removed and the new cells are added to the geometry conformal model, thereby finalizing step three.

Algorithm 6 Tetrahedron step 3

```
1: Bool split = false
2: for each polygon m of cell l do                                          ▷ Polygon split
3:     if vertex k is inside polygon m then
4:         Find opposite vertex to intersected polygon
5:         for each line n of intersected polygon m do
6:             plit = true
7:             add intersection vertex k to cornerVertices
8:             add end vertices of line n to cornerVertices
9:             add opposite vertex to intersected polygon m to cornerVertices
10:            generate new tetrahedron cell with vertices in cornerVertices and add to newTetrahedron
11:        end for
12:    end if
13: end for
14: if split = false then
15:    for each line m of cell l do                                          ▷ Line split
16:        if vertex k is on line m then
17:            split = true
18:            find the 2 vertices opposite vertices to the intersected line
19:            for each end vertex n of linem do
20:                add intersection vertex k to cornerVertices
21:                add end vertex n to cornerVertices
22:                add the 2 vertices not connected to the intersected line to cornerVertices
23:                generate new tetrahedron cell with vertices in cornerVertices and add to newTetrahedron
24:            end for
25:        end if
26:    end for
27: end if
28: if split = true then                                 ▷ adjust geometry conformal model
29:    Remove the original cell, triangles, lines and vertices from the geometry conformal model
30:    Add the new cells, triangles, lines and vertices to the geometry conformal model
31: end if
```

4 Demonstration and analysis

In this section, multiple demonstrations will be shown of the developed geometry conformal methods. These demonstrations are chosen from the experience of the writer following from the 83 amount of NOBSD used to check all different functionalities of both geometry conformal methods during its development process (see appendix B for all the designs tested). In section 4.1, the quad-hexahedron method is demonstrated by describing multiple aspects of the quad-hexahedron method on the basis of some examples. These will show the strengths and weaknesses of both sub-methods and the so called triangular polygon challenges. In section 4.2, the tetrahedron method is described on the hand of two examples which shows the process of obtaining a conformal model. After which both methods will be demonstrated on a full building size NOBSD example in section 4.3. These methods will then be compared and analysed in section 4.4 according to the predefined requirements and preferences in section 3.1.

4.1 Quad-hexahedron method

The quad-hexahedron method contains two sub-methods, each of which has its strength and weaknesses. Considering the preference for suitable polygon partitions for the finite element mesh, as well as building like partitions is, in example of figure 23, the following preferred. For space 2, the 'perpendicular to intersected' sub-method is preferred and for space 1 the 'perpendicular to opposite' sub-method is preferred. Therefore would the ideal geometry conformal model exist out of space 1 split by the perpendicular to opposite sub-method and space 2 split by the perpendicular to intersected, resulting in the best distributed partitions sizes as well as having the most building like partitions.

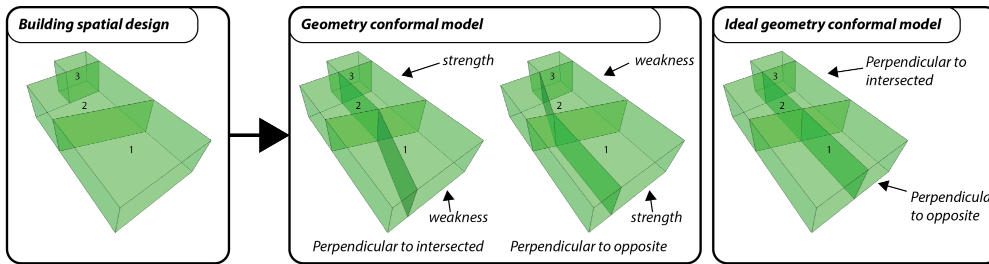


Figure 23: Weaknesses and advantages of both sub-methods within the quad-hexahedron method

In section 3.3.2, the 'check' and 'repeat' processes used in step one were introduced to solve the '*follow-up cells within a space*' and '*follow-up cell between different spaces*' cases. To prove the functionality of these 'check' and 'repeat' processes, both sub-methods are applied to two typical examples of the '*follow-up cells within a space*'(figure 24) and the '*follow-up cell between different spaces*' (figure 25) cases. The '*follow-up cells within a space*' example consists of space 2 till 5 intersecting with a large space 1. In this case the cells of space 2 till 5 first generate multiple cell layers. All these cell layers are then split in accordance to the intersections found between the cell of space 1 with the cell of space 5. These splits have been done through multiple intercell checks with the help of repeat 2, resulting in the fully continued blue indicated surface in figure 24. In the '*follow-up cell between different spaces*' example space 1 and 2 have no intersection with any spaces until space 3 is split in accordance to space 5, followed directly with space 4 being split in accordance to the split space 3 during the first interspace check. Therefore, the blue indicated planes where generated during the second interspace check, resulting in a conformal geometry.

There was a challenge found with triangular polygons on four levels of geometry, as is illustrated in figure 26. Neither the triangular polygons nor the resulting cell(s) of a triangular prism and possibly pentagonal prism spaces is currently supported in the geometry conformal method. Resulting in triangular challenges on the following levels of geometry: a sub-method, a polygon intersection, a multiple cell intersection, and a building spatial design level.

On a sub-method level there are two kinds of cases found. In case 1 (in figure 26), both sub-methods find a new corner vertex outside the boundaries of the opposite polygon, thereby finding a triangular prism and pentagonal prism cell. In case 2 (in figure 26), both sub-methods find a new corner vertex outside the boundaries of the intersected polygon, since the intersected polygon is split the same way in both sub-methods. On a polygon intersection level, the method-defined order in which the polygon intersection vertices are inserted results in some cases in a triangle polygon (see figure 26). This method-defined order is due to the predefined sorted order in which the cell corner-vertex information is stored in the cell entities. On a multiple cell intersection

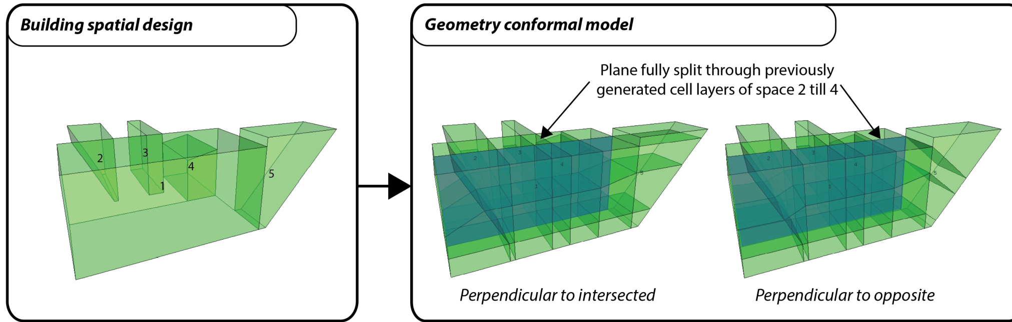


Figure 24: Demonstration of the quad-hexahedron method on a '*follow-up cell between different spaces*' case

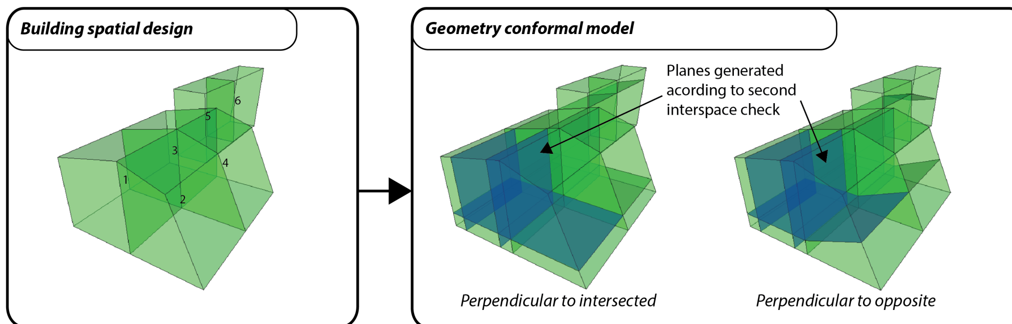


Figure 25: Demonstration of the quad-hexahedron method on a '*follow-up cells within a space*' case

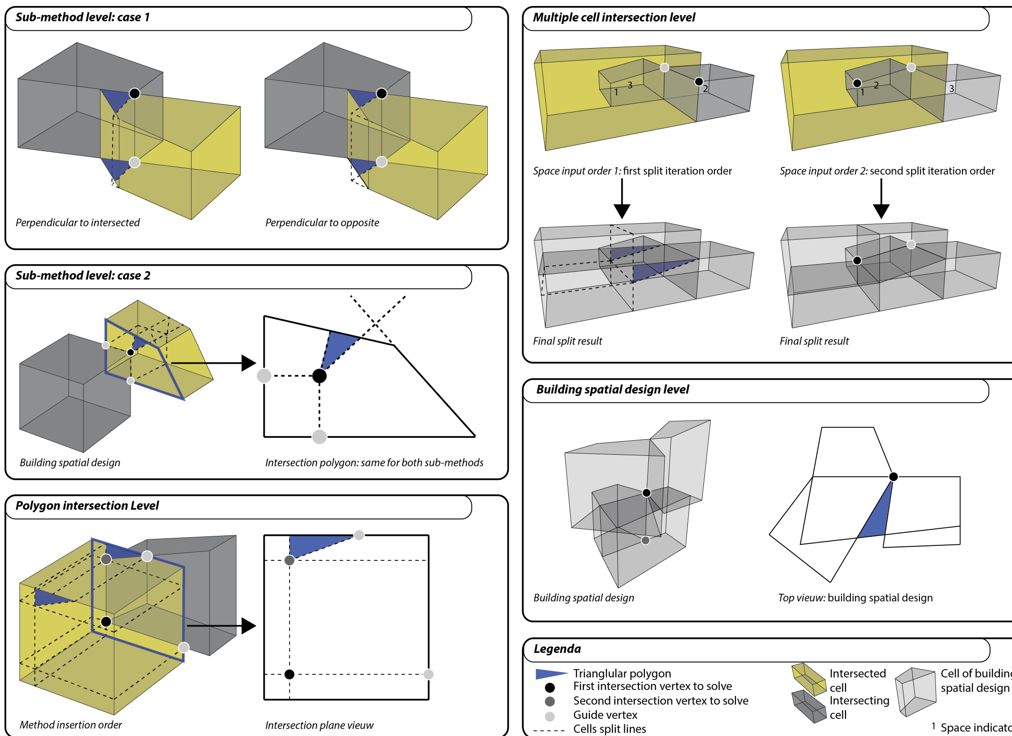


Figure 26: Triangular polygon challenges found with the quad-hexahedron method on four geometry levels depicted in four cases

level a triangular polygon is generated in the user-defined space input order one (see figure 26) and not in the user-defined space input order two (see figure 26). This user-defined space input order is used for the step one 'interspace' check, as was previously explained in section 3.3.2. In case of space input order one, first the cell of space 1 is split on intersections found with the cell of space 2. Followed by the cell of space 1 with the cell of space 3, thereby generating a triangular polygon. In space input order two by solving the intersections between

space 1 and space 2 the intersections between space 3 and 1 are automatically solved. These user-defined space input orders show thereby the influence of the space input order on the resulting geometry conformal model. Although, in this case, the triangular polygon can be avoided, however, there are multiple more complicated cases where this is not possible. An example of such a case can be seen in the last triangular challenge case, namely on a building spatial design level. In this case, the BSD itself already encloses a triangular polygon, thereby determining that it is impossible to obtain a geometry conformal model without dealing with triangular polygons in the quad-hexahedron geometry conformal method.

4.2 Tetrahedron method

Two examples are shown to demonstrate and describe multiple aspects of the tetrahedron method.

The first example shows in which order the split of a single tetrahedron is carried out to obtain a geometry conformal model, as can be seen in figure 27. How this goal is obtained is not necessarily the same for each triangle with the same intersection vertices found, as can be seen in the difference between the lines of the 'non-symmetric triangle' in comparison to the 'starting triangle'. This difference occurs due to the iteration order through the sorted vertices within the tetrahedron entity, since this is sorted differently due to the orientation of both triangles.

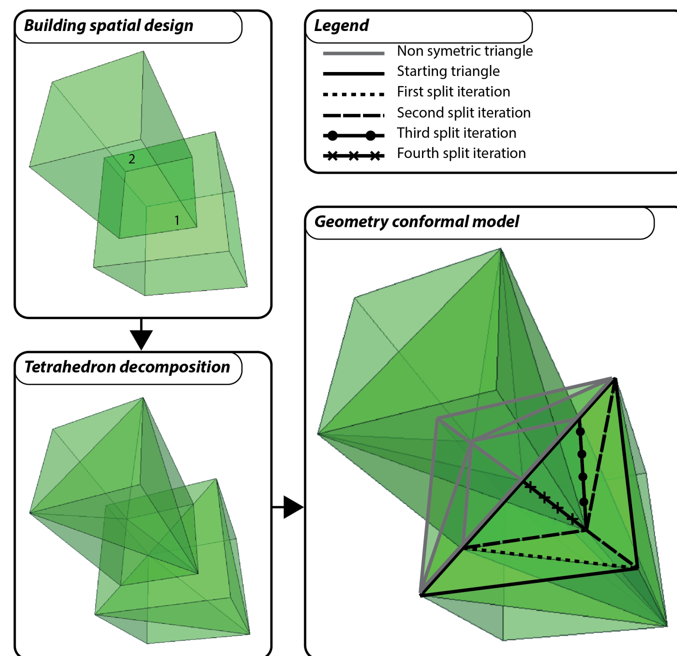


Figure 27: Multiple iteration explained on the basis of a conformal triangle within a full conformal geometry example

The second example seems to be able to obtain a geometry conformal model when performing it in a different manual splitting order that the splitting order performed by the geometry conformal method (see figure 28). The second example will therefore show a manually performed splitting order and a method performed splitting order. This will be shown on the basis of the first split according to the *first four vertices* and the mid stages of the *repeat iterations* as presented in algorithm 4 and as the 'were new intersection found' decision in figure 20. To keep the image as clear as possible only the intersection face between the two cells will be indicated and analysed. At first, there will be looked at the 'method' *first four vertices* inserted in step three example in figure 28. In this case it can be seen that the lines generated by the split according to vertex 2 and 4 intersects. Also, the line generated by vertex 2 intersects with the tetrahedron of space 1. Lastly, the split according to the first 4 vertices makes sure that all the splits according to the rest of the vertices can only generate new intersections. This can be seen in the result of the *first repeat iteration* of the 'method' insertion order. In this *first repeat iteration* can also be seen that vertex 5 does not generate the split from vertex 5 till vertex 4 in space 1, although this should have been done. This is due the removing and adding of tetrahedrons in the considered space, since the considered index tetrahedron is removed. This results in the change of minus one index value of the next tetrahedron to consider, while the iteration mechanism will consider the index+1 tetrahedron next, thereby skipping a tetrahedron entity. This bug happens every time a tetrahedron entity is split. In the 'manual iterated version of the *first four vertices* can be seen that only the split according to vertex 1 generates

an intersection with the tetrahedron of space 1. The rest of the splits performed only result in a conformal geometry, as can be seen in the *first repeat iteration*. Therefore, when the last intersection vertex splits are performed during the *second repeat iteration* is the geometry conformal model obtained, whereas the 'method' insertion order still generates more intersection vertices as can be seen in the results from the *second repeat iteration*. Eventually, the 'method' iteration is stopped at the fourth iteration due to the error of finding all new tetrahedron cell corner vertices on one plane. This error could have two explanations, either all these iterations are performed in a poor iteration order that new intersection vertices are continuously generated, therefore resulting in a non-convergent of the conformal model. Or the third and fourth iteration could be caused by tolerance errors, which may result in the malfunctioning geometry checks having its influences throughout the tetrahedron method steps. Therefore, it can be concluded that a geometry conformal model in this case could be obtained, however, this is not obtained in the tetrahedron method due to the mentioned reasons.

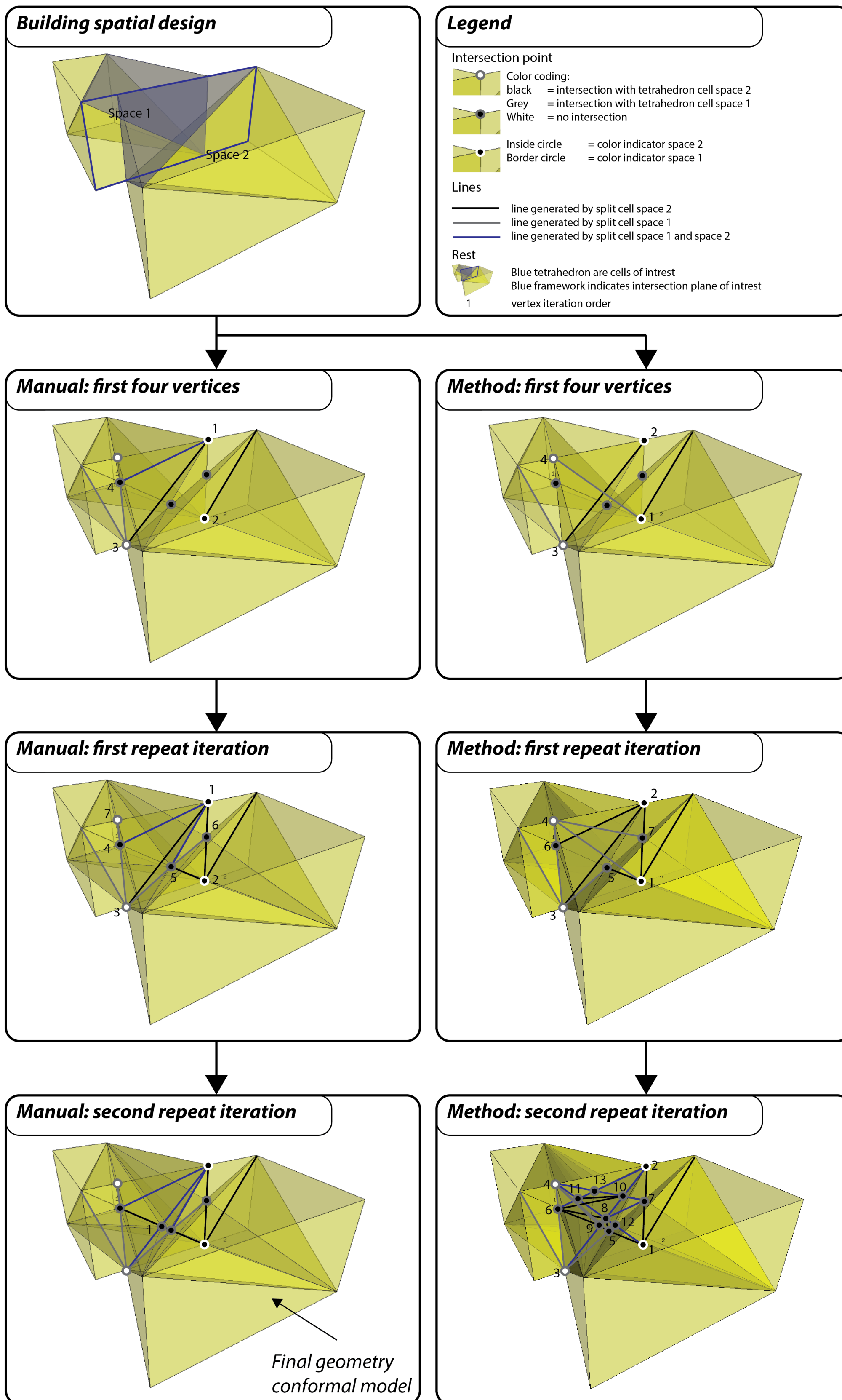


Figure 28: Tetrahedron iteration example for which the 'manual' method iterations are compared

4.3 non-orthogonal BSD example

The following example is developed (see figure 30) to prove that a full building size NOBSD can be made into a conformal model by the quad-hexahedron method. This example is inspired by the 'ZAC maséna' building in Paris (Archi Tonic, n.d.), but is adapted to fit the current triangular challenges restrictions and avoid tolerance issues of the quad-hexahedron method. The adaptation made concise of no protrusions starting from the middle of the main towers side, no terraces at back of the building and a rectangular based building as is presented in figure 29. Furthermore, the BSD is transformed to consist of quad-hexahedron NOBSD shapes, therefore no recessed balconies. The right tower (tower B) has to solve only line-line split cases. The left tower (tower A) mostly considers polygon-line split cases, wherein the rectangular base conecting both towers show both the intercell and interspace check. this full building example is therefore a good representation of all the check mechanisms within the quad-hexahedron method. The tetrahedron method was unable to split this full building example due to the error of finding all new tetrahedron cell corner vertices on one plane, as was also seen in the second iteration example discussed in section 4.2.

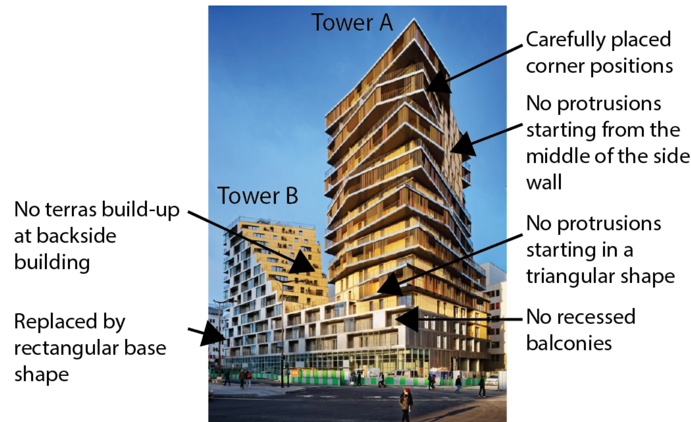


Figure 29: 'ZAC maséna' building in Paris with marked adaptations to fit the current triangular challenges restrictions, to avoid tolerance issues, and to fit the quad-hexahedron BSD build-up (Archi Tonic, n.d.)

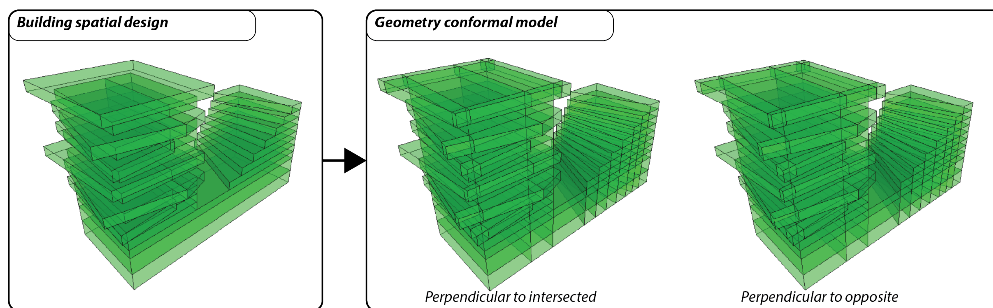


Figure 30: Full building scale BSD solved with both sub-method of the quad-hexahedron method

4.4 Comparison

To compare both geometry conformal methods, the previous defined requirements and preferences in section 3.1 will be considered.

The first and most important requirement consists of generating a conformal model, for which it is preferred to generate a conformal model for as many, if not all, quad-hexahedron BSD. The quad-hexahedron method can do so for a large scope of NOBSD, only being geometrically restricted by the triangular challenges previously explained in section 4.1. The BSD scope of the tetrahedron method is limited due to the not optimized split iteration order of the intersection vertices and possibly tolerance issues, as previously discussed in section 4.2. Considering both methods in their current form, it can therefore be concluded that the quad-hexahedron method can solve a larger scope of NOBSDs.

The second requirement is that the geometry conformal method applies to other building spatial design shapes, consisting of non-orthogonal triangular prism and a non-orthogonal pentagonal prism. The quad-hexahedron method could satisfy this requirement by the addition of a similar step three as in section 3.3.2 for the triangular prism cells. In addition, the pentagonal prism cell inserted or generated should be decomposed in a triangular prism and quad-hexahedron cell. This will assure a suiting step three splitting method and a suiting meshing technique for the faces of the cell. The tetrahedron method could allow for a triangular prism and a pentagonal prism space by defining decomposition functions, just like it has been done for the quad-hexahedron NOBSD as explained in section 3.4. Concluding that both geometry conformal methods could be applied to triangular prisms and pentagonal prisms.

The third requirement for the geometry conformal methods is that, for the same method and sub-method, the inserted building spatial design always results in the same conformal model. This is the case for both methods if the spaces within the BSD input are inserted in the same order. On the other hand, the insertion order of the corner vertices within a single space definition input is allowed to vary in order since, in the rest of the steps, all the iteration orders are regulated by the storing order of vertices, lines, rectangles, triangles, quad-hexahedron, and tetrahedrons within the entities. This regulation by sorting could also be applied to the inserted space input in the future, thereby regulating the insertion order making the input order of spaces no factor in the resulting conformal model.

The first preferences for the geometry conformal model is to result in building like partitions. In general, the resulting partitions from the quad-hexahedron method are more building like then the partitions resulting from the tetrahedron method, since the quad-hexahedron geometry elements are more often seen in building practices then the tetrahedron geometry element. Therefore, it can be concluded that the quad-hexahedron method results in building like partitions more often.

The second preference for the geometry conformal model, from the structural discipline-specific model, is to have resulting polygon partitions that are suitable for the local meshing process. If the BSD inserted does not consist of intersection vertices close to edges of polygons or on ends of lines, then the quad-hexahedron methods result mostly in constant partition sizes, as can been seen in the previously shown demonstrations. This isn't necessarily the case for the tetrahedron method, as shown in figure 28. Furthermore, both methods have their advantages and disadvantages considering the preference for close to convex polygon partitions. The quad-hexahedron method generally results in a conformal model of convex nature. Unless the inserted BSD has an intersection plane that is non-convex like in the triangular challenge in figure 26 case 4. In the tetrahedron method, the triangular polygon of the cells is already of convex nature, therefore are non-convex cells non-existing. But the triangular faces can tend toward a very narrow triangle, making the triangular face not so close to a convex region (all 60-degree angles). Taking this analysis into consideration, it can in general be concluded that the quad-hexahedron method results more often in a suitable polygon partition for the local meshing process.

The third preference for the geometry conformal method is to be of short process time. This has not yet been analysed since the focus of this research consist of making the geometry conformal methods work for the full scope of NOBSD, then analysing the process time.

The last preference is a clear and structured conformal model for reviewing the results of discipline-specific models. In general, the quad-hexahedron method is more structured and easier to follow how the conformal model is build-up due to less amount of cells. Also, the lines within the quad-hexahedron method switch less from cell to cell and in lower degrees.

Considering all these preferences and requirements, the quad-hexahedron method, in its current form, is the preferred method since it can solve a larger scope of NOBSD, have the most building like partitions, results in a clear and structured geometry conformal model, and results most often in optimal partitions for mesh generation.

5 Discussion

In this paper, two newly developed geometry conformal methods are presented to obtain a conformal model for NOBSD. The first, the quad-hexahedron method, splits the quad-hexahedron BSD in quad-hexahedron geometry entities via one of the two sub-methods. The second, the tetrahedron method, splits the quad-hexahedron BSD in tetrahedron geometry elements. In this section, some critical remarks on the developed methods and the presented work are given.

Firstly, the quad-hexahedron method is unable to obtain a conformal model for all possible NOBSDs, since this method is restricted by the triangular challenges previously explained in section 4.1 and figure 26. However, it seems that most of these challenges can be solved for vertical walls and horizontal floors by adding a triangular prism and pentagonal prism with the pentagonal prism decomposition function and a triangular

prism split function as explained in section 4.4. This way, the BSD scope is still significantly increased compared to the original orthogonal rectangular BSD scope. However, it is very difficult to oversee the full NOBSD scope. Therefore, specific cases could occur which are unsupported by the geometry entities represented in the quad-hexahedron method. One way to minimize the change on these specific cases and to enlarge the NOBSD scope would consist of the following. At first more sub-methods to split the quad-hexahedron cells should be added. Combining these additional sub-methods with a grammar would optimize which sub-method to use at which splitting action. Thereby avoiding the limitations of each individual sub-method, as well as resulting in more favorable geometry conformal models containing more building like partitions, equally sized partitions, and closer to convex partitions, as was previously stated as a preference in section 3.1.

Secondly, the tetrahedron method is unable to obtain a conformal model for all possible NOBSDs due to a sub-optimal iteration order, which was previously explained in section 4.2 and figure 28. As a result, continual new intersection vertices where generated, resulting in non-convergence of the geometry conformal model. An initial idea to solve this problem would be to regulate the iteration order. However, it is unsure if this new iteration order will solve all the cases in which new intersection vertices may be generated since at least some partitions with newly generated intersection vertices are needed to obtain a conformal model, as can been seen in the manual iterated version of figure 28. In addition, it is currently unsure what will happen when more than 2 spaces are considered since an extensive analysis of this has not yet been made. For example, the split of the first space in accordance to the second space might generate new intersection on the third space, which in turn might generate new intersection on the first split space. Thereby resulting in a non-converting loop between different spaces. So when the iteration order between the split of two separate spaces are solve, it might still result in iteration order problems between multiple spaces. In case these insertion order problems are solves for all cases, then the tetrahedron method could be a promising method applicable to all NOBSDs consisting of tetrahedron build-up. Thereby having a larger scope of possible NOBSD then the quad-hexahedron method could support.

Thirdly, both methods may have tolerance issues due to round-of-errors and in some specific cases wrongly performed geometry checks. Within the quad-hexahedron method, these cases has been identified in multiple cases from which some are listed in appendix C. One of these examples is a line intersection wrongly identified as a polygon intersection by the line-polygon check, where-after a polygon split is executed instead of a line split. Within the tetrahedron method, no such specific examples have been found yet, although the same intersection vertex definition and geometry checks are performed within the tetrahedron method.

Fourthly, not all partitions generated in these geometry conformal models will always result in suitable partitions for mesh generation as has been shown in figure 23, figure 28, section 4.1, and section 4.2. For this reasons, some adaptations are proposed in the following. A adaptations already proposed for the quad-hexahedron method consist of a grammar to distinguishes which sub-method is best. This can be used to improve the suitable polygon partitions for mesh generation when using the requirement of equally sized partitions, and close to convex partitions. Within the tetrahedron method, the initial decomposition in tetrahedrons could be improved to reduce the amount of initial intersections related to the decomposition of the quad-hexahedron space. Also, the resulting geometry conformal model could be post-processed by partitioning the bigger partition to match the size of the smaller partitions.

Fifthly, not all possible quad-hexahedron NOBSDs are tested due to the large scope considered and the preference for further development before extensively testing. This further development would consist of solving the triangular challenges in the quad-hexahedron method, tolerance issues in both methods, and the not-optimized split iteration order of the intersection vertices in the tetrahedron method. It is expected that these would change some of the conclusions made on the current functionalities regarding the following requirement and preferences: (1) generating a conformal model for quad-hexahedron NOBSD, (2) applicability to a wider NOBSD scope, and (3) fastest processing time. Nonetheless, the already made conclusions about building like partitions, preference for clear and structured conformal models, and suitable partitions for mesh generation, despite the possible improvements, are not expected to change with further research due to the intrinsic qualities of the geometry entities used. Therefore, regarding the not expected to change requirement and preferences, the quad-hexahedron method is still the preferred method. However, the quad-hexahedron method will not be able to address tetrahedron NOBSD unless a hybrid between the two methods is made. Therefore, it might still be interesting to develop the tetrahedron method further since the possible scope of all NOBSD consisting of tetrahedron entities is larger and more non-orthogonal then the quad-hexahedron method could support.

Sixthly, within this research only two kinds of geometry conformal methods are considered, while there are more kinds possible. Some examples of this are already proposed, like the geometry conformal method considering the triangular prism. However, more geometry entities are also possible. In addition, a conformal model may also be reached by other methods, as already presented in the literature section 2.2 where some conformal meshing methods where described.

The above remarks shows that future work on the geometry conformal methods is needed. However, this

does not discount the promise of both method that has already been observed. The quad-hexahedron method, in its current form, is already able to generate a conformal model for a certain scope of NOBSD. In addition, it seams to be able to make suitable geometry conformal models for the defined use-cases. Furthermore, in case the iterator order problem is solve, the tetrahedron method might be able to make a conformal model for a larger and more non-orthogonal BSD scope then the quad-hexahedron method could support. This would result in two geometry conformal methods, each applicable in their own field of strength.

6 Conclusions

Two geometry conformal methods have been presented that generate a conformal model for the use in a design and optimisation framework, which transforms a quad-hexahedron NOBSD to multiple discipline-specific models. This conformal model has multiple use-cases regarding discipline-specific models definitions, such as element connection and intersection problems, definition of properties and loads, and it allows for the grouping of sub-parts (cells) of spaces into zones for spatial layouts that are more logical from a disciplines point of view.

The first method, the quad-hexahedron method, initiates a conformal model by first decomposing NOBSD in quad-hexahedron geometry entities, where after an iterative adapt phase is initiated. This adapt phase consist of a nested steps one till three, which are as follows. Step one finds the intersection lines and accompanying intersection vertices. Step two makes, with the help of the intersection lines, a selection of vertices for each intersection vertex found in step one to guide one single splitting action of step three. Step three splits the quad-hexahedron according to the selection made in step two and either one of the sub-method used for all splitting actions. The sub-methods are named the 'perpendicular to opposite' and the 'perpendicular to intersected' sub-method. At the end of the adapt phase, the conformal model is obtained.

The second method, the tetrahedron method, initiates a conformal model by first decomposing NOBSD in tetrahedron geometry entities, where after an iterative adapt phase is initiated. This adapt phase consist of a nested steps one till three, which are as follows. Step one finds all the intersection vertices, then initiates step two per vertix within the geometry conformal model for each space in the building conformal model. Step two then checks if the intersection is already solved in the previous iterations, if not, then step two sends the vertex and the according cell to step three. Step three then splits the tetrahedron cell. After multiple iteration of all the steps a conformal model is obtained, which consisting of only tetrahedron cells.

Both methods have been demonstrated, analysed and compared according to previously defined requirements and preferences. The demonstration contained some typical examples to show the aspects of both methods. For the quad-hexahedron method, these were the strengths and weaknesses of the sub-methods and the triangular challenges. For the tetrahedron method, these were the non-convergence due to continues intersection vertex generation in the splitting action and a demonstration on how a geometry conformal model is obtained through multiple iterations. These aspects of both methods were discovered during the testing of 83 designs over the course of the development of the geometry conformal methods. In addition, a full building size NOBSD has been demonstrated where after the methods were compared according to the requirements and preferences.

Regarding the requirements and current functionalities of both methods, the following conclusions can be made. Both geometry conformal methods may not always result in a conformal model. In case of the quad-hexahedron method, this is due to the triangular challenges. In case of the tetrahedron method, this is due to the non-convergence of the conformal model, resulting from continual intersection vertex generation due to the non-optimal iteration order. However, considering both methods in their current it form, it can be concluded that the quad-hexahedron method can solve a larger scope of NOBSDs. Furthermore, both geometry conformal methods may be able to allow for a triangular prism and a pentagonal prism with a few adjustments. The last requirement is that the inserted building spatial design, with the same method, always results in the same conformal model. This is the case for both methods if the spaces within the BSD input are inserted in the same order.

Regarding the preferences, the following conclusions can be made. The quad-hexahedron method in comparison to the tetrahedron method results in more building like partitions and results more often in a suitable polygon partition for a (future) local meshing process. No conclusions has been made regarding the preference for short process time, since the focus within this research contained the functioning of both geometry conformal methods for all NOBSD. Lastly, the quad-hexahedron method results in a more clear and structured conformal model for the reviewing of the results.

Considering all the preferences, requirements, and both methods in their current form the following conclusions can be made. The quad-hexahedron method is the preferred method. However, further research is needed since significant improvement is still expected to enlarge the applicability scope of the quad-hexahedron method. Furthermore, the tetrahedron method could be interesting for the NOBSD cases in which the quad-hexahedron method is not applicable, such as the NOBSDs build-up from tetrahedron geometry entities. However, only in

the case the iteration order challenge is solved. Resulting in two geometry conformal methods each applicable in their own field of strength.

Acknowledgements

I would like to express my gratitude towards professor H. Hofmeyer for the guidance, availability, and enthusiasm throughout the supervisions of the graduation project. For the same efforts, I would like to thanks S. Boonstra, with additional thanks for the continued supervisions after finishing his PhD and starting his following carrier path. The member of the graduation committee P. Pauwels is thanked for his enthusiasm and short notice availability for the final-colloquium. Furthermore, this work would not have been possible without all the previous performed efforts on the Toolbox. Therefore, a special thanks is given to all the contributors. Lastly, a special thanks is given to my family, friends, and boyfriend Victor Hoeksema for the continual support throughout the graudation project, as wel as my father Jan Ezendam and boyfriend Victor Hoeksema for reviewing my graduation thesis.

References

Anderson, J. E., Wulfhorst, G., & Lang, W. (2015). Energy analysis of the built environment - A review and outlook. *Renewable and Sustainable Energy Reviews*, *44*, 149–158. https://doi.org/10.1016/j.rser.2014.12.027

Archi Tonic. (n.d.). Batiment Home, Zac Masséna, Paris XIII. https://www.architonic.com/en/project/hamonic-masson-associes-batiment-home-zac-massena-paris-xiii/5103555

Baldock, R., & Shea, K. (2006). Structural Topology Optimization of Braced Steel Frameworks Using Genetic Programming. In I. F. C. Smith (Ed.), *Intelligent computing in engineering and architecture* (pp. 54–61). Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/11888598_6

Bols, J., Taelman, L., De Santis, G., Degroote, J., Verhegghe, B., Segers, P., & Vierendeels, J. (2016). Unstructured hexahedral mesh generation of complex vascular trees using a multi-block grid-based approach. *Computer Methods in Biomechanics and Biomedical Engineering*, *19*(6). https://doi.org/10.1080/10255842.2015.1058925

Boonstra, S., & Hofmeyer, H. (2020). TUe-excellent-buildings/BSO-toolbox: Official release of the BSO toolbox. https://doi.org/10.5281/zenodo.3924556

Boonstra, S., van der Blom, K., Hofmeyer, H., & Emmerich, M. T. (2020). Conceptual structural system layouts via design response grammars and evolutionary algorithms. *Automation in Construction*, *116*. https://doi.org/10.1016/j.autcon.2019.103009

Boonstra, S., van der Blom, K., Hofmeyer, H., Emmerich, M. T., van Schijndel, J., & de Wilde, P. (2018). Toolbox for super-structured and super-structure free multi-disciplinary building spatial design optimisation. *Advanced Engineering Informatics*, *36*, 86–100. https://doi.org/10.1016/j.aei.2018.01.003

Caldas, L. (2008). Generation of energy-efficient architecture solutions applying GENE_ARCH: An evolution-based generative design system. *Advanced Engineering Informatics*, *22*(1), 59–70. https://doi.org/10.1016/j.aei.2007.08.012

Chen, H., Lu, Z., & Guo, T. (2017). A Hybrid Dynamic Mesh Generation Method for Multi-Block Structured Grid. *Advances in Applied Mathematics and Mechanics*, *9*(4). https://doi.org/10.4208/aamm.2016.m1423

Claessens, D. P., Boonstra, S., & Hofmeyer, H. (2020). Spatial zoning for better structural topology design and performance. *Advanced Engineering Informatics*, *46*. https://doi.org/10.1016/j.aei.2020.101162

Dino, I. G., & Üçoluk, G. (2017). Multiobjective Design Optimization of Building Space Layout, Energy, and Daylighting Performance. *Journal of Computing in Civil Engineering*, *31*(5), 04017025. https://doi.org/10.1061/(asce)cp.1943-5487.0000669

European Commission. (2005). Challenging and Changing Europe's Built Environment: A vision for a sustainable and competitive construction sector by 2030. www.ectp.org

Fogg, H. J., Armstrong, C. G., & Robinson, T. T. (2015). Automatic generation of multiblock decompositions of surfaces. *International Journal for Numerical Methods in Engineering*, *101*(13). https://doi.org/10.1002/nme.4825

Frey, P. J., & George, P.-L. (2000). *Mesh Generation*. HERMES Science Publishing.

Geyer, P. (2008). Multidisciplinary grammars supporting design optimization of buildings. *Research in Engineering Design*, *18*, 197–216. https://doi.org/10.1007/s00163-007-0038-6

Granadeiro, V., Duarte, J. P., Correia, J. R., & Leal, V. M. (2013). Building envelope shape design in early stages of the design process: Integrating architectural design systems and energy simulation. *Automation in Construction*, *32*, 196–209. https://doi.org/10.1016/j.autcon.2012.12.003

Hofmeyer, H., Roosmalen, M. V., & Gelbal, F. (2011). Pre-processing parallel and orthogonally positioned structural design elements to be used within the finite element method. *Advanced Engineering Informatics*, *25*(2), 245–258. https://doi.org/10.1016/j.aei.2010.06.004

Hofmeyer, H., & Russell, P. (2009). Interaction between spatial and structural building design: A finite element based program for the analysis of kinematically indeterminable structural topologies. In X. Wang & N. Gu (Eds.), *Convr2009, proceedings of the 9th international conference on construction applications of virtual reality, november 5-6, 2009* (pp. 247–256). University of Sydney.

Hofmeyer, H., & Davila Delgado, J. M. (2015). Coevolutionary and genetic algorithm based building spatial and structural design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, *29*, 351–370. https://doi.org/10.1017/S0890060415000384

Huang, C.-Y. (1997). Recent progress in multiblock hybrid structured and unstructured mesh generation. *Computer Methods in Applied Mechanics and Engineering*, *150*(1-4), 1–24. https://doi.org/10.1016/S0045-7825(97)00105-9

Klimaatakkoord (tech. rep.). (2019). Den Haag. https://www.klimaatakkoord.nl/binaries/klimaatakkoord/documenten/publicaties/2019/06/28/klimaatakkoord/klimaatakkoord.pdf

Kowalski, N., Ledoux, F., & Frey, P. (2015). Automatic domain partitioning for quadrilateral meshing with line constraints. *Engineering with Computers*, *31*, 405–421. https://doi.org/10.1007/s00366-014-0387-5

Li, Z., Chen, H., Lin, B., & Zhu, Y. (2018). Fast bidirectional building performance optimization at the early design stage. *Building Simulation*, *11*, 647–611. https://doi.org/10.1007/s12273-018-0432-1

Liang, Q. Q., Xie, Y. M., & Steven, G. P. (2000). Optimal Topology Design of Bracing Systems for Multistory Steel Frames. *Journal of Structural Engineering*, *126*(7), 823–829. https://doi.org/10.1061/(ASCE)0733-9445(2000)126:7(823)

Lin, S. H. E., & Gerber, D. J. (2014). Designing-in performance: A framework for evolutionary energy performance feedback in early stage design. *Automation in Construction*, *38*, 59–73. https://doi.org/10.1016/j.autcon.2013.10.007

Lo, S. H. (1995). Automatic mesh generation over intersecting surfaces. *International Journal for Numerical Methods in Engineering*, *38*(6), 943–954. https://doi.org/10.1002/nme.1620380605

Lu, F., Pang, Y., Jiang, X., Sun, J., Huang, Y., Wang, Z., & Ju, J. (2018). Automatic generation of structured multiblock boundary layer mesh for aircrafts. *Advances in Engineering Software*, *115*, 297–313. https://doi.org/10.1016/j.advengsoft.2017.10.003

Nguyen-Van-Phai. (1982). Automatic mesh generation with tetrahedron elements. *International Journal for Numerical Methods in Engineering*, *18*(2), 273–289. https://doi.org/10.1002/nme.1620180209

Nicolas, G., & Fouquet, T. (2013). Adaptive mesh refinement for conformal hexahedralmeshes. *Finite Elements in Analysis and Design*, *67*, 1–12. https://doi.org/10.1016/j.finel.2012.11.008

Oh, H. S., & Lim, J. K. (1996). Modified r-method for the finite element adaptive analysis of plane elastic problems. *KSME Journal*, *10*, 190–202. https://doi.org/10.1007/BF02953658

Paris Agreement (tech. rep.). (2015). https://unfccc.int/files/essential_background/convention/application/pdf/english_paris_agreement.pdf

Sharafi, P., Teh, L. H., & Hadi, M. N. (2015). Conceptual design optimization of rectilinear building frames: A knapsack problem approach. *Engineering Optimization*, *47*(10), 1303–1323. https://doi.org/10.1080/0305215X.2014.963068

Sigmund, O. (2001). A 99 line topology optimization code written in Matlab. *Struct Multidisc Optim*, *21*, 120–127. https://doi.org/https://doi.org/10.1007/s001580050176

Smith, R. E., & Eriksson, L.-E. (1987). Algebraic grid generation. *Computer Methods in Applied Mechanics and Engineering*, *64*(1-3), 285–300. https://doi.org/10.1016/0045-7825(87)90044-2

Smulders, C., & Hofmeyer, H. (2012). An automated stabilisation method for spatial to structural design transformations. *Advanced Engineering Informatics*, *26*(4), 691–704. https://doi.org/10.1016/j.aei.2012.03.002

Staten, M. L., Shepherd, J. F., Ledoux, F., & Shimada, K. (2010). Hexahedral Mesh Matching: Converting non-conforming hexahedral-to-hexahedral interfaces into conforming interfaces. *International Journal for Numerical Methods in Engineering*, *82*(12), 1475–1509. https://doi.org/10.1002/nme.2800

Tugilimana, A., Thrall, A. P., Descamps, B., & Coelho, R. F. (2017). Spatial orientation and topology optimization of modular trusses. *Structural and Multidisciplinary Optimization*, *55*, 459–476. https://doi.org/10.1007/s00158-016-1501-7

van der Blom, K., Boonstra, S., Hofmeyer, H., Back, T., & Emmerich, M. T. (2017). Configuring advanced evolutionary algorithms for multicriteria building spatial design optimisation. *In 2017 IEEE Congress*

on Evolutionary Computation (CEC) Porceeding, june 5-8,2017, Donostia - San sebastián, Span, 1803–1810. https://doi.org/10.1109/CEC.2017.7969520

van der Blom, K., Boonstra, S., Hofmeyer, H., & Emmerich, M. T. M. (2016). Multicriteria Building Spatial Design with Mixed Integer Evolutionary Algorithms. In J. Handl, E. Hart, P. R. Lewis, M. O. G. López-Ibáñez, & B. Paechter (Eds.), *Parallel problem solving from nature ppsn xiv. ppsn* (pp. 453–462). Springer. https://doi.org/https://doi.org/10.1007/978-3-319-45823-6{_}42

Vassberg, J. C. (2000). Multi-block mesh extrusion driven by a globally elliptic system. *International Journal for Numerical Methods in Engineering*, *49*, 3–15.

Wang, L., Shen, W., Xie, H., Neelamkavil, J., & Pardasani, A. (2002). Collaborative conceptual design - state of the art and future trends. *Comput. Aided Des.*, *34*, 981–996. https://www.semanticscholar.org/paper/Collaborative-conceptual-design-state-of-the-art-Wang-Shen/6125fb7b51d57c894c73e724f1be06999ba00fa9

Yi, Y. K., & Malkawi, A. M. (2009). Optimizing building form for energy performance based on hierarchical geometry relation. *Automation in Construction*, *18*, 825–833. https://doi.org/10.1016/j.autcon.2009.03.006

A C++ code

The code of the two geometry conformal methods is made in an already developed framework named the BSO toolbox. Therefore, in this sections, only the files which are adjusted for the development of the two NOBSD geometry conformal methods will be shown and briefly explained in this introduction of the C++ code appendix. To understand and run the full toolbox framework is referred to the BSO toolbox code (Boonstra and Hofmeyer, 2020). To run the geometry conformal methods should be looked at the 'main' code. This file represent the commands needed to initiate a building spatial design model, which is then used to initiate the geometry conformal method to result in a conformal model. The input file 'design_1' consist the following information per space: the input method N, the space ID, an the coordinates of eight vertices to define a quad-hexahedron space. An example of such a file is given in figure 31.

Before the two geometry conformal methods where developed, it was first needed to allow for NOBSD inputs in the toolbox. Adjustments for this purpose were made in the 'ms_building' code, which represent the building spatial design model shown in figure 4. Furthermore, the code of the two geometry conformal methods are as follows dispersed over the numerous code files. In the 'cf_building_model' code is the conformal model initiated, containing both the building conformal model and the geometry conformal model which is inherited from the 'cf_geometry_model' code. Furthermore, the step one process of both geometry conformal methods are defined within this 'cf_building_model' code. The 'cf_entity' class is developed to be inherited by all the geometry entities to properly store all the geometric entities within the rest of the conformal model code. This cf_entity class is inherited through the class called the cf_geometry_entity, which is not included in this annex since it was not adjusted. The step two procedure of both geometry conformal methods can be found in the 'cf_space' code. The step three process of the quad-hexahedron geometry conformal method can be found in the 'cf_cuboid' code. The step three process of the tetrahedron geometry conformal method can be found in the 'cf_tetrahedron' code. Furthermore, the 'cf_rectangle', 'cf_triangle', and 'cf_line' code present the code for the polygons (rectangle), triangles, and lines split according to the in step three performed split procedure. Therefore, these codes describe the conformal model adjustments code for conserving the conformal model links presented in figure 8 and 11. Finally, the code in 'cf_building' is part of the visualisation packages used in the toolbox. In this section the tetrahedron en triangle entities are added to allow for the visualisation of these geometric entities.

```
1  N,  1,  2000,4000,3000,     6000,4000,3000,     4000,8000,3000,
   2000,8000,3000,     2000,4000,6000,     6000,4000,6000,     4000,8000,6000,
   2000,8000,6000
2  N,  2,  2000,0,3000,        8000,0,3000,        8000,4000,3000,
   3000,4000,3000,     2000,0,6000,        8000,0,6000,    8000,4000,6000,
   3000,4000,6000
3  N,  3,  6000,4000,3000,     12000,4000,3000,       13000,6000,3000,
   5000,6000,3000,     6000,4000,8000,     12000,4000,8000,
   13000,6000,8000,    5000,6000,8000
```

Figure 31: An example of the content of an input file 'design_1' for the running of the main file

A.1 main

```
1   #include <iostream>
2   #include <string>
3
4   #include <bso/spatial_design/ms_building.hpp>
5   #include <bso/spatial_design/cf_building.hpp>
6   #include <bso/structural_design/sd_model.hpp>
7   #include <bso/building_physics/bp_model.hpp>
8   #include <bso/grammar/grammar.hpp>
9   #include <bso/visualization/visualization.hpp>
10
11  int end, begin = 0;
12  template<class T>
13  void out(const T& t, const bool& e = false,
14                    const bool& i = false, const bool& verbose = false){
15      if (!verbose) return;
16      std::cout << t;
17      end = clock();
18      if (i) std::cout << " (" << 1000*(end-begin)/CLOCKS_PER_SEC << " ms)";
19      if (e) std::cout << std::endl;
20      begin = end;
21  } // out()
22
23  using namespace bso;
24  int main(int argc, char* argv[])
25  {
26      // Generate building spatial design model with input file 'design_1'
27      bso::spatial_design::ms_building MS1("design_1");
28      out("Created an MS model",true,true,true);
29
30      // select the geometry conformal method, including the used sub-method
31      // I = perpendicular to intersection side
32      // O = perpendicular to oposide side
33      // T = tetrahedron method
34      // Generate geometry conformal model
35      bso::spatial_design::cf_building CF1(MS1, 'I', 0.001);
36      bso::spatial_design::cf_building CF2(MS1, 'O', 0.001);
37      bso::spatial_design::cf_building CF3(MS1, 'T', 0.001);
38      out("Created an CF1 model",true,true,true);
39
40      // visualize building spation design model and geometry conformal model
41      bso::visualization::initVisualization(argc, argv);
42      bso::visualization::visualize(MS1);
43      bso::visualization::visualize(CF1,"rectangle");
44      bso::visualization::visualize(CF1,"cuboid");
45      bso::visualization::visualize(CF2,"rectangle");
46      bso::visualization::visualize(CF2,"cuboid");
47      bso::visualization::visualize(CF3,"triangle");
48      bso::visualization::visualize(CF3,"tetrahedron");
49      bso::visualization::endVisualization();
50
51      return 0;
52  }
```

A.2 ms_building

```
1   #ifndef MS_BUILDING_HPP
2   #define MS_BUILDING_HPP
3
4   #include <vector>
5   #include <map>
6   #include <utility>
7   #include <bso/spatial_design/ms_space.hpp>
8   #include <bso/spatial_design/sc_building.hpp>
9
10  namespace bso { namespace spatial_design {
11
12  class ms_building
13  {
14  private:
15      std::vector<ms_space*> mSpaces;
16      mutable unsigned int mLastSpaceID;
17      void checkValidity() const;
18  public:
19      ms_building(); // empty constructor
20      ms_building(std::string fileName); // initilization by string or text
        file
21      ms_building(const ms_building& rhs); // copy constructor
22      ms_building(const sc_building& sc); // convert SC to MS
23      ~ms_building(); // destructor
24
25      void writeToFile(std::string fileName) const;
26      std::vector<ms_space*> getSpacePtrs() const;
27      ms_space* getSpacePtr(const ms_space& space) const;
28      ms_space* getSpacePtr(const ms_space* spacePtr) const; // the spacePtr
        that was passed may belong to another instance of ms_building
29      unsigned int getLastSpaceID() const;
30      double getVolume() const;
31      double getFloorArea() const;
32
33      std::vector<ms_space*> selectSpacesGeometrically(
34          const bso::utilities::geometry::vertex& location,
35          const bso::utilities::geometry::vector& direction,
36          const bool includePartialSpaces = false) const;
37
38      void setZZero();
39      void addSpace(const ms_space& space);
40      void deleteSpace(ms_space* spacePtr);
41      void deleteSpace(ms_space& space);
42      void cutOff(const bso::utilities::geometry::vertex& location,
43          const bso::utilities::geometry::vector& direction, const double&
            tol = 1e-3);
44      void sweep(const bso::utilities::geometry::vertex& location,
45          const std::vector<std::pair<unsigned int, double> >& distances =
46          {{0,1.0},{1,0.0},{2,0.0}}, const double& tol = 1e-3);
47      void scale(const std::vector<std::pair<unsigned int, double> >& scales
        = {{0,sqrt(2.0)},{1,sqrt(2.0)}});
48      void splitSpace(ms_space* spacePtr, const
        std::vector<std::pair<unsigned int, unsigned int> >& splits =
        {{0,2},{1,2}});
49      void splitSpace(ms_space& space, const std::vector<std::pair<unsigned
        int, unsigned int> >& splits = {{0,2},{1,2}});
```

```
50        void snapOn(const std::vector<std::pair<unsigned int, double> >& snaps
          = {{0,10},{1,10}});
51
52        const auto begin() const {return mSpaces.begin();}
53        const auto end() const {return mSpaces.end();}
54
55        bool hasOverlappingSpaces(std::multimap<ms_space*,ms_space*>&
          overlappingSpaces,
56            const double tol = 1e-3) const;
57        bool hasFloatingSpaces(std::vector<ms_space*>& floatingSpaces,
58            const double tol = 1e-3) const;
59
60        bool operator == (const ms_building& rhs);
61        bool operator != (const ms_building& rhs);
62
63        operator sc_building() const; // convert MS to SC
64
65        friend std::ostream& operator<< (std::ostream& stream, const
          ms_building& building);
66    };
67
68    } // namespace spatial_design
69    } // namespace bso
70
71    #include <bso/spatial_design/ms_building.cpp>
72
73    #endif // MS_BUILDING_HPP
```

```
1    #ifndef MS_BUILDING_CPP
2    #define MS_BUILIDNG_CPP
3
4    #include <sstream>
5    #include <fstream>
6    #include <vector>
7    #include <algorithm>
8    #include <utility>
9    #include <stdexcept>
10   #include <exception>
11   #include <boost/algorithm/string.hpp>
12   #include <bitset>
13
14   namespace bso { namespace spatial_design {
15
16   ms_building::ms_building()
17   { // empty constructor
18       mLastSpaceID = 0;
19   } // ms_building() (empty constructor)
20
21   ms_building::ms_building(std::string fileName)
22   { // initilization by string or text file
23       mLastSpaceID = 0;
24       std::ifstream input;
25       if (!fileName.empty()) input.open(fileName.c_str());
26
27       if (fileName.empty() || !input.is_open())
28       {
29           std::stringstream errorMessage;
30           errorMessage << "Could not initialize an MS building spatial design
             with the following input file:" << std::endl
31                                        << ((fileName.empty())? "no input file
                                          given" : fileName) << std::endl
32                                        << "(bso/spatial_design/ms_building.cpp).
                                          " << std::endl;
33           throw std::invalid_argument(errorMessage.str());
34       }
35
36       std::string line;
37
38
39       while (!input.eof()) // Parse the input file line by line
40       {
41           getline(input,line); // get next line from the file
42           try
43           {
44               boost::algorithm::trim(line); // remove white space from start
                 and end of line (to see if it is an empty line, removes any
                 incidental white space)
45               if (line == "") //skip empty lines (tokenizer does not like it)
46               {
47                   continue; // continue to next line
48               }
49               else if (line.substr(0,2) == "R," || line.substr(0,2) == "r,"
                 || line.substr(0,2) == "N," || line.substr(0,2) == "n,")
50               {
```

```
51                  //line.erase(0,2); // line.erase now happens in ms_space
                    for sDefmethod definition (space definitiond method)
52                  mSpaces.push_back(new ms_space(line));
53              }
54              else
55              {
56                  continue;
57              }
58          }
59          catch(std::exception& e)
60          {
61              std::stringstream errorMessage;
62              errorMessage << "Encountered an error while parsing the
                following line of an MS input file:" << std::endl
63                                       << line << std::endl
64                                       <<
                                         "(bso/spatial_design/ms_building.cpp).
                                         Got the following error: " << std::endl
65                                       << e.what() << std::endl;
66              throw std::invalid_argument(errorMessage.str());
67          }
68      }
69
70      mLastSpaceID = this->getLastSpaceID();
71      this->checkValidity();
72  } // ms_building() (constructor using input file)
73
74  ms_building::ms_building(const ms_building& rhs)
75  { // copy constructor
76      mLastSpaceID = rhs.mLastSpaceID;
77      for (auto i : rhs.mSpaces)
78      {
79          mSpaces.push_back(new ms_space(*i));
80      }
81
82      this->checkValidity();
83  } // ms_building() (copy constructor)
84
85  ms_building::ms_building(const sc_building& sc)
86  { // convert SC to MS
87      try
88      {
89          // initialize variables and variable size
90          mLastSpaceID = 0;
91          std::vector<unsigned int> originIndices = {sc.getWSize(),
            sc.getDSize(), sc.getHSize()};
92          std::vector<std::vector<double> > globalCoords(3);
93          std::vector<std::vector<double> > dimensions(3);
94          for (unsigned int i = 0; i < 3; i++)
95          {
96              globalCoords[i] = std::vector<double>(originIndices[i]+1);
97              dimensions[i] = std::vector<double>(originIndices[i]);
98          }
99          for (unsigned int i = 0; i < originIndices[0]; i++)
            dimensions[0][i] = sc.getWValue(i);
100         for (unsigned int i = 0; i < originIndices[1]; i++)
```

```
            dimensions[1][i] = sc.getDValue(i);
101         for (unsigned int i = 0; i < originIndices[2]; i++)
            dimensions[2][i] = sc.getHValue(i);
102
103         // find the lowest cell indices (w,d,h) that index an active cell
            (i.e. they will represent the origin)
104         for (unsigned int i = 1; i < sc.getBRowSize(); i++)
105         { // for each cell
106             std::vector<unsigned int> indices = {sc.getWIndex(i),
                sc.getDIndex(i), sc.getHIndex(i)}; // temporarily store the
                indices (w,d,h)
107             for (unsigned int j = 0; j < sc.getBSize(); j++)
108             { // for each space
109                 if (sc.getBValue(j,i) == 1)
110                 { // if the cell i is active for space j
111                     for (unsigned int k = 0; k < 3; k++)
112                     { // check for each index
113                         if (indices[k] < originIndices[k])
114                         { // if it is lower than the lowest found so far to
                            index an active cell
115                             originIndices[k] = indices[k]; // if it is, set
                                the lowest found to it.
116                         }
117                     }
118                 }
119             }
120         }
121
122         for (unsigned int i = 0; i < 3; i++)
123         { // for each index
124             globalCoords[i][originIndices[i]] = 0.0; // set the origins
                coordinate value to zero
125             for (unsigned int j = originIndices[i] + 1; j <
                globalCoords[i].size(); j++)
126             { // for each consequent index value increment with the width
                of the dimensions described by the supercube
127                 globalCoords[i][j] = globalCoords[i][j-1] +
                    dimensions[i][j-1];
128             }
129         }
130
131         for (unsigned int i = 0; i < sc.getBSize(); i++)
132         { // for each space
133             // find the indices of the minium and maximum cell index that
                is active for space i
134             unsigned int max = 0, min = 0;
135             for (unsigned int j = 1; j <= sc.getBRowSize(); j++)
136             {
137                 if (min == 0 && sc.getBValue(i,j) == 1) min = j;
138                 if (sc.getBValue(i,j) == 1) max = j;
139             }
140
141             //get the locations and dimensions from these indices
142             utilities::geometry::vertex location;
143             location << globalCoords[0][sc.getWIndex(min)],
144                                     globalCoords[1][sc.getDIndex(min)],
```

```
145                                    globalCoords[2][sc.getHIndex(min)];
146                 location.round(0);
147 
148                 utilities::geometry::vector dimensions;
149                 dimensions << globalCoords[0][sc.getWIndex(max)+1],
150                                                 globalCoords[1][sc.getDIndex(max)+1],
151                                                 globalCoords[2][sc.getHIndex(max)+1];
152                 dimensions -= location;
153                 dimensions.round(0);
154 
155                 //initialize a new space with the found location and dimensions
156                 mSpaces.push_back(new ms_space(sc.getBValue(i,0), location,
                    dimensions));
157                 // check if the last space ID is up to date.
158                 if (mLastSpaceID < mSpaces.back()->getID())
159                 { // if it is not, up date it
160                     mLastSpaceID = mSpaces.back()->getID();
161                 }
162             }
163         }
164         catch (std::exception& e)
165         {
166             std::stringstream errorMessage;
167             errorMessage << "Could not convert SC building into MS building" <<
            std::endl
168                                          << "(bso/spatial_design/ms_building.cpp)."
                                             << std::endl
169                                          << "Got the following error: " << std::endl
170                                          << e.what() << std::endl;
171             throw std::invalid_argument(errorMessage.str());
172         }
173 
174 
175     } // ms_building() (convert SC to MS)
176 
177     ms_building::~ms_building()
178     { // destructor
179         for (auto i : mSpaces) delete i;
180     } // ~ms_building()
181 
182     void ms_building::checkValidity() const
183     {
184 
185     } // checkValidity()
186 
187     void ms_building::writeToFile(std::string fileName) const
188     {
189         std::ofstream output;
190         if (!fileName.empty()) output.open(fileName.c_str());
191 
192         if (fileName.empty() || !output.is_open())
193         {
194             std::stringstream errorMessage;
195             errorMessage << "Could not open the following file to write an MS
            building spatial design to:" << std::endl
196                                          << ((fileName.empty())? "no input file
```

```
                                         given" : fileName) << std::endl
197                                      << "(bso/spatial_design/ms_building.cpp).
                                         " << std::endl;
198           throw std::invalid_argument(errorMessage.str());
199       }
200
201       output << *this;
202   } // writeToFile()
203
204   std::vector<ms_space*> ms_building::getSpacePtrs() const
205   {
206       return mSpaces;
207   } //getSpacePtrs()
208
209   ms_space* ms_building::getSpacePtr(const ms_space& space) const
210   {
211       for (auto i : mSpaces)
212       {
213           if (/*i == &space ||*/ *i == space)
214           {
215               return i;
216           }
217       }
218
219       // if this part of code is reached, the space was not found in the
          preceding for loop
220       std::stringstream errorMessage;
221       errorMessage << "Could not find the following space in an MS building
          spatial design:" << std::endl
222                                    << space << std::endl
223                                    << "(bso/spatial_design/ms_building.cpp). " <<
                                       std::endl;
224       throw std::runtime_error(errorMessage.str());
225   } // getSpacePtr()
226
227   ms_space* ms_building::getSpacePtr(const ms_space* spacePtr) const
228   {
229       return getSpacePtr(*spacePtr);
230   }
231
232   unsigned int ms_building::getLastSpaceID() const
233   {
234       for (auto i : mSpaces)
235       {
236           if (mLastSpaceID < i->getID()) mLastSpaceID = i->getID();
237       }
238       return mLastSpaceID;
239   } // getLastSpaceID()
240
241   double ms_building::getVolume() const
242   {
243       double volume = 0;
244       for (auto i : mSpaces) volume += i->getVolume();
245
246       return volume;
247   } // getVolume()
```

```
248
249 double ms_building::getFloorArea() const
250 {
251     double area = 0;
252     for (const auto& i : mSpaces) area += i->getFloorArea();
253     return area;
254 }
255
256 std::vector<ms_space*> ms_building::selectSpacesGeometrically(
257         const bso::utilities::geometry::vertex& location,
258         const bso::utilities::geometry::vector& direction,
259         const bool includePartialSpaces /*= false*/) const
260 {
261     std::vector<ms_space*> spaceSelection;
262     for (const auto& i : mSpaces)
263     {
264         bool allVerticesSelected = true;
265         bool oneOrMoreVerticesSelected = false;
266         for (const auto& j : i->getGeometry())
267         {
268             double checkValue = direction.dot(j - location);
269             if (checkValue >= -1e-3)
270             { // check if point j lies behind the plane defined by the
                location and the direction
271                 if (checkValue >= 1e-3) oneOrMoreVerticesSelected = true;
272             }
273             else
274             {
275                 allVerticesSelected = false;
276             }
277         }
278         if (allVerticesSelected) spaceSelection.push_back(i);
279         else if (oneOrMoreVerticesSelected && includePartialSpaces)
            spaceSelection.push_back(i);
280     }
281     return spaceSelection;
282 }
283
284 void ms_building::setZZero()
285 {
286     double min = mSpaces[0]->getCoordinates()(2); // set an initial value
        to the minimum
287
288     // find the minimum value of the z-coordinates in the building
289     for (auto i : mSpaces) if (i->getCoordinates()(2) < min) min =
        i->getCoordinates()(2);
290     if (min <= 0) return;
291
292     utilities::geometry::vector coordDifference;
293     coordDifference << 0, 0, -min;
294
295     for (auto i : mSpaces) i->setCoordinates(i->getCoordinates() +
        coordDifference);
296 } // setZZero()
297
298 void ms_building::addSpace(const ms_space& space)
```

```
299  {
300      try
301      {
302          mSpaces.push_back(new ms_space(space));
303          checkValidity();
304      }
305      catch(std::exception& e)
306      {
307          std::stringstream errorMessage;
308          errorMessage << "Could not add the following space to an MS
             building spatial design: " << std::endl
309                                       << space << std::endl
310                                       << "(bso/spatial_design/ms_building.cpp).
                                          Got the following error message: " <<
                                          std::endl
311                                       << e.what() << std::endl;
312          throw std::invalid_argument(errorMessage.str());
313      }
314  } // addSpace
315
316  void ms_building::deleteSpace(ms_space* spacePtr)
317  {
318      if (std::find(mSpaces.begin(), mSpaces.end(), spacePtr) ==
         mSpaces.end()) spacePtr = getSpacePtr(spacePtr);
319      try
320      {
321          mSpaces.erase(std::remove(mSpaces.begin(), mSpaces.end(),
             spacePtr), mSpaces.end());
322      }
323      catch(std::exception& e)
324      {
325          std::stringstream errorMessage;
326          errorMessage << "Could not delete the following space from an MS
             building spatial design: " << std::endl
327                                       << *spacePtr << std::endl
328                                       << "(bso/spatial_design/ms_building.cpp).
                                          Got the following error message: " <<
                                          std::endl
329                                       << e.what() << std::endl;
330          throw std::invalid_argument(errorMessage.str());
331      }
332  } // deleteSpace()
333
334  void ms_building::deleteSpace(ms_space& space)
335  {
336      this->deleteSpace(this->getSpacePtr(space));
337  }
338
339  void ms_building::cutOff(const bso::utilities::geometry::vertex& location,
340          const bso::utilities::geometry::vector& direction, const double&
             tol /*= 1e-3*/)
341  {
342      // normalize the direction for convenience of later use
343      auto n = direction.normalized();
344      // check if the cutting plane is orhtogonal
345      bool isOrthogonal = false;
```

```
346         for (unsigned int i = 0; i < 3; ++i)
347         {
348             bso::utilities::geometry::vector check = {0,0,0};
349             check(i) = 1;
350             if (check.isParallel(n, tol))
351             {
352                 isOrthogonal = true;
353                 break;
354             }
355         }
356         if (!isOrthogonal)
357         {
358             std::stringstream errorMessage;
359             errorMessage << "\nError, could not cut-off part of the building in
                following location:\n"
360                                          << location << ", and the following
                                             direction:\n"
361                                          << direction << ". Direction is not
                                             orthogonal.\n"
362                                          << "(bso/spatial_design/ms_building.cpp)"
                                             << std::endl;
363             throw std::invalid_argument(errorMessage.str());
364         }
365 
366         // remove spaces that are completely included by the cut-off plane
367         auto selection = this->selectSpacesGeometrically(location,n,false);
368         for (auto& i : selection) this->deleteSpace(i);
369 
370         // select spaces that are cut by the cut-off plane
371         selection = this->selectSpacesGeometrically(location,n,true);
372         for (auto& i : selection)
373         {// cut each space at the cutting plane
374             bso::utilities::geometry::vertex coords = i->getCoordinates();
375             bso::utilities::geometry::vector tempSum = coords +
                i->getDimensions();
376             bso::utilities::geometry::vector vec1 = coords - location;
377 
378             if (n.dot(vec1) < 0)
379             {
380                 vec1 = tempSum - location;
381                 double distance = n.dot(vec1);
382                 tempSum -= n* distance;
383             }
384             else
385             {
386                 double distance = n.dot(vec1);
387                 coords -= n * distance;
388             }
389 
390             i->setCoordinates(coords);
391             i->setDimensions(tempSum - coords);
392         }
393     }
394 
395     void ms_building::sweep(const bso::utilities::geometry::vertex& location,
396             const std::vector<std::pair<unsigned int, double> >& distances
```

```
397             /*= {{0,1.0},{1,0.0},{2,0.0}}*/, const double& tol /*=1e-3*/)
398     {
399         bso::utilities::geometry::vertex coords;
400         bso::utilities::geometry::vector tempSum;
401
402         unsigned int axis;
403         double distance;
404         bool checkDouble[3] = {false};
405
406         for (auto i : distances)
407         {
408             axis = i.first;
409             if (axis != 0 && axis != 1 && axis != 2)
410             {
411                 std::stringstream errorMessage;
412                 errorMessage << "Trying to sweep an MS building spatial design
                    over the \n"
413                                              << "following non-existent axis: " <<
                                                 axis << std::endl
414                                              <<
                                                 "(bso/spatial_design/ms_building.cpp)."
                                                 << std::endl;
415                 throw std::invalid_argument(errorMessage.str());
416             }
417
418             if (checkDouble[axis])
419             {
420                 std::stringstream errorMessage;
421                 errorMessage << "Defined a scaling factor to scale and MS
                    building \n"
422                                              << "spatial design twice for the same
                                                 axis, axis: " << axis << std::endl
423                                              <<
                                                 "(bso/spatial_design/ms_building.cpp)."
                                                 << std::endl;
424                 throw std::invalid_argument(errorMessage.str());
425             }
426             else checkDouble[axis] = true;
427         }
428
429         for (auto i : mSpaces)
430         {
431             coords = i->getCoordinates();
432             tempSum = coords + i->getDimensions();
433
434             for (auto j : distances)
435             {
436                 axis = j.first;
437                 distance = j.second;
438
439                 if (distance < 0)
440                 {
441                     if (coords(axis)  + tol < location(axis)) coords(axis)   +=
                        distance;
442                     if (tempSum(axis) + tol < location(axis)) tempSum(axis) +=
                        distance;
```

```
443                 }
444                 else
445                 {
446                     if (coords(axis)  - tol > location(axis)) coords(axis)   +=
                        distance;
447                     if (tempSum(axis) - tol > location(axis)) tempSum(axis) +=
                        distance;
448                 }
449             }
450 
451             i->setCoordinates(coords);
452             i->setDimensions(tempSum - coords);
453         }
454     } // sweep()
455 
456     void ms_building::scale(const std::vector<std::pair<unsigned int, double>
    >& scales /*= {{0,sqrt(2.0)},{1,sqrt(2.0)}}*/)
457     {
458         utilities::geometry::vertex coords;
459         utilities::geometry::vector tempSum;
460 
461         unsigned int axis;
462         double n;
463         bool checkDouble[3] = {false};
464 
465         for (auto i : scales)
466         {
467             axis = i.first;
468             if (axis != 0 && axis != 1 && axis != 2)
469             {
470                 std::stringstream errorMessage;
471                 errorMessage << "Trying to sweep an MS building spatial design
                over the \n"
472                                          << "following non-existent axis: " <<
                                         axis << std::endl
473                                          <<
                                         "(bso/spatial_design/ms_building.cpp)."
                                         << std::endl;
474                 throw std::invalid_argument(errorMessage.str());
475             }
476 
477             if (checkDouble[axis])
478             {
479                 std::stringstream errorMessage;
480                 errorMessage << "Defined a scaling factor to scale and MS
                building \n"
481                                          << "spatial design twice for the same
                                         axis, axis: " << axis << std::endl
482                                          <<
                                         "(bso/spatial_design/ms_building.cpp)."
                                         << std::endl;
483                 throw std::invalid_argument(errorMessage.str());
484             }
485             else checkDouble[axis] = true;
486         }
487 
```

```
488         for (auto i : mSpaces)
489         {
490             coords = i->getCoordinates();
491             tempSum = coords + i->getDimensions();
492 
493             for (auto j : scales)
494             {
495                 axis = j.first;
496                 n = j.second;
497 
498                 coords(axis)  *= n;
499                 tempSum(axis) *= n;
500             }
501 
502             i->setCoordinates(coords);
503             i->setDimensions(tempSum - coords);
504         }
505     } // scale()
506 
507     void ms_building::splitSpace(ms_space* spacePtr, const
        std::vector<std::pair<unsigned int, unsigned int> >& splits /*=
        {{0,2},{1,2}}*/)
508     {
509         this->getLastSpaceID();
510         utilities::geometry::vertex coords  = spacePtr->getCoordinates();
511         utilities::geometry::vector tempSum = coords + spacePtr->getDimensions();
512 
513         std::vector<double> splitValues[3];
514         for (unsigned int i = 0; i < 3; i++)
515         {
516             splitValues[i] = std::vector<double>();
517             splitValues[i].push_back(coords(i));
518             splitValues[i].push_back(tempSum(i));
519         }
520 
521         unsigned int axis, div;
522         bool checkDouble[3] = {false};
523 
524         for (auto i : splits)
525         {
526             axis = i.first;
527             div = i.second;
528 
529             if (axis != 0 && axis != 1 && axis != 2)
530             {
531                 std::stringstream errorMessage;
532                 errorMessage << "Trying to split space \"" << spacePtr->getID()
                    << "\" over a non-existing axis: " << axis << std::endl
533                                                  <<
                                                     "(bso/spatial_design/ms_building.cpp)."
                                                     << std::endl;
534                 throw std::invalid_argument(errorMessage.str());
535             }
536 
537             if (div < 2)
538             {
```

```
539                 std::stringstream errorMessage;
540                 errorMessage << "Trying to split space \"" << spacePtr->getID()
                    << "\" into less than two new spaces: n_split = " << div <<
                    std::endl
541                                                 <<
                                                 "(bso/spatial_design/ms_building.cpp)."
                                                 << std::endl;
542                 throw std::invalid_argument(errorMessage.str());
543             }
544 
545             if (checkDouble[axis])
546             {
547                 std::stringstream errorMessage;
548                 errorMessage << "Defined a split for space \"" <<
                    spacePtr->getID() << "\" twice for the same axis, axis: " <<
                    axis << std::endl
549                                                 <<
                                                 "(bso/spatial_design/ms_building.cpp)."
                                                 << std::endl;
550                 throw std::invalid_argument(errorMessage.str());
551             }
552             else checkDouble[axis] = true;
553 
554             double delta = floor((splitValues[axis][1] -
                splitValues[axis][0])/div);
555 
556             for (unsigned int i = 0; i < div-1; i++)
557             {
558                 double nextCoord;
559                 nextCoord = splitValues[axis][i] + delta;
560                 splitValues[axis].insert(splitValues[axis].begin() + i +
                    1,nextCoord);
561             }
562         }
563 
564         ms_space* temp;
565         for (unsigned int i = 0; i < splitValues[0].size() - 1; i++)
566         {
567             for (unsigned int j = 0; j < splitValues[1].size() - 1; j++)
568             {
569                 for (unsigned int k = 0; k < splitValues[2].size() - 1; k++)
570                 {
571                     temp = new ms_space(*spacePtr);
572                     temp->setID(++mLastSpaceID);
573                     coords << splitValues[0][i], splitValues[1][j],
                        splitValues[2][k];
574                     temp->setCoordinates(coords);
575                     tempSum << splitValues[0][i+1], splitValues[1][j+1],
                        splitValues[2][k+1];
576                     temp->setDimensions(tempSum - coords);
577                     mSpaces.push_back(temp);
578                 }
579             }
580         }
581 
582         deleteSpace(spacePtr);
```

```
583 } // splitSpace()
584
585 void ms_building::splitSpace(ms_space& space, const
    std::vector<std::pair<unsigned int, unsigned int> >& splits /*=
    {{0,2},{1,2}}*/)
586 {
587     this->splitSpace(this->getSpacePtr(space), splits);
588 } // splitSpace()
589
590 void ms_building::snapOn(const std::vector<std::pair<unsigned int, double>
    >& snaps /*= {{0,10},{1,10}}*/)
591 {
592     utilities::geometry::vertex coords;
593     utilities::geometry::vector tempSum;
594
595     unsigned int axis;
596     double n;
597     bool checkDouble[3] = {false};
598
599     for (auto i : snaps)
600     {
601         axis = i.first;
602         if (axis != 0 && axis != 1 && axis != 2)
603         {
604             std::stringstream errorMessage;
605             errorMessage << "Trying to snap-on an MS building spatial to a
                grid over the following non-existent axis: " << axis << std::endl
606                                                          <<
                                                             "(bso/spatial_design/ms_building.cpp)."
                                                             << std::endl;
607             throw std::invalid_argument(errorMessage.str());
608         }
609
610         if (checkDouble[axis])
611         {
612             std::stringstream errorMessage;
613             errorMessage << "Defined a snap-on for an MS building spatial
                design twice for the same axis, axis: " << axis << std::endl
614                                                          <<
                                                             "(bso/spatial_design/ms_building.cpp)."
                                                             << std::endl;
615             throw std::invalid_argument(errorMessage.str());
616         }
617         else checkDouble[axis] = true;
618     }
619
620     for (auto i : mSpaces)
621     {
622         coords = i->getCoordinates();
623         tempSum = coords + i->getDimensions();
624
625         for (auto j : snaps)
626         {
627             axis = j.first;
628             n = j.second;
629
```

```
630                 coords(axis)  = round(coords(axis)  / n) * n;
631                 tempSum(axis) = round(tempSum(axis) / n) * n;
632             }
633 
634             i->setCoordinates(coords);
635             i->setDimensions(tempSum - coords);
636         }
637     } // snapOn()
638 
639     bool ms_building::hasOverlappingSpaces(std::multimap<ms_space*,ms_space*>&
        overlappingSpaces,
640         const double tol /*= 1e-3*/) const
641     {
642         std::vector<bso::utilities::geometry::quad_hexahedron> spaceGeoms;
643         for (const auto& i : mSpaces)
644         {
645             spaceGeoms.push_back(i->getGeometry());
646         }
647 
648         for (unsigned int i = 0; i < spaceGeoms.size(); ++i)
649         {
650             for (unsigned int j = 0; j < spaceGeoms.size(); ++j)
651             {
652                 if (i == j) continue;
653                 bool foundOverlap = false;
654                 for (const auto& k : spaceGeoms[i])
655                 {
656                     if (spaceGeoms[j].isInside(k,tol))
657                     {
658                         overlappingSpaces.emplace(mSpaces[j], mSpaces[i]);
659                         foundOverlap = true;
660                         break;
661                     }
662                 }
663                 if (foundOverlap) break;
664                 unsigned int linePolygonIntersections = 0;
665                 for (const auto& k : spaceGeoms[j].getLines())
666                 {
667                     for (const auto& l : spaceGeoms[i].getPolygons())
668                     {
669                         if (l->intersectsWith(k,tol))
670                         {
671                             ++linePolygonIntersections;
672                             if (linePolygonIntersections == 2)
673                             {
674                                 overlappingSpaces.emplace(mSpaces[j],
                                    mSpaces[i]);
675                                 foundOverlap = true;
676                                 break;
677                             }
678                         }
679                     }
680                     if (foundOverlap) break;
681                 }
682             }
683         }
```

```
684
685         return (overlappingSpaces.size() != 0);
686     } // hasOverlappingSpaces()
687
688     bool ms_building::hasFloatingSpaces(std::vector<ms_space*>& floatingSpaces,
689         const double tol /*= 1e-3*/) const
690     {
691         std::vector<bso::utilities::geometry::quad_hexahedron>
            groundedSpaceGeoms;
692         floatingSpaces = mSpaces;
693         bool hasFloatingSpaces = false;
694         bool addedSpaceToGroundedSpaceGeoms = true; // initial
695         while (addedSpaceToGroundedSpaceGeoms)
696         {
697             addedSpaceToGroundedSpaceGeoms = false;
698             auto spaceIt = floatingSpaces.begin();
699             while (spaceIt != floatingSpaces.end())
700             {
701                 auto spaceGeom = (*spaceIt)->getGeometry();
702
703                 // check if this space has any vertex with a z-coordinate at or
                    below zero
704                 bool isGrounded = false;
705                 for (const auto& j : spaceGeom)
706                 {
707                     if (j(2) < abs(tol))
708                     {
709                         isGrounded = true;
710                         break;
711                     }
712                 }
713                 if (isGrounded)
714                 {
715                     groundedSpaceGeoms.push_back(spaceGeom);
716                     addedSpaceToGroundedSpaceGeoms = true;
717                     floatingSpaces.erase(spaceIt);
718                     continue;
719                 }
720
721                 // check if this space is connected to any of the grounded spaces
722                 bool isConnectedToGroundedSpace = false;
723                 for (const auto& i : groundedSpaceGeoms)
724                 {
725                     for (const auto& j : spaceGeom)
726                     {
727                         if (i.isInside(j,tol))
728                         { // if any vertex j of the space is inside a grounded
                            space
729                             isConnectedToGroundedSpace = true;
730                             break;
731                         }
732                         for (const auto& k : i.getPolygons())
733                         {
734                             if (k->isInside(j,tol))
735                             { // if any vertex j of the space is inside the
                                surface of a grounded space
```

```
736                                 isConnectedToGroundedSpace = true;
737                                 break;
738                             }
739                         }
740                         if (isConnectedToGroundedSpace) break;
741                         for (const auto& k : i.getLines())
742                         {
743                             if (k.isOnLine(j,tol))
744                             { // if any vertex j of the space is on a line
                             segment of a grounded space
745                                 isConnectedToGroundedSpace = true;
746                                 break;
747                             }
748                         }
749                         if (isConnectedToGroundedSpace) break;
750                         for (const auto& k : i)
751                         {
752                             if (k.isSameAs(j,tol))
753                             { // if any vertex j of the space is collocated
                             with a vertex of a grounded space
754                                 isConnectedToGroundedSpace = true;
755                                 break;
756                             }
757                         }
758                         if (isConnectedToGroundedSpace) break;
759                     }
760                     for (const auto& j : i)
761                     {
762                         if (spaceGeom.isInside(j,tol))
763                         { // if any vertex j of a grounded space i is inside
                         the space
764                             isConnectedToGroundedSpace = true;
765                             break;
766                         }
767                     }
768                     if (isConnectedToGroundedSpace) break;
769                 }
770 
771                 if (isConnectedToGroundedSpace)
772                 {
773                     groundedSpaceGeoms.push_back(spaceGeom);
774                     addedSpaceToGroundedSpaceGeoms = true;
775                     floatingSpaces.erase(spaceIt);
776                     continue;
777                 }
778 
779                 ++spaceIt;
780             }
781         }
782 
783 
784         return (floatingSpaces.size() != 0);
785     } // hasFloatingSpaces()
786 
787     bool ms_building::operator == (const ms_building& rhs)
788     {
```

```
789        if (mSpaces.size() != rhs.mSpaces.size()) return false;
790
791        // check if rhs contains the same spaces, assuming they may be in
           different order
792        std::vector<ms_space*> cmp = rhs.mSpaces; // copy the vector containing
           the spaces to safe computation time, matches can be removed from this
793        std::reverse(cmp.begin(), cmp.end()); // reverse the vector, since
           items will be removed we want to iterate from the back, however if
           vectors are same order we want to benefit from that to safe computation
           time
794
795        for (auto i : mSpaces)
796        {
797            bool found = false;
798            for (unsigned int j = cmp.size(); j > 0; j--)
799            {
800                if (*i == *(cmp[j - 1]))
801                {
802                    found = true;
803                    cmp.erase(cmp.begin() + j - 1);
804                    break;
805                }
806            }
807            if (!found) return false;
808        }
809        return true;
810    }
811
812    bool ms_building::operator != (const ms_building& rhs)
813    {
814        return !(*this == rhs);
815    }
816
817    ms_building::operator sc_building() const
818    { // convert MS to SC
819        try
820        {
821            sc_building sc;
822            bso::utilities::geometry::vertex minMSPoint = {0,0,0};
823            std::vector<std::vector<double> > coordValues(3); //
               {xValues,yValues,Zvalues}
824
825            // store all coordinate values in this MS building model in the
               three std::vectors
826            for (auto i : mSpaces)
827            {
828                utilities::geometry::vertex p1 = i->getCoordinates();
829                utilities::geometry::vertex p2 = i->getDimensions() + p1;
830                for (unsigned int j = 0; j < 3; j++)
831                {
832                    if (p1[j] < minMSPoint[j]) minMSPoint[j] = p1[j];
833                    coordValues[j].push_back(p1(j));
834                    coordValues[j].push_back(p2(j));
835                }
836            }
837
```

```
838             // sort the three vectors with coordinate values, and remove
                duplicates
839             for (auto& i : coordValues)
840             {
841                 sort(i.begin(), i.end());
842                 i.erase(std::unique(i.begin(), i.end()), i.end());
843             }
844
845             // create the width, depth and height vectors of the supercube, and
                compute the size of the bitmask of the b vectors
846             unsigned int cubeSize = 1;
847             for (unsigned int i = 0; i < coordValues.size(); i++)
848             {
849                 std::vector<double> tempDimensions;
850                 for (unsigned int j = 0; j < coordValues[i].size()-1; j++)
851                 {
852                     tempDimensions.push_back(coordValues[i][j+1] -
                        coordValues[i][j]);
853                 }
854
855                 cubeSize *= tempDimensions.size();
856                 if (i == 0) sc.mWValues = tempDimensions;
857                 else if (i == 1) sc.mDValues = tempDimensions;
858                 else if (i == 2) sc.mHValues = tempDimensions;
859             }
860
861             // for each space, create an emtpy bit mask, and subsequently find
                out which cells in that mask belong to the space, and should be
                activated
862             for (auto i : mSpaces)
863             {
864                 sc.mBValues.push_back(std::vector<int>(cubeSize + 1));
865                 sc.mBValues.back()[0] = i->getID();
866                 utilities::geometry::vertex p1 = i->getCoordinates();
867                 utilities::geometry::vertex p2 = i->getDimensions() + p1;
868
869                 for (unsigned int j = 1; j < sc.mBValues.back().size(); j++)
870                 {
871                     utilities::geometry::vertex pCheck = minMSPoint;
872
873                     for (unsigned int k = 0; k < sc.getWIndex(j); ++k)
                        pCheck[0] += sc.getWValue(k);
874                     for (unsigned int k = 0; k < sc.getDIndex(j); ++k)
                        pCheck[1] += sc.getDValue(k);
875                     for (unsigned int k = 0; k < sc.getHIndex(j); ++k)
                        pCheck[2] += sc.getHValue(k);
876
877                     bool belongsToSpace = true;
878                     for (unsigned int k = 0; k < 3; k++)
879                     {
880                         if (pCheck[k] < p1[k] || pCheck[k] >= p2[k])
881                         {
882                             belongsToSpace = false;
883                             break;
884                         }
885                     }
```

```
886                         sc.mBValues.back()[j] = belongsToSpace;
887                     }
888                 }
889                 sc.checkValidity();
890                 return sc;
891             }
892             catch (std::exception& e)
893             {
894                 std::stringstream errorMessage;
895                 errorMessage << "Could not convert MS building into SC building" <<
                    std::endl
896                                          << "(bso/spatial_design/ms_building.cpp)."
                                             << std::endl
897                                          << "Got the following error: " << std::endl
898                                          << e.what() << std::endl;
899                 throw std::invalid_argument(errorMessage.str());
900             }
901         } // sc_building() (convert MS to SC)
902
903         std::ostream& operator <<(std::ostream& stream, const ms_building& building)
904         {
905             bool first = true;
906             for (auto i : building.getSpacePtrs())
907             {
908                 if (!first) stream << std::endl;
909                 else first = false;
910                 stream << *i;
911             }
912
913             return stream;
914         } // << operator
915
916         } // namespace spatial_design
917         } // namespace bso
918
919         #endif // MS_BUILDING_CPP
```

A.3 cf_building_model

```
1   #ifndef CF_BUILDING_MODEL_HPP
2   #define CF_BUILDING_MODEL_HPP
3
4   #include <bso/spatial_design/ms_building.hpp>
5
6   namespace bso { namespace spatial_design { namespace conformal {
7
8       class cf_building_model : public cf_geometry_model
9       {
10      private:
11          std::vector<cf_point*> mCFPoints;
12          std::vector<cf_edge*> mCFEdges;
13          std::vector<cf_surface*> mCFSurfaces;
14          std::vector<cf_space*> mCFSpaces;
15
16          ms_building mMSModel; // safe it, in case copy consttructor is called
17          double mTol;
18
19          void addSpace(const ms_space& msSpace); // initializer buiding- and
            geometry-conformal model quad-hexahedron method; add space
            rectanular/orthogonal and non-orthogonal quad-hexahedron
20          void addSpaceT(const ms_space& msSpace); // initializer building-
            and geometry-conformal model tetrahedron method;add space
            tetrahedron entities
21          void makeConformal(); // rectangular and orthogonal makeConformal
22          void makeConformalN(char method); // NOBSD quad-hexahedorn geometry
            conformal method (method I,O,M)
23          void makeConformalT(); // NOBSD tetrahedron geometry conformal method
24
25          cf_building_model& operator = (cf_building_model& rhs) = default;
26          friend class cf_geometry_entity;
27      public:
28          cf_building_model(const cf_building_model& rhs);
29          cf_building_model(const ms_building& msModel, const double& tol =
            1e-3); // makes use of standart method 'I'
30          cf_building_model(const ms_building& msModel, char method, const
            double& tol = 1e-3);
31          ~cf_building_model();
32
33          const std::vector<cf_point*         >& cfPoints()       const {
            return mCFPoints;}
34          const std::vector<cf_edge*          >& cfEdges()            const {
            return mCFEdges;}
35          const std::vector<cf_surface*       >& cfSurfaces()     const {
            return mCFSurfaces;}
36          const std::vector<cf_space*         >& cfSpaces()       const {
            return mCFSpaces;}
37
38      };
39
40  } // conformal
41  } // spatial_design
42  } // bso
43
44  #endif // CF_BUILDING_MODEL_HPP
```

```
1    #ifndef CF_BUILDING_MODEL_CPP
2    #define CF_BUILDING_MODEL_CPP
3
4    #include <bitset>
5
6    namespace bso { namespace spatial_design { namespace conformal {
7
8        void cf_building_model::addSpace(const ms_space& msSpace) // initialize
         quad-hexahedorn geometry conformal model
9        { //
10           mCFSpaces.push_back(new cf_space(msSpace.getGeometry(), this));
11           mCFSpaces.back()->setSpaceID(msSpace.getID());
12           std::string possibleSpaceType;
13           if (msSpace.getSpaceType(possibleSpaceType))
14           {
15               mCFSpaces.back()->setSpaceType(possibleSpaceType);
16           }
17
18           auto spPtr = mCFSpaces.back();
19           for (const auto& i : *spPtr)
20           {
21               mCFPoints.push_back(new cf_point(i, this));
22               mCFPoints.back()->addSpace(spPtr);
23               spPtr->addPoint(mCFPoints.back());
24           }
25
26           for (const auto& i : spPtr->getLines())
27           {
28               mCFEdges.push_back(new cf_edge(i, this));
29               mCFEdges.back()->addSpace(spPtr);
30               spPtr->addEdge(mCFEdges.back());
31           }
32
33           std::vector<std::string> possibleSurfaceTypes;
34           bool surfaceTypesAvailable =
             msSpace.getSurfaceTypes(possibleSurfaceTypes);
35           if (surfaceTypesAvailable)
36           {
37               std::swap(possibleSurfaceTypes[0],possibleSurfaceTypes[2]);
38               std::swap(possibleSurfaceTypes[4],possibleSurfaceTypes[5]);
39           }
40           auto typeIte = possibleSurfaceTypes.begin();
41
42           for (const auto& i : spPtr->getPolygons())
43           {
44               mCFSurfaces.push_back(new cf_surface(*i, this));
45               mCFSurfaces.back()->addSpace(spPtr);
46               spPtr->addSurface(mCFSurfaces.back());
47               if (surfaceTypesAvailable)
48               {
49                   mCFSurfaces.back()->setSurfaceType(*typeIte);
50                   ++typeIte;
51               }
52           }
53       } // addSpace
54
```

```
55      void cf_building_model::addSpaceT(const ms_space& msSpace) //
        initialize tetrahedron geometry conformal model
56      {
57          bso::utilities::geometry::quad_hexahedron
            temp(msSpace.getGeometry());
58          mCFSpaces.push_back(new cf_space(msSpace.getGeometry(), this ,
            (temp.getTetrahedrons())));
59
60
61          // Add space types
62          mCFSpaces.back()->setSpaceID(msSpace.getID());
63          std::string possibleSpaceType;
64          if (msSpace.getSpaceType(possibleSpaceType))
65          {
66              mCFSpaces.back()->setSpaceType(possibleSpaceType);
67          }
68
69
70          // generate the triangle entities of the tetrahedron, since
            otherwise the polygon does not convert
71          std::vector<utilities::geometry::triangle*> tri;
72          for(const auto i: ((mCFSpaces.back() -> getTetrahedrons())))
73          {
74              for(const auto j: i.getPolygons())
75              {
76                  std::vector <utilities::geometry::vertex> triVertex;
77                  for(const auto k: j->getVertices())
78                  {
79                      triVertex.push_back(k);
80                  }
81                  tri.push_back(new utilities::geometry::triangle(triVertex));
82              }
83          }
84
85
86          // Add the geometry entities to cf_geometry_model and
            cf_building_model
87          auto spPtr = mCFSpaces.back();
88          for (const auto& i : *spPtr)
89          {
90              mCFPoints.push_back(new cf_point(i, this)); // add points of
                space spPtr to mCFSpaces and initate constoctor, which adds the
                points to cf_geometry_model mCFVertices
91              mCFPoints.back()->addSpace(spPtr); // store the space to which
                the points belongs in the mCFSpaces of cf_point
92              spPtr->addPoint(mCFPoints.back()); // add the cf_point to the
                mCFSpaces of the cf_space to which it belongs point belongs
93          }
94
95          for (const auto& i : spPtr->getLines())
96          {
97              mCFEdges.push_back(new cf_edge(i, this));
98              mCFEdges.back()->addSpace(spPtr);
99              spPtr->addEdge(mCFEdges.back());
100         }
101
```

```
102             std::vector<std::string> possibleSurfaceTypes;
103             bool surfaceTypesAvailable =
                msSpace.getSurfaceTypes(possibleSurfaceTypes);
104             if (surfaceTypesAvailable)
105             {
106                 std::swap(possibleSurfaceTypes[0],possibleSurfaceTypes[2]);
107                 std::swap(possibleSurfaceTypes[4],possibleSurfaceTypes[5]);
108             }
109             auto typeIte = possibleSurfaceTypes.begin();
110
111             for (const auto& i : spPtr->getPolygons())
112             {
113                 std::vector<utilities::geometry::triangle*> triOnPoly;
114                 double tol = tolerance();
115
116                 //find triangles from tetrahedron which are on the space
                    polygon to link to this surface
117                 for(const auto l: tri)
118                 {
119                     bool onFace = true;
120                     for(const auto k: l->getVertices())
121                     {
122                         if
                            ((std::find((i->getVertices()).begin(),(i->getVertices())
                            .end(), k) == (i->getVertices()).end()) == false)
123                         {
124                             onFace = false;
125                         }
126                     }
127
128                     if(onFace == true)
129                     {
130                         triOnPoly.push_back(l);
131                     }
132                 }
133
134                 // add surface and triangles
135
136                 mCFSurfaces.push_back(new cf_surface(*i, this, triOnPoly));
137                 mCFSurfaces.back()->addSpace(spPtr);
138                 spPtr->addSurface(mCFSurfaces.back());
139                 if (surfaceTypesAvailable)
140                 {
141                     mCFSurfaces.back()->setSurfaceType(*typeIte);
142                     ++typeIte;
143                 }
144
145             }
146         } //addSpaceT
147
148         void cf_building_model::makeConformalN(char method) // Step one
            quad-hexahedron method
149         { //
150             std::vector <cf_vertex> pIntersectionV;
151             std::vector <utilities::geometry::line_segment> pIntersectionL;
152             utilities::geometry::vertex pIntersection;
```

```
153
154             int t = 1;
155             for (int i=0; i<t; i++)
156             {
157                 bool isSplitDone = false;
158                 for (const auto& i: mCFSpaces)
159                 {
160                     for (const auto& j: mCFSpaces)
161                     {
162                         if(i == j)
163                         {
164                             continue;
165                         }
166
167                         for(const auto& o: (i->cfCuboids()))
168                         {
169                             for(const auto& p: (j->cfCuboids()))
170                             {
171                                 // check for points inside space
172
173
174                                 // check for line - rectangle intersections,
                                    and add the found vertex to the geometry model
                                    and pIntersection
175                                 for (const auto k: o->getPolygons())
176                                 {
177                                     for (const auto l: p->getLines())
178                                     {
179                                         if (k->intersectsWith(l, pIntersection,
                                            mTol))
180                                         {
181                                             this->addVertex(pIntersection);
182
183                                             bool doubleInsert = false;
184                                             for (const auto& m : pIntersectionV)
185                                             {// check if pIntersection already
                                                exist in pIntersectionV
186                                                 if (m.isSameAs(pIntersection,
                                                    mTol))
187                                                 {
188                                                     doubleInsert = true;
189                                                 }
190                                             }
191
192                                             if (doubleInsert == false)
193                                             {
194
                                                    pIntersectionV.push_back(pInterse
                                                    ction);
195                                             }
196                                             //}
197                                         }
198
199                                         else if(k->isInside(l[0]) &&
                                            k->isInside(l[1]))
200                                         {
```

```
201                                         pIntersectionL.push_back(l);
202                                     }
203                                 }
204                             }
205 
206                             // check for line line intersections, and add
                                the found vertex to the geometry model and
                                pIntersection
207                             for (const auto k: o->getLines())
208                             {
209                                 for (const auto l: p->getLines())
210                                 {
211                                     if (k.intersectsWith(l, pIntersection,
                                        mTol))
212                                     {
213                                         this->addVertex(pIntersection);
214                                         pIntersectionL.push_back(l);
215 
216                                         bool doubleInsert = false;
217                                         for (const auto& m : pIntersectionV)
218                                         {// check if pIntersection already
                                            exist in pIntersectionV
219                                             if (m.isSameAs(pIntersection,
                                                mTol))
220                                             {
221                                                 doubleInsert = true;
222                                             }
223                                         }
224 
225                                         if (doubleInsert == false)
226                                         {
227 
                                                pIntersectionV.push_back(pInterse
                                                ction);
228                                         }
229                                     }
230                                 }
231                             }
232                         }
233                     }
234 
235                     // remove ducplicates from pIntersectionL
236                     auto pIntersectionL_end = pIntersectionL.end();
237                     for(auto it = pIntersectionL.begin(); it !=
                        pIntersectionL_end; ++it)
238                     {
239                         pIntersectionL_end = std::remove(it + 1,
                            pIntersectionL_end, *it);
240                     }
241                     pIntersectionL.erase(pIntersectionL_end,
                        pIntersectionL.end());
242 
243 
244 
245                     if(pIntersectionV.size() >= 1)
246                     {
```

```
247                         i->checkVertexN(pIntersectionV, pIntersectionL,
                            method);
248
249                         if(isSplitDone == false)
250                         {
251                             t++;
252                             isSplitDone = true;
253                         }
254                     }
255                     pIntersectionV.clear();
256                     pIntersectionL.clear();
257                 }
258             }
259
260             for (const auto& h: mCFSpaces) // intercell
261             {
262                 std::vector <cf_vertex> pIntersectionV;
263                 std::vector <utilities::geometry::line_segment>
                    pIntersectionL;
264                 utilities::geometry::vertex pIntersection;
265
266
267                 int s = 1;
268                 for (int i=0; i<s; i++) //Repeat two
269                 {
270                     bool isSplitDone = false;
271                     for(const auto& o: (h->cfCuboids()))
272                     {
273                         for(const auto& p: (h->cfCuboids()))
274                         {
275                             // check for points inside space
276
277
278                             // check for line - rectangle intersections,
                                and add the found vertex to the geometry model
                                and pIntersection
279                             for (const auto k: o->getPolygons())
280                             {
281                                 for (const auto l: p->getLines())
282                                 {
283                                     if (k->intersectsWith(l, pIntersection,
                                        mTol))
284                                     {
285                                         bool doubleInsert = false;
286                                         for (const auto& m : pIntersectionV)
287                                         {// check if pIntersection already
                                            exist in pIntersectionV
288                                             if (m.isSameAs(pIntersection,
                                                mTol))
289                                             {
290                                                 doubleInsert = true;
291                                             }
292                                         }
293
294                                         if (doubleInsert == false)
295                                         {
```

```
296                         pIntersectionV.push_back(pInterse
                            ction);
297                     }
298                 }
299 
300                 else if(k->isInside(l[0], mTol) &&
                    k->isInside(l[1], mTol))
301                 {
302                     pIntersectionL.push_back(l);
303                 }
304             }
305         }
306 
307         // check for line line intersections, and add
            the found vertex to the geometry model and
            pIntersection
308         for (const auto k: o->getLines())
309         {
310             for (const auto l: p->getLines())
311             {
312                 if (k.intersectsWith(l, pIntersection,
                    mTol))
313                 {
314                     pIntersectionL.push_back(l);
315 
316                     bool doubleInsert = false;
317                     for (const auto& m : pIntersectionV)
318                     {// check if pIntersection already
                        exist in pIntersectionV
319                         if (m.isSameAs(pIntersection,
                            mTol))
320                         {
321                             doubleInsert = true;
322                         }
323                     }
324 
325                     if (doubleInsert == false)
326                     {
327 
                            pIntersectionV.push_back(pInterse
                            ction);

328                     }
329                 }
330             }
331         }
332 
333         // remove ducplicates from pIntersectionL
334         auto pIntersectionL_end = pIntersectionL.end();
335         for(auto it = pIntersectionL.begin(); it !=
            pIntersectionL_end; ++it)
336         {
337             pIntersectionL_end = std::remove(it + 1,
                pIntersectionL_end, *it);
```

```
338                             }
339                             pIntersectionL.erase(pIntersectionL_end,
                                pIntersectionL.end());
340
341
342                             if(pIntersectionV.size() >= 1)
343                             {
344                                 h->checkVertexN(pIntersectionV,
                                    pIntersectionL, method);
345
346                                 if(isSplitDone == false)
347                                 {
348                                     s++;
349                                     isSplitDone = true;
350                                 }
351                             }
352                             pIntersectionV.clear();
353                             pIntersectionL.clear();
354                         }
355                     }
356
357                     // delete the lines, rectangles, and cuboids that were
                        tagged for deletion
358                     mCFLines.erase(std::remove_if(mCFLines.begin(),
                        mCFLines.end(), [](const auto& i){ return
                        i->deletion(); }), mCFLines.end());
359
                        mCFRectangles.erase(std::remove_if(mCFRectangles.begin(),
                         mCFRectangles.end(), [](const auto& i){ return
                        i->deletion(); }), mCFRectangles.end());
360                     mCFCuboids.erase(std::remove_if(mCFCuboids.begin(),
                        mCFCuboids.end(), [](const auto& i){ return
                        i->deletion(); }), mCFCuboids.end());
361                 }
362             }
363
364             // delete the lines, rectangles, and cuboids that were tagged
                for deletion
365             mCFLines.erase(std::remove_if(mCFLines.begin(), mCFLines.end(),
                [](const auto& i){ return i->deletion(); }), mCFLines.end());
366             mCFRectangles.erase(std::remove_if(mCFRectangles.begin(),
                mCFRectangles.end(), [](const auto& i){ return i->deletion();
                }), mCFRectangles.end());
367             mCFCuboids.erase(std::remove_if(mCFCuboids.begin(),
                mCFCuboids.end(), [](const auto& i){ return i->deletion(); }),
                mCFCuboids.end());
368         }
369     } //makeConformalN()
370
371
372     void cf_building_model::makeConformalT()
373     {
374         unsigned int t = 0;
375         while (t < 1) //Repeat
376         {
377             t = mCFVertices.size();
```

```
378
379             // check for line line intersections, and add the found vertex
                to the geometry model
380             utilities::geometry::vertex pIntersection;
381             for (unsigned int i = 0; i < mCFLines.size(); ++i)
382             {
383                 for (unsigned int j = i+1; j < mCFLines.size(); ++j)
384                 {
385                     if (mCFLines[i]->intersectsWith(*(mCFLines[j]),
                        pIntersection, mTol))
386                     {
387                         this->addVertex(pIntersection);
388                     }
389                 }
390             }
391
392             // check for line - rectangle intersections, and add the found
                vertex to the geometry model
393             for (unsigned int i = 0; i < mCFTriangles.size(); ++i)
394             {
395                 for (unsigned int j = 0; j < mCFLines.size(); ++j)
396                 {
397                     if (mCFTriangles[i]->intersectsWith(*(mCFLines[j]),
                        pIntersection, mTol))
398                     {
399                         this->addVertex(pIntersection);
400                     }
401                 }
402             }
403
404             // Send to step 2
405             unsigned int i = 0;
406             while (i < mCFVertices.size())
407             {
408                 for (auto& j : mCFSpaces)
409                 {
410                     j->checkVertexT(mCFVertices[i]);
411                 }
412                 ++i;
413             }
414
415
416             // delete the lines, rectangles, and cuboids that were tagged
                for deletion
417             mCFLines.erase(std::remove_if(mCFLines.begin(), mCFLines.end(),
                [](const auto& i){ return i->deletion(); }), mCFLines.end());
418             mCFTriangles.erase(std::remove_if(mCFTriangles.begin(),
                mCFTriangles.end(), [](const auto& i){ return i->deletion();
                }), mCFTriangles.end());
419             mCFTetrahedrons.erase(std::remove_if(mCFTetrahedrons.begin(),
                mCFTetrahedrons.end(), [](const auto& i){ return i->deletion();
                }), mCFTetrahedrons.end());
420         }
421     } //makeConformalT()
422
423
```

```
424
425        void cf_building_model::makeConformal() // Origonal orthogonal
           rectangular geometry conformal method
426        { //
427            // check for line line intersections, and add the found vertex to
               the geometry model
428            utilities::geometry::vertex pIntersection;
429            for (unsigned int i = 0; i < mCFLines.size(); ++i)
430            {
431                for (unsigned int j = i+1; j < mCFLines.size(); ++j)
432                {
433                    if (mCFLines[i]->intersectsWith(*(mCFLines[j]),
                       pIntersection, mTol))
434                    {
435                        this->addVertex(pIntersection);
436                    }
437                }
438            }
439
440
441            // check for line - rectangle intersections, and add the found
               vertex to the geometry model
442            for (unsigned int i = 0; i < mCFRectangles.size(); ++i)
443            {
444                for (unsigned int j = 0; j < mCFLines.size(); ++j)
445                {
446                    if (mCFRectangles[i]->intersectsWith(*(mCFLines[j]),
                       pIntersection, mTol))
447                    {
448                        this->addVertex(pIntersection);
449                    }
450                }
451            }
452
453
454            // check each vertex if it intersects with any space
455            unsigned int i = 0;
456            while (i < mCFVertices.size())
457            {
458                for (auto& j : mCFSpaces)
459                {
460                    j->checkVertex(mCFVertices[i]);
461                }
462                ++i;
463            }
464
465
466            // delete the lines, rectangles, and cuboids that were tagged for
               deletion
467            mCFLines.erase(std::remove_if(mCFLines.begin(), mCFLines.end(),
               [](const auto& i){ return i->deletion(); }), mCFLines.end());
468            mCFRectangles.erase(std::remove_if(mCFRectangles.begin(),
               mCFRectangles.end(), [](const auto& i){ return i->deletion(); }),
               mCFRectangles.end());
469            mCFCuboids.erase(std::remove_if(mCFCuboids.begin(),
               mCFCuboids.end(), [](const auto& i){ return i->deletion(); }),
```

```
            mCFCuboids.end());
470
471         } //  makeConformal()
472
473         cf_building_model::cf_building_model(const cf_building_model& rhs)
474         {
475             auto newPtr = new cf_building_model(rhs.mMSModel, rhs.mTol);
476             *this = *newPtr;
477         } // copy ctor()
478
479         cf_building_model::cf_building_model(const ms_building& msModel, const
            double& tol /*= 1e-3*/)
480         :   cf_geometry_model(tol), mMSModel(msModel), mTol(tol) //constructor
            using standart geometry conformal method
481         { //
482             // select method to use:
483             //Standart method
484             char method = 'I';
485             // I = quad-hexahedron; sub-method: perpendicular to intersection
                side
486             // O = quad-hexahedorn; sub-method: perpendicular to oposide side
487             // M = quad-hexahedorn; sub-method: middle ratio method (not
                finisched)
488             // T = tetrahedron method
489
490             if (method == 'I' || method == 'O' || method == 'M')
491             {
492                 for (const auto& i : mMSModel)
493                 {
494                     addSpace(*i);
495                 }
496                 this->makeConformalN(method);
497             }
498             else if (method == 'T')
499             {
500                 for (const auto& i : mMSModel)
501                 {
502                     addSpaceT(*i);
503                 }
504                 this->makeConformalT();
505             }
506         } //
507
508         cf_building_model::cf_building_model(const ms_building& msModel, char
            method, const double& tol /*= 1e-3*/)
509         :   cf_geometry_model(tol), mMSModel(msModel), mTol(tol) //constructor
            for user defined geometry conformal method
510         { //
511             // select method to use:
512
513             //char method = 'I';
514             // I = perpendicular to intersection side
515             // O = perpendicular to oposide side
516             // M = middle ratio method
517             // T = tetrahedron method
518
```

```
519             if (method == 'I' || method == 'O' || method == 'M')
520             {
521                 for (const auto& i : mMSModel)
522                 {
523                     addSpace(*i);
524                 }
525                 this->makeConformalN(method);
526             }
527             else if (method == 'T')
528             {
529                 for (const auto& i : mMSModel)
530                 {
531                     addSpaceT(*i);
532                 }
533                 this->makeConformalT();
534             }
535             else
536             {
537                 std::stringstream errorMessage;
538                 errorMessage << "\nError, the choosen split method is
                    non-existing \n"
539                              << "(bso/spatial_design/conformal/cf_building_model)."
                                 << std::endl;
540                 throw std::runtime_error(errorMessage.str());
541             }
542         } //
543 
544         cf_building_model::~cf_building_model()
545         { //
546             for (auto& i : mCFSpaces) delete i;
547             for (auto& i : mCFSurfaces) delete i;
548             for (auto& i : mCFEdges) delete i;
549             for (auto& i : mCFPoints) delete i;
550         } //
551 
552     } // conformal
553     } // spatial_design
554     } // bso
555 
556     #endif // CF_BUILDING_MODEL_CPP
```

A.4 cf_geometry_model

```
1   #ifndef CF_GEOMETRY_MODEL_HPP
2   #define CF_GEOMETRY_MODEL_HPP
3
4   namespace bso { namespace spatial_design { namespace conformal {
5
6       class cf_geometry_model
7       {
8       protected:
9           double mTol;
10          int mDec;
11
12          std::vector<cf_vertex*> mCFVertices;
13          std::vector<cf_line*> mCFLines;
14          std::vector<cf_rectangle*> mCFRectangles;
15          std::vector<cf_triangle*> mCFTriangles;
16          std::vector<cf_cuboid*> mCFCuboids;
17          std::vector<cf_tetrahedron*> mCFTetrahedrons;
18      public:
19          cf_geometry_model(const double& tol = 1e-3);
20          ~cf_geometry_model();
21
22          const double& tolerance() const {return mTol;}
23
24          cf_vertex* addVertex(const bso::utilities::geometry::vertex& p);
25          cf_line* addLine(const bso::utilities::geometry::line_segment& l);
26          cf_rectangle* addRectangle(const
            bso::utilities::geometry::quadrilateral& quad);
27          cf_triangle* addTriangle(const bso::utilities::geometry::triangle&
            tri);
28          cf_cuboid* addCuboid(const
            bso::utilities::geometry::quad_hexahedron& qhex);
29          cf_tetrahedron* addTetrahedron(const
            bso::utilities::geometry::tetrahedron& tetra);
30
31          void removeLine(cf_line* lPtr);
32          void removeRectangle(cf_rectangle* recPtr);
33          void removeTriangle(cf_triangle* triPtr);
34          void removeCuboid(cf_cuboid* cubPtr);
35          void removeTetrahedron(cf_tetrahedron* tetPtr);
36
37          const std::vector<cf_vertex*>&      cfVertices()        const {
            return mCFVertices;}
38          const std::vector<cf_line*>&        cfLines()           const {
            return mCFLines;}
39          const std::vector<cf_rectangle*>&   cfRectangles()      const {
            return mCFRectangles;}
40          const std::vector<cf_triangle*>&    cfTriangles()       const {
            return mCFTriangles;}
41          const std::vector<cf_cuboid*>&      cfCuboids()         const {
            return mCFCuboids;}
42          const std::vector<cf_tetrahedron*>& cfTetrahedrons()    const {
            return mCFTetrahedrons;}
43      };
44
45  } // conformal
46  } // spatial_design
```

```
47  } // bso
48
49  #endif // CF_GEOMETRY_MODEL_HPP
```

```
1   #ifndef CF_BUILDING_MODEL_CPP
2   #define CF_BUILDING_MODEL_CPP
3
4   #include <bitset>
5
6   namespace bso { namespace spatial_design { namespace conformal {
7
8       void cf_building_model::addSpace(const ms_space& msSpace) // initialize
        quad-hexahedorn geometry conformal model
9       { //
10          mCFSpaces.push_back(new cf_space(msSpace.getGeometry(), this));
11          mCFSpaces.back()->setSpaceID(msSpace.getID());
12          std::string possibleSpaceType;
13          if (msSpace.getSpaceType(possibleSpaceType))
14          {
15              mCFSpaces.back()->setSpaceType(possibleSpaceType);
16          }
17
18          auto spPtr = mCFSpaces.back();
19          for (const auto& i : *spPtr)
20          {
21              mCFPoints.push_back(new cf_point(i, this));
22              mCFPoints.back()->addSpace(spPtr);
23              spPtr->addPoint(mCFPoints.back());
24          }
25
26          for (const auto& i : spPtr->getLines())
27          {
28              mCFEdges.push_back(new cf_edge(i, this));
29              mCFEdges.back()->addSpace(spPtr);
30              spPtr->addEdge(mCFEdges.back());
31          }
32
33          std::vector<std::string> possibleSurfaceTypes;
34          bool surfaceTypesAvailable =
            msSpace.getSurfaceTypes(possibleSurfaceTypes);
35          if (surfaceTypesAvailable)
36          {
37              std::swap(possibleSurfaceTypes[0],possibleSurfaceTypes[2]);
38              std::swap(possibleSurfaceTypes[4],possibleSurfaceTypes[5]);
39          }
40          auto typeIte = possibleSurfaceTypes.begin();
41
42          for (const auto& i : spPtr->getPolygons())
43          {
44              mCFSurfaces.push_back(new cf_surface(*i, this));
45              mCFSurfaces.back()->addSpace(spPtr);
46              spPtr->addSurface(mCFSurfaces.back());
47              if (surfaceTypesAvailable)
48              {
49                  mCFSurfaces.back()->setSurfaceType(*typeIte);
50                  ++typeIte;
51              }
52          }
53      } // addSpace
54
```

```
55      void cf_building_model::addSpaceT(const ms_space& msSpace) //
        initialize tetrahedron geometry conformal model
56      {
57          bso::utilities::geometry::quad_hexahedron
            temp(msSpace.getGeometry());
58          mCFSpaces.push_back(new cf_space(msSpace.getGeometry(), this ,
            (temp.getTetrahedrons())));
59
60
61          // Add space types
62          mCFSpaces.back()->setSpaceID(msSpace.getID());
63          std::string possibleSpaceType;
64          if (msSpace.getSpaceType(possibleSpaceType))
65          {
66              mCFSpaces.back()->setSpaceType(possibleSpaceType);
67          }
68
69
70          // generate the triangle entities of the tetrahedron, since
            otherwise the polygon does not convert
71          std::vector<utilities::geometry::triangle*> tri;
72          for(const auto i: ((mCFSpaces.back() -> getTetrahedrons())))
73          {
74              for(const auto j: i.getPolygons())
75              {
76                  std::vector <utilities::geometry::vertex> triVertex;
77                  for(const auto k: j->getVertices())
78                  {
79                      triVertex.push_back(k);
80                  }
81                  tri.push_back(new utilities::geometry::triangle(triVertex));
82              }
83          }
84
85
86          // Add the geometry entities to cf_geometry_model and
            cf_building_model
87          auto spPtr = mCFSpaces.back();
88          for (const auto& i : *spPtr)
89          {
90              mCFPoints.push_back(new cf_point(i, this)); // add points of
                space spPtr to mCFSpaces and initate constoctor, which adds the
                points to cf_geometry_model mCFVertices
91              mCFPoints.back()->addSpace(spPtr); // store the space to which
                the points belongs in the mCFSpaces of cf_point
92              spPtr->addPoint(mCFPoints.back()); // add the cf_point to the
                mCFSpaces of the cf_space to which it belongs point belongs
93          }
94
95          for (const auto& i : spPtr->getLines())
96          {
97              mCFEdges.push_back(new cf_edge(i, this));
98              mCFEdges.back()->addSpace(spPtr);
99              spPtr->addEdge(mCFEdges.back());
100         }
101
```

```
102         std::vector<std::string> possibleSurfaceTypes;
103         bool surfaceTypesAvailable =
            msSpace.getSurfaceTypes(possibleSurfaceTypes);
104         if (surfaceTypesAvailable)
105         {
106             std::swap(possibleSurfaceTypes[0],possibleSurfaceTypes[2]);
107             std::swap(possibleSurfaceTypes[4],possibleSurfaceTypes[5]);
108         }
109         auto typeIte = possibleSurfaceTypes.begin();
110
111         for (const auto& i : spPtr->getPolygons())
112         {
113             std::vector<utilities::geometry::triangle*> triOnPoly;
114             double tol = tolerance();
115
116             //find triangles from tetrahedron which are on the space
                polygon to link to this surface
117             for(const auto l: tri)
118             {
119                 bool onFace = true;
120                 for(const auto k: l->getVertices())
121                 {
122                     if
                        ((std::find((i->getVertices()).begin(),(i->getVertices())
                        .end(), k) == (i->getVertices()).end()) == false)
123                     {
124                         onFace = false;
125                     }
126                 }
127
128                 if(onFace == true)
129                 {
130                     triOnPoly.push_back(l);
131                 }
132             }
133
134             // add surface and triangles
135
136             mCFSurfaces.push_back(new cf_surface(*i, this, triOnPoly));
137             mCFSurfaces.back()->addSpace(spPtr);
138             spPtr->addSurface(mCFSurfaces.back());
139             if (surfaceTypesAvailable)
140             {
141                 mCFSurfaces.back()->setSurfaceType(*typeIte);
142                 ++typeIte;
143             }
144
145         }
146     } //addSpaceT
147
148     void cf_building_model::makeConformalN(char method) // Step one
        quad-hexahedron method
149     { //
150         std::vector <cf_vertex> pIntersectionV;
151         std::vector <utilities::geometry::line_segment> pIntersectionL;
152         utilities::geometry::vertex pIntersection;
```

A.5 cf_entity

```
1   #ifndef CF_ENTITY_HPP
2   #define CF_ENTITY_HPP
3
4   namespace bso { namespace spatial_design { namespace conformal {
5
6       class cf_entity
7       {
8       protected:
9           std::vector<cf_vertex*> mCFVertices;
10          std::vector<cf_line*> mCFLines;
11          std::vector<cf_rectangle*> mCFRectangles;
12          std::vector<cf_cuboid*> mCFCuboids;
13          std::vector<cf_tetrahedron*> mCFTetrahedrons;
14          std::vector<cf_triangle*> mCFTriangles;
15
16          std::vector<cf_point*> mCFPoints;
17          std::vector<cf_edge*> mCFEdges;
18          std::vector<cf_surface*> mCFSurfaces;
19          std::vector<cf_space*> mCFSpaces;
20      public:
21          void addLine                (cf_line*          lPtr);
22          void addRectangle           (cf_rectangle*     recPtr);
23          void addTriangle            (cf_triangle*      triPtr);
24          void addCuboid              (cf_cuboid*        cubPtr);
25          void addTetrahedron         (cf_tetrahedron*   tetPtr);
26          void removeLine             (cf_line*          lPtr);
27          void removeRectangle        (cf_rectangle*     recPtr);
28          void removeTriangle         (cf_triangle*      triPtr);
29          void removeCuboid           (cf_cuboid*        cubPtr);
30          void removeTetrahedron      (cf_tetrahedron*   tetPtr);
31          void addPoint               (cf_point*         pPtr);
32          void addEdge                (cf_edge*          ePtr);
33          void addSurface             (cf_surface*       srfPtr);
34          void addSpace               (cf_space*         spPtr);
35
36          const std::vector<cf_vertex*       >& cfVertices()      const {
            return mCFVertices;}
37          const std::vector<cf_line*         >& cfLines()         const {
            return mCFLines;}
38          const std::vector<cf_rectangle*    >& cfRectangles()    const {
            return mCFRectangles;}
39          const std::vector<cf_triangle*     >& cfTriangles()     const {
            return mCFTriangles;}
40          const std::vector<cf_cuboid*       >& cfCuboids()       const {
            return mCFCuboids;}
41          const std::vector<cf_tetrahedron*  >& cfTetrahedrons()  const {
            return mCFTetrahedrons;}
42          const std::vector<cf_point*        >& cfPoints()        const {
            return mCFPoints;}
43          const std::vector<cf_edge*         >& cfEdges()         const {
            return mCFEdges;}
44          const std::vector<cf_surface*      >& cfSurfaces()      const {
            return mCFSurfaces;}
45          const std::vector<cf_space*        >& cfSpaces()        const {
            return mCFSpaces;}
46      };
```

```
47
48  } // conformal
49  } // spatial_design
50  } // bso
51
52  #endif // CF_ENTITY_HPP
```

```
1   #ifndef CF_GEOMETRY_ENTITY_CPP
2   #define CF_GEOMETRY_ENTITY_CPP
3
4   namespace bso { namespace spatial_design { namespace conformal {
5
6       void cf_entity::addLine(cf_line* lPtr)
7       { //
8           if (std::find(mCFLines.begin(),mCFLines.end(),lPtr) ==
            mCFLines.end())
9           {
10              mCFLines.push_back(lPtr);
11          }
12      } //
13
14      void cf_entity::addRectangle(cf_rectangle* recPtr)
15      { //
16          if (std::find(mCFRectangles.begin(),mCFRectangles.end(),recPtr) ==
            mCFRectangles.end())
17          {
18              mCFRectangles.push_back(recPtr);
19          }
20      } //
21
22      void cf_entity::addTriangle(cf_triangle* triPtr)
23      { //
24          if (std::find(mCFTriangles.begin(),mCFTriangles.end(),triPtr) ==
            mCFTriangles.end())
25          {
26              mCFTriangles.push_back(triPtr);
27          }
28      } //
29
30      void cf_entity::addCuboid(cf_cuboid* cubPtr)
31      { //
32          if (std::find(mCFCuboids.begin(),mCFCuboids.end(),cubPtr) ==
            mCFCuboids.end())
33          {
34              mCFCuboids.push_back(cubPtr);
35          }
36      } //
37
38      void cf_entity::addTetrahedron(cf_tetrahedron*  tetPtr)
39      { //
40          if (std::find(mCFTetrahedrons.begin(),mCFTetrahedrons.end(),tetPtr)
            == mCFTetrahedrons.end())
41          {
42              mCFTetrahedrons.push_back(tetPtr);
43          }
44      } //
45
46      void cf_entity::removeLine(cf_line* lPtr)
47      { //
48          mCFLines.erase(std::remove(mCFLines.begin(),mCFLines.end(),lPtr),
            mCFLines.end());
49      } //
50
```

```
51      void cf_entity::removeRectangle(cf_rectangle* recPtr)
52      { //
53
            mCFRectangles.erase(std::remove(mCFRectangles.begin(),mCFRectangles.e
            nd(),recPtr), mCFRectangles.end());
54      } //
55
56      void cf_entity::removeTriangle(cf_triangle* triPtr)
57      { //
58
            mCFTriangles.erase(std::remove(mCFTriangles.begin(),mCFTriangles.end(
            ),triPtr), mCFTriangles.end());
59      } //
60
61      void cf_entity::removeCuboid(cf_cuboid* cubPtr)
62      { //
63
            mCFCuboids.erase(std::remove(mCFCuboids.begin(),mCFCuboids.end(),cubP
            tr), mCFCuboids.end());
64      } //
65
66      void cf_entity::removeTetrahedron(cf_tetrahedron* tetPtr)
67      { //
68
            mCFTetrahedrons.erase(std::remove(mCFTetrahedrons.begin(),mCFTetrahed
            rons.end(),tetPtr), mCFTetrahedrons.end());
69      } //
70
71      void cf_entity::addPoint(cf_point* pPtr)
72      { //
73          if (std::find(mCFPoints.begin(),mCFPoints.end(),pPtr) ==
            mCFPoints.end())
74          {
75              mCFPoints.push_back(pPtr);
76          }
77      } //
78
79      void cf_entity::addEdge(cf_edge* ePtr)
80      { //
81          if (std::find(mCFEdges.begin(),mCFEdges.end(),ePtr) ==
            mCFEdges.end())
82          {
83              mCFEdges.push_back(ePtr);
84          }
85      } //
86
87      void cf_entity::addSurface(cf_surface* srfPtr)
88      { //
89          if (std::find(mCFSurfaces.begin(),mCFSurfaces.end(),srfPtr) ==
            mCFSurfaces.end())
90          {
91              mCFSurfaces.push_back(srfPtr);
92          }
93      } //
94
95      void cf_entity::addSpace(cf_space* spPtr)
```

```
96          { //
97              if (std::find(mCFSpaces.begin(),mCFSpaces.end(),spPtr) ==
                mCFSpaces.end())
98              {
99                  mCFSpaces.push_back(spPtr);
100             }
101         } //
102
103
104     } // conformal
105     } // spatial_design
106     } // bso
107
108     #endif // CF_GEOMETRY_ENTITY_CPP
```

A.6 cf_space

```
1   #ifndef CF_SPACE_HPP
2   #define CF_SPACE_HPP
3
4   namespace bso { namespace spatial_design { namespace conformal {
5
6       class cf_space : public utilities::geometry::quad_hexahedron,
7                                         public cf_building_entity
8       {
9       private:
10          std::string mSpaceType = "";
11          unsigned int mSpaceID = 0;
12      public:
13          cf_space(const utilities::geometry::quad_hexahedron& rhs,
            cf_building_model* buildingModel);
14          cf_space(const utilities::geometry::quad_hexahedron& rhs,
            cf_building_model* buildingModel,
            std::vector<utilities::geometry::tetrahedron> tet);
15
16          void checkVertex(cf_vertex* pPtr);
17          void checkVertexN(std::vector<cf_vertex>& intsecV, std::vector
            <utilities::geometry::line_segment> intsecL, char method);
18          void checkVertexT(cf_vertex* pPtr);
19
20          void splitLines(const std::vector<cf_vertex>& intsecV, std::vector
            <utilities::geometry::line_segment>& intsecL);
21
22          void addLine                (cf_line*        lPtr    )   =
            delete;
23          void addRectangle           (cf_rectangle*   recPtr)     =
            delete;
24          void addTriangle            (cf_triangle*      triPtr)   =
            delete;
25          void removeLine             (cf_line*        lPtr    )   =
            delete;
26          void removeRectangle        (cf_rectangle*   recPtr)     =
            delete;
27          void removeTriangle         (cf_triangle*      triPtr)   =
            delete;
28          void addSpace               (cf_space*       spPtr   )   =
            delete;
29
30          void setSpaceID(const unsigned int& spaceID) {mSpaceID = spaceID;}
31          void setSpaceType(const std::string& spaceType) {mSpaceType =
            spaceType;}
32
33          const std::vector<cf_vertex*       >& cfVertices()     const {
            return mCFVertices;}
34          const std::vector<cf_line*         >& cfLines()            const {
            return mCFLines;}
35          const std::vector<cf_rectangle*    >& cfRectangles() const {
            return mCFRectangles;}
36          const std::vector<cf_space*        >& cfSpaces()       const {
            return mCFSpaces;}
37
38          const unsigned int getSpaceID() const {return mSpaceID;}
39          const std::string getSpaceType() const {return mSpaceType;}
```

```
40        };
41
42    } // conformal
43    } // spatial_design
44    } // bso
45
46    #endif // CF_SPACE_HPP
```

```
1    #ifndef CF_SPACE_CPP
2    #define CF_SPACE_CPP
3
4    #include <bso/utilities/sort_clockwise.hpp>
5
6    namespace bso { namespace spatial_design { namespace conformal {
7
8        cf_space::cf_space(const utilities::geometry::quad_hexahedron& rhs,
         cf_building_model* buildingModel)
9        :  utilities::geometry::quad_hexahedron(rhs,buildingModel->tolerance())
10       { //
11           mBuildingModel = buildingModel;
12           mCFCuboids.push_back(mBuildingModel->addCuboid(*this));
13           mCFCuboids.back()->addSpace(this);
14       } //
15
16       cf_space::cf_space(const utilities::geometry::quad_hexahedron& rhs,
         cf_building_model* buildingModel,
         std::vector<utilities::geometry::tetrahedron> tet)
17       :  utilities::geometry::quad_hexahedron(rhs,buildingModel->tolerance())
18       { //
19           mBuildingModel = buildingModel;
20           for(const auto i : tet)
21           {
22               mCFTetrahedrons.push_back(mBuildingModel->addTetrahedron(i));
23               mCFTetrahedrons.back()->addSpace(this);
24           }
25       } //
26
27       void cf_space::checkVertex(cf_vertex* pPtr) // orthogonal_rectangular
         step 2 of geometry conformal method
28       { //
29
30           // check if the vertex is in or on the space's geometry
31           bool onGeometry = false;
32
33           if ((this->isInside(*pPtr, mBuildingModel->tolerance())))
             onGeometry = true;
34
35           for (const auto& i : mPolygons)
36           {
37               if (onGeometry) break;
38               if (i->isInside(*pPtr, mBuildingModel->tolerance()))
39               {
40                   onGeometry = true;
41                   break;
42               }
43           }
44
45           for (const auto& i : mLineSegments)
46           {
47               if (onGeometry) break;
48               if (i.isOnLine(*pPtr, mBuildingModel->tolerance()))
49               {
50                   onGeometry = true;
51                   break;
```

```
52                  }
53              }
54
55              if (!onGeometry) return;  // if it isn't return the function
                immediately
56
57              // check each cuboid if it can be split by the vertex
58              for (unsigned int i = 0; i < mCFCuboids.size(); ++i)
59              {
60                  mCFCuboids[i]->split(pPtr);
61              }
62          } // checkVertex
63
64
65          void cf_space::checkVertexN(std::vector<cf_vertex>& intsecV,
            std::vector <utilities::geometry::line_segment> intsecL, char method)
            // quad-hexahedron method step 2
66          { //
67              // split intsecL on the hand of the found intsecV
68              cf_space::splitLines(intsecV, intsecL);
69
70
71              // check if there are vertexes on a polygon of the space's geometry
72              for (int i=0; i < intsecV.size(); i++)
73              {
74                  bool onGeometry = false;
75                  for (unsigned int k = 0; k < mCFCuboids.size(); ++k)
76                  {
77                      std::vector <int> onPolyVIndex;
78                      std::vector <utilities::geometry::line_segment>
                        linesToAccound;
79                      for (const auto& j : mCFCuboids[k]->cfRectangles())
80                      {
81                          if (onGeometry) {break;}
82                          if (j->isInside(intsecV[i], mBuildingModel->tolerance()))
83                          {
84                              onGeometry = true;
85
86                              // find the conected lines to onPolyIntersection to
                                accound for
87                              for(const auto l : intsecL)
88                              {
89                                  if(l[0].isSameAs(intsecV[i],
                                    mBuildingModel->tolerance()) ||
                                    l[1].isSameAs(intsecV[i],
                                    mBuildingModel->tolerance()))
90                                  {
91                                      linesToAccound.push_back(l);
92                                  }
93                              }
94
95                              // generate the vector of vertices to split with
96                              std::vector<cf_vertex> splitVertexen;
97                              splitVertexen.push_back(intsecV[i]);
98
99                              for(const auto k : linesToAccound)
```

```
100                         {
101                             if(k[0].isSameAs(intsecV[i],
                                mBuildingModel->tolerance()))
102                             {
103                                 splitVertexen.push_back(k[1]);
104                             }
105                             if(k[1].isSameAs(intsecV[i],
                                mBuildingModel->tolerance()))
106                             {
107                                 splitVertexen.push_back(k[0]);
108                             }
109                         }
110
111
112                         // remove ducplicates from splitVertexen
113                         auto splitVertexen_end = splitVertexen.end();
114                         for(auto it = splitVertexen.begin(); it !=
                            splitVertexen_end; ++it)
115                         {
116                             splitVertexen_end = std::remove(it + 1,
                                splitVertexen_end, *it);
117                         }
118                         splitVertexen.erase(splitVertexen_end,
                            splitVertexen.end());
119
120                         // pars split vertexes to split function
121                         mCFCuboids[k]->splitN(&splitVertexen, method);
122                         linesToAccound.clear();
123                     }
124                 }
125             }
126         }
127
128
129         // check if there are vertexes on the line_segments of the space's
            geometry
130         for (int i=0; i < intsecV.size(); i++)
131         {
132             for (unsigned int k = 0; k < mCFCuboids.size(); ++k)
133             {
134                 for (const auto& j : mCFCuboids[k]->cfLines())
135                 {
136                     if (j->isOnLine(intsecV[i], mBuildingModel->tolerance()))
137                     {
138                         std::vector <utilities::geometry::line_segment>
                            linesToAccound;
139
140                         // find the conected lines to onPolyIntersection to
                            accound for
141                         for(const auto l : intsecL)
142                         {
143                             if(l[0].isSameAs(intsecV[i],
                                mBuildingModel->tolerance()) ||
                                l[1].isSameAs(intsecV[i],
                                mBuildingModel->tolerance()))
144                             {
```

```
145                                 if ((mCFCuboids[k])->isInsideOrOn(l[0],
                                    mBuildingModel->tolerance()) &&
                                    (mCFCuboids[k])->isInsideOrOn(l[1],
                                    mBuildingModel->tolerance()))
146                                 {
147                                     linesToAccound.push_back(l);
148                                 }
149                             }
150                         }
151
152                         // generate the vector of vertices to split with
153                         std::vector<cf_vertex> splitVertexen;
154                         splitVertexen.push_back(intsecV[i]);
155
156                         for(const auto k : linesToAccound)
157                         {
158                             if(k[0].isSameAs(intsecV[i],
                                mBuildingModel->tolerance()))
159                             {
160                                 splitVertexen.push_back(k[1]);
161                             }
162                             if(k[1].isSameAs(intsecV[i],
                                mBuildingModel->tolerance()))
163                             {
164                                 splitVertexen.push_back(k[0]);
165                             }
166                         }
167
168                         // send vector of vertices to step 3 split
169                         mCFCuboids[k]->splitN(&splitVertexen, method);
170                         linesToAccound.clear();
171                     }
172                 }
173             }
174         }
175     } // checkVertexN
176
177
178     void cf_space::checkVertexT(cf_vertex* pPtr)
179     { //
180         // check if the vertex is in or on the space's geometry
181         bool onGeometry = false;
182
183         if ((this->isInside(*pPtr, mBuildingModel->tolerance())))
            onGeometry = true;
184
185         for (const auto& i : mPolygons)
186         {
187             if (onGeometry) break;
188             if (i->isInside(*pPtr, mBuildingModel->tolerance()))
189             {
190                 onGeometry = true;
191                 break;
192             }
193         }
194
```

```
195            for (const auto& i : mLineSegments)
196            {
197                if (onGeometry) break;
198                if (i.isOnLine(*pPtr, mBuildingModel->tolerance()))
199                {
200                    onGeometry = true;
201                    break;
202                }
203            }
204
205
206            if (!onGeometry) return;  // if it isn't return the function
            immediately
207
208            // check each cuboid if it can be split by the vertex
209
210            for (unsigned int i = 0; i < mCFTetrahedrons.size(); ++i)
211            {
212                mCFTetrahedrons[i]->split(pPtr);
213            }
214
215        } // checkVertexT
216
217
218        void cf_space::splitLines(const std::vector<cf_vertex>& intsecV,
        std::vector<utilities::geometry::line_segment>& intsecL)
219        {
220            // split lines in acordance to intersection points
221            for(int k; k<intsecL.size(); k++)
222            {
223                for(const auto l :intsecV)
224                {
225                    if(intsecL[k].isOnLine(l))
226                    {
227                        utilities::geometry::line_segment one =
                        {{(intsecL[k])[0]},{l}};
228                        utilities::geometry::line_segment two =
                        {{(intsecL[k])[1]},{l}};
229
230                        intsecL.push_back(one);
231                        intsecL.push_back(two);
232
233                        intsecL.erase(intsecL.begin() + k);
234                    }
235                }
236            }
237
238            // remove ducplicates from intsecL
239            auto intsecL_end = intsecL.end();
240            for(auto it = intsecL.begin(); it != intsecL_end; ++it)
241            {
242                intsecL_end = std::remove(it + 1, intsecL_end, *it);
243            }
244            intsecL.erase(intsecL_end, intsecL.end());
245
246
```

```
247             // remove ducplicates which has switched vertexes in intsecL
248             for(int k; k<intsecL.size(); k++)
249             {
250                 for(auto m : intsecL)
251                 {
252                     if (m == intsecL[k]){continue;}
253                     if (m.isSameAs(intsecL[k])){intsecL.erase(intsecL.begin() +
                        k);}
254                 }
255             }
256         }//cf_space::splitLines
257
258
259
260
261
262
263
264
265
266     } // conformal
267     } // spatial_design
268     } // bso
269
270     #endif // CF_SPACE_CPP
```

A.7 cf_cuboid

```
1   #ifndef CF_CUBOID_HPP
2   #define CF_CUBOID_HPP
3
4   namespace bso { namespace spatial_design { namespace conformal {
5
6       class cf_cuboid : public bso::utilities::geometry::quad_hexahedron,
7                                             public cf_geometry_entity
8       {
9       private:
10
11      public:
12          cf_cuboid(const utilities::geometry::quad_hexahedron& rhs,
            cf_geometry_model* geomModel);
13          ~cf_cuboid();
14
15          void split(cf_vertex* pPtr); // orthogonal rectangular split function
16          void splitN(std::vector<cf_vertex>* intsecV, char method); //
            non-orthogonal split function
17          void checkAssociated(cf_vertex* pPtr); // orthogonal rectangular
            split function to split the other cf entities to conformal entities
18          void checkAssociatedN(std::vector<cf_cuboid*> newCuboids); //
            non-orthogonal split function; send geometry entities to other
            cf_geometry for splitting
19
20          void addLine                (cf_line*          lPtr)       =
            delete ;
21          void addRectangle           (cf_rectangle*   recPtr)       =
            delete ;
22          void addTriangle            (cf_triangle*      triPtr)     =
            delete;
23          void addCuboid              (cf_cuboid*        cubPtr)     =
            delete ;
24          void addTetrahedron         (cf_tetrahedron*   tetPtr)     =
            delete ;
25          void removeLine             (cf_line*          lPtr)       =
            delete ;
26          void removeRectangle        (cf_rectangle*   recPtr)       =
            delete ;
27          void removeTriangle         (cf_triangle*      triPtr)     =
            delete;
28          void removeCuboid           (cf_cuboid*        cubPtr)     =
            delete ;
29          void removeTetrahedron      (cf_tetrahedron*   tetPtr)     =
            delete ;
30          void addPoint               (cf_point*         pPtr)       =
            delete ;
31          void addEdge                (cf_edge*              ePtr)   =
            delete ;
32          void addSurface             (cf_surface*       srfPtr)     =
            delete ;
33
34          const std::vector<cf_triangle*      >& cfTriangles()        = delete;
35          const std::vector<cf_cuboid*        >& cfCuboids()      = delete;
36          const std::vector<cf_tetrahedron*   >& cfTetrahedrons() = delete;
37          const std::vector<cf_point*         >& cfPoints()       = delete;
38          const std::vector<cf_edge*          >& cfEdges()        = delete;
```

```
39          const std::vector<cf_surface*       >& cfSurfaces()      = delete;
40      };
41
42  } // conformal
43  } // spatial_design
44  } // bso
45
46  #endif // CF_CUBOID_HPP
```

```
1    #ifndef CF_CUBOID_CPP
2    #define CF_CUBOID_CPP
3
4    namespace bso { namespace spatial_design { namespace conformal {
5
6        cf_cuboid::cf_cuboid(const utilities::geometry::quad_hexahedron& rhs,
         cf_geometry_model* geometryModel)
7        : utilities::geometry::quad_hexahedron(rhs, geometryModel->tolerance())
8        {
9            mGeometryModel = geometryModel;
10           for (const auto& i : mVertices)
11           {
12               mCFVertices.push_back(mGeometryModel->addVertex(i));
13               mCFVertices.back()->addCuboid(this);
14           }
15           for (const auto& i : mLineSegments)
16           {
17               mCFLines.push_back(mGeometryModel->addLine(i));
18               mCFLines.back()->addCuboid(this);
19           }
20           for (const auto& i : mPolygons)
21           {
22               mCFRectangles.push_back(mGeometryModel->addRectangle(*i));
23               mCFRectangles.back()->addCuboid(this);
24           }
25       } // ctor
26
27       cf_cuboid::~cf_cuboid()
28       {
29           for (const auto& i : mCFVertices) i->removeCuboid(this);
30           for (const auto& i : mCFLines) i->removeCuboid(this);
31           for (const auto& i : mCFRectangles) i->removeCuboid(this);
32       } // dtor
33
34
35       void cf_cuboid::splitN(std::vector<cf_vertex>* intsecV, char method) //
         step three quad-hexahedron split method
36       { //
37           // The first intsecV is the intersection vertex. The rest of the
             two point are guidance points.
38
39           //Declerations
40           std::vector <bool> isInside;
41           std::vector <bool> onPolygon;
42           std::vector <bool> onLine;
43
44           std::vector <utilities::geometry::vertex> isInsideV;
45           std::vector <utilities::geometry::vertex> onPolygonV;
46           std::vector <utilities::geometry::vertex> onLineV;
47
48           std::vector <utilities::geometry::polygon*> intersecP;
49           std::vector <utilities::geometry::line_segment> intersecL;
50
51           int counter = 0;
52
53           //Collect information for diferentiation between split methods for
```

```
            later use
54
55          for (const auto i: *intsecV)
56          {
57              if (this->isInside(i, mGeometryModel->tolerance()))
58              {
59                  isInsideV.push_back(i);
60              }
61              else
62              {
63                  isInside.push_back(false);
64              }
65          }
66
67          bool firstVertexOnPoly = false; // only if the first vertex is
            onPoly then the poly-split may continue
68          counter = 0;
69          for (const auto i: *intsecV)
70          {
71              unsigned int index = 0;
72              for (const auto& j : mPolygons)
73              {
74                  utilities::geometry::vertex temp = j->getPointClosestTo(i);
75
76                  if (j->isInside(i, mGeometryModel->tolerance()) &&
                    i.isSameAs(temp, mGeometryModel->tolerance())) // temp is
                    needed when the polygon consist of a (0,0,0) vertex,
                    otherwise its finds a intersection when there is none
77                  {
78                      onPolygonV.push_back(i);
79                      intersecP.push_back(j);
80                      onPolygon.push_back(true);
81                      if (counter == 0)
82                      {
83                          firstVertexOnPoly = true;
84                      }
85                      index++;
86                  }
87                  else
88                  {
89                      onPolygon.push_back(false);
90                  }
91              }
92              counter++;
93          }
94
95          bool firstVertexOnLine = false; // only if the first vertex is
            onLine then the line-split may continue
96          counter = 0;
97          for (const auto i: *intsecV)
98          {
99              for (const auto& j : mLineSegments)
100             {
101                 if (j.isOnLine(i,mGeometryModel->tolerance()))
102                 {
103                     onLineV.push_back(i);
```

```
104                         intersecL.push_back(j);
105                         onLine.push_back(true);
106                         if (counter == 0)
107                         {
108                             firstVertexOnLine = true;
109                         }
110                     }
111                     else
112                     {
113                         onLine.push_back(false);
114                     }
115 
116                 }
117                 counter++;
118             }
119 
120 
121             //split methods execution
122 
123             // I = perpendicular to intersection side
124             // O = perpendicular to oposide side
125             // M = middle ratio method
126 
127             std::vector<cf_vertex*> newVertices;
128             std::vector<cf_cuboid*> newCuboids;
129             bool split = false;
130 
131             // Not written yet
132             // split method for intersection points inside space which defide
            space into 8 cells
133             if (isInsideV.size() >= 1)
134             {
135                 //split = true;
136                 if (isInsideV.size() >= 2)
137                 {
138 
139                 }
140                 else
141                 {
142 
143                 }
144             } // split inside space
145 
146 
147             // split method for intersection point on polygon, split space into
            4 "cuboids"
148             if(!split)
149             {
150                 if (onPolygonV.size() >= 1 && firstVertexOnPoly == true)
151                 {
152                     split = true;
153 
154                     // find the opposite polygon to the intersection polygon
155                     std::vector <bool> sameCornpoint;
156                     unsigned int opposite = 6;
157 
```

```
158
159                 for (unsigned int j = 0; j < mPolygons.size(); j++)
160                 {
161                     bool sameCornp;
162
163                     for (auto m = (*mPolygons[j]).begin(); m !=
                        (*mPolygons[j]).end(); m++)
164                     {
165                         for (auto l = (*intersecP[0]).begin(); l !=
                            (*intersecP[0]).end(); l++)
166                         {
167                             if (m->isSameAs(*l, mGeometryModel->tolerance()))
168                             {
169                                 sameCornpoint.push_back(true);
170                             }
171                             else
172                             {
173                                 sameCornpoint.push_back(false);
174                             }
175                         }
176                     }
177
178                     bool isOposite = true;
179                     for (int a=0; a < sameCornpoint.size(); a++)
180                     {
181                         if (sameCornpoint[a] == true)
182                         {
183                             isOposite = false;
184                         }
185                     }
186
187                     if (isOposite == true)
188                     {
189                         opposite = j;
190                     }
191                     sameCornpoint.clear();
192                 }
193
194                 if (opposite == 6)
195                 {
196                     std::stringstream errorMessage;
197                     errorMessage << "\nError, could not find opposite
                    surface when\n"
198                                                  << "splitting a cuboid from a
                                                 point on a surface\n"
199                                                  <<
                                                 "(bso/spatial_design/conformal/c
                                                 f_cuboid)." << std::endl;
200                     throw std::runtime_error(errorMessage.str());
201                 }
202
203                 // Find the relevant line_segments to generate points on
                and store the first 2 known cornerpoints, namely the
                intersection point and the intersected side cornerpoint.
204                 for (const auto j : *(intersecP[0]))
205                 {
```

```
206                     // Declerations
207                     std::vector<utilities::geometry::vertex> cornerPoints;
208                     std::vector<utilities::geometry::line_segment>
                        intersPoly_line;
209                     std::vector<utilities::geometry::line_segment>
                        opositePoly_line;
210
211                     // Add first known cornerpoints
212                     cornerPoints.push_back(onPolygonV[0]);
213                     cornerPoints.push_back(j);
214
215
216
217                     // Find the lines within the polygon which are attached
                        to the cornerPoints of the cuboid, and the attached
                        line to vertex j
218                     for (const auto& k : mLineSegments)
219                     {
220                         for (auto l = (k).begin(); l != (k).end(); l++)
221                         {
222
223                             if(!(l->isSameAs(cornerPoints[1],
                                mGeometryModel->tolerance()))) // (*l !=
                                cornerPoints[1])
224                             { // only allow the continueation of
                                line_segments with are attached to the relevant
                                vertex
225                                 continue;
226                             }
227
228                             std::vector <bool> isPartOf;
229                             isPartOf.clear();
230
231                             for (const auto& m : intersecP[0]-> getLines())
232                             {// Determinde if line_segments is the same of
                                the polygon
                                line_segments
233                                 if (k.isSameAs(m,
                                    mGeometryModel->tolerance()))
234                                 { //if so then add closest point to
                                    intersection point to cornerpoints
235                                     intersPoly_line.push_back(k);
236                                     isPartOf.push_back(true);
237                                     continue;
238                                 }
239                                 else
240                                 {
241                                     isPartOf.push_back(false);
242                                 }
243                             }
244
245
246                             bool isPartOfPoly = false;
247                             for (int a=0; a < isPartOf.size(); a++)
248                             {
249                                 if (isPartOf[a] == true)
```

```
250                                 {
251                                     isPartOfPoly = true;
252                                 }
253                             }
254                             if (isPartOfPoly == true)
255                             { // line_segment is attacht to vertex j, but
                                is not in polygon
256                                 continue;
257                             }
258
259
260
261                             // find vertix on the oposide side conected to
                                vertex j
262
263
264                             // determine which of the point of non polygon
                                line is not the same as the vertex space, so
                                vertex of origonal cuboid on oposide side
265                             if(!(k[0].isSameAs(cornerPoints[1],
                                mGeometryModel->tolerance()))) //(k[0] !=
                                cornerPoints[1])
266                             {
267                                 cornerPoints.push_back(k[0]);
268                                 for (const auto& m : mPolygons[opposite]->
                                    getLines())
269                                 {
270                                     for (auto n = (m).begin(); n !=
                                        (m).end(); n++)
271                                     {
272                                         if(!(k[0].isSameAs(*n,
                                            mGeometryModel->tolerance())))//(k[0]
                                             != *n)
273                                         {
274                                             continue;
275                                         }
276
                                            opositePoly_line.push_back(m);

277                                     }
278                                 }
279                             }
280                             else if(!(k[1].isSameAs(cornerPoints[1],
                                mGeometryModel->tolerance()))) // (k[1] !=
                                cornerPoints[1])
281                             {
282                                 cornerPoints.push_back(k[1]);
283
284                                 for (const auto& m : mPolygons[opposite]->
                                    getLines())
285                                 {
286                                     for (auto n = (m).begin(); n !=
                                        (m).end(); n++)
287                                     {
288                                         if(!(k[1].isSameAs(*n,
                                            mGeometryModel->tolerance())))
```

```
                                    //(k[1] != *n)
289                                 {
290                                     continue;
291                                 }
292                                 opositePoly_line.push_back(m);
293                             }
294                         }
295                     }
296                 }
297             }
298
299
300
301
302             // Intersection Poly
303
304             //if onLineVertex are on the intersPoly_line then add
                the intersectionPoly_line_segments points to the
                cornerPoints value
305             //else if there are no intersection points on the
                intersectionPoly_linesegments, add the perpendicular
                points to intersection point
306             std::vector<bool> spaceInsecVInsert;
307             for (int b = 0; b < intersPoly_line.size(); b++)
308             {
309                 bool isFound = false;
310                 if (onPolygonV.size() > 1) // check for guidepoints
                    on polygon and project these points to the opposide
                    side
311                 {
312                     for (int l=1; l<onPolygonV.size(); l++)
313                     {
314                         bool ProjectionIntersect = false;
315                         bso::utilities::geometry::vertex
                            polyIntersectP;
316                         utilities::geometry::line_segment
                            polyIntersectLine =
                            {{cornerPoints[0]},{onPolygonV[l]}};
317
318                         ProjectionIntersect =
                            polyIntersectLine.vIntersects(intersPoly_line
                            [b],polyIntersectP);
319
320                         utilities::geometry::line_segment
                            cuboidLine =
                            {{cornerPoints[0]},{polyIntersectP}};
321
322                         if(ProjectionIntersect == true &&
                            intersPoly_line[b].isOnLine(polyIntersectP,
                            mGeometryModel->tolerance()) &&
                            cuboidLine.isOnLine(onPolygonV[l],
                            mGeometryModel->tolerance()))
323                         {
324                             cornerPoints.push_back(polyIntersectP);
325                             isFound = true;
326                             spaceInsecVInsert.push_back(true);
```

```
327                             }
328                         }
329                     }
330 
331                     for (const auto l: onLineV)
332                     {
333                         if
                            (intersPoly_line[b].isOnLine(l,mGeometryModel->to
                            lerance()))
334                         {
335                             cornerPoints.push_back(l);
336                             isFound = true;
337                             spaceInsecVInsert.push_back(true);
338                             continue;
339                         }
340                     }
341                     if(isFound == false) //
342                     {
343 
                            cornerPoints.push_back(intersPoly_line[b].getPoin
                            tClosestTo(onPolygonV[0]));
344                         spaceInsecVInsert.push_back(false);
345                     }
346                 }
347 
348                 //Opposide Poly, perpendicular to oposite line
349                 if (method == 'O')
350                 {
351 
                        cornerPoints.push_back(mPolygons[opposite]->getPointC
                        losestTo(onPolygonV[0])); // middle point on
                        opposite polygon
352                     for (int b = 0; b < intersPoly_line.size(); b++)
353                     {
354                         for(int c = 0; c < mPolygons.size(); c++)
355                         {
356                             bool isBothOnPoly1 = false;
357 
358                             if
                                (mPolygons[c]->isCoplanarN((intersPoly_line[b
                                ])[0], mGeometryModel->tolerance()))
359                             {
360                                 if
                                    (mPolygons[c]->isCoplanarN((intersPoly_li
                                    ne[b])[1], mGeometryModel->tolerance()))
361                                 {
362                                     isBothOnPoly1 = true;
363                                 }
364                             }
365                             if (isBothOnPoly1 == false){continue;}
366 
367                             for (int d = 0; d <
                                opositePoly_line.size(); d++)
368                             {
369                                 bool isBothOnPoly2 = false;
370                                 if
```

```
                                        (mPolygons[c]->isCoplanarN((opositePoly_l
                                        ine[d])[0],
                                        mGeometryModel->tolerance()))
371                                     {
372                                         if
                                            (mPolygons[c]->isCoplanarN((opositePo
                                            ly_line[d])[1],
                                            mGeometryModel->tolerance()))
373                                         {
374                                             isBothOnPoly2 = true;
375                                         }
376                                     }
377                                     if (isBothOnPoly2 == false){continue;}
378 
379                                     // found opposite line to
                                        intersPoly_line, therefore add the
                                        acording conerpoint (guidepoint or not):
380                                     if (spaceInsecVInsert[b] == false)
381                                     {
382 
                                            cornerPoints.push_back(opositePoly_li
                                            ne[d].getPointClosestTo(onPolygonV[0]
                                            ));
383                                     }
384                                     if (spaceInsecVInsert[b] == true)
385                                     {
386 
                                            cornerPoints.push_back(opositePoly_li
                                            ne[d].getPointClosestTo(cornerPoints[
                                            3+b]));
387                                     }
388                                 }
389                             }
390                         } //Opposide Poly, perpendicular to oposite line
391                     } // if (method == 'O')
392 
393 
394                     //Opposide Poly, perpendicular to intersection polygon.
395                     if (method == 'I')
396                     {
397                         for (int b = 0; b < intersPoly_line.size(); b++)
398                         {
399                             for(int c = 0; c < mPolygons.size(); c++)
400                             {
401                                 bool isBothOnPoly1 = false;
402 
403 
404                                 if
                                    (mPolygons[c]->isCoplanarN((intersPoly_line[b
                                    ])[0], mGeometryModel->tolerance()))
405                                 {
406                                     if
                                        (mPolygons[c]->isCoplanarN((intersPoly_li
                                        ne[b])[1], mGeometryModel->tolerance()))
407                                     {
408                                         isBothOnPoly1 = true;
```

```
409                 }
410             }
411             if (isBothOnPoly1 == false){continue;}
412 
413             for (int d = 0; d <
                opositePoly_line.size(); d++)
414             {
415                 bool isBothOnPoly2 = false;
416                 if
                    (mPolygons[c]->isCoplanarN((opositePoly_l
                    ine[d])[0],
                    mGeometryModel->tolerance()))
417                 {
418                     if
                        (mPolygons[c]->isCoplanarN((opositePo
                        ly_line[d])[1],
                        mGeometryModel->tolerance()))
419                     {
420 
421                         isBothOnPoly2 = true;
422                     }
423                 }
424                 if (isBothOnPoly2 == false){continue;}
425 
426 
427                 // found opposite line to
                    intersPoly_line, therefore add
                    conerpoint acordingly:
428                 if (spaceInsecVInsert[b] == false)
429                 {
430                     //determine where the line
                        intersects with the plane
431                     //
                        (http://geomalgorithms.com/a05-_inter
                        sect-1.html) background information
                        intersection point line-plane
432                     // defines plane to be intersected
433                     utilities::geometry::vector v3 =
                        cornerPoints[3+b]-cornerPoints[0];

434                     utilities::geometry::vector v4 =
                        intersecP[0]->normal().normalized();


435 
436                     //define intersection variables
437                     utilities::geometry::vertex pInt;
438                     double tol =
                        mGeometryModel->tolerance();
439                     utilities::geometry::line_segment
                        l1 = opositePoly_line[d];
440                     utilities::geometry::vector mNormal
                        = v3.cross(v4).normalized();
441 
442                     if
                        (l1.getVector().isPerpendicular(mNorm
                        al, tol))
```

```
443                                 {
444                                     std::stringstream errorMessage;
445                                     errorMessage << "\nError, could
                                        not find the intersection point
                                        on the opposite side \n"
446                                              << "during the polygon
                                                 split process, since
                                                 the line to intersect
                                                 is perpendicular to
                                                 the plane\n"
447                                              <<
                                                 "(bso/spatial_design/con
                                                 formal/cf_cuboid)." <<
                                                 std::endl;
448                                     throw
                                        std::runtime_error(errorMessage.s
                                        tr());
449                                 }
450                                 else
451                                 {
452                                     utilities::geometry::vector v1
                                        = l1[1]-l1[0];
453                                     utilities::geometry::vector v2
                                        = cornerPoints[3+b]-l1[0];
454
455                                     //check if the line intersects
                                        with the plane
456                                     double r =
                                        (mNormal.dot(v2))/(mNormal.dot(v1
                                        ));
457
458                                     if (r > -tol && r < (1 + tol))
459                                     { // if it does, compute the
                                        point
460                                         pInt = l1[0] + r * v1;
461                                         cornerPoints.push_back(pInt);
462                                     }
463                                     else
464                                     {
465                                         std::stringstream
                                            errorMessage;
466                                         errorMessage << "\nError,
                                            could not find the
                                            intersection point on the
                                            opposite side \n"
467                                                  << "during the
                                                     polygon split
                                                     process
                                                     perpendicular to
                                                     intersection
                                                     plane\n"
468                                                  <<
                                                     "(bso/spatial_design
                                                     /conformal/cf_cuboid
                                                     )." << std::endl;
469                                         throw
```

```
                            std::runtime_error(errorMessa
                            ge.str());
470                     }
471                 }
472
473
474             }
475             if (spaceInsecVInsert[b] == true)
476             {
477                 //determine where the line
                    intersects with the plane
478
479                 // defines plane to be intersected
480                 utilities::geometry::vector v3 =
                    cornerPoints[3+b]-cornerPoints[0];

481                 utilities::geometry::vector v4 =
                    intersecP[0]->normal().normalized();


482
483                 //define intersection variables
484                 utilities::geometry::vertex pInt;
485                 double tol =
                    mGeometryModel->tolerance();
486                 utilities::geometry::line_segment
                    l1 = opositePoly_line[d];
487                 utilities::geometry::vector mNormal
                    = v3.cross(v4).normalized();
488
489                 if
                    (l1.getVector().isPerpendicular(mNorm
                    al, tol))
490                 {
491                     std::stringstream errorMessage;
492                     errorMessage << "\nError, could
                        not find the intersection point
                        on the opposite side \n"
493                              << "during the polygon
                                 split process, since
                                 the line to intersect
                                 is perpendicular to
                                 the plane\n"
494                              <<
                                 "(bso/spatial_design/con
                                 formal/cf_cuboid)." <<
                                 std::endl;
495                     throw
                        std::runtime_error(errorMessage.s
                        tr());
496                 }
497                 else
498                 {
499                     utilities::geometry::vector v1
                        = l1[1]-l1[0];
500                     utilities::geometry::vector v2
                        = cornerPoints[3+b]-l1[0];
```

```
501
502                     //check if the line intersects
                        with the plane
503                     double r =
                        (mNormal.dot(v2))/(mNormal.dot(v1
                        ));
504
505                     if (r > -tol && r < (1 + tol))
506                     { // if it does, compute the
                        point
507                         pInt = l1[0] + r * v1;
508                         cornerPoints.push_back(pInt);
509                     }
510                     else
511                     {
512                         std::stringstream
                            errorMessage;
513                         errorMessage << "\nError,
                            could not find the
                            intersection point on the
                            opposite side \n"
514                                 << "during the
                                    polygon split
                                    process
                                    perpendicular to
                                    intersection
                                    plane\n"
515                                 <<
                                    "(bso/spatial_design
                                    /conformal/cf_cuboid
                                    )." << std::endl;
516                         throw
                            std::runtime_error(errorMessa
                            ge.str());
517                     }
518                 }
519             }
520         }
521     }
522
523
524     if(b == 0)
525     {// intersectionpoint on oposite polygon
526         utilities::geometry::vertex pInt;
527         double tol = mGeometryModel->tolerance();
528
529         utilities::geometry::vector normalIntsect =
            intersecP[0]->normal().normalized();
530         utilities::geometry::vector normalOposite =
            mPolygons[opposite]->normal().normalized();
531
532         utilities::geometry::vector v3 =
            cornerPoints[3] -cornerPoints[0];
533         utilities::geometry::vector v4 =
            cornerPoints[4] -cornerPoints[0];
534
```

```
535
536
537                 if (normalOposite.isPerpendicular(v3,
                    mGeometryModel->tolerance()) &&
                    normalOposite.isPerpendicular(v4,
                    mGeometryModel->tolerance()))
538                 {//check if normals oposite plane is
                    Perpendicular to the 2 vector within
                    intersection plane
539
                        cornerPoints.push_back(mPolygons[opposite
                        ]->getPointClosestTo(onPolygonV[0]));
                        // point on opposite
                        polygon
540                 }
541                 else
542                 {//determine where the line intersects with
                    the plane
543
544                     utilities::geometry::line_segment l1 =
                        {(cornerPoints[0]+50000*normalIntsect),co
                        rnerPoints[0]};
545                     utilities::geometry::vector mNormal =
                        mPolygons[opposite]->normal().normalized(
                        );
546
547                     if
                        (l1.getVector().isPerpendicular(mNormal,
                        tol))
548                     {
549                         std::stringstream errorMessage;
550                         errorMessage << "\nError, could not
                            find the intersection point on the
                            opposite side \n"
551                                   << "during the polygon
                                   split process, since the
                                   line to intersect is
                                   perpendicular to the
                                   plane\n"
552                                   <<
                                   "(bso/spatial_design/conform
                                   al/cf_cuboid)." <<
                                   std::endl;
553                         throw
                            std::runtime_error(errorMessage.str()
                            );
554                     }
555                     else
556                     {
557                         utilities::geometry::vector v1 =
                            l1[1]-l1[0];
558                         utilities::geometry::vector v2 =
                            mVertices[0]-l1[0];
559
560                         //check if the line intersects with
                            the plane
```

```
561                                     double r =
                                        (mNormal.dot(v2))/(mNormal.dot(v1));
562                                     if (r > -tol /*&& r < (1 + tol)*/)
563                                     { // if it does, compute the point
564                                         pInt = l1[0] + r * v1;
565                                         cornerPoints.push_back(pInt);
566                                     }
567                                     else
568                                     {
569                                         std::stringstream errorMessage;
570                                         errorMessage << "\nError, could
                                            not find the intersection point
                                            on the opposite side \n"
571                                                  << "during the split
                                                     process perpendicular
                                                     to intersection plane\n"
572                                                  <<
                                                     "(bso/spatial_design/con
                                                     formal/cf_cuboid)." <<
                                                     std::endl;
573                                         throw
                                            std::runtime_error(errorMessage.s
                                            tr());
574                                     }
575                                 }
576                             }
577                         }
578                     } //Opposide Poly, perpendicular to intersection
                        polygon.
579                 } // if (method == 'I')
580
581
582                 // Opposide Poly, middle ratio method
583                 if (method == 'M')
584                 {
585
586                 } // if (method == 'M')
587
588
589
590                 // safety checks
591                 // Check for duplicates
592                 bool duplicate = false;
593                 for(int t=0; t<cornerPoints.size(); t++)
594                 {
595                     for(int s=0; s<cornerPoints.size(); s++)
596                     {
597                         if(t == s)
598                         {
599                             continue;
600                         }
601                         if(cornerPoints[t].isSameAs(cornerPoints[s],
                            mGeometryModel->tolerance()))
602                         {
603                             std::stringstream errorMessage;
604                             errorMessage << "\nError, There are
```

```
                                duplicates inside the to generate
                                quad-hexahedron cornerPoints.\n"
605                                          << "In case the inserted
                                             tolerance is 0.1 mm, then the
                                             polygon intersection check can
                                             mistake a line \n"
606                                          << "for a polygon
                                             intersection. Therefore
                                             performing the wrong split
                                             case resulting in duplicate
                                             value's"
607                                          << "Solution, make tolerance
                                             smaller\n"
608                                      <<
                                         "(bso/spatial_design/conformal/cf_cu
                                         boid)." << std::endl;
609                             throw std::runtime_error(errorMessage.str());
610                         }
611                     }
612                 }
613 
614                 if (cornerPoints.size() < 8)
615                 {
616                     std::stringstream errorMessage;
617                     errorMessage << "\nError, There are not enough
                        value's in cornerpoints to\n"
618                              << "make a cuboids to split space during
                                 the split polygon precedure\n"
619                              <<
                                 "(bso/spatial_design/conformal/cf_cuboid)."
                                 << std::endl;
620                     throw std::runtime_error(errorMessage.str());
621                 }
622                 if (cornerPoints.size() > 8)
623                 {
624                     std::stringstream errorMessage;
625                     errorMessage << "\nError, There are too manny
                        value's in cornerpoints to\n"
626                              << "make a cuboids to split space during
                                 the split polygon precedure\n"
627                              <<
                                 "(bso/spatial_design/conformal/cf_cuboid)."
                                 << std::endl;
628                     throw std::runtime_error(errorMessage.str());
629                 }
630 
631                 // The new cuboids generator
632                 newCuboids.push_back(mGeometryModel->addCuboid(
633 
                        utilities::geometry::quad_hexahedron(cornerPoints
                        , mGeometryModel->tolerance())));
634             }
635         }
636     } // split polygon
637 
638 
```

```
639
640             // split method for intersection points on line_segments which
                defide the cell into 2 cells
641             if(!split)
642             {
643                 if (onLineV.size() >= 1 && firstVertexOnLine == true)
644                 {
645                     split = true;
646
647                     // check for multiple intersection vertexen/line
648
649                     if (onLineV.size() > 3)
650                     {// check if multiple intersection vertexen are on the same
                        polygons
651                         std::stringstream errorMessage;
652                             errorMessage << "\nError, can't split space when
                                there are more than 3 vertexes intersecting with
                                lines \n"
653                                                         << "with one or multiple
                                                            line_segments.\n"
654                                                         << "There are the
                                                            following amound of points
                                                            intersecting: " <<
                                                            onLineV.size() << " \n"
655                                                         <<
                                                            "(bso/spatial_design/conform
                                                            al/cf_cuboid)." <<
                                                            std::endl;
656                             throw std::runtime_error(errorMessage.str());
657                     }
658
659                     // if check is fine, then preceed on
660
661                     for (auto i = (intersecL[0]).begin(); i !=
                        (intersecL[0]).end(); i++)
662                     {
663                         // Declerations
664                         std::vector<utilities::geometry::vertex> cornerPoints;
665                         std::vector<utilities::geometry::polygon*>
                            polyInCuboid; //PolyInCuboid are the polygons which are
                            the polygons to the side which does not need splitting
666                         std::vector<utilities::geometry::line_segment>
                            lineToSplit; //Lines which need to be split for the
                            generation of a new cuboid
667
668                         // Add first known cornerPoints
669                         cornerPoints.push_back(onLineV[0]);
670
671                         // Find the PolyInCuboid geometry
672                         for (const auto& j : mPolygons)
673                         {
674                             for (auto k = (*j).begin(); k != (*j).end(); k++)
675                             {
676                                 if(!(i->isSameAs(*k,
                                    mGeometryModel->tolerance()))) //(*i != *k)
677                                 {
```

```
678                                     continue;
679                                 }
680
681                                 bool polyWanted = true;
682                                 for (const auto l: j->getLines())
683                                 {
684                                     if((l.isSameAs(intersecL[0],
                                        mGeometryModel->tolerance()))) // (l ==
                                        intersecL[0])
685                                     {
686                                         polyWanted = false;
687                                         continue;
688                                     }
689
690                                 }
691
692                                 if(polyWanted == true)
693                                 {
694                                     polyInCuboid.push_back(j);
695                                 }
696                             }
697                         }
698
699                         if (polyInCuboid.size() >= 2)
700                         {
701                             std::stringstream errorMessage;
702                             errorMessage << "\nError, found multiple
                            PolyInCuboid, therefore having to manny vertexen to
                            initiate a cuboid \n"
703                                                          <<
                                                         "(bso/spatial_design/conform
                                                         al/cf_cuboid)." <<
                                                         std::endl;
704                             throw std::runtime_error(errorMessage.str());
705                         }
706
707                         // find lineToSplit
708                         for (auto j = (*polyInCuboid[0]).begin(); j !=
                        (*polyInCuboid[0]).end(); j++)
709                         {
710                             std::vector<utilities::geometry::line_segment> Temp;
711                             for (auto k : mLineSegments)
712                             {
713                                 for (auto l = k.begin(); l != k.end(); l++)
714                                 {
715                                     if(!(l->isSameAs(*j,
                                    mGeometryModel->tolerance()))) // (*l != *j)
716                                     {
717                                         continue;
718                                     }
719
720                                     Temp.push_back(k);
721                                 }
722                             }
723
724                             for (auto k: Temp)
```

```
725                             {
726                                 bool partOfPoly = false;
727                                 for (auto j: polyInCuboid[0]->getLines())
728                                 {
729                                     if(j.isSameAs(k,
                                    mGeometryModel->tolerance())) //(j == k)
730                                     {
731                                         partOfPoly = true;
732                                     }
733                                 }
734
735                                 if (partOfPoly == false)
736                                 {
737                                     lineToSplit.push_back(k);
738                                 }
739                             }
740                             Temp.clear();
741                         }
742
743
744
745                         // Add all the vertexes of the cuboid to variable
                        cornerPoints
746
747                         // perpendicular to opposite side method
748                         if (method == 'O')
749                         {
750                             // add the already known in new cuboids vertexes
751                             for (auto j = (*polyInCuboid[0]).begin(); j !=
                            (*polyInCuboid[0]).end(); j++)
752                             {//polygon of the new cuboid which goes unchanged
                            into the new cuboid
753                                 cornerPoints.push_back(*j);
754                             }
755
756                             // determine if there is a guidance line
757                             utilities::geometry::vertex guidanceV;
758                             utilities::geometry::line_segment guidanceL;
759                             bool guidanceVertex = false;
760                             for (auto j : lineToSplit)
761                             {//the new vertixen in the middle of the to split
                            cuboids
762                                 if (j == intersecL[0])
763                                 {
764                                     continue;
765                                 }
766                                 // check if line already has an intersection
                                point
767                                 for(int k=1; k < onLineV.size(); k++)
768                                 {
769                                     if(j.isOnLine(onLineV[k],
                                    mGeometryModel->tolerance()))
770                                     {
771                                         guidanceVertex = true;
772                                         cornerPoints.push_back(onLineV[k]);
773                                         guidanceV = onLineV[k];
```

```
774                     guidanceL = j;
775                     continue;
776                 }
777             }
778         }
779
780         // Not finesched yet
781         // find intersection point on rest of LineToSplit
782         for (auto j : lineToSplit)
783         {
784             if (j == intersecL[0])
785             {
786                 continue;
787             }
788
789             if (guidanceL == j)
790             {
791                 continue;
792             }
793
794
795             //define intersection variables
796             utilities::geometry::vertex pInt;
797             double tol = mGeometryModel->tolerance();
798             utilities::geometry::line_segment l1 = j;
799             utilities::geometry::vector mNormal;
800             // assign mNormal
801             if (guidanceVertex == true)
802             {
803                 utilities::geometry::polygon*
                    intersectionPolygon;
804                 utilities::geometry::polygon* opositePolygon;
805
806                 // Find intersectionPolygon with guideline
                    on it
807                 for(int c = 0; c < mPolygons.size(); c++)
808                 {
809                     bool isBothOnPoly1 = false;
810
811                     if
                        (mPolygons[c]->isCoplanarN((onLineV[0]),
                        mGeometryModel->tolerance()))
812                     {
813                         if
                            (mPolygons[c]->isCoplanarN(guidanceV,
                             mGeometryModel->tolerance()))
814                         {
815                             isBothOnPoly1 = true;
816                         }
817                     }
818                     if (isBothOnPoly1 == false){continue;}
819
820                     intersectionPolygon = mPolygons[c];
821                     break;
822                 }
823
```

```
824                 // Find oppositePolygon to
                    intersectionPolygon
825                 std::vector <bool> sameCornpoint;
826                 unsigned int opposite = 6;
827
828                 for (unsigned int j = 0; j <
                    mPolygons.size(); j++)
829                 {
830                     bool sameCornp;
831
832                     for (auto m = (*mPolygons[j]).begin();
                        m != (*mPolygons[j]).end(); m++)
833                     {
834                         for (auto l =
                            (*intersectionPolygon).begin(); l
                            != (*intersectionPolygon).end(); l++)
835                         {
836                             if (*m == *l)
837                             {
838
                                    sameCornpoint.push_back(true)
                                    ;
839                             }
840                             else
841                             {
842
                                    sameCornpoint.push_back(false
                                    );
843                             }
844                         }
845                     }
846
847                     bool isOposite = true;
848                     for (int a=0; a < sameCornpoint.size();
                        a++)
849                     {
850                         if (sameCornpoint[a] == true)
851                         {
852                             isOposite = false;
853                         }
854                     }
855
856                     if (isOposite == true)
857                     {
858                         opposite = j;
859                         opositePolygon = mPolygons[j];
860                     }
861                     sameCornpoint.clear();
862                 }
863
864                 if (opposite == 6)
865                 {
866                     std::stringstream errorMessage;
867                     errorMessage << "\nError, could not
                        find opposite surface when\n"
868                                                  << "splitting
```

```
                                    a cuboid from
                                    a point on a
                                    surface\n"
869                                 <<
                                    "(bso/spatial_de
                                    sign/conformal/c
                                    f_cuboid)." <<
                                    std::endl;
870                 throw
                    std::runtime_error(errorMessage.str());
871             }
872
873             // Use the normal of the oppositePolygon to
                define the normal of the cell plane for
                line intersections
874             utilities::geometry::line_segment
                guidanceLine = {{onLineV[0]},{guidanceV}};
875             utilities::geometry::vector guidanceLVector
                = guidanceLine.getVector();
876             utilities::geometry::vector
                intersectionPlaneV =
                opositePolygon->getNormal();
877             mNormal =
                (intersectionPlaneV.cross(guidanceLVector));

878
879
880             // compute rest of intersection points by
                plane intersection with the to split lines
881             if (l1.getVector().isPerpendicular(mNormal,
                tol))
882             {
883                 std::stringstream errorMessage;
884                 errorMessage << "\nError, could not
                    find the intersection point on the
                    opposite side \n"
885                          << "during the line split
                             process, since the line to
                             intersect is perpendicular to
                             the plane\n"
886                          <<
                             "(bso/spatial_design/conformal/c
                             f_cuboid)." << std::endl;
887                 throw
                    std::runtime_error(errorMessage.str());
888             }
889             else
890             {
891                 utilities::geometry::vector v1 =
                    l1[1]-l1[0];
892                 utilities::geometry::vector v2 =
                    onLineV[0]-l1[0];
893
894                 //check if the line intersects with the
                    plane
895                 double r =
```

```
                                                (mNormal.dot(v2))/(mNormal.dot(v1));
896                                             if (r > -tol && r < (1 + tol))
897                                             { // if it does, compute the point
898                                                 pInt = l1[0] + r * v1;
899                                                 cornerPoints.push_back(pInt);
900                                             }
901                                             else
902                                             {
903                                                 std::stringstream errorMessage;
904                                                 errorMessage << "\nError, could not
                                                 find the intersection point of the
                                                 \n"
905                                                          << "perpendicular line
                                                          with the oposide side
                                                          line, \n"
906                                                          << "since the vertex is
                                                          not within the oposite
                                                          line borders \n"
907                                                          << "during the line split
                                                          process perpendicular to
                                                          opposite plane\n"
908                                                          <<
                                                          "(bso/spatial_design/conform
                                                          al/cf_cuboid)." <<
                                                          std::endl;
909                                                 throw
                                                 std::runtime_error(errorMessage.str()
                                                 );
910                                             }
911                                         }
912                                     }
913                                     else
914                                     {
915                                         // get closest point to intersection point
916
                                         //cornerPoints.push_back(j.getPointClosestTo(
                                         cornerPoints[0]));
917
918
919
920                                         // find opositePolygon, there are 2 polygon
                                         opposites
921                                         utilities::geometry::polygon*
                                         opositePolygon1;
922                                         utilities::geometry::polygon*
                                         opositePolygon2;
923                                         int k = 0;
924                                         for(int c = 0; c < mPolygons.size(); c++)
925                                         {
926                                             bool isBothNotOnPoly1 = true;
927
928                                             if
                                             ((mPolygons[c]->isCoplanarN((intersecL[0]
                                             )[0], mGeometryModel->tolerance())))
929                                             {
930                                                 isBothNotOnPoly1 = false;
```

```
931                     }
932                     else if
                        ((mPolygons[c]->isCoplanarN((intersecL[0]
                        )[1], mGeometryModel->tolerance())))
933                     {
934                         isBothNotOnPoly1 = false;
935                     }
936                     if (isBothNotOnPoly1 == false)
937                     {
938                         continue;
939                     }
940
941
942                     if(k == 0)
943                     {
944                         opositePolygon1 = mPolygons[c];
945                         k++;
946                     }
947                     else if (k == 1)
948                     {
949                         opositePolygon2 = mPolygons[c];
950                         k++;
951                     }
952                     else
953                     {
954                         std::stringstream errorMessage;
955                         errorMessage << "\nError, found to
                            manny opposite polygons \n"
956                                   << "during the line split
                                      process of split to
                                      opposite \n"
957                                   <<
                                      "(bso/spatial_design/conform
                                      al/cf_cuboid)." <<
                                      std::endl;
958                         throw
                            std::runtime_error(errorMessage.str()
                            );
959                     }
960                 }
961
962                 // Use the normal of the two
                    oppositePolygon to define the normal of the
                    cell plane for line intersections
963                 utilities::geometry::vector
                    opositePolygon2N =
                    opositePolygon2->getNormal();
964                 utilities::geometry::vector
                    opositePolygon1N =
                    opositePolygon1->getNormal();
965                 mNormal =
                    (opositePolygon1N.cross(opositePolygon2N));

966
967
968                 // compute rest of intersection points by
```

```
                plane intersection with the to split lines
969             if (l1.getVector().isPerpendicular(mNormal,
                tol))
970             {
971                 std::stringstream errorMessage;
972                 errorMessage << "\nError, could not
                    find the intersection point on the
                    opposite side \n"
973                          << "during the line split
                             process, since the line to
                             intersect is perpendicular to
                             the plane\n"
974                          <<
                             "(bso/spatial_design/conformal/c
                             f_cuboid)." << std::endl;
975                 throw
                    std::runtime_error(errorMessage.str());
976             }
977             else
978             {
979                 utilities::geometry::vector v1 =
                    l1[1]-l1[0];
980                 utilities::geometry::vector v2 =
                    onLineV[0]-l1[0];
981
982                 //check if the line intersects with the
                    plane
983                 double r =
                    (mNormal.dot(v2))/(mNormal.dot(v1));
984                 //std::cout << "r is: " << r <<
                    std::endl;
985                 if (r > -tol && r < (1 + tol))
986                 { // if it does, compute the point
987                     pInt = l1[0] + r * v1;
988                     cornerPoints.push_back(pInt);
989                 }
990                 else
991                 {
992                     std::stringstream errorMessage;
993                     errorMessage << "\nError, could not
                        find the intersection point of the
                        \n"
994                              << "perpendicular line
                                 with the oposide side
                                 line, \n"
995                              << "since the vertex is
                                 not within the oposite
                                 line borders \n"
996                              << "during the line split
                                 process perpendicular to
                                 opposite plane\n"
997                              <<
                                 "(bso/spatial_design/conform
                                 al/cf_cuboid)." <<
                                 std::endl;
998                     throw
```

```
                                    std::runtime_error(errorMessage.str()
                                    );
 999                            }
1000                        }
1001                    }
1002                }
1003            } // if (method == 'O')
1004
1005
1006            // perpendicular to intersection side method
1007            if (method == 'I')
1008            {
1009                // add the already known in cornerPoints
1010                for (auto j = (*polyInCuboid[0]).begin(); j !=
                    (*polyInCuboid[0]).end(); j++)
1011                {//polygon of the new cuboid which goes unchanged
                    into the new cuboid
1012                    cornerPoints.push_back(*j);
1013                }
1014
1015                // determine if there is a guidance line
1016                utilities::geometry::vertex guidanceV;
1017                utilities::geometry::line_segment guidanceL;
1018                bool guidanceVertex = false;
1019                for (auto j : lineToSplit)
1020                {//the new vertixen in the middle of the to split
                    cuboids
1021                    if (j == intersecL[0])
1022                    {
1023                        continue;
1024                    }
1025                    // check if line already has an intersection
                        point
1026                    for(int k=1; k < onLineV.size(); k++)
1027                    {
1028                        if(j.isOnLine(onLineV[k]))
1029                        {
1030                            guidanceVertex = true;
1031                            cornerPoints.push_back(onLineV[k]);
1032                            guidanceV = onLineV[k];
1033                            guidanceL = j;
1034                            continue;
1035                        }
1036                    }
1037                }
1038
1039                // find intersection point on rest of LineToSplit
1040                for (auto j : lineToSplit)
1041                {
1042                    if (j == intersecL[0])
1043                    {
1044                        continue;
1045                    }
1046
1047                    if (guidanceL == j)
1048                    {
```

```
1049                continue;
1050            }
1051
1052
1053            //define intersection variables
1054            utilities::geometry::vertex pInt;
1055            double tol = mGeometryModel->tolerance();
1056            utilities::geometry::line_segment l1 = j;
1057            utilities::geometry::vector mNormal;
1058            // assign mNormal
1059            if (guidanceVertex == true)
1060            {
1061                utilities::geometry::line_segment
                    guidanceLine = {{onLineV[0]},{guidanceV}};
1062                utilities::geometry::vector intersecLVector
                    = (intersecL[0]).getVector();
1063                utilities::geometry::vector guidanceLVector
                    = guidanceLine.getVector();
1064
1065                utilities::geometry::vector
                    intersectionPlaneV =
                    (intersecLVector.cross(guidanceLVector)).norm
                    alized();
1066                mNormal =
                    (intersectionPlaneV.cross(guidanceLVector));

1067            }
1068            else {mNormal = (intersecL[0]).getVector();}
1069
1070            // compute rest of intersection points
1071
1072
1073            if (l1.getVector().isPerpendicular(mNormal, tol))
1074            {
1075                std::stringstream errorMessage;
1076                errorMessage << "\nError, could not find
                    the intersection point on the opposite side
                    \n"
1077                        << "during the line split process,
                        since the line to intersect is
                        perpendicular to the plane\n"
1078                        <<
                        "(bso/spatial_design/conformal/cf_cu
                        boid)." << std::endl;
1079                throw std::runtime_error(errorMessage.str());
1080            }
1081            else
1082            {
1083                utilities::geometry::vector v1 = l1[1]-l1[0];
1084                utilities::geometry::vector v2 =
                    onLineV[0]-l1[0];
1085
1086                //check if the line intersects with the plane
1087                double r =
                    (mNormal.dot(v2))/(mNormal.dot(v1));
1088                if (r > -tol && r < (1 + tol))
```

```
1089                            { // if it does, compute the point
1090                                pInt = l1[0] + r * v1;
1091                                cornerPoints.push_back(pInt);
1092                            }
1093                            else
1094                            {
1095                                std::stringstream errorMessage;
1096                                errorMessage << "\nError, could not
                                find the intersection point of the \n"
1097                                         << "perpendicular line with
                                         the oposide side line, \n"
1098                                         << "since the vertex is not
                                         within the oposite line
                                         borders \n"
1099                                         << "during the line split
                                         process perpendicular to
                                         intersection plane\n"
1100                                         <<
                                         "(bso/spatial_design/conformal/c
                                         f_cuboid)." << std::endl;
1101                                throw
                                std::runtime_error(errorMessage.str());
1102                            }
1103                        }
1104                    }
1105                } // if (method == 'I')
1106
1107
1108                // not fully finished yet
1109                // Ratio method
1110                if (method == 'M')
1111                {
1112
1113                } // if (method == 'M')
1114
1115
1116                // remove duplicates to make sure that the safety
                checks work correctly
1117                auto cornerPoints_end = cornerPoints.end();
1118                for(auto it = cornerPoints.begin(); it !=
                cornerPoints_end; ++it)
1119                {
1120                    cornerPoints_end = std::remove(it + 1,
                    cornerPoints_end, *it);
1121                }
1122                cornerPoints.erase(cornerPoints_end, cornerPoints.end());
1123
1124                // safety checks
1125                if (cornerPoints.size() < 8)
1126                {
1127                    std::stringstream errorMessage;
1128                    errorMessage << "\nError, There are not enough
                    value's in cornerpoints to\n"
1129                             << "make a cuboids to split space during
                             the split line precedure\n"
1130                             <<
```

```
                                    "(bso/spatial_design/conformal/cf_cuboid)."
                                    << std::endl;
1131                            throw std::runtime_error(errorMessage.str());
1132                        }
1133                        if (cornerPoints.size() > 8)
1134                        {
1135                            std::stringstream errorMessage;
1136                            errorMessage << "\nError, There are too manny
                            value's in cornerpoints to\n"
1137                                    << "make a cuboids to split space during
                                    the split line precedure\n"
1138                                    <<
                                    "(bso/spatial_design/conformal/cf_cuboid)."
                                    << std::endl;
1139                            throw std::runtime_error(errorMessage.str());
1140                        }
1141
1142                        // The new generator for the cuboids should be placed
                        here
1143                        newCuboids.push_back(mGeometryModel->addCuboid(
1144
                                utilities::geometry::quad_hexahedron(cornerPoints
                                , mGeometryModel->tolerance())));
1145                    } // cornerpoint line_segment
1146                }
1147            } // split line_segment
1148
1149
1150
1151            if (split)
1152            {
1153                mDeletion = true;
1154
1155                for (auto& i : mCFSpaces)
1156                {
1157                    i->removeCuboid(this);
1158                    for (auto& j : newCuboids)
1159                    {
1160                        i->addCuboid(j);
1161                        j->addSpace(i);
1162                    }
1163                    checkAssociatedN(newCuboids);
1164                }
1165
1166                mCFSpaces.clear();
1167            }
1168        } // splitN
1169
1170
1171
1172
1173
1174        void cf_cuboid::split(cf_vertex* pPtr)
1175        { //
1176            for (auto& i : mCFVertices)
1177            {
```

```
1178                if (pPtr == i) return;
1179            }
1180
1181            std::vector<cf_vertex*> newVertices;
1182            std::vector<cf_cuboid*> newCuboids;
1183            bool split = false;
1184
1185            std::vector<utilities::geometry::vector> normals; // the three
                normals of the cuboid
1186
1187            for (auto& i : mPolygons)
1188            {
1189                bool normalFound = false;
1190                for (const auto& j : normals)
1191                {
1192                    if (i->normal().isParallel(j, mGeometryModel->tolerance()))
                        normalFound = true;
1193                }
1194                if (!normalFound) normals.push_back(i->normal().normalized());
1195                if (normals.size() == 3) break;
1196            }
1197            if (normals.size() != 3)
1198            {
1199                std::stringstream errorMessage;
1200                errorMessage << "\nError, expected to find 3 normals of a
                cuboid\n"
1201                                         << "found: " << normals.size() << ".\n"
1202                                         <<
                                             "(bso/spatial_design/conformal/cf_cuboid
                                             .cpp)" << std::endl;
1203                throw std::runtime_error(errorMessage.str());
1204            }
1205
1206            if (this->isInside(*pPtr, mGeometryModel->tolerance()))
1207            {
1208                std::cout << "The cuboid split section has been used" <<
                std::endl;
1209
1210                split = true;
1211                newVertices.push_back(pPtr);
1212
1213                for (const auto& i : mPolygons)
1214                { // find the point closest on each surface
1215
                    newVertices.push_back(mGeometryModel->addVertex(i->getPointCl
                    osestTo(*pPtr)));
1216                    if (!(i->isInside(*(newVertices.back()),
                    mGeometryModel->tolerance())))
1217                    {
1218                        std::stringstream errorMessage;
1219                        errorMessage << "\nError, found a point closest to a
                        rectangle,\n"
1220                                                 << "but that point is not
                                                 inside the rectangle.\n"
1221                                                 << "While splitting a cuboid
                                                 at a point inside that
```

```
                                                    cuboid.\n"
1222                                                << "Rectangle: " << *i << "\n"
1223                                                << "Split point: " <<
                                                    pPtr->transpose() << "\n"
1224                                                << "Found point: " <<
                                                    newVertices.back()->transpose()
                                                    << "\n"
1225                                                <<
                                                    "(bso/spatial_design/conformal/c
                                                    f_cuboid.cpp)" << std::endl;
1226                        throw std::runtime_error(errorMessage.str());
1227                    }
1228                }
1229
1230                for (const auto& i : mLineSegments)
1231                { // ifnd the point closest on each line
1232
                    newVertices.push_back(mGeometryModel->addVertex(i.getPointClo
                    sestTo(*pPtr)));
1233                    if (!(i.isOnLine(*(newVertices.back()),
                    mGeometryModel->tolerance())))
1234                    {
1235                        std::stringstream errorMessage;
1236                        errorMessage << "\nError, found a point closest to a
                        line segment,\n"
1237                                                << "but that point is not on
                                                    the line segment.\n"
1238                                                << "While splitting a cuboid
                                                    at a point inside that
                                                    cuboid.\n"
1239                                                << "Line: " << i  << "\n"
1240                                                << "Split point: " <<
                                                    pPtr->transpose() << "\n"
1241                                                << "Found point: " <<
                                                    newVertices.back()->transpose()
                                                    << "\n"
1242                                                <<
                                                    "(bso/spatial_design/conformal/c
                                                    f_cuboid.cpp)" << std::endl;
1243                        throw std::runtime_error(errorMessage.str());
1244                    }
1245                }
1246
1247                for (const auto& i: mVertices)
1248                {
1249                    std::vector<utilities::geometry::vertex> cornerPoints;
1250                    cornerPoints.push_back(*pPtr);
1251                    cornerPoints.push_back(i);
1252
1253                    for (const auto& j : normals)
1254                    {
1255                        double projectedDistance = j.dot(cornerPoints[1] -
                        cornerPoints[0]);
1256                        cornerPoints.push_back(cornerPoints[0] +
                        projectedDistance * j);
1257                        cornerPoints.push_back(cornerPoints[1] -
```

```
                        projectedDistance * j);
1258                }
1259
1260                newCuboids.push_back(mGeometryModel->addCuboid(
1261                    utilities::geometry::quad_hexahedron(cornerPoints,
                        mGeometryModel->tolerance())));
1262            }
1263        }
1264
1265        if (!split)
1266        {
1267            for (unsigned int i = 0; i < 6; ++i)
1268            { // check if the point is on any of the rectangles of this
            cuboid
1269
1270                if (mPolygons[i]->isInside(*pPtr,
                mGeometryModel->tolerance()))
1271                {
1272                 // if it is, lets split the cuboid into four new ones
1273
1274                    split = true;
1275                    newVertices.push_back(pPtr);
1276                    unsigned int opposite = 6;
1277                    // first find the rectangle opposite to rectangle i
1278                    for (unsigned int j = 0; j < 6; ++j)
1279                    {
1280                        if (j != i &&
                        mPolygons[i]->normal().isParallel(mPolygons[j]->norma
                        l(), mGeometryModel->tolerance()))
1281                        {
1282                            opposite = j;
1283                            break;
1284                        }
1285                    }
1286                    if (opposite == 6)
1287                    {
1288                        std::stringstream errorMessage;
1289                        errorMessage << "\nError, could not find opposite
                        surface when\n"
1290                                                     << "splitting a cuboid
                                                     from a point on a surface\n"
1291                                                     <<
                                                     "(bso/spatial_design/conform
                                                     al/cf_cuboid)." <<
                                                     std::endl;
1292                        throw std::runtime_error(errorMessage.str());
1293                    }
1294
1295
                    newVertices.push_back(mGeometryModel->addVertex(mPolygons
                    [opposite]->getPointClosestTo(*pPtr)));
1296
1297                    if
                    (!(mPolygons[opposite]->isInside(*(newVertices.back()))))
1298                    {
1299                        std::stringstream errorMessage;
```

```
1300                            errorMessage << "\nError, found a point closest to
                                a surface,\n"
1301                                                         << "but that point is not
                                                         inside the surface.\n"
1302                                                         << "When splitting cuboid
                                                         at surface.\n"
1303                                                         << "Surface:\n" <<
                                                         *mPolygons[opposite] << "\n"
1304                                                         << "Split point: " <<
                                                         pPtr->transpose() << "\n"
1305                                                         << "Found point: " <<
                                                         (newVertices.back())->transp
                                                         ose() << "\n"
1306                                                         <<
                                                         "(bso/spatial_design/conform
                                                         al/cf_cuboid)." <<
                                                         std::endl;
1307                            throw std::runtime_error(errorMessage.str());
1308                        }
1309
1310                        unsigned int splitSurfaces[2] = {i,opposite};
1311
1312                        // next find the vertices on each of the lines of the
                            surface
1313                        for (unsigned int j = 0; j < 2; ++j)
1314                        {
1315                            for (const auto& k :
                                mPolygons[splitSurfaces[j]]->getLines())
1316                            {
1317
                                    newVertices.push_back(mGeometryModel->addVertex(k
                                    .getPointClosestTo(*(newVertices[j]))));
1318
1319                                if (!k.isOnLine(*(newVertices.back()),
                                    mGeometryModel->tolerance()))
1320                                {
1321                                    std::stringstream errorMessage;
1322                                    errorMessage << "\nError, found a point
                                        closest to a line segment,\n"
1323                                                                 << "but that point
                                                                 is not on the line
                                                                 segment.\n"
1324                                                                 << "When splitting
                                                                 cuboid at
                                                                 surface.\n"
1325                                                                 << "Surface:\n" <<
                                                                 *mPolygons[splitSurf
                                                                 aces[j]] << "\n"
1326                                                                 << "Line:\n" << k
                                                                 << "\n"
1327                                                                 << "Split point: "
                                                                 <<
                                                                 pPtr->transpose()
                                                                 << "\n"
1328                                                                 << "Found point: "
                                                                 <<
```

```
                                                                        newVertices.back()->
                                                                        transpose() << "\n"
1329                                                                    <<
                                                                        "(bso/spatial_design
                                                                        /conformal/cf_cuboid
                                                                        )." << std::endl;
1330                                    throw std::runtime_error(errorMessage.str());
1331                                }
1332                            }
1333                        }
1334
1335                        for (const auto j : *(mPolygons[opposite]))
1336                        {
1337                            std::vector<utilities::geometry::vertex>
                                cornerPoints;
1338                            cornerPoints.push_back(*pPtr);
1339                            cornerPoints.push_back(j);
1340                            for (const auto& k : normals)
1341                            {
1342                                double projectedDistance =
                                    k.dot(cornerPoints[1] - cornerPoints[0]);
1343                                cornerPoints.push_back(cornerPoints[0] +
                                    projectedDistance * k);
1344                                cornerPoints.push_back(cornerPoints[1] -
                                    projectedDistance * k);
1345                            }
1346                            newCuboids.push_back(mGeometryModel->addCuboid(
1347
                                    utilities::geometry::quad_hexahedron(cornerPoints
                                    , mGeometryModel->tolerance())));
1348                        }
1349                        break;
1350                    }
1351                }
1352            }
1353
1354            if (!split)
1355            {
1356                for (unsigned int i = 0; i < 12; ++i)
1357                {
1358                    if
                        (mLineSegments[i].isOnLine(*pPtr,mGeometryModel->tolerance())
                        )
1359                    {
1360                        split = true;
1361                        newVertices.push_back(pPtr);
1362                        utilities::geometry::vector v1 =
                            mLineSegments[i].getVector();
1363
1364                        for (unsigned int j = 0; j < 12; ++j)
1365                        {
1366                            if (j == i) continue;
1367
1368                            if (mLineSegments[j].getVector().isParallel(v1,
                                mGeometryModel->tolerance()))
1369                            {
```

```
1370                                    newVertices.push_back(mGeometryModel->addVertex(m
                                        LineSegments[j].getPointClosestTo(*pPtr)));
1371                                    if
                                        (!mLineSegments[j].isOnLine(*(newVertices.back())
                                        , mGeometryModel->tolerance()))
1372                                    {
1373                                        std::stringstream errorMessage;
1374                                        errorMessage << "\nError, found a point
                                            closest to a line segment,\n"
1375                                                         << "but that point
                                                         is not on the line
                                                         segment.\n"
1376                                                         << "When splitting
                                                         cuboid at a line
                                                         segment.\n"
1377                                                         << "Line segment:
                                                         " <<
                                                         mLineSegments[j]
                                                         << "\n"
1378                                                         << "Split point: "
                                                         <<
                                                         pPtr->transpose()
                                                         << "\n"
1379                                                         << "Found point: "
                                                         <<
                                                         newVertices.back()->
                                                         transpose() << "\n"
1380                                                         <<
                                                         "(bso/spatial_design
                                                         /conformal/cf_cuboid
                                                         )." << std::endl;
1381                                        throw std::runtime_error(errorMessage.str());
1382                                    }
1383                                }
1384                            }
1385
1386                            if (newVertices.size() != 4)
1387                            {
1388                                std::stringstream errorMessage;
1389                                errorMessage << "\nError, expected to find 4 new
                                vertices,\n"
1390                                                         << "when dividing a cuboid
                                                         at a line segment.\n"
1391                                                         << "Found: " <<
                                                         newVertices.size() << "\n"
1392                                                         <<
                                                         "(bso/spatial_design/conform
                                                         al/cf_cuboid)." <<
                                                         std::endl;
1393                                throw std::runtime_error(errorMessage.str());
1394                            }
1395
1396                            for (const auto& j : mPolygons)
1397                            {
1398                                if (v1.isParallel(j->normal(),
```

```
                                mGeometryModel->tolerance()))
1399                            {
1400                                std::vector<utilities::geometry::vertex>
                                cornerPoints;
1401                                for (unsigned int k = 0; k < 4; ++k)
1402                                {
1403                                    cornerPoints.push_back(*newVertices[k]);
1404                                }
1405
                                cornerPoints.insert(cornerPoints.end(),j->begin()
                                , j->end());
1406
1407                                newCuboids.push_back(mGeometryModel->addCuboid(
1408
                                    utilities::geometry::quad_hexahedron(cornerPo
                                    ints, mGeometryModel->tolerance())));
1409                            }
1410                        }
1411                        if (newCuboids.size() != 2)
1412                        {
1413                            std::stringstream errorMessage;
1414                            errorMessage << "\nError, expected to find 2 new
                            cuboids,\n"
1415                                                         << "when dividing a cuboid
                                                         at a line segment.\n"
1416                                                         << "Found: " <<
                                                         newCuboids.size() << "\n"
1417                                                         <<
                                                         "(bso/spatial_design/conform
                                                         al/cf_cuboid)." <<
                                                         std::endl;
1418                            throw std::runtime_error(errorMessage.str());
1419                        }
1420                        break;
1421                    }
1422                }
1423            }
1424
1425            if (split)
1426            {
1427                mDeletion = true;
1428                for (auto& i : mCFSpaces)
1429                {
1430                    i->removeCuboid(this);
1431                    for (auto& j : newCuboids)
1432                    {
1433                        i->addCuboid(j);
1434                        j->addSpace(i);
1435                    }
1436                }
1437
1438
1439                for (const auto& i : newVertices) this->checkAssociated(i);
1440                mCFSpaces.clear();
1441            }
1442        } // split
```

```
1443
1444
1445
1446
1447
1448
1449        void cf_cuboid::checkAssociated(cf_vertex* pPtr)
1450        { //
1451            for (auto& i : mCFSpaces)
1452            {
1453                i->checkVertex(pPtr);
1454                for (auto& j : i->cfEdges())
1455                {
1456                    j->checkVertex(pPtr);
1457                }
1458                for (auto& j : i->cfSurfaces())
1459                {
1460                    j->checkVertex(pPtr);
1461                }
1462            }
1463        } // checkAssociated
1464
1465
1466        void cf_cuboid::checkAssociatedN(std::vector<cf_cuboid*> newCuboids)
1467        {
1468            // insert the found polygons in the regtangular split function to
            split the origonal polygons
1469            for (const auto& j : cfRectangles()) // origonal polygons
1470            {
1471                std::vector<cf_rectangle*> newRect;
1472
1473
1474                for (auto i : newCuboids)
1475                {
1476                    std::vector <bool> origonalPolycheck;
1477
1478                    for (auto m : (i->cfRectangles())) // new polygons of new
                    cuboids
1479                    {
1480                        for (auto l = m->begin(); l != m->end(); l++) //
                        vertixes of new polygons
1481                        {
1482
1483                            if((j->isInsideOrOn(*l,
                            mGeometryModel->tolerance()))) // isInsideOrOn does
                            not reconize a common vertex {0,0,0}
1484                            {
1485                                origonalPolycheck.push_back(true);
1486                            }
1487
1488                            else
1489                            {
1490                                for (auto n = j->begin(); n != j->end(); n++)
1491                                {
1492                                    if(l->isSameAs(*n,
                                    mGeometryModel->tolerance()))
```

```
1493                                {
1494                                    origonalPolycheck.push_back(true);
1495                                }
1496                            }
1497
1498                        }
1499                    }
1500
1501                    if(origonalPolycheck.size() == 4) // all vertixes of
                    new polygons are on the orgigonal polygon
1502                    {
1503                        newRect.push_back(m);
1504                    }
1505                    else if (origonalPolycheck.size() > 4)
1506                    {
1507                        std::stringstream errorMessage;
1508                        errorMessage << "\nError, something went wrong when
                        searching for the new polygons within the old one,\n"
1509                                                          <<
                                                          "(bso/spatial_design/conform
                                                          al/cf_cuboid)." <<
                                                          std::endl;
1510                        throw std::runtime_error(errorMessage.str());
1511                    }
1512                    origonalPolycheck.clear();
1513                }
1514            }
1515
1516            // Insert new polygons to rectangular split function
1517            if(newRect.size() > 1)
1518            {
1519                j -> splitN(newRect);
1520            }
1521        }
1522
1523
1524
1525        // insert the found line segments in the line split function to
        split the origonal lines
1526        for (const auto& j : cfLines()) // origonal line segments
1527        {
1528            std::vector<cf_line*> newLine;
1529
1530            for (auto i : newCuboids)
1531            {
1532                std::vector <bool> origonalLinecheck;
1533
1534                for (auto m : (i->cfLines())) // lines of the new cuboids
1535                {
1536                    for (auto n = m->begin(); n != m->end(); n++)
1537                    {
1538                        if(j->isOnLine(*n, mGeometryModel->tolerance()))
1539                        {
1540                            origonalLinecheck.push_back(true);
1541                        }
1542                        else
```

```
1543                        {
1544                            for (auto k = j->begin(); k != j->end(); k++)
1545                            {
1546                                if(k->isSameAs(*n,
                                mGeometryModel->tolerance()))
1547                                {
1548                                    origonalLinecheck.push_back(true);
1549                                }
1550                            }
1551 
1552                        }
1553                    }
1554 
1555                    if(origonalLinecheck.size() == 2) // all vertixes of
                    new polygons are on the orgigonal polygon
1556                    {
1557                        newLine.push_back(m);
1558                    }
1559                    else if (origonalLinecheck.size() > 2)
1560                    {
1561                        std::stringstream errorMessage;
1562                        errorMessage << "\nError, something went wrong when
                        searching for the new line within the old one,\n"
1563                                                     <<
                                                     "(bso/spatial_design/confor
                                                     mal/cf_cuboid)." <<
                                                     std::endl;
1564                        throw std::runtime_error(errorMessage.str());
1565                    }
1566                    origonalLinecheck.clear();
1567                }
1568            }
1569 
1570 
1571            // Insert new lines to line split function
1572            if(newLine.size() > 1)
1573            {
1574                j -> splitN(newLine);
1575            }
1576        }
1577 
1578 
1579    }//checkAssociatedN
1580 
1581 } // conformal
1582 } // spatial_design
1583 } // bso
1584 
1585 #endif // CF_CUBOID_CPP
```

A.8 cf_tetrahedron

```
1   #ifndef CF_TETRAHEDRON_HPP
2   #define CF_TETRAHEDRON_HPP
3
4   namespace bso { namespace spatial_design { namespace conformal {
5
6       class cf_tetrahedron : public bso::utilities::geometry::tetrahedron,
7                                         public cf_geometry_entity
8       {
9       private:
10
11      public:
12          cf_tetrahedron(const utilities::geometry::tetrahedron& rhs,
            cf_geometry_model* geometryModel);
13          ~cf_tetrahedron();
14
15          void split(cf_vertex* pPtr);
16          void checkAssociatedT(std::vector<cf_tetrahedron*> newTetrahedron);
17
18          void addLine              (cf_line*         lPtr)      =
            delete ;
19          void addRectangle         (cf_rectangle*    recPtr)    =
            delete ;
20          void addTriangle          (cf_triangle*     triPtr)    =
            delete;
21          void addCuboid            (cf_cuboid*       cubPtr)    =
            delete ;
22          void addTetrahedron       (cf_tetrahedron*  tetPtr)    =
            delete ;
23          void removeLine           (cf_line*         lPtr)      =
            delete ;
24          void removeRectangle      (cf_rectangle*    recPtr)    =
            delete ;
25          void removeTriangle       (cf_triangle*     triPtr)    =
            delete;
26          void removeCuboid         (cf_cuboid*       cubPtr)    =
            delete ;
27          void removeTetrahedron    (cf_tetrahedron*  tetPtr)    =
            delete ;
28          void addPoint             (cf_point*        pPtr)      =
            delete ;
29          void addEdge              (cf_edge*         ePtr)      =
            delete ;
30          void addSurface           (cf_surface*      srfPtr)    =
            delete ;
31
32          const std::vector<cf_rectangle*     >& cfRectangles()   = delete;
33          const std::vector<cf_tetrahedron*   >& cfTetrahedrons() = delete;
34          const std::vector<cf_cuboid*        >& cfCuboids()      = delete;
35          const std::vector<cf_point*         >& cfPoints()       = delete;
36          const std::vector<cf_edge*          >& cfEdges()        = delete;
37          const std::vector<cf_surface*       >& cfSurfaces()     = delete;
38      };
39
40  } // conformal
41  } // spatial_design
42  } // bso
```

```
43
44  #endif // CF_TETRAHEDRON_HPP
```

```
1    #ifndef CF_TETRAHEDRON_CPP
2    #define CF_TETRAHEDRON_CPP
3
4    namespace bso { namespace spatial_design { namespace conformal {
5
6        cf_tetrahedron::cf_tetrahedron(const utilities::geometry::tetrahedron&
         rhs, cf_geometry_model* geometryModel)
7        : utilities::geometry::tetrahedron(rhs, geometryModel->tolerance())
8        {
9            mGeometryModel = geometryModel;
10           for (const auto& i : mVertices)
11           {
12               mCFVertices.push_back(mGeometryModel->addVertex(i));
13               mCFVertices.back()->addTetrahedron(this);
14           }
15           for (const auto& i : mLineSegments)
16           {
17               mCFLines.push_back(mGeometryModel->addLine(i));
18               mCFLines.back()->addTetrahedron(this);
19           }
20           for (const auto& i : mPolygons)
21           {
22               mCFTriangles.push_back(mGeometryModel->addTriangle(*i));
23               mCFTriangles.back()->addTetrahedron(this);
24           }
25       } // ctor
26
27       cf_tetrahedron::~cf_tetrahedron()
28       {
29           for (const auto& i : mCFVertices) i->removeTetrahedron(this);
30           for (const auto& i : mCFLines) i->removeTetrahedron(this);
31           for (const auto& i : mCFTriangles) i->removeTetrahedron(this);
32       } // dtor
33
34
35
36       void cf_tetrahedron::split(cf_vertex* pPtr)
37       { //
38           for (auto& i : mCFVertices)
39           {
40               if (pPtr == i) return;
41           }
42
43           std::vector<cf_vertex*> newVertices;
44           std::vector<cf_tetrahedron*> newTetrahedron;
45           bool split = false;
46
47           /*
48           if (this->isInside(*pPtr, mGeometryModel->tolerance()))
49           {
50
51           }
52           */
53
54           if (!split)
55           {
```

```
56              for (unsigned int i = 0; i < 4; ++i)
57              { // check if the point is on any of the rectangles of this
                cuboid
58                  if (mPolygons[i]->isInside(*pPtr,
                    mGeometryModel->tolerance()))
59                  {
60                   // if it is, lets split the tetrahedron into three new ones
61
62                      split = true;
63                      newVertices.push_back(pPtr);
64                      unsigned int opposite = 4;
65
66                      // first find the vertex opposite to triangle i
67                      for (unsigned int j = 0; j < 4; ++j)
68                      {
69                          if
                            ((std::find((mPolygons[i]->getVertices()).begin(),(mP
                            olygons[i]->getVertices()).end(), mVertices[j]) ==
                            (mPolygons[i]->getVertices()).end()))
70                          {
71                              opposite = j;
72                              break;
73                          }
74                      }
75                      if (opposite == 4)
76                      {
77                          std::stringstream errorMessage;
78                          errorMessage << "\nError, could not find opposite
                            surface when\n"
79                                                   << "splitting a cuboid
                                                     from a point on a surface\n"
80                                                   <<
                                                     "(bso/spatial_design/conform
                                                     al/cf_cuboid)." <<
                                                     std::endl;
81                          throw std::runtime_error(errorMessage.str());
82                      }
83
84                      // generate new tetrahedrons
85                      for (const auto k: mPolygons[i]->getLines())
86                      {
87                          std::vector<utilities::geometry::vertex>
                            cornerPoints;
88                          cornerPoints.push_back(*pPtr);
89                          cornerPoints.push_back(mVertices[opposite]);
90                          cornerPoints.push_back(k[0]);
91                          cornerPoints.push_back(k[1]);
92
93
                            newVertices.push_back(mGeometryModel->addVertex(k[0])
                            );
94
                            newVertices.push_back(mGeometryModel->addVertex(k[1])
                            );
95
96                          if (cornerPoints.size() != 4)
```

```
97                              {
98                                  std::stringstream errorMessage;
99                                  errorMessage << "\nError, expected to find 4
                                    new vertices,\n"
100                                                              << "when dividing a
                                                                 cuboid at a triangle
                                                                 intersection.\n"
101                                                              << "Found: " <<
                                                                 newVertices.size() <<
                                                                 "\n"
102                                                              <<
                                                                 "(bso/spatial_design/con
                                                                 formal/cf_tetrahedron)."
                                                                  << std::endl;
103                                 throw std::runtime_error(errorMessage.str());
104                             }
105
106
                                newTetrahedron.push_back(mGeometryModel->addTetrahedr
                                on(
107                                 utilities::geometry::tetrahedron(cornerPoints,
                                    mGeometryModel->tolerance())));
108                         }
109
110                         if (newTetrahedron.size() != 3)
111                         {
112                             std::stringstream errorMessage;
113                             errorMessage << "\nError, expected to find 2 new
                                tetrahedrons,\n"
114                                                         << "when dividing a
                                                            tetrahedron at a line
                                                            segment.\n"
115                                                         << "Found: " <<
                                                            newTetrahedron.size() <<
                                                            "\n"
116                                                         <<
                                                            "(bso/spatial_design/conform
                                                            al/cf_tetrahedron)." <<
                                                            std::endl;
117                             throw std::runtime_error(errorMessage.str());
118                         }
119                         break;
120                     }
121                 }
122             }
123
124
125             if (!split)
126             {
127                 for (unsigned int i = 0; i < 6; ++i)
128                 {
129                     if
                        (mLineSegments[i].isOnLine(*pPtr,mGeometryModel->tolerance())
                        )
130                     {
131                         split = true;
```

```
132
133                 newVertices.push_back(pPtr);
134                 std::vector<int> opposite;
135
136                 // find the 2 vertexes not conected to the intersected
                    line
137                 for (unsigned int j = 0; j < 4; ++j)
138                 {
139                     if (mLineSegments[i][0] != mVertices[j])
140                     {
141                         if (mLineSegments[i][1] != mVertices[j])
142                         {
143                             opposite.push_back(j);
144                         }
145                     }
146                 }
147
148                 if (opposite.size() != 2)
149                 {
150                     std::stringstream errorMessage;
151                     errorMessage << "\nError, could not find 2 opposite
                        vertexes to line_segment \n"
152                                  << " when splitting a
                                     tetrahedron from a point
                                     on a triangle\n"
153                                  <<
                                     "(bso/spatial_design/conform
                                     al/cf_tetrahedron)." <<
                                     std::endl;
154                     throw std::runtime_error(errorMessage.str());
155                 }
156
157                 // generate new tetrahedrons
158                 for (unsigned int j = 0; j < 2; ++j)
159                 {
160                     std::vector<utilities::geometry::vertex>
                        cornerPoints;
161                     cornerPoints.push_back(*pPtr);
162                     cornerPoints.push_back((mLineSegments[i])[j]);
163
                        newVertices.push_back(mGeometryModel->addVertex(((mLi
                        neSegments[i])[j])));
164                     for(const int k: opposite)
165                     {
166                         cornerPoints.push_back(mVertices[k]);
167
                            newVertices.push_back(mGeometryModel->addVertex((
                            mVertices[k])));
168                     }
169
170                     if (cornerPoints.size() != 4)
171                     {
172                         std::stringstream errorMessage;
173                         errorMessage << "\nError, expected to find 4
                            new vertices,\n"
174                                      << "when dividing a
```

```
                                                    tetrahedron at a line
                                                    segment.\n"
175                                                 << "Found: " <<
                                                    newVertices.size() <<
                                                    "\n"
176                                                 <<
                                                    "(bso/spatial_design/con
                                                    formal/cf_tetrahedron)."
                                                     << std::endl;
177                         throw std::runtime_error(errorMessage.str());
178                     }
179
180
                        newTetrahedron.push_back(mGeometryModel->addTetrahedr
                        on(
181                         utilities::geometry::tetrahedron(cornerPoints,
                            mGeometryModel->tolerance())));
182                 }
183
184                 if (newTetrahedron.size() != 2)
185                 {
186                     std::stringstream errorMessage;
187                     errorMessage << "\nError, expected to find 2 new
                        tetrahedrons,\n"
188                                                 << "when dividing a
                                                tetrahedron at a line
                                                segment.\n"
189                                                 << "Found: " <<
                                                newTetrahedron.size() <<
                                                "\n"
190                                                 <<
                                                "(bso/spatial_design/conform
                                                al/cf_tetrahedron)." <<
                                                std::endl;
191                     throw std::runtime_error(errorMessage.str());
192                 }
193                 break;
194             }
195         }
196     }
197
198     if (split)
199     {
200         mDeletion = true;
201         for (auto& i : mCFSpaces)
202         {
203             i->removeTetrahedron(this);
204             for (auto& j : newTetrahedron)
205             {
206                 i->addTetrahedron(j);
207                 j->addSpace(i);
208             }
209         }
210
211         this->checkAssociatedT(newTetrahedron);
212
```

```
213                 mCFSpaces.clear();
214             }
215         } // split
216 
217 
218 
219         void cf_tetrahedron::checkAssociatedT(std::vector<cf_tetrahedron*>
            newTetrahedron)
220         {
221 
222             // insert the found triangles in the triangle split function to
                split the origonal triangle
223             for (const auto& j : cfTriangles()) // origonal triangles
224             {
225                 std::vector<cf_triangle*> newTri;
226 
227                 for (const auto i : newTetrahedron)
228                 {
229                     for (const auto k : i->cfTriangles()) // new triangles
230                     {
231                         std::vector <bool> OnOrigonal;
232                         for (const auto l : k->getVertices()) // vertices of
                            new triangle
233                         {
234                             bool found = false;
235                             for (const auto m : j->getVertices()) // new
                                triangles
236                             {
237                                 if(m.isSameAs(l))
238                                 {
239                                     OnOrigonal.push_back(true);
240                                     found = true;
241                                 }
242                             }
243                             if(j->isInsideOrOn(l) && found == false)
244                             {
245                                 OnOrigonal.push_back(true);
246                             }
247                             else if(found == false)
248                             {
249                                 OnOrigonal.push_back(false);
250                             }
251                         }
252 
253                         bool isOnOgigonal = true;
254 
255                         for(const auto o: OnOrigonal)
256                         {
257                             if(o == false)
258                             {
259                                 isOnOgigonal = false;
260                                 break;
261                             }
262                         }
263 
264                         if(isOnOgigonal == false)
```

```
265                     {
266                         continue;
267                     }
268                     else if(isOnOgigonal == true)
269                     {
270                         newTri.push_back(k);
271                     }
272                 }
273             }
274 
275             if(newTri.size() > 2)
276             {
277                 j -> splitT(newTri);
278             }
279         }
280 
281         // insert the found line segments in the line split function to
            split the origonal lines
282         for (const auto& j : cfLines()) // origonal line segments
283         {
284             std::vector<cf_line*> newLine;
285 
286             for (auto i : newTetrahedron)
287             {
288                 std::vector <bool> origonalLinecheck;
289 
290                 for (auto m : (i->cfLines())) // lines of the new tetrahedron
291                 {
292                     for (auto n = m->begin(); n != m->end(); n++)
293                     {
294                         if(j ->isOnLine(*n))
295                         {
296                             origonalLinecheck.push_back(true);
297                         }
298                         else
299                         {
300                             for (auto k = j->begin(); k != j->end(); k++)
301                             {
302                                 if(*k == *n)
303                                 {
304                                     origonalLinecheck.push_back(true);
305                                 }
306                             }
307                         }
308                     }
309 
310                     if(origonalLinecheck.size() == 2) // all vertixes of
                        new polygons are on the orgigonal polygon
311                     {
312                         newLine.push_back(m);
313                     }
314                     else if (origonalLinecheck.size() > 2)
315                     {
316                         std::stringstream errorMessage;
317                         errorMessage << "\nError, something went wrong when
                            searching for the new line within the old one,\n"
```

```
318                                                        <<
                                                           "(bso/spatial_design/confor
                                                           mal/cf_cuboid)." <<
                                                           std::endl;
319                             throw std::runtime_error(errorMessage.str());
320                         }
321                         origonalLinecheck.clear();
322                     }
323                 }
324
325
326                 // Insert new lines to line split function
327                 if(newLine.size() > 1)
328                 {
329                     j -> splitN(newLine);
330                 }
331             }
332         }//checkAssociatedT
333
334     } // conformal
335     } // spatial_design
336     } // bso
337
338     #endif // CF_TETRAHEDRON_CPP
```

A.9 cf_rectangle

```
1   #ifndef CF_RECTANGLE_HPP
2   #define CF_RECTANGLE_HPP
3
4   namespace bso { namespace spatial_design { namespace conformal {
5
6       class cf_rectangle : public bso::utilities::geometry::quadrilateral,
7                                                   public cf_geometry_entity
8       {
9       private:
10
11      public:
12          cf_rectangle(const utilities::geometry::quadrilateral& rhs,
            cf_geometry_model* geomModel);
13          ~cf_rectangle();
14
15          void split(cf_vertex* pPtr);
16          void splitN(std::vector<cf_rectangle*> newRect);
17          void checkAssociated(cf_vertex* pPtr);
18
19          void addLine                (cf_line*           lPtr    )   =
            delete;
20          void addRectangle           (cf_rectangle*      recPtr)     =
            delete;
21          void addTriangle            (cf_triangle*       triPtr)     =
            delete;
22          void addTetrahedron         (cf_tetrahedron*    tetPtr)     =
            delete;
23          void removeLine             (cf_line*           lPtr    )   =
            delete;
24          void removeRectangle        (cf_rectangle*      recPtr)     =
            delete;
25          void removeTriangle         (cf_triangle*       triPtr)     =
            delete;
26          void removeTetrahedron      (cf_tetrahedron*    tetPtr)     =
            delete;
27          void addPoint               (cf_point*          pPtr    )   =
            delete;
28          void addEdge                (cf_edge*           ePtr    )   =
            delete;
29          void addSpace               (cf_space*          spPtr   )   =
            delete;
30
31          const std::vector<cf_rectangle*>     >& cfRectangles()   = delete;
32          const std::vector<cf_triangle*>      >& cfTriangle()     = delete;
33          const std::vector<cf_tetrahedron*    >& cfTetrahedrons() = delete;
34          const std::vector<cf_point*          >& cfPoints()       = delete;
35          const std::vector<cf_edge*           >& cfEdges()        = delete;
36          const std::vector<cf_space*          >& cfSpaces()       = delete;
37      };
38
39  } // conformal
40  } // spatial_design
41  } // bso
42
43  #endif // CF_RECTANGLE_HPP
```

```
1   #ifndef CF_RECTANGLE_CPP
2   #define CF_RECTANGLE_CPP
3
4   namespace bso { namespace spatial_design { namespace conformal {
5
6       cf_rectangle::cf_rectangle(const utilities::geometry::quadrilateral&
        rhs, cf_geometry_model* geometryModel)
7       : utilities::geometry::quadrilateral(rhs, geometryModel->tolerance())
8       {
9           mGeometryModel = geometryModel;
10          for (const auto& i : mVertices)
11          {
12              mCFVertices.push_back(mGeometryModel->addVertex(i));
13              mCFVertices.back()->addRectangle(this);
14          }
15          for (const auto& i : mLineSegments)
16          {
17              mCFLines.push_back(mGeometryModel->addLine(i));
18              mCFLines.back()->addRectangle(this);
19          }
20      } // ctor
21
22      cf_rectangle::~cf_rectangle()
23      {
24          for (const auto& i : mCFVertices) i->removeRectangle(this);
25          for (const auto& i : mCFLines) i->removeRectangle(this);
26      } // dtor
27
28
29      void cf_rectangle::splitN(std::vector<cf_rectangle*> newRect)
30      {
31          //Declarations
32          std::vector<cf_rectangle*> newRectangles;
33          bool split = false;
34
35          //Definition of new rectangles
36          int k = 0;
37          for (const auto i : newRect)
38          {
39              split = true;
40              std::vector<utilities::geometry::vertex> cornerVertices;
41              for(auto j = i->begin(); j != i->end(); j++)
42              {
43                  cornerVertices.push_back(*j);
44              }
45
46              newRectangles.push_back(mGeometryModel->addRectangle(
47                      utilities::geometry::quadrilateral(cornerVertices,
                        mGeometryModel->tolerance())));
48              k++;
49          }
50
51          //Removing origonal rectangle and replace it with new rectangles
52          if (split)
53          {
54              mDeletion = true;
```

```
55              for (auto& i : mCFSurfaces)
56              {
57                  i->removeRectangle(this);
58                  for (const auto& j : newRectangles)
59                  {
60                      i->addRectangle(j);
61                      j->addSurface(i);
62                  }
63              }
64              mCFSurfaces.clear();
65          }
66      }
67
68
69
70
71
72      void cf_rectangle::split(cf_vertex* pPtr)
73      { //
74          for (auto& i : mCFVertices)
75          {
76              if (pPtr == i) return;
77          }
78
79          std::vector<cf_vertex*> newVertices;
80          std::vector<cf_rectangle*> newRectangles;
81          bool split = false;
82          // check if the point is inside the rectangle
83          if (this->isInside(*pPtr, mGeometryModel->tolerance()))
84          {
85              split = true;
86              newVertices.push_back(pPtr);
87              // get the four new vertices on each of the rectangles line
                segments
88              for (const auto& i : mLineSegments)
89              { // for each
90
                    newVertices.push_back(mGeometryModel->addVertex(i.getPointClo
                    sestTo(*pPtr)));
91                  if (!(i.isOnLine(*(newVertices.back()),
                    mGeometryModel->tolerance())))
92                  {
93                      std::stringstream errorMessage;
94                      errorMessage << "Error. When finding a point on a
                        rectangle's\n"
95                                               << "line closest to a point
                                                 inside the rectangle.\n"
96                                               << "Found a point that is not
                                                 on that line.\n"
97                                               << "rectangle: " << *this <<
                                                 "\n"
98                                               << "point: " << *pPtr << "\n"
99                                               <<
                                                 "(bso/spatial_design/conformal/c
                                                 f_rectangle.cpp)" << std::endl;
100                     throw std::runtime_error(errorMessage.str());
```

```
101                 }
102             }
103 
104             for (unsigned int i = 0; i < 4; ++i)
105             { // create four new rectangles
106                 std::vector<utilities::geometry::vertex> cornerVertices;
107                 cornerVertices.push_back(*newVertices[0]);
108                 cornerVertices.push_back(*newVertices[(i)+1]);
109                 cornerVertices.push_back(*newVertices[(i+1)%4+1]);
110                 // find the vertex that both line segments have in common
111                 if (mLineSegments[i][0].isSameAs(mLineSegments[(i+1)%4][0],
                    mGeometryModel->tolerance()) ||
112 
                            mLineSegments[i][0].isSameAs(mLineSegments[(i+1)%4][1
                            ], mGeometryModel->tolerance()))
113                 {
114                     cornerVertices.push_back(mLineSegments[i][0]);
115                 }
116                 else
117                 {
118                     cornerVertices.push_back(mLineSegments[i][1]);
119                 }
120                 newRectangles.push_back(mGeometryModel->addRectangle(
121                     utilities::geometry::quadrilateral(cornerVertices,
                        mGeometryModel->tolerance())));
122             }
123 
124         }
125 
126         if (!split)
127         { // check if the point is on any of the rectangle's lines
128 
129             for (unsigned int i = 0; i < 4; ++i)
130             {
131                 if (mLineSegments[i].isOnLine(*pPtr,
                    mGeometryModel->tolerance()))
132                 {
133                     split = true;
134                     newVertices.push_back(pPtr);
135 
                        newVertices.push_back(mGeometryModel->addVertex(mLineSegm
                        ents[(i+2)%4].getPointClosestTo(*pPtr)));
136                     if
                        (!mLineSegments[(i+2)%4].isOnLine(*(newVertices.back()),
                        mGeometryModel->tolerance()))
137                     {
138                         std::stringstream errorMessage;
139                         errorMessage << "\nError. When finding a point on a
                            rectangle's\n"
140                                                         << "opposite to a point
                                                            intersecting with one of
                                                            the rectangle's lines.\n"
141                                                         << "Found a point that is
                                                            not on that line.\n"
142                                                         << "Rectangle:\n" << *this
                                                            << "\n"
```

```
143                                                     << "Line: " <<
                                                        mLineSegments[(i+2)%4] <<
                                                        "\n"
144                                                     << "Split point: " <<
                                                        pPtr->transpose() << "\n"
145                                                     << "Found point: " <<
                                                        newVertices.back()->transpos
                                                        e() << "\n"
146                                                     <<
                                                        "(bso/spatial_design/conform
                                                        al/cf_rectangle.cpp)" <<
                                                        std::endl;
147                         throw std::runtime_error(errorMessage.str());
148                     }
149
                        newRectangles.push_back(mGeometryModel->addRectangle(util
                        ities::geometry::quadrilateral({
150                         *newVertices[0],
151                         *newVertices[1],
152                         mLineSegments[(i+1)%4][0],
153                         mLineSegments[(i+1)%4][1]
154                     }, mGeometryModel->tolerance())));
155
                        newRectangles.push_back(mGeometryModel->addRectangle(util
                        ities::geometry::quadrilateral({
156                         *newVertices[0],
157                         *newVertices[1],
158                         mLineSegments[(i-1)%4][0],
159                         mLineSegments[(i-1)%4][1]
160                     }, mGeometryModel->tolerance())));
161                     break;
162                 }
163             }
164         }
165
166         if (split)
167         {
168             mDeletion = true;
169             for (auto& i : mCFSurfaces)
170             {
171                 i->removeRectangle(this);
172                 for (auto& j : newRectangles)
173                 {
174                     i->addRectangle(j);
175                     j->addSurface(i);
176                 }
177             }
178             for (const auto& i : newVertices) this->checkAssociated(i);
179             mCFSurfaces.clear();
180         }
181     } //
182
183     void cf_rectangle::checkAssociated(cf_vertex* pPtr)
184     { //
185         for (auto& i : mCFSurfaces)
186         {
```

```
187                 i->checkVertex(pPtr);
188                 for (auto& j : i->cfEdges())
189                 {
190                     j->checkVertex(pPtr);
191                 }
192                 for (auto& j : i->cfSpaces())
193                 {
194                     j->checkVertex(pPtr);
195                 }
196             }
197         } //
198 
199     } // conformal
200     } // spatial_design
201     } // bso
202 
203     #endif // CF_RECTANGLE_CPP
```

A.10 cf_triangle

```
1  #ifndef CF_TRIANGLE_HPP
2  #define CF_TRIANGLE_HPP
3
4  namespace bso { namespace spatial_design { namespace conformal {
5
6      class cf_triangle : public bso::utilities::geometry::triangle,
7                                              public cf_geometry_entity
8      {
9      private:
10
11     public:
12         cf_triangle(const utilities::geometry::triangle& rhs,
           cf_geometry_model* geomModel);
13         ~cf_triangle();
14
15         void splitT(std::vector<cf_triangle*> newTri);
16
17         void addLine                   (cf_line*          lPtr    )   =
           delete;
18         void addRectangle              (cf_rectangle*     recPtr)     =
           delete;
19         void addTriangle               (cf_triangle*      triPtr)     =
           delete;
20         void addCuboid                 (cf_cuboid*        tetPtr)     =
           delete;
21         void removeLine                (cf_line*          lPtr    )   =
           delete;
22         void removeRectangle           (cf_rectangle*     recPtr)     =
           delete;
23         void removeTriangle            (cf_triangle*      triPtr)     =
           delete;
24         void removeCuboid              (cf_cuboid*        tetPtr)     =
           delete;
25         void addPoint                  (cf_point*         pPtr    )   =
           delete;
26         void addEdge                   (cf_edge*          ePtr    )   =
           delete;
27         void addSpace                  (cf_space*         spPtr   )   =
           delete;
28
29         const std::vector<cf_rectangle*    >& cfRectangles()    = delete;
30         const std::vector<cf_triangle*     >& cfTriangle()      = delete;
31         const std::vector<cf_cuboid*       >& cfCuboids()       = delete;
32         const std::vector<cf_point*        >& cfPoints()        = delete;
33         const std::vector<cf_edge*         >& cfEdges()         = delete;
34         const std::vector<cf_space*        >& cfSpaces()        = delete;
35     };
36
37 } // conformal
38 } // spatial_design
39 } // bso
40
41 #endif // CF_TRIANGLE_HPP
```

```
1     #ifndef CF_TRIANGLE_CPP
2     #define CF_TRIANGLE_CPP
3
4     namespace bso { namespace spatial_design { namespace conformal {
5
6         cf_triangle::cf_triangle(const utilities::geometry::triangle& rhs,
          cf_geometry_model* geometryModel)
7         : utilities::geometry::triangle(rhs, geometryModel->tolerance())
8         {
9             mGeometryModel = geometryModel;
10            for (const auto& i : mVertices)
11            {
12                mCFVertices.push_back(mGeometryModel->addVertex(i));
13                mCFVertices.back()->addTriangle(this);
14            }
15            for (const auto& i : mLineSegments)
16            {
17                mCFLines.push_back(mGeometryModel->addLine(i));
18                mCFLines.back()->addTriangle(this);
19            }
20        } // ctor
21
22        cf_triangle::~cf_triangle()
23        {
24            for (const auto& i : mCFVertices) i->removeTriangle(this);
25            for (const auto& i : mCFLines) i->removeTriangle(this);
26        } // dtor
27
28
29        void cf_triangle::splitT(std::vector<cf_triangle*> newTri)
30        {
31            //Declarations
32            std::vector<cf_triangle*> newTriangles;
33            bool split = false;
34
35            //Definition of new triangles
36            int k = 0;
37            for (const auto i : newTri)
38            {
39                split = true;
40                std::vector<utilities::geometry::vertex> cornerVertices;
41                for(auto j = i->begin(); j != i->end(); j++)
42                {
43                    cornerVertices.push_back(*j);
44                }
45
46                newTriangles.push_back(mGeometryModel->addTriangle(
47                        utilities::geometry::triangle(cornerVertices,
                          mGeometryModel->tolerance())));
48                k++;
49            }
50
51            //Removing origonal rectangle and replace it with new rectangles
52            if (split)
53            {
54                mDeletion = true;
```

```
55              for (auto& i : mCFSurfaces)
56              {
57                  i->removeTriangle(this);
58                  for (const auto& j : newTriangles)
59                  {
60                      i->addTriangle(j);
61                      j->addSurface(i);
62                  }
63              }
64
65              mCFSurfaces.clear();
66          }
67      }
68
69
70  } // conformal
71  } // spatial_design
72  } // bso
73
74  #endif // CF_TRIANGLE_CPP
```

A.11 cf_line

```
1   #ifndef CF_LINE_HPP
2   #define CF_LINE_HPP
3
4   namespace bso { namespace spatial_design { namespace conformal {
5
6       class cf_line : public utilities::geometry::line_segment,
7                                       public cf_geometry_entity
8       {
9       private:
10
11      public:
12          cf_line(const utilities::geometry::line_segment& l,
            cf_geometry_model* geomModel);
13          ~cf_line();
14
15          void split(cf_vertex* pPtr);
16          void splitN(std::vector<cf_line*> newLine);
17          void splitT(std::vector<cf_line*> newLine);
18          void checkAssociated(cf_vertex* pPtr);
19          void checkAssociatedT(cf_vertex* pPtr);
20
21          void addLine                    (cf_line*           lPtr    ) =
            delete;
22          void removeLine                (cf_line*           lPtr    ) = delete;
23          void addPoint                   (cf_point*          pPtr    ) =
            delete;
24          void addSurface                (cf_surface*        srfPtr) = delete;
25          void addSpace                   (cf_space*          spPtr   ) =
            delete;
26
27          const std::vector<cf_line*         >& cfLines()            const =
            delete;
28          const std::vector<cf_point*        >& cfPoints()       const =
            delete;
29          const std::vector<cf_surface*      >& cfSurfaces()     const =
            delete;
30          const std::vector<cf_space*        >& cfSpaces()       const =
            delete;
31      };
32
33  } // conformal
34  } // spatial_design
35  } // bso
36
37  #endif // CF_LINE_HPP
```

```
1    #ifndef CF_LINE_CPP
2    #define CF_LINE_CPP
3
4    namespace bso { namespace spatial_design { namespace conformal {
5
6        cf_line::cf_line(const utilities::geometry::line_segment& l,
         cf_geometry_model* geometryModel)
7        : bso::utilities::geometry::line_segment(l)
8        {
9            mGeometryModel = geometryModel;
10           for (const auto& i : mVertices)
11           {
12               mCFVertices.push_back(mGeometryModel->addVertex(i));
13               mCFVertices.back()->addLine(this);
14           }
15       }
16
17       cf_line::~cf_line()
18       {
19           for (auto& i : mCFVertices) i->removeLine(this);
20       } // dtor
21
22
23       void cf_line::splitN(std::vector<cf_line*> newLine)
24       {
25           bool split = false;
26           if(newLine.size() > 1)
27           {
28               split = true;
29           }
30
31
32           if(split == true)
33           {
34               mDeletion = true;
35
36               for(const auto j: newLine)
37               {
38                   mGeometryModel->addLine(*j);
39               }
40
41               for (auto& i : mCFEdges)
42               {
43                   i->removeLine(this);
44
45                   for(const auto j: newLine)
46                   {
47                       j->addEdge(i);
48                       i->addLine(j);
49                   }
50               }
51
52               mCFEdges.clear();
53           }
54       } // splitN()
55
```

```
56        void cf_line::splitT(std::vector<cf_line*> newLine)
57        {
58            bool split = false;
59            if(newLine.size() > 1)
60            {
61                split = true;
62            }
63
64
65            if(split == true)
66            {
67                mDeletion = true;
68
69                for(const auto j: newLine)
70                {
71                    mGeometryModel->addLine(*j);
72                }
73
74                for (auto& i : mCFEdges)
75                {
76                    i->removeLine(this);
77
78                    for(const auto j: newLine)
79                    {
80                        j->addEdge(i);
81                        i->addLine(j);
82                    }
83                }
84
85                mCFEdges.clear();
86            }
87        } // splitT()
88
89        void cf_line::split(cf_vertex* pPtr)
90        {
91            for (auto& i : mCFVertices)
92            {
93                if (pPtr == i) return;
94            }
95
96            if (this->isOnLine(*pPtr, mGeometryModel->tolerance()))
97            {
98                mDeletion = true;
99                cf_line* l1 = mGeometryModel->addLine({mVertices[0], *pPtr});
100               cf_line* l2 = mGeometryModel->addLine({mVertices[1], *pPtr});
101
102               for (auto& i : mCFEdges)
103               {
104                   l1->addEdge(i);
105                   l2->addEdge(i);
106                   i->addLine(l1);
107                   i->addLine(l2);
108                   i->removeLine(this);
109               }
110
111               this->checkAssociated(pPtr);
```

```
112                mCFEdges.clear();
113            }
114        } // split()
115
116        void cf_line::checkAssociated(cf_vertex* pPtr)
117        {
118            for (auto& i : mCFEdges)
119            {
120                i->checkVertex(pPtr);
121                for (auto& j : i->cfSurfaces())
122                {
123                    j->checkVertex(pPtr);
124                }
125                for (auto& j : i->cfSpaces())
126                {
127                    j->checkVertex(pPtr);
128                }
129            }
130        } // checkAssociated()
131
132        void cf_line::checkAssociatedT(cf_vertex* pPtr)
133        {
134            for (auto& i : mCFEdges)
135            {
136                i->checkVertex(pPtr);
137                for (auto& j : i->cfSurfaces())
138                {
139                    j->checkVertex(pPtr);
140                }
141                for (auto& j : i->cfSpaces())
142                {
143                    j->checkVertex(pPtr);
144                }
145            }
146        } // checkAssociated()
147
148    } // conformal
149    } // spatial_design
150    } // bso
151
152    #endif // CF_LINE_CPP
```

A.12 cf_building

```cpp
#ifndef CONFORMAL_MODEL_HPP
#define CONFORMAL_MODEL_HPP

#include <bso/spatial_design/cf_building.hpp>
#include <bso/visualization/models/model_base.hpp>
#include <bso/visualization/bsp/bsp.hpp>

#include <cstdlib>

namespace bso { namespace visualization
{

class Conformal_Model : public model_base
{
public:
        Conformal_Model(const bso::spatial_design::cf_building& cf,
                                            const std::string& type, const
                                            std::string& title);
        ~Conformal_Model();
        void render(const camera &cam) const;
        const std::string get_description();

        bool key_pressed(int key);

protected:

private:
        std::string mTitle;
        std::list<polygon*> polygons;
        std::list<label*>   labels;

        polygon_props  pprops_rectangle;
        polygon_props  pprops_cuboid;
        polygon_props  pprops_cuboid_overlap;

        line_props     lprops;
        label_props    lbprops;
        random_bsp     *pbsp;
};

Conformal_Model::Conformal_Model(const bso::spatial_design::cf_building& cf,
                                                                                const
std::string& type, const std::string& title)
{
    std::cout << "Conformal_Model::Conformal_Model() is used" << std::endl;

    mTitle = title;
    pprops_rectangle.ambient = rgba(0.5f, 0.5f, 0.05f, 0.3f);
    pprops_rectangle.diffuse = rgba(1.0f, 1.0f, 0.1f, 0.3f);
    pprops_rectangle.specular = rgba(1.0f, 1.0f, 0.2f, 0.3f);
    pprops_rectangle.emission = rgba(0.0f, 0.0f, 0.0f, 0.0f);
    pprops_rectangle.shininess = 60.0;
    pprops_rectangle.translucent = true;
    pprops_rectangle.twosided = true;

    bool standart = true;
```

```
55
56        if(standart == true)
57        {
58            pprops_cuboid.ambient = rgba(0.1f, 0.5f, 0.1f, 0.3f);
59            pprops_cuboid.diffuse = rgba(0.2f, 1.0f, 0.2f, 0.3f);
60            pprops_cuboid.specular = rgba(0.2f, 1.0f, 0.2f, 0.3f);
61            pprops_cuboid.emission = rgba(0.0f, 0.0f, 0.0f, 0.0f);
62            pprops_cuboid.shininess = 60.0;
63            pprops_cuboid.translucent = true;
64            pprops_cuboid.twosided = true;
65
66            pprops_cuboid_overlap.ambient = rgba(1.0f, 0.1f, 0.1f, 1.0f);
67            pprops_cuboid_overlap.diffuse = rgba(1.0f, 0.2f, 0.2f, 1.0f);
68            pprops_cuboid_overlap.specular = rgba(1.0f, 0.2f, 0.2f, 1.0f);
69            pprops_cuboid_overlap.emission = rgba(0.0f, 0.0f, 0.0f, 0.0f);
70            pprops_cuboid_overlap.shininess = 60.0;
71            pprops_cuboid_overlap.translucent = true;
72            pprops_cuboid_overlap.twosided = true;
73        }
74        else
75        {
76            pprops_cuboid.ambient = rgba(0.7f, 0.7f, 0.7f, 0.3f);
77            pprops_cuboid.diffuse = rgba(0.2f, 0.2f, 0.2f, 0.3f);
78            pprops_cuboid.specular = rgba(0.2f, 0.2f, 0.2f, 0.3f);
79            pprops_cuboid.emission = rgba(0.0f, 0.0f, 0.0f, 0.0f);
80            pprops_cuboid.shininess = 60.0;
81            pprops_cuboid.translucent = true;
82            pprops_cuboid.twosided = true;
83
84            pprops_cuboid_overlap.ambient = rgba(1.0f, 0.1f, 0.1f, 1.0f);
85            pprops_cuboid_overlap.diffuse = rgba(1.0f, 0.2f, 0.2f, 1.0f);
86            pprops_cuboid_overlap.specular = rgba(1.0f, 0.2f, 0.2f, 1.0f);
87            pprops_cuboid_overlap.emission = rgba(0.0f, 0.0f, 0.0f, 0.0f);
88            pprops_cuboid_overlap.shininess = 60.0;
89            pprops_cuboid_overlap.translucent = true;
90            pprops_cuboid_overlap.twosided = true;
91        }
92
93        double scale = 1.0;//0.8; cell scale factor
94        namespace geom = bso::utilities::geometry;
95
96        //Add the space indixes to the visualization models
97        for (const auto& i : cf.cfSpaces())
98        {
99
              this->addLabel(labels,&lbprops,std::to_string(i->getSpaceID()),i->get
              Center());
100       }
101
102       if (type == "line_segment")
103       {
104           for (const auto& i : cf.cfLines())
105           {
106               std::vector<geom::vertex> lineVertices;
107               for (const auto& j : *i)
108               {
```

```
109                 lineVertices.push_back(i->getCenter() + scale*(j -
                    i->getCenter()));
110             }
111             geom::line_segment lineGeometry(lineVertices);
112
                this->addLineSegment(polygons,&lineGeometry,&pprops_rectangle,&lp
                rops);
113         }
114     }
115     else if (type == "rectangle")
116     {
117         for (const auto& i : cf.cfRectangles())
118         {
119             std::vector<geom::vertex> rectangleVertices;
120             for (const auto& j : *i)
121             {
122                 rectangleVertices.push_back(i->getCenter() + scale*(j -
                    i->getCenter()));
123             }
124             geom::quadrilateral rectangleGeometry(rectangleVertices);
125
                this->addPolygon(polygons,&rectangleGeometry,&pprops_rectangle,&l
                props,0.0);
126         }
127     }
128     else if (type == "triangle")
129     {
130         std::cout << "iterate tough cf.cfTriangles() of size: " <<
            (cf.cfTriangles()).size() << std::endl;
131         for (const auto& i : cf.cfTriangles())
132         {
133             std::vector<geom::vertex> triangleVertices;
134             for (const auto& j : *i)
135             {
136                 triangleVertices.push_back(i->getCenter() + scale*(j -
                    i->getCenter()));
137             }
138             geom::triangle triangleGeometry(triangleVertices);
139
                this->addPolygon(polygons,&triangleGeometry,&pprops_rectangle,&lp
                rops,0.0);
140         }
141     }
142     else if (type == "cuboid")
143     {
144         for (const auto& i : cf.cfCuboids())
145         {
146             std::vector<geom::vertex> cuboidVertices;
147             for (const auto& j : *i)
148             {
149                 cuboidVertices.push_back(i->getCenter() + scale*(j -
                    i->getCenter()));
150             }
151             geom::quad_hexahedron cuboidGeometry(cuboidVertices);
152             if (i->cfSpaces().size() > 1)
153             {
```

```
154
                    this->addPolyhedron(polygons,&cuboidGeometry,&pprops_cuboid_o
                    verlap,&lprops);
155             }
156             else
157             {
158
                    this->addPolyhedron(polygons,&cuboidGeometry,&pprops_cuboid,&
                    lprops);
159             }
160         }
161     }
162     else if (type == "tetrahedron")
163     {
164         std::cout << "iterate tough cf.cfTetrahedrons() of size: " <<
            (cf.cfTetrahedrons()).size() << std::endl;
165         for (const auto& i : cf.cfTetrahedrons())
166         {
167             std::vector<geom::vertex> tetrahedronVertices;
168             for (const auto& j : *i)
169             {
170                 tetrahedronVertices.push_back(i->getCenter() + scale*(j -
                    i->getCenter()));
171             }
172             geom::tetrahedron tetrahedronGeometry(tetrahedronVertices);
173             if (i->cfSpaces().size() > 1)
174             {
175
                    this->addPolyhedron(polygons,&tetrahedronGeometry,&pprops_cub
                    oid_overlap,&lprops);
176             }
177             else
178             {
179
                    this->addPolyhedron(polygons,&tetrahedronGeometry,&pprops_cub
                    oid,&lprops);
180             }
181         }
182     }
183
184     pbsp = new random_bsp(polygons);
185 }
186
187 Conformal_Model::~Conformal_Model()
188 {
189         delete pbsp;
190
191         for (std::list<polygon*>::iterator pit = polygons.begin();
192                  pit != polygons.end(); pit++)
193                 delete *pit;
194
195         for (std::list<label*>::iterator lbit = labels.begin();
196                  lbit != labels.end(); lbit++)
197                 delete *lbit;
198 }
199
```

```
200  const std::string Conformal_Model::get_description()
201  {
202          return mTitle;
203  }
204
205  void Conformal_Model::render(const camera &cam) const
206  {
207      glPushAttrib(GL_ENABLE_BIT);
208
209      glDisable(GL_DEPTH_TEST);
210
211      pbsp->render_btf(cam);
212
213      std::list<label*>::const_iterator lbit;
214      for (lbit = labels.begin(); lbit != labels.end(); lbit++)
215      {
216          (*lbit)->render();
217      }
218      glPopAttrib();
219  }
220
221  bool Conformal_Model::key_pressed(int key)
222  {
223          switch (key)
224          {
225          case 't':
226          case 'T':
227              //toggle geometry translucency
228              pprops_rectangle.translucent = !pprops_rectangle.translucent;
229              pprops_cuboid.translucent = !pprops_cuboid.translucent;
230
231              return true;
232          };
233      return false;
234  }
235
236  } // namespace Visualisation
237  } // namespace BSO
238
239
240  #endif // CONFORMAL_MODEL_HPP
241
```

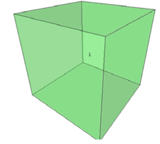
B Designs tested

Multiple NOBSD were tested to check all different functionalities of both the geometry conformal methods during its development process. Most of these will be presented within this section. These NOBSD are subdivided into the design tested according to the quad-hexahedron method, tetrahedron method, and the developed design for the demonstration of the methods.

Quad-hexahedron method

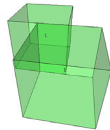
Design_1

- R, 1,5000,5000,5000,0,0,0



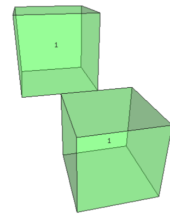
Design_2

- R, 1,5000,5000,5000,0,0,0
- R, 1,5000,5000,5000,5000,2500,0



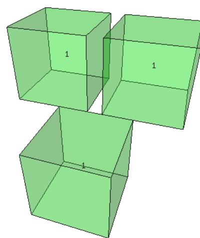
Design_3

- R, 1,5000,5000,5000,0,0,0
- R, 1,5000,5000,5000,5000,2500,5000



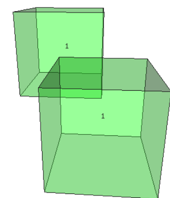
Design_4

- R, 1,5000,5000,5000,0,0,0
- R, 1,5000,5000,5000,5000,3000,5000
- R, 1,5000,5000,5000,5000,-3000,5000



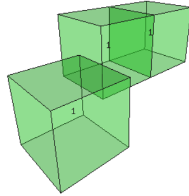
Design_5

- R, 1,5000,5000,5000,0,0,0
- R, 1,5000,5000,5000,5000,2500,2500



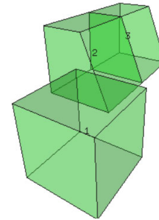
Design_6

- R, 1,5000,5000,5000,0,0,0
- R, 1,5000,5000,5000,5000,2500,2500
- R, 1,5000,5000,5000,10000,2500,2500



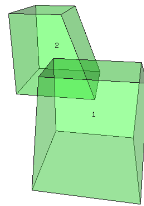
Design_7

- N, 1, 0,0,0, 5000,0,0,
  0,5000,0, 5000,5000,0, 0,0,5000,
  5000,0,5000, 0,5000,5000,
  5000,5000,5000
- N, 2, 5000,2500,2500, 10000,2500,2500,
  5000,7500,2500, 10000,7500,2500,
  5000,4500,7500, 10000,4500,7500,
  5000,7500,7500,
  10000,7500,7500
- N, 3, 10000,2500,2500, 15000,2500,2500,
  10000,7500,2500, 15000,7500,2500,
  10000,4500,7500, 15000,4500,7500,
  10000,7500,7500,
  15000,7500,7500



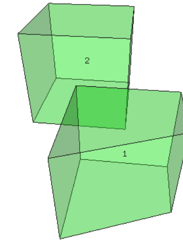
Design_8

- N, 1,0,0,0, 5000,0,0, 0,5000,0,
  5000,5000,0, 2000,0,5000,
  5000,0,5000, 2000,5000,5000,
  5000,5000,5000
- N, 2,5000,2500,2500, 10000,2500,2500,
  5000,7500,2500, 10000,7500,2500,
  5000,4500,7500, 10000,4500,7500,
  5000,7500,7500,
  10000,7500,7500



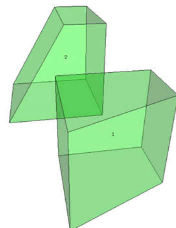
Design_9

- N, 1, 2000,0,0, 5000,0,0,
  0,5000,0, 5000,5000,0,
  2000,0,5000, 5000,0,5000,
  0,5000,5000,
  5000,5000,5000
- N, 2, 5000,2500,2500,
  10000,2500,2500,
  5000,7500,2500,
  10000,7500,2500,
  5000,2500,7500,
  10000,2500,7500,
  5000,7500,7500,
  10000,7500,7500



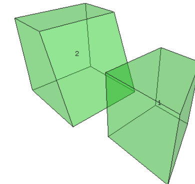
Design_10

- N, 1, 2000,0,0, 5000,0,0,
  0,5000,0, 5000,5000,0,
  2000,0,5000, 5000,0,5000,
  0,5000,5000,
  5000,5000,5000
- N, 2, 5000,2500,2500,
  10000,2500,2500,
  5000,7500,2500,
  10000,7500,2500,
  5000,4500,7500,
  10000,4500,7500,
  5000,2500,7500,
  10000,2500,7500



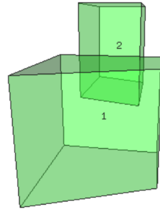
Design_11

- N, 1, 2000,0,0, 5000,0,0,
  0,5000,0, 5000,5000,0,
  2000,0,5000, 5000,0,5000,
  0,5000,5000,
  5000,5000,5000
- N, 2, 5000,2500,2500,
  10000,2500,2500,
  5000,7500,2500,
  10000,7500,2500,
  7000,2500,7500,
  10000,2500,7500,
  7000,7500,7500,
  10000,7500,7500



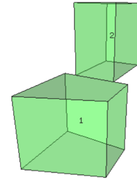
Design_12

- N, 1, 2000,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 2000,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,1000,2500, 10000,1000,2500, 5000,4000,2800, 10000,4000,2800, 5000,1000,7500, 10000,1000,7500, 5000,4000,7500, 10000,4000,7500



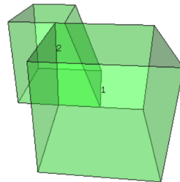
Design_13

- N, 1, 2000,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 2000,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,1000,5000, 10000,1000,5000, 5000,4000,5000, 10000,4000,5000, 5000,1000,10000, 10000,1000,10000, 5000,4000,10000, 10000,4000,10000



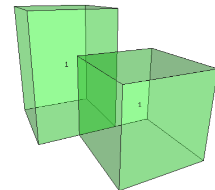
Design_14

- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,2500,0, 10000,2500,0, 5000,7500,0, 10000,7500,0, 5000,4500,5000, 10000,4500,5000, 5000,7500,5000, 10000,7500,5000



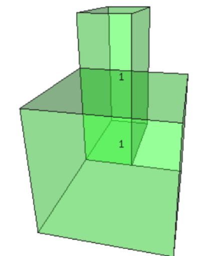
Design_15

- R, 1,5000,5000,5000,0,0,0
- R, 1,5000,5000,7500,5000,2500,0



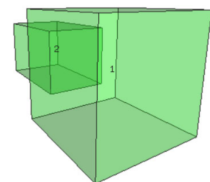
Design_16

- R, 1,5000,5000,5000,0,0,0
- R, 1,5000,2500,7500,5000,2500,0



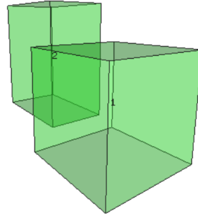
Design_17

- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,1000,1000, 10000,1000,1000, 5000,4000,1000, 10000,4000,1000, 5000,1000,4000, 10000,1000,4000, 5000,4000,4000, 10000,4000,4000



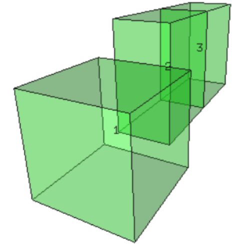
Design_18

- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,1000,1000, 10000,1000,1000, 5000,4000,1000, 10000,4000,1000, 5000,1000,7000, 10000,1000,7000, 5000,4000,7000, 10000,4000,7000



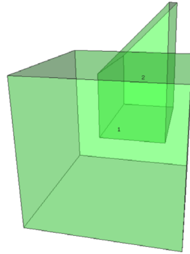
Design_19

- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,1000,1000, 10000,1000,1000, 5000,4000,1000, 10000,4000,1000, 5000,1000,7000, 10000,1000,7000, 5000,4000,7000, 10000,4000,7000
- N, 3, 10000,1000,1000, 15000,1000,1000, 10000,4000,1000, 15000,4000,1000, 10000,1000,7000, 15000,1000,7000, 10000,4000,7000, 15000,4000,7000



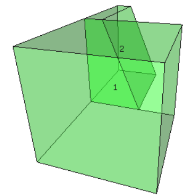
Design_20

- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,1000,1000, 10000,1000,1000, 5000,4000,1000, 10000,4000,1000, 5000,1000,7000, 10000,1000,7000, 5000,4000,4000, 10000,4000,4000



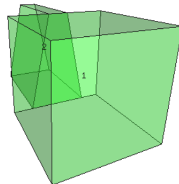
Design_21

- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,2000,0, 10000,2000,0, 5000,5000,0, 10000,5000,0, 5000,4000,5000, 10000,4000,5000, 5000,5000,5000, 10000,5000,5000



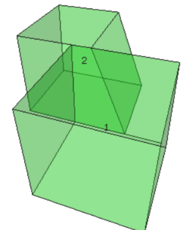
Design_22

- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,1000,0, 10000,1000,0, 5000,4000,0, 10000,4000,0, 5000,2000,5000, 10000,2000,5000, 5000,3000,5000, 10000,3000,5000



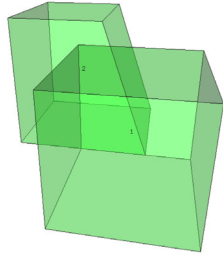
Design_23

- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,2000,0, 10000,2000,0, 5000,8000,0, 10000,8000,0, 5000,4000,5000, 10000,4000,5000, 5000,8000,5000, 10000,8000,5000



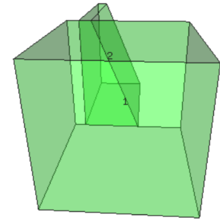
Design_24

- N, 1, 0,0,0, 5000,0,0, 0,5000,0,
  5000,5000,0, 0,0,5000,
  5000,0,5000, 0,5000,5000,
  5000,5000,5000
- N, 2, 5000,2000,0, 10000,2000,0,
  5000,8000,0, 10000,8000,0,
  5000,4000,6000,
  10000,4000,6000,
  5000,8000,6000,
  10000,8000,6000



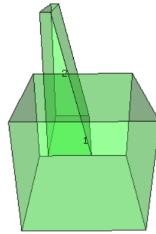
Design_25

- N, 1, 0,0,0, 5000,0,0, 0,5000,0,
  5000,5000,0, 0,0,5000,
  5000,0,5000, 0,5000,5000,
  5000,5000,5000
- N, 2, 5000,2000,0, 10000,2000,0,
  5000,4500,0, 10000,4500,0,
  5000,4000,5000,
  10000,4000,5000,
  5000,4500,5000,
  10000,4500,5000



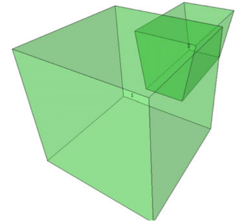
Design_26

- N, 1, 0,0,0, 5000,0,0, 0,5000,0,
  5000,5000,0, 0,0,5000,
  5000,0,5000, 0,5000,5000,
  5000,5000,5000
- N, 2, 5000,2000,0, 10000,2000,0,
  5000,4500,0, 10000,4500,0,
  5000,4000,8000,
  10000,4000,8000,
  5000,4500,8000,
  10000,4500,8000



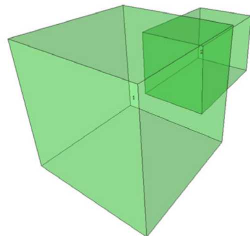
Design_27

- N, 1, 0,0,0, 5000,0,0, 0,5000,0,
  5000,5000,0, 0,0,5000,
  5000,0,5000, 0,5000,5000,
  5000,5000,5000
- N, 2, 5000,1500,1000,
  10000,1500,1000,
  5000,3500,1000,
  10000,3500,1000,
  5000,1000,4000,
  10000,1000,4000,
  5000,4000,4000,
  10000,4000,4000



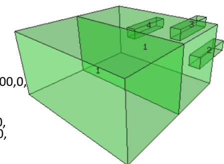
Desing_28

- N, 1, 0,0,0, 5000,0,0, 0,5000,0,
  5000,5000,0, 0,0,5000,
  5000,0,5000, 0,5000,5000,
  5000,5000,5000
- N, 2, 5000,1000,1000,
  10000,1000,1000,
  5000,4000,1000,
  10000,4000,1000,
  5000,1100,4000,
  10000,1100,4000,
  5000,4000,4000,
  10000,4000,4000

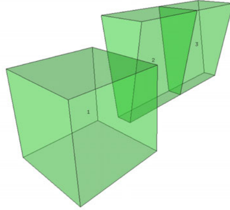


Design_29

- N, 1, 0,0,0, 5000,0,0, 0,10000,0, 5000,10000,0,
  0,0,5000, 5000,0,5000, 0,10000,5000,
  5000,10000,5000
- N, 1, -5000,0,0, 0,0,0, -5000,10000,0, 0,10000,0,
  -5000,0,5000, 0,0,5000, -5000,10000,5000,
  0,10000,5000
- N, 2, 5000,1000,1000, 10000,1000,1000,
  5000,2000,1000, 10000,2000,1000,
  5000,1000,2000, 10000,1000,2000,
  5000,2000,2000, 10000,2000,2000
- N, 3, 5000,3000,3000, 10000,3000,3000,
  5000,4000,3000, 10000,4000,3000,
  5000,3000,4000, 10000,3000,4000,
  5000,4000,4000, 10000,4000,4000
- N, 4, 5000,8000,2000, 10000,8000,2000,
  5000,9000,2000, 10000,9000,2000,
  5000,8000,3000, 10000,8000,3000,
  5000,9000,3000, 10000,9000,3000

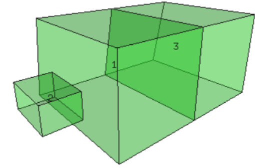


Design_30



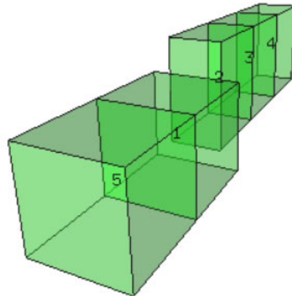
- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,2000,1000, 10000,2000,1000, 5000,3000,1000, 10000,3000,1000, 5000,1000,7000, 10000,1000,7000, 5000,4000,7000, 10000,4000,7000
- N, 3, 10000,2000,1000, 15000,2000,1000, 10000,3000,1000, 15000,3000,1000, 10000,1000,7000, 15000,1000,7000, 10000,4000,7000, 15000,4000,7000

Design_31



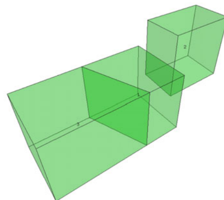
- N, 1, 0,0,0, 8000,0,0, 0,5000,0, 8000,5000,0, 1000,0,5000, 8000,0,5000, 1000,5000,5000, 8000,5000,5000
- N, 2, 2300,0,800, 5300,0,800, 2300,-2000,800, 5300,-2000,800, 2300,0,2300, 5300,0,2300, 2300,-2000,2300, 5300,-2000,2300
- N, 3, 0,5000,0, 8000,5000,0, 0,10000,0, 8000,10000,0, 1000,5000,5000, 8000,5000,5000, 1000,10000,5000, 8000,10000,5000

Design_32



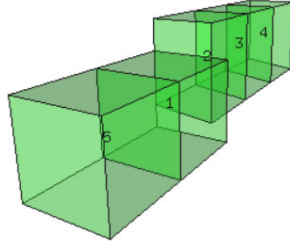
- N, 5,-5000,0,0, 0,0,0, -5000,5000,0, 0,5000,0, -5000,0,5000, 0,0,5000, -5000,5000,5000, 0,5000,5000
- N, 3,10000,1000,1000, 15000,1000,1000, 10000,4000,1000, 15000,4000,1000, 10000,1000,7000, 15000,1000,7000, 10000,4000,7000, 15000,4000,7000
- N, 1,0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2,5000,1000,1000, 10000,1000,1000, 5000,4000,1000, 10000,4000,1000, 5000,1000,7000, 10000,1000,7000, 5000,4000,7000, 10000,4000,7000
- N, 4,15000,1000,1000, 20000,1000,1000, 15000,4000,1000, 20000,4000,1000, 15000,1000,7000, 20000,1000,7000, 15000,4000,7000, 20000,4000,7000

Design_33



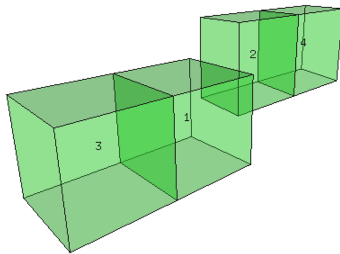
- N, 3, 2000,0,0, -5000,0,0, 0,5000,0, -5000,5000,0, 2000,0,5000, -5000,0,5000, 0,5000,5000, -5000,5000,5000
- N, 1, 2000,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 2000,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,1000,2500, 10000,1000,2500, 5000,4000,2800, 10000,4000,2800, 5000,1000,7500, 10000,1000,7500, 5000,4000,7500, 10000,4000,7500

Design_34



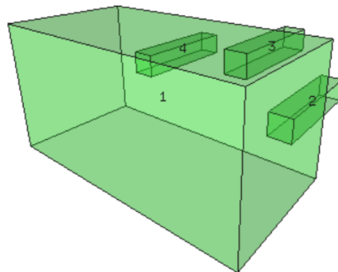
- N, 5,-5000,0,0, 0,0,0, -5000,5000,0, 0,5000,0, -5000,0,5000, 0,0,5000, -5000,5000,5000, 0,5000,5000
- N, 3,10000,1000,1000, 15000,1000,1000, 10000,5000,1500, 15000,5000,1500, 10000,1000,7000, 15000,1000,7000, 10000,5000,7000, 15000,5000,7000
- N, 1,0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2,5000,1000,1000, 10000,1000,1000, 5000,5000,1500, 10000,5000,1500, 5000,1000,7000, 10000,1000,7000, 5000,5000,7000, 10000,5000,7000
- N, 4,15000,1000,1000, 20000,1000,1000, 15000,5000,1500, 20000,5000,1500, 15000,1000,7000, 20000,1000,7000, 15000,5000,7000, 20000,5000,7000

Design_35



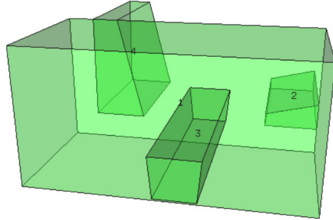
- N, 3,-5000,0,0, 0,0,0, -5000,5000,0, 0,5000,0, -5000,0,5000, 0,0,5000, -5000,5000,5000, 0,5000,5000
- N, 1,0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2,5000,1000,2500, 10000,1000,2500, 5000,4000,2800, 10000,4000,2800, 5000,1000,7500, 10000,1000,7500, 5000,4000,7500, 10000,4000,7500
- N, 4,10000,1000,2500, 15000,1000,2500, 10000,4000,2800, 15000,4000,2800, 10000,1000,7500, 15000,1000,7500, 10000,4000,7500, 15000,4000,7500

Design_36



- N, 1,0,0,0, 5000,0,0, 0,10000,0, 5000,10000,0, 0,0,5000, 5000,0,5000, 0,10000,5000, 5000,10000,5000
- N, 2,5000,1000,1000, 10000,1000,1000, 5000,2000,1000, 10000,2000,1000, 5000,1000,2000, 10000,1000,2000, 5000,2000,2000, 10000,2000,2000
- N, 3,5000,3000,3000, 10000,3000,3000, 5000,4000,3000, 10000,4000,3000, 5000,3000,4000, 10000,3000,4000, 5000,4000,4000, 10000,4000,4000
- N, 4,5000,8000,2000, 10000,8000,2000, 5000,9000,2000, 10000,9000,2000, 5000,8000,3000, 10000,8000,3000, 5000,9000,3000, 10000,9000,3000

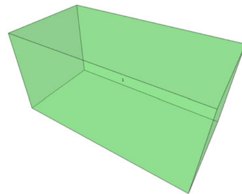
Design_37



- N, 1,0,0,0, 5000,0,0, 0,10000,0, 5000,10000,0, 0,0,5000, 5000,0,5000, 0,10000,5000, 5000,10000,5000
- N, 2,5000,0,1000, 10000,0,1000, 5000,2000,1000, 10000,2000,1000, 5000,0,2000, 10000,0,2000, 5000,2000,1500, 10000,2000,1500
- N, 3,0,3000,3000, -5000,3000,3000, 0,4000,3000, -5000,4000,3000, 0,2900,4000, -5000,2900,4000, 0,4100,4000, -5000,4100,4000
- N, 4,5000,7000,1000, 10000,7000,1000, 5000,9000,1000, 10000,9000,1000, 5000,8000,5000, 10000,8000,5000, 5000,9000,5000, 10000,9000,5000

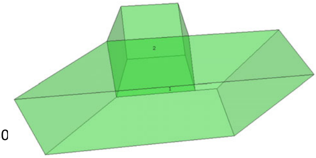
Design_38

- N, 1,0,0,0, 5000,0,0, 0,10000,0, 5000,10000,0, 0,0,5000, 5000,0,5000, 0,10000,5000, 5000,10000,5000

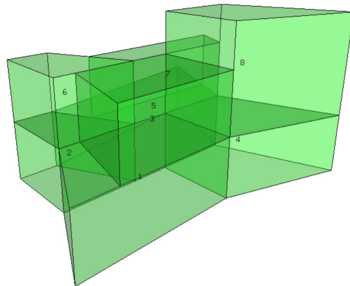


Design_39

- N, 1, -2000,0,0, 5000,0,0, 5000,2500,0, 500,2500,0, -2000,0,3000, 7000,0,3000, 7000,2500,30 500,2500,3000
- N, 2, 500,2500,0, 4000,2500,0, 1000,5000,0, 4000,5000,0, 500,2500,3000, 3500,2500,3000, 1000,5000,3000, 3500,5000,3000



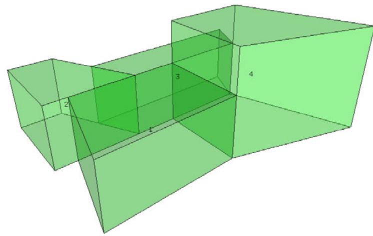
Design_40



- N, 1, 0,0,0, 8000,0,0, 8000,4000,0, 2000,4000,0, 0,0,3000, 8000,0,3000, 8000,4000,3000, 2000,4000,3000
- N, 3, 6000,4000,0, 12000,4000,0, 15000,10000,0, 4000,8000,0, 6000,4000,3000, 12000,4000,3000, 15000,10000,3000, 4000,8000,3000
- N, 2, 2000,4000,0, 6000,4000,0, 4000,8000,0, 2000,8000,0, 2000,4000,3000, 6000,4000,3000, 4000,8000,3000, 2000,8000,3000
- N, 4, 8000,0,0, 14000,-2000,0, 12000,4000,0, 8000,4000,0, 8000,0,3000, 14000,-2000,3000, 12000,4000,3000, 8000,4000,3000,
- N, 5, 2000,0,3000, 8000,0,3000, 8000,4000,3000, 3000,4000,3000, 2000,0,6000, 8000,0,6000, 8000,4000,6000, 3000,4000,6000

- N, 6, 2000,4000,3000, 6000,4000,3000, 4000,8000,3000, 2000,8000,3000, 2000,4000,6000, 6000,4000,6000, 4000,8000,6000, 2000,8000,6000
- N, 7, 6000,4000,3000, 12000,4000,3000, 13000,6000,3000, 5000,6000,3000, 6000,4000,6000, 12000,4000,6000, 13000,6000,6000, 5000,6000,6000
- N, 8, 8000,0,3000, 14000,-2000,3000, 12000,4000,3000, 8000,4000,3000, 8000,0,8000, 14000,-2000,8000, 12000,4000,8000, 8000,4000,8000,

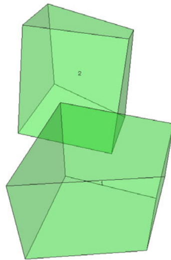
Design_41



- N, 1, 2000,0,3000, 8000,0,3000, 8000,4000,3000, 3000,4000,3000, 2000,0,6000, 8000,0,6000, 8000,4000,6000, 3000,4000,6000
- N, 2, 2000,4000,3000, 6000,4000,3000, 4000,8000,3000, 2000,8000,3000, 2000,4000,6000, 6000,4000,6000, 4000,8000,6000, 2000,8000,6000
- N, 3, 6000,4000,3000, 12000,4000,3000, 13000,6000,3000, 5000,6000,3000, 6000,4000,6000, 12000,4000,6000, 13000,6000,6000, 5000,6000,6000
- N, 4, 8000,0,3000, 14000,-2000,3000, 12000,4000,3000, 8000,4000,3000, 8000,0,8000, 14000,-2000,8000, 12000,4000,8000, 8000,4000,8000,

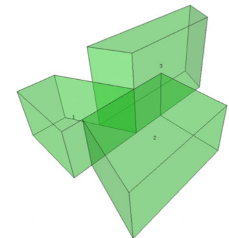
Design_42

- N, 1, 2000,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 2000,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,2500,2500, 8000,2500,2500, 5000,7500,2500, 10000,7500,2500, 7000,2500,7500, 8000,2500,7500, 7000,7500,7500, 10000,7500,7500



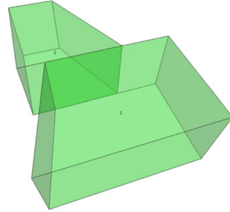
Design_43

- N,1, 2000,4000,3000, 6000,4000,3000, 4000,8000,3000, 2000,8000,3000, 2000,4000,6000, 6000,4000,6000, 4000,8000,6000, 2000,8000,6000
- N,2, 2000,0,3000, 8000,0,3000, 8000,4000,3000, 3000,4000,3000, 2000,0,6000, 8000,0,6000, 8000,4000,6000, 3000,4000,6000
- N,3, 6000,4000,3000, 12000,4000,3000, 13000,6000,3000, 5000,6000,3000, 6000,4000,8000, 12000,4000,8000, 13000,6000,8000, 5000,6000,8000



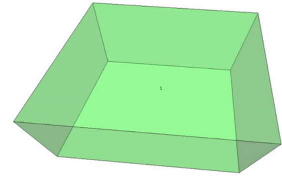
Design_44

- N,1, 2000,0,3000,
8000,0,3000,
8000,4000,3000,
3000,4000,3000,
2000,0,6000,
8000,0,6000, 8000,4000,6000,
3000,4000,6000
- N,2, 2000,4000,3000,
6000,4000,3000,
4000,8000,3000,
2000,8000,3000,
2000,4000,6000,
6000,4000,6000,
4000,8000,6000,
2000,8000,6000



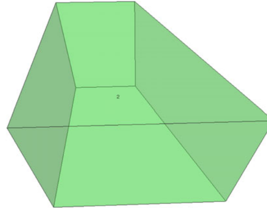
Design_45

- N,1, 2000,0,3000,
8000,0,3000,
8000,4000,3000,
3000,4000,3000,
2000,0,6000,
8000,0,6000,
8000,4000,6000,
3000,4000,6000



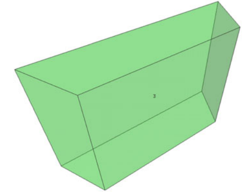
Design_46

- N,2, 2000,4000,3000,
6000,4000,3000,
4000,8000,3000,
2000,8000,3000,
2000,4000,6000,
6000,4000,6000,
4000,8000,6000,
2000,8000,6000



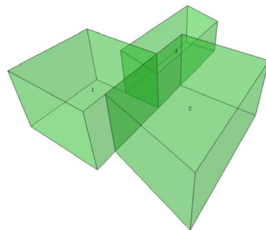
Design_47

- N,3, 6000,4000,3000,
12000,4000,3000,
13000,6000,3000,
5000,6000,3000,
6000,4000,8000,
12000,4000,8000,
13000,6000,8000,
5000,6000,8000



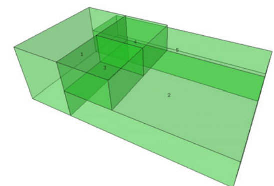
Design_48

- N,1, 2000,4000,0, 6000,4000,0,
6000,8000,0, 2000,8000,0,
2000,4000,3000,
6000,4000,3000,
6000,8000,3000,
2000,8000,3000
- N,2, 2000,0,0, 8000,0,0,
8000,4000,0, 3000,4000,0,
2000,0,3000, 8000,0,3000,
8000,4000,3000,
3000,4000,3000
- N,3, 6000,4000,0, 12000,4000,0,
12000,6000,0, 6000,6000,0,
6000,4000,2000, 12000,4000,2000,
12000,6000,2500, 6000,6000,2500



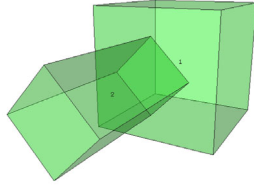
Design_49

- R, 1,5000,10000,6000,0,0,0
- R, 2,15000,10000,3000,5000,0,0
- R, 3,5000,5000,3000,5000,0,3000
- R, 4,5000,5000,3000,5000,5000,3000
- R, 5,20000,5000,3000,0,10000,0



Design_50

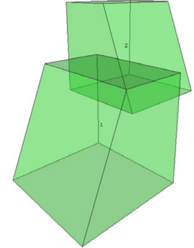
- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000, 5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 2, 5000,1000,2500, 10000,1000,2500, 5000,4000,2500, 10000,4000,2500, 5000,2500,4000, 10000,2500,4000, 5000,2500,1000, 10000,2500,1000



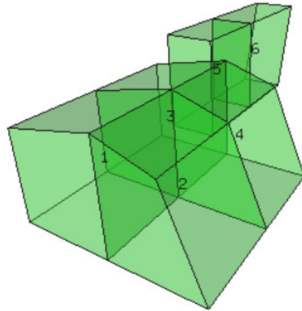
Demo_cases developement

Design_1a

- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 2000,0,5000, 5000,0,5000, 2000,5000,5000, 5000,5000,5000
- N, 2, 5000,2500,2500, 10000,2500,2500, 5000,7500,2500, 10000,7500,2500, 5000,4500,7500, 10000,4500,7500, 5000,7500,7500, 10000,7500,7500

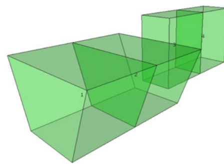


Design_18



- N, 1,0,0,0, -5000,0,0, 0,5000,0, -5000,5000,0, 0,0,6000, -5000,0,6000, 0,5000,5000, -5000,5000,5000
- N, 2,0,0,0, -5000,0,0, 0,-5000,0, -5000,-5000,0, 0,0,6000, -5000,0,6000, 0,-3000,5000, -5000,-3000,5000
- N, 3,0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,6000, 5000,0,6000, 0,5000,5000, 5000,5000,5000
- N, 4,0,0,0, 5000,0,0, 0,-5000,0, 5000,-5000,0, 0,0,6000, 5000,0,6000, 0,-3000,5000, 5000,-3000,5000
- N, 5,5000,1000,1000, 10000,1000,1000, 5000,4000,1000, 10000,4000,1000, 5000,1000,7000, 10000,1000,7000, 5000,4000,7000, 10000,4000,7000
- N, 6,10000,1000,1000, 15000,1000,1000, 10000,4000,1000, 15000,4000,1000, 10000,1000,7000, 17000,1000,7000, 10000,4000,7000, 17000,4000,7000

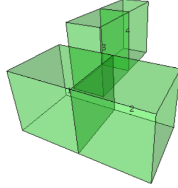
Design_19



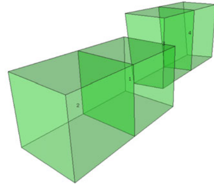
- N, 1, 0,0,0, -5000,0,0, 0,5000,0, -5000,5000,0, 0,-2000,5000, -5000,-2000,5000, 0,7000,5000, -5000,7000,5000
- N, 2, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,-2000,5000, 5000,-2000,5000, 0,7000,5000, 5000,7000,5000
- N, 3,5000,1000,1000, 10000,1000,1000, 5000,4000,1000, 10000,4000,1000, 5000,1000,7000, 10000,1000,7000, 5000,4000,7000, 10000,4000,7000
- N, 4,10000,1000,1000, 15000,1000,1000, 10000,4000,1000, 15000,4000,1000, 10000,1000,7000, 15000,1000,7000, 10000,4000,7000, 15000,4000,7000

Design_20

- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0,
  0,0,5000, 5000,0,5000, 0,5000,5000,
  5000,5000,5000
- N, 2, 0,0,0, 5000,0,0, 0,-5000,0, 5000,-5000,0,
  0,0,5000, 5000,0,5000, 0,-5000,5000,
  5000,-5000,5000
- N, 3, 5000,1000,1000, 10000,1000,1000,
  5000,4000,1000, 10000,4000,1000,
  5000,1000,7000, 10000,1000,7000,
  5000,4000,7000, 10000,4000,7000
- N, 4, 10000,1000,1000, 15000,1000,1000,
  10000,4000,1000, 15000,4000,1000,
  10000,1000,7000, 15000,1000,7000,
  10000,4000,7000, 15000,4000,7000

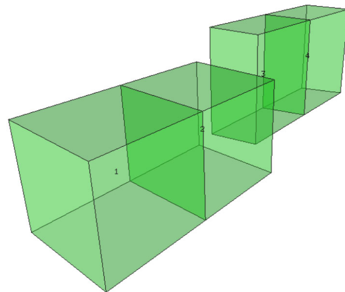


Design_21



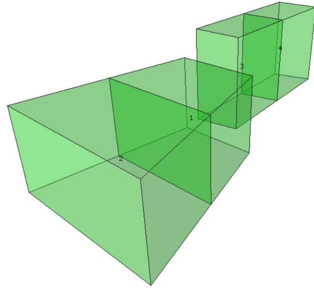
- N, 1, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0,
  0,0,5000, 5000,0,5000, 0,5000,5000,
  5000,5000,5000
- N, 2, 0,0,0, -5000,0,0, 0,5000,0, -5000,5000,0,
  0,0,5000, -5000,0,5000, 0,5000,5000,
  -5000,5000,5000
- N, 3, 5000,1000,1300, 10000,1000,1300, 5000,4000,1000,
  10000,4000,1000, 5000,1000,7000, 10000,1000,7000,
  5000,4800,7000, 10000,4800,7000
- N, 4, 10000,1000,1300, 15000,1000,1300, 10000,4000,1000,
  15000,4000,1000, 10000,1000,7000, 15000,1000,7000,
  10000,4800,7000, 15000,4800,7000

Design_22



- N, 1, 0,0,0, -5000,0,0, 0,5000,0,
  -5000,5000,0, 0,0,5000, -5000,0,5000,
  0,5000,5000, -5000,5000,5000
- N, 2, 0,0,0, 5000,0,0, 0,5000,0, 5000,5000,0, 0,0,5000,
  5000,0,5000, 0,5000,5000, 5000,5000,5000
- N, 3, 5000,1000,1300, 10000,1000,1300, 5000,4000,1000,
  10000,4000,1000, 5000,1000,7000, 10000,1000,7000,
  5000,4000,7000, 10000,4000,7000
- N, 4, 10000,1000,1300, 15000,1000,1300, 10000,4000,1000,
  15000,4000,1000, 10000,1000,7000, 15000,1000,7000,
  10000,4000,7000, 15000,4000,7000

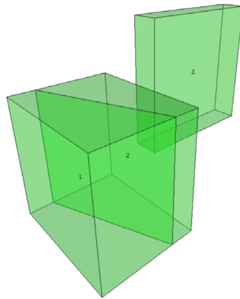
Design_23



- N, 1, 0,-1000,0, 5000,0,0, 0,6000,0, 5000,5000,0, 0,-1000,5000, 5000,0,5000, 0,6000,5000, 5000,5000,5000
- N, 2, 0,-1000,0, -5000,-2000,0, 0,6000,0, -5000,7000,0, 0,-1000,5000, -5000,-2000,5000, 0,6000,5000, -5000,7000,5000
- N, 3, 5000,1000,1300, 10000,1000,1300, 5000,4000,1000, 10000,4000,1000, 5000,1000,7000, 10000,1000,7000, 5000,4000,7000, 10000,4000,7000
- N, 4, 10000,1000,1300, 15000,1000,1300, 10000,4000,1000, 15000,4000,1000, 10000,1000,7000, 15000,1000,7000, 10000,4000,7000, 15000,4000,7000

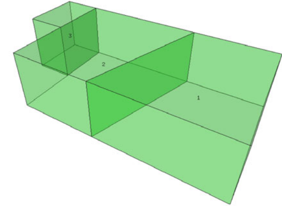
Design_33

- N, 1, 4000,0,0, 1000,0,0, 2000,5000,0, 1000,5000,0, 4000,0,5000, 1000,0,5000, 2000,5000,5000, 1000,5000,5000
- N, 2, 4000,0,0, 5000,0,0, 2000,5000,0, 5000,5000,0, 4000,0,5000, 5000,0,5000, 2000,5000,5000, 5000,5000,5000
- N, 3, 5000,2000,2500, 10000,2000,2500, 5000,3000,2500, 10000,3000,2500, 5000,2000,7500, 10000,2000,7500, 5000,3000,7500, 10000,3000,7500



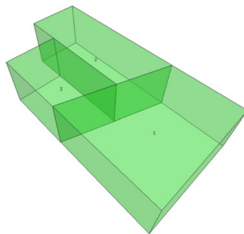
Design_34

- N, 1, 0,0,0, 6000,0,0, 6000,4000,0, 0,6000,0, 0,0,3000, 6000,0,3000, 6000,4000,3000, 0,6000,3000
- N, 2, 0,6000,0, 6000,4000,0, 6000,10000,0, 0,10000,0, 0,6000,3000, 6000,4000,3000, 6000,10000,3000, 0,10000,3000
- N, 3, 3000,10000,0, 6000,10000,0, 6000,12000,0, 3000,12000,0, 3000,10000,3000, 6000,10000,3000, 6000,12000,3000, 3000,12000,3000

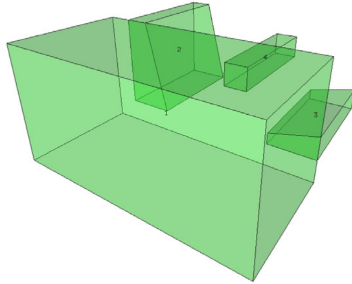


Design_35

- N, 1, 0,0,0, 6000,0,0, 6000,4000,0, 0,6000,0, 0,0,3000, 6000,0,3000, 6000,4000,3000, 0,6000,3000
- N, 2, 3000,5000,0, 6000,4000,0, 6000,12000,0, 3000,12000,0, 3000,5000,3000, 6000,4000,3000, 6000,12000,3000, 3000,12000,3000
- N, 3, 3000,5000,0, 0,6000,0, 0,10000,0, 3000,10000,0, 3000,5000,3000, 0,6000,3000, 0,10000,3000, 3000,10000,3000



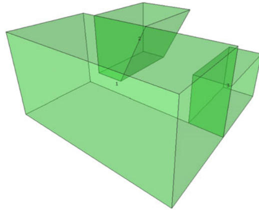
Design_37



- N, 1,0,0,0, 5000,0,0, 0,10000,0, 5000,10000,0, 0,0,5000, 5000,0,5000, 0,10000,5000, 5000,10000,5000
- N, 2,5000,7000,1000, 10000,7000,1000, 5000,9000,1000, 10000,9000,1000, 5000,8000,5000, 10000,8000,5000, 5000,9000,5000, 10000,9000,5000
- N, 3,5000,0,1000, 10000,0,1000, 5000,2000,1000, 10000,2000,1000, 5000,0,2000, 10000,0,2000, 5000,2000,1500, 10000,2000,1500
- N, 4,5000,3000,3000, 10000,3000,3000, 5000,4000,3000, 10000,4000,3000, 5000,3000,4000, 10000,3000,4000, 5000,4000,4000, 10000,4000,4000

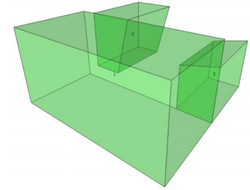
Design_38

- N, 1, 0,0,0, 5000,0,0, 0,10000,0, 5000,10000,0, 0,0,5000, 5000,0,5000, 0,10000,5000, 5000,10000,5000
- N, 2, 5000,7000,1000, 10000,7000,1000, 5000,9000,1000, 10000,9000,1000, 5000,5000,5000, 10000,5000,5000, 5000,9000,5000, 10000,9000,5000
- N, 3, 5000,0,0, 10000,0,0, 5000,2000,0, 10000,2000,0, 5000,0,3000, 10000,0,3000, 5000,2000,3000, 10000,2000,3000

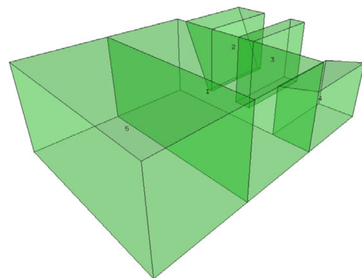


Design_39

- N, 1, 0,0,0, 5000,0,0, 0,10000,0, 5000,10000,0, 0,0,5000, 5000,0,5000, 0,10000,5000, 5000,10000,5000
- N, 2, 5000,7000,1000, 10000,7000,1000, 5000,9000,1000, 10000,9000,1000, 5000,6000,5000, 10000,6000,5000, 5000,9000,5000, 10000,9000,5000
- N, 3, 5000,0,0, 10000,0,0, 5000,2250,0, 10000,2250,0, 5000,0,3500, 10000,0,3500, 5000,2250,3000, 10000,2250,3000

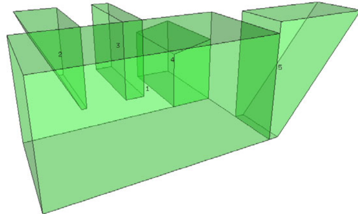


Design_40



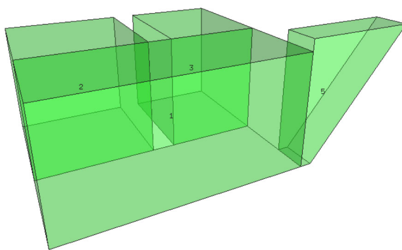
- N, 1,0,0,0, 5000,0,0, 0,10000,0, 5000,10000,0, 0,0,5000, 5000,0,5000, 0,10000,5000, 5000,10000,5000
- N, 2,5000,7000,1000, 10000,7000,1000, 5000,7250,1000, 10000,7250,1000, 5000,7000,5000, 10000,7000,5000, 5000,9250,5000, 10000,9250,5000
- N, 3,5000,4000,1000, 10000,4000,1000, 5000,5000,1000, 10000,5000,1000, 5000,4000,5000, 10000,4000,5000, 5000,5000,5000, 10000,5000,5000
- N, 4,5000,0,0, 10000,0,0, 5000,2250,0, 10000,2250,0, 5000,0,3500, 10000,0,3500, 5000,2250,3000, 10000,2250,3000
- N, 5,0,0,0, -5000,0,0, 0,10000,0, -5000,10000,0, 0,0,5000, - 5000,0,5000, 0,10000,5000, -5000,10000,5000

Design_41



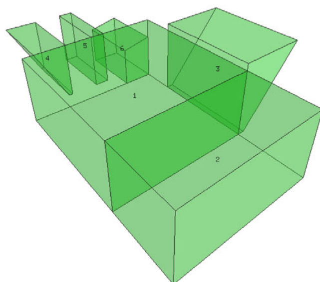
- N, 1,0,0,0, 5000,0,0, 0,10000,0, 5000,10000,0, 0,0,5000, 5000,0,5000, 0,10000,5000, 5000,10000,5000
- N, 2,5000,7000,1000, 10000,7000,1000, 5000,7250,1000, 10000,7250,1000, 5000,7000,5000, 10000,7000,5000, 5000,9250,5000, 10000,9250,5000
- N, 3,5000,4000,1000, 10000,4000,1000, 5000,5000,1000, 10000,5000,1000, 5000,4000,5000, 10000,4000,5000, 5000,5000,5000, 10000,5000,5000
- N, 4,5000,0,0, 10000,0,0, 5000,2250,0, 10000,2250,0, 5000,0,3500, 10000,0,3500, 5000,2250,3000, 10000,2250,3000
- N, 5,0,0,0, 2500,0,0, 2500,-500,0, 0,-500,0, 0,0,5500, 2500,0,5500, 2500,-5000,5500, 0,-5000,5500

Design 42



- N, 1,1000,0,0, 5000,0,0, 1000,10000,0, 5000,10000,0, 1000,0,5000, 5000,0,5000, 1000,10000,5000, 5000,10000,5000
- N, 2,5000,10000,0, 10000,10000,0, 5000,5000,0, 10000,5000,0, 5000,10000,5000, 10000,10000,5000, 5000,5000,5000, 10000,5000,5000
- N, 3,5000,4000,0, 10000,4000,0, 5000,0,0, 10000,0,0, 5000,4000,5000, 10000,4000,5000, 5000,0,5000, 10000,0,5000
- N, 5,1000,0,0, 2500,0,0, 2500,-500,0,1000,-500,0, 1000,0,5500, 2500,0,5500, 2500,-5000,5500, 1000,-5000,5500

Design_43



C Geometry check errors

Within the geometry conformal methods is there currently still some errors occurring due to tolerance issues and in some specific cases wrongly performed geometry checks. Some of these encountered errors during the quad-hexahedron method are listed below.

Error 1: The geometry check for checking if a vertex in on a plane (isInside()) found that the vertex (5000,7000,1000) is located on the following plane: (0,0,0); (5000,0,0); (0,10000,0);(5000,10000,0). However, this plane is on the z=0 plane and the vertex is on the z=1000 plane.

Error 2: The geometry check (isCoPlannar()) has difficulty dealing with a polygon containing the vertex (0,0,0). Specifically, the difficulty exist due the the generation of two vectors to perform the check with. These vectors are generated, in this case, by a vertex minus the (0,0,0) vertex, therefore resulting in an invalid vector for the check performed.

Error 3: The inserted tolerance of 0.1 was not well accounted for in the line-polygon check since it was found that an line intersection vertex is intersecting a polygon in the case of figure 32.

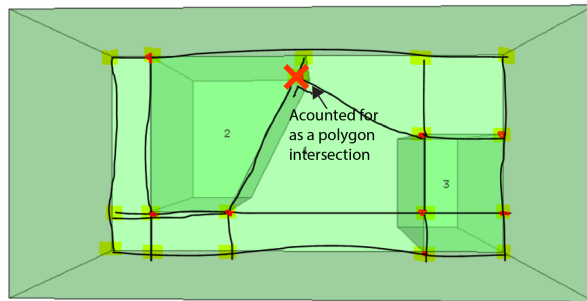


Figure 32: line intersection vertex is intersecting with a polygon according to the line-polygon check

Error 4: An intersection vertex (0,6235.28,4052.82) is found with a tolerance of 0,01, while this intersection vertex is the corner vertex of a cell. See figure 33 for the console output. Thereby, the iteration will not convert.

```
The cuboid mCFCuboids[k] being split: {{5000,2000,3000},{5000,6235.29,4058.82},{5000,7000,1000},{5000,2000,1000},{0,2
000,3000},{0,6235.29,4058.82},{0,7000,1000},{0,2000,1000}}
IntersectionPolygon punten
isFound is added: {5000,6235.28,4058.82}
isFound is added: {5000,2000,3000}

The values within cornerpoints after intersection poly adds to cornerpoints are:
{5000,6235.28,4058.82}{5000,2000,3000}{5000,2000,1000}{5000,6235.28,4058.82}{5000,2000,3000}

The values within cornerpoints just before cuboid generation are:
{5000,6235.28,4058.82}{5000,2000,3000}{5000,2000,1000}{5000,6235.28,4058.82}{5000,2000,3000}{5000,6235.28,1000}{5000,
6235.28,1000}{5000,2000,1000}
terminate called after throwing an instance of 'std::runtime_error'
  what():
Error, There are duplicates inside the to generate quad-hexahedron cornerPoints.
In case the inserted tolerance is 0.1 mm, then the polygon intersection check can mistake a line
for a polygon intersection. Therefore performing the wrong split case resulting in duplicate value'sSolution, make to
lerance smaller
(bso/spatial_design/conformal/cf_cuboid).

Aborted (core dumped)
tessa@vrt-9:~/Tessa code/split/Demo case$
```

Figure 33: Corner vertex found as intersection

Error 5: It has been observed that when a tolerance of 0,001 is accounted for that, in some cases, the iteration continues on without ending. Such a case can be observed in the console output in figure 34 where the iteration was manually terminated after t=4929 iterations.

```
t is:4929
The size of mCFCuboids is: 12
mCFCuboids[k] is: {{5000,2000,3000},{5000,2000,5000},{5000,6000,5000},{5000,6235.29,4058.82},{0,2000,3000},{0,2000,50
00},{0,6000,5000},{0,6235.29,4058.82}}
The Poly splitVertexen are:
{5000,6235.3,4058.82}
The Poly splitVertexen after remove duplicates are:
{5000,6235.3,4058.82}
The cuboid mCFCuboids[k] being split: {{5000,2000,3000},{5000,2000,5000},{5000,6000,5000},{5000,6235.29,4058.82},{0,2
000,3000},{0,2000,5000},{0,6000,5000},{0,6235.29,4058.82}}
```

Iteration amount

Figure 34: Non-converting iteration mechanism