Ingenieurfakultät Bau Geo Umwelt
Lehrstuhl für Computergestützte Modellierung und Simulation
Prof. Dr.-Ing. André Borrmann

Implementation of Geometry Representations for Infrastructure

Carlos Mauricio Platteau

Bachelorthesis
für den Bachelor of Science Studiengang Umweltingenieurwesen

Autor:	Carlos Mauricio Platteau
Matrikelnummer:	[redacted]
Betreuer:	Prof. Dr.-Ing. André Borrmann
	Štefan Jaud
Ausgabedatum:	01. Januar 2020
Abgabedatum:	15. September 2020

Abstract

Due to the high demand in architecture, engineering and construction industry for standardization in infrastructure projects, the IFC-Bridge, -Road, -Rail projects were launched. This thesis presents the implementation of the entity *IfcSectionedSolidHorizontal*, introduced with the IFC 4x1 (IFC-Bridge) version of the Industry Foundation Classes standards. This entity is implemented in the IFC 4x3_RC1 format to support the IFC-Road project in the workflow of the open source BIM visualization software being developed at the Technical University of Munich (TUM) called OpenInfraPlatform (OIP).
As such, the geometry converter of the OIP was expanded to interpret the *IfcSectioned-SolidHorizontal*. This entity is defined by the sweeping operation of multiple CrossSections(profiles) along the *IfcAlignmentCurve* which is very useful for supporting bridge and roadway geometry.
The visualization of the example files demostrated that the implementation works for the line segment of the alignment curve when the *CrossSections* all have the same number of points. Despite this, the algorithm should still be further improved to work correctly for *CrossSections* that have a variable number of profile points and also for the transitions between the curve segments and the line segments.

Zusammenfassung

Aufgrund der hohen Nachfrage der Architecture, Engineering and Construction (AEC) Industrie nach einer Lösung für Standardisierung bei Infrastrukturprojekten wurden die Projekte IFC-Bridge, -Road und -Rail gestartet. Diese Arbeit beschreibt die Implementierung der in der Version IFC 4x1 (IFC-Bridge) neu definierten Entität *IfcSectionedSolidHorizontal* in das Workflow der von der TUM entwickelten Open Source BIM Visualisierungssoftware. Diese Software trägt den Namen OIP. Die Implementierung erfolgte unter dem IFC 4x3_RC1 Standard, um das IFC-Road Format zu unterstützen.
Für die Implementierung von *IfcSectionedSolidHorizontal* wurde der Geometriekonverter der OIP erweitert, um die Geometrie interpretieren zu können. Diese neue Entität wird anhand einer Extrusion von mehreren Querschnittsprofilen(CrossSections) entlang einer Kurve (*IfcAlignmentCurve*) beschrieben und ist daher sehr nützlich, um sowohl Straßen als auch Brücken Geometrie zu beschreiben.
Durch die Visualisierung der Beispieldateien wurde festgestellt, dass die Extrusion von den Profilen entlang den Linien Segmenten der *IfcAlignmentCurve* funktioniert, wenn alle Querschnittsprofile die gleiche Anzahl an Punkten haben. Für die Extrusion von Profilen mit verschiedenen Anzahl an Profil-Punkten muss der Algorithmus noch verändert werden sowie für die Extrusion an den Übergängen von Kurvensegmenten zu Liniensegmenten.

Contents

Abbrevations

AEC	architecture, engineering and construction
BCP	BasisCurvePoints
BIM	Building Information Modeling
bSI	buildingSMART International
CAD	computer-aided design
CMS	Chair of Computational Modeling and Simulation
CS	coordinate system
CSG	Construtive solid geometry
CSP	CrossSectionPoints
FM	facility management
IAI	International Alliance of interoperability
ISO	Intenational Organization for Standardization
IFC	Industry Foundation Classes
OIP	OpenInfraPlatform
OKSTRA	Objekt Katalog Straße
STEP	Standard for the Exchange of Product model data
TUM	Technical University of Munich
2D	two-dimensional
3D	three-dimensional

Chapter 1

Introduction

The world is shifting towards a more digitalized approach in the architecture, engineering and construction (AEC) industry. With the steadily increasing demand for digitalization, Building Information Modeling (BIM) has begun replacing conventional computer-aided design (CAD). BIM takes the planning of buildings and infrastructure from a two-dimensional (2D) point of view to a three-dimensional (3D) one, with BIM also adding the semantic information to the represented geometry [19]. Furthermore BIM could lead the AEC industry to a paper-free, and hence less wasteful future [18, 19].

The digital models and data exchanged between the different stakeholders of a given project are of vital importance, and have a substantial impact on its efficiency and long-term success. Since there are a variety of data formats and software available, the interoperability between project partners must be ensured. Industry Foundation Classes (IFC) is one of the most established open data models [8]; it is currently being developed by the non profit organization buildingSMART International (bSI) and was admitted into the ISO standards in 2013[2].

Due to the fact that most of the visualization software for BIM models do not support all IFC versions the Chair of Computational Modeling and Simulation (CMS), open-source software is being developed at the Technical University of Munich (TUM) named TUM OpenInfraPlatform (OIP) to overcome this compatibility issue [4]. In addition to being open-source TUM OIP is also vendor-neutral and bases its data model on IFC standards [18, 2].

1.1 Motivation

While all IFC versions before IFC 4 primarily focused on buildings and as such where made to support strucural engineering applications, the new versions intend to provide extensions to make the description of bridges, roads and tunnels possible [3].

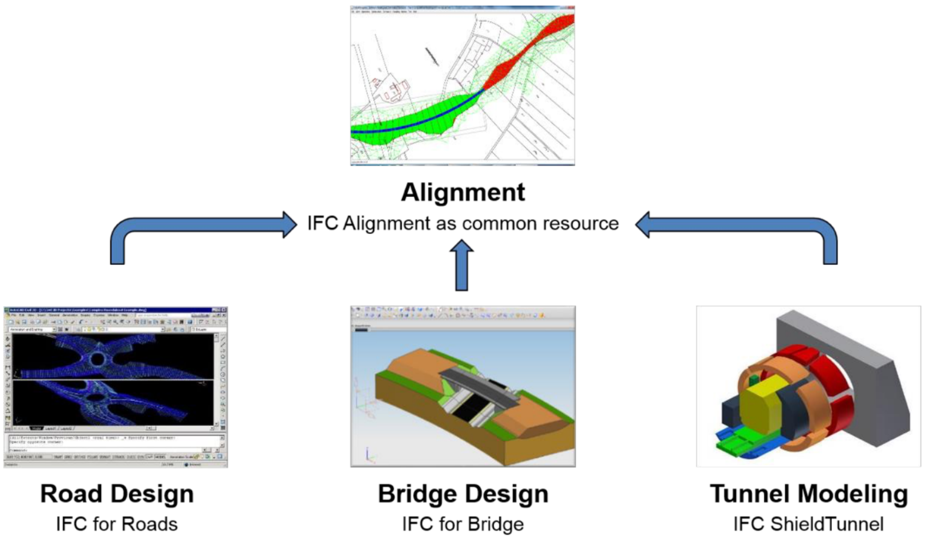


Figure 1.1: IFC Alignment as basis for the development of IFC-Bridge, IFC-Road and IFC-Tunnel projects. Retrieved from Amann et al.[3].

In 2013 buildingSMART International founded the "Infra Room" to start developing extensions for infrastructure [8]. With the completion of the IFC-Alignment project , the extension to describe an alignment for infrastructure was made possible. Due to stakeholder's high demand regarding extensions pertaining to infrastructure and ones based on the IFC-Alignment, the IFC Infra Overall Architecture project was commenced [6].

In addition to the two completed IFC projects resulting in the release of the IFC4x1 which provided support for linear civil engineering structures [21], it also laid the foundations for the new IfcRoad, Ifc-Rail, IfcBridge and IfcTunnel projects. As shown in Figure 1.1 the projects IFC-Roads, -Bridge and -Tunnel all share the alignment as a common ressource. The alignment will later on be explained in more detail later on in Chapter 2.

1.2 Related Work

Since the 1980s there has been a great deal of interest for the data exchange between different CAD systems. The Standard for the Exchange of Product model data (STEP) was proposed out of a need for standardization [7]. Moreover the process was very bureaucratic and was not advancing at the speed required by the AEC industry, and the International Alliance of interoperability (IAI) was consequently founded in 1995 by construction, engineering and software manufacturers such as Autodesk.

As shown in Figure 1.2, the first version was introduced in 1997 under the name of "IFC" as version 1.0, with the intention to improve the interoperability of BIM. One year later, in

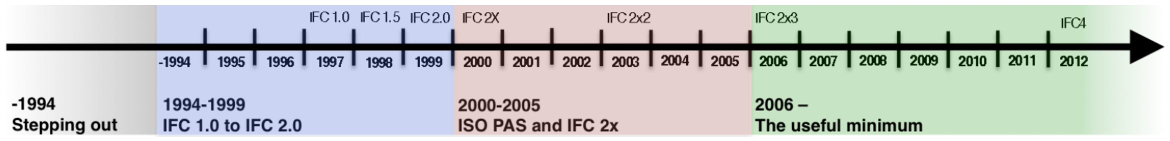


Figure 1.2: IFC timeline. Retrieved from Laakso et al. [20]

1998, IFC 1.5 was released, simultaneously being the first version to be applied to construction software [20, 7].

With the introduction of IFC 2x in the year 2000, the concept of a *platform* was introduced into the context of IFC. In addition to the platform the idea of two extension models within the framework was also developed under the name of ST-4. This extension included structural analysis as well as steel constructions [23].

After the rebranding of the International Alliance of interoperability (IAI) in 2005 as the non-profit organization buildingSMART International, the next big release was introduced: IFC 2x3. This new version included new classes (such as *IfcBuildingElemntComponent*) to express profile properties as well as structural members and matereials[24]. IFC 2x3 was the most widespread version of Industry Foundation Classes until 2013, when it slowly began to be replaced by IFC 4 [7].

The next major release was IFC. IFC 4's novel approach consisted of putting "quality over speed" to achieve the goal of gaining a full ISO standard status. In 2013 IFC was finally admitted as an ISO standard [2]. As a result many countries have adopted it as an obligatory data exchange format in the AEC industry [7].

As previously mentioned in Section 1.1, all versions prior to IFC4 were focused on buildings, but in 2013 the founding of a subdivision from buildingSMART International called Infra Room changed the landscape. The Infra Room subsequently started working on extensions necessary for infrastructure projects[8]. The IFC-Alignment project in 2015 and the IFC Infra Overall Architecture which ended in 2017, laid the theoretical foundations with the alignment curve on which future infrastructure projects would build upon as mentioned in Section (cf.1.1) [3, 6].

Since the IFC4x1 release in 2018 and the IfcBridge project completion in 2019 [8], the extensions now support the modelling of bridges. Finally, the latest release is the IFC 4x3RC1 in 2020. The ongoing projects of IfcBridge, IfcRoad and IfcRail build upon the next major release IFC 5 [16]. Other Future projects include IfcTunnel and Ifc Ports and Waterways.

1.3 Problem Statement

The purpose of this thesis is the expansion of the TUM OpenInfraPlatform geometry converter so that it can support newly introduced geometric entities such as *IfcSectionedSolid-Horizontal* an entity that was introduced in IFC 4x1. To achieve this, these new entities must to be implemented in the workflow of OIP as well as several other supporting entities (*IfcDistanceExpression, IfcProfileDef*). Furthermore, example files shall be produced to test the new code.

1.4 Objective

In order to test and implement *IfcSectionedSolidHorizontal* in the workflow of OIP the objectives of this thesis are outlined by the following:

1. Development of an algorithm to interpret the attributes of this new geometric entity.
2. Development of an algorithm to calculate the geometry in every profile and point along the alignment curve.
3. Production of examples that build upon each other in matters of complexity.
4. Production of an example that is flawed and will not work for this implementation in order to demonstrate the limits of this proposed implementation.

1.5 Thesis Structure

This thesis is structured into the following seven chapters; Chapter 1 is a short introduction of BIM and IFC as well as a recapitulation of its chronology. Chapter 2 elaborates the theoretical framework behind the Industry Foundation Classes, new entities and attributes. The method for the development of the implementation for new geometry representation is addressed in Chapter 3. The implementation is described in Chapter 4. In Chapter 5 the results of the implementation are presented and subsequently discussed in Chapter 6. Lastly Chapter 7 concludes this thesis with a summary of the findings and serves to incite further research into this field (Section 7.2).

Chapter 2

Theoretical Framework

To understand the method and implementation of the IFC entities into the OpenInfraPlatform workflow, the theoretical framework must first of all be addressed to gain a deeper insight. This begins with Section 2.1, in which the IFC is explained in greater depth. Furthermore, the entities and attributes of *IfcAlignment* and *IfcSectionedSolidHorizontal* are described. The second section presents an overview of theTUM OpenInfraPlatform project. Finally, Section 2.3 elaborates the mathematical foundations needed for the geometrical interpretation of *IfcSectionedSolidHorizontal*.

2.1 Industry Foundation Classes (IFC)

As previously mentioned in Chapter 1, the Industry Foundation Classes (IFC) is a vendor neutral open data model standard [2]. It has been in development since 1995 by the former International Alliance of interoperability now known as buildingSMART International (Chapter 1.2), in order to improve the interoperability between different BIM software from separate vendors[20]. Since BIM can be employed in a variety of development stages, from the designing of a building through the construction and later on in the facility management (FM), there are wide range of stakeholders involved in its lifecycle [7]. Moreover a study conducted by Gallaher et al. in 2002 in the USA showed that the lack of interoperability among the software tools and redundant data accounts for 15.8 billion USD in losses per year [17]. This is approximately an 1-2% of the AEC industry's dividend [17, 20].

With the release of IFC 4x1 new entities were introduced to help support geometry for bridges decks and then the IFC 4x3 roadways, which are specifically relevant to this thesis. One of the new entities is the *IfcSectionedSolidHorizontal* which will be described in Section 2.1.2. However, before getting into the details of this entity another one must first be explained: *IfcAlignmentCurve* since its representation is vital for *IfcSectionedSolidHorizontal* [6].

2.1.1 IfcAlignmentCurve

The *IfcAlignmentCurve* has the attributes Horizontal, Vertical, Tag as well as the attributes inherited from the *IfcBoundedCurve* and *IfcCurve*. It can be divided into *IfcAlignment2DHorizontal* (which represents the horizontal Alignment) and *IfcAlignment2DVertical* (which represents the vertical Alignment) [12, 15].

The horizontal alignment is defined in the X-Y plane and has the attribute CurveGeometry which can be of type *IfcLineSegment2D* if the curve segment is a line, *IfcCircularArcSegment2D* for a circular segment ot *IfcCltohoidalArcSegment2D* if it is an clothoidal arc segment (used in roads for a transition curve) [10, 15].

On the other hand the *IfcAlignment2Dvertical* is defined by the distance along the z-axis and connects the segments from the beginning to end. In this case the segments can be of type *IfcAlignment2DVerCircularArc*, *IfcAlignment2DVerSegLine* or *IfcAlignment2DVerSegParabolicArc* [11, 15].

2.1.2 IfcSectionedSolidHorizontal

To represent solid geometry the *IfcSolidModel* is required. This entity is an abstract supertype of *IfcCsgSolid*, *IfcManifoldSolidBrep*, *IfcSectionedSolid*, *IfcSweptAreaSolid* and *IfcSweptDiskSolid* [14]. For this thesis, only *IfcSectionedSolid* will be relevant, as it is the supertype of *IfcSectionedSolidHorizontal*.

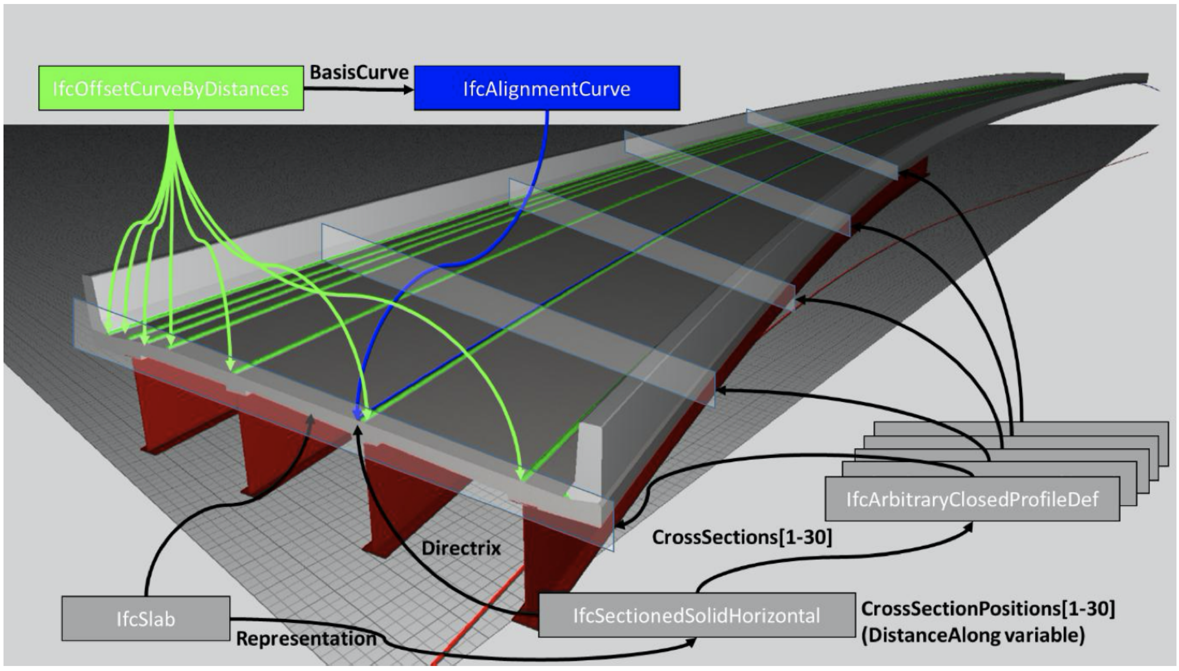

IfcOffsetCurveByDistances
BasisCurve
IfcAlignmentCurve
IfcArbitraryClosedProfileDef
CrossSections[1-30]
Directrix
IfcSlab
Representation
IfcSectionedSolidHorizontal
CrossSectionPositions[1-30]
(DistanceAlong variable)

Figure 2.1: IfcSectionedSolidHorizontal. Retrieved from Borrmann et al. [6]

As it is a subtype *IfcSectionedSolidHorizontal* inherits the attributes from all types above it in the hierarchy. As such, it has the attribute *Dim* (always equals to 3) from *IfcSolidModel* and the attributes *Directrix* and *CrossSections* from the supertype *IfcSectionedSolid*. In addition to these attributes *IfcSectionedSolidHorizontal* also has *CrossSectionPositions* and *FixedAxisVertical*[13]. Figure 2.2 clarifies this relationship with an instance diagram.

Figure 2.1 shows how all entities and attributes are connected to one another. The *CrossSections* attribute is of type *IfcProfileDef* and defines the profiles for each section along the *directrix* curve, which in this case is an *IfcAlignmentCurve*. *CrossSectionPositions* is of type *IfcDistanceExpression* and indicates the position of each paired CrossSection also along the *Directrix*; there are as a consequence just as many *CrossSections* as *CrossSectionPositions*. Lastly, FixedAxisVertical which is of type *IfcBoolean* and may have the values TRUE or FALSE, which indicate the orientation of the *CrossSections*. For FixedAxisVertical TRUE, the Y-axis of the profiles faces in positive Z-direction and for FALSE they are perpendicular to the *directrix* [13].

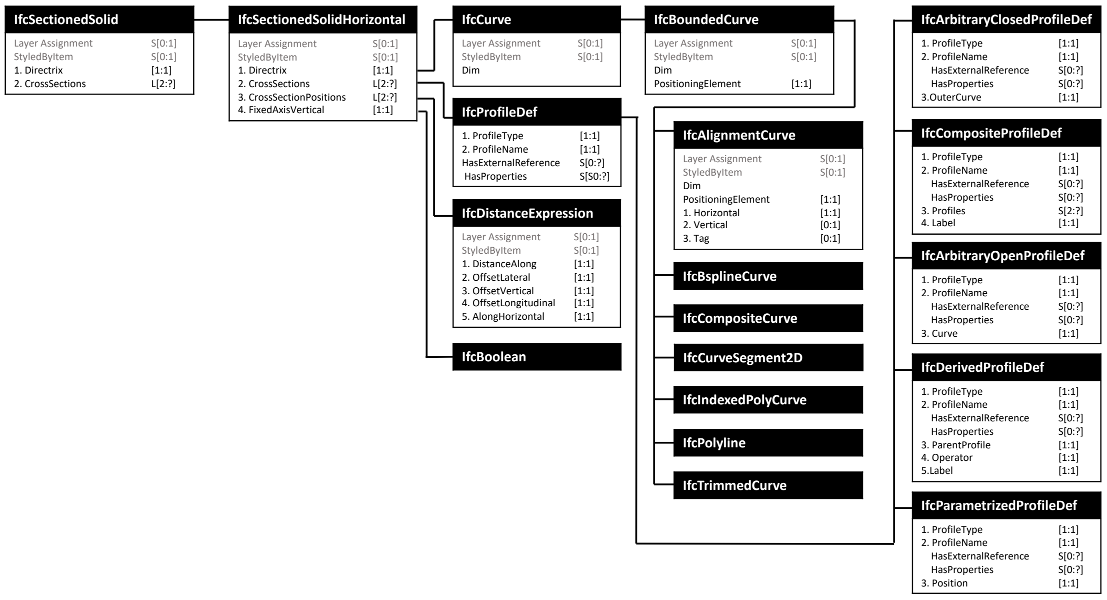


Figure 2.2: *IfcSectionedSolidHorizontal* inheritance diagram. Based on The *ifcAlignment* instance diagram from the IFC4x2 Release.[9]

Since this thesis will use an alignment curve to describe the *Directrix*, only the attributes of the *IfcAlignmentCurve* are relevant and therefore the other subtypes of the *IfcBoundedCurve* in Figure 2.2 will not be discussed in further detail.

2.2 OpenInfraPlatform (OIP)

As mentioned in Chapter 1 TUM OpenInfraPlatform is an open-source, vendor-neutral software package that is capable of visualizing Building Information Modeling models. Its data model is based on the Industry Foundation Classes standard [18, 2]. Furthermore the sofware is written in the programming language C++ and uses CMake to manage the building process of the sofware [18]. The architecture is best explained in the referenced paper by H. Hecht and Š. Jaud (2019) [18].

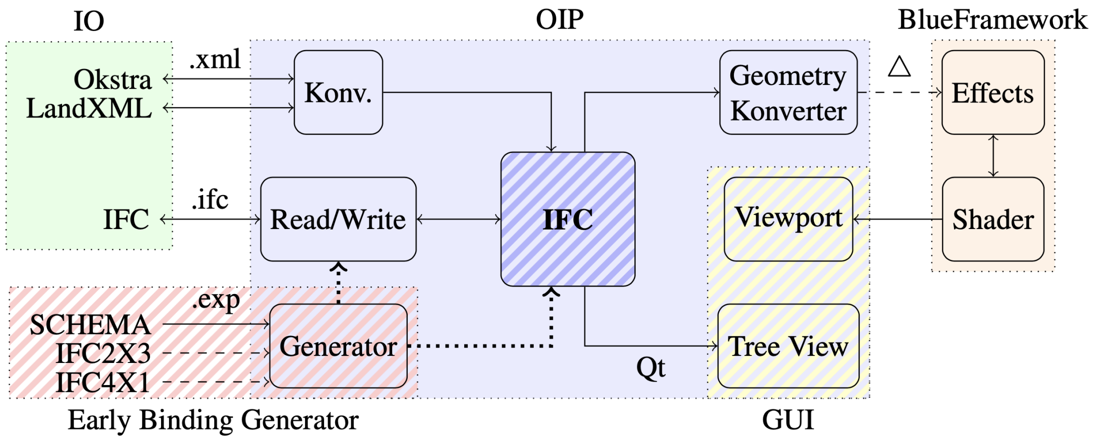


Figure 2.3: Open Infra Platform sofware components. Retrieved from Hecht et al. [18]

Figure 2.3 presents an overview of the main components of this program. If the data imported is in another format such as Objekt Katalog Straße (OKSTRA) or LandXML it will be converted into IFC by the converter (Konv. in Figure 2.3). The Early Binding Generator will model the IFC objects as a C++ library so that each class doesn't have to be implemented manually. This works by mirroring the EXPRESS schema and translating it into C++ [18].

Although Qt has its own rendering engine, the TUM has developed a library named BlueFramework which is 3D rendering engine also written in C++. What are initally abstractabstract geometric representation are converted in into triangules(triangulated) in the Geometry Converter so that the graphics rendering engine can understand and visualize the geometry [18]. This part is important to this thesis since it was necessary to expand the geometry converter in order to understand *IfcSectionedSolidHorizontal*, more on this can be found in Chapter 4.

The OIP can also add other functionalities to process data by adding the appropiate plugin module. These plugins can be selected in CMake before generating the project [18]. It is also worth mentioning that boost is used to supplement and expand the C++ standard library[18].

2.3 Linear Transformations

Linear transformations are used to translate, scale and rotate geometric objects[22]. But before explaining the operations and matrices behind the linear transformations first of all it is important to begin with a look at the notation used in computer graphics to describe 3D transformations and its advantages. Since it impossible to describe a translation in a matrix form for the Cartesian coordinate system (CS) the homogeneous coordinates must to be introduced.

The conversion between Cartesian and homogeneous coordinates adds an extra dimension to the system. For example, a 2D point with coordinates $P_c = (x, y)^T$ would be represented as $P_h = (x \times w, y \times w, w)^T$ in homogeneous coordinates [22]. If there is any scaling factor present w will be unequal to 0. For the purpose of this thesis, w is always equal to 1. This comes with the great advantage that now the translation may now also be written as a matrix and consequently may be multiplied with other matrices to create a single transformation matrix T (see below 2.1) [5].

$$\begin{bmatrix} x' \\ y' \\ w' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = T \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, [5] \tag{2.1}$$

This new transformation matrix T is can be devided into the following parts [5]:

$$\begin{bmatrix} \text{Scaling and rotating} & \text{Translations} \\ \text{0 for the homogeneus representation} & 1 \end{bmatrix}$$

In the next subsections the matrices with homogeneous coordinates behind these so called function blocks will be outlined in more detail.

2.3.1 Scaling

To scale a point $P = (x, y, z, 1)$, the following matrix in 3D with homogeneous coordinates is used:

$$S = \begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.2}$$

with scaling factors $a, b, c, d \in \mathbb{R}$

The equation for the scaling would be $P' = P \times S$ such that $P' = (x \times a, y \times b, z \times c, 1)$ [22].

2.3.2 Translation

A translation moves the point P along the x,y and z axes. For the translation the following matrix is used:

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{pmatrix} \tag{2.3}$$

When multiplied with the matrix 2.3 the point P is moved to the coordinates $P' = (x + dx, y + dy, z + dz, 1)$ [22].

2.3.3 Rotation

In the case of the rotations there are three rotations possible in a 3D space, R_x for a rotation around the x-axis, R_y for a rotation around the y-axis and R_z for the rotation around the z-axis (see below 2.4) [22, 5].

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad R_y = \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.4}$$

$$R_z = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

If necessary, more than one transformation can be applied to the same point P, in this case the transformation matrices may be mulitplied with each other to create a new transformation matrix and then multiply that last one with the point.

$$T = T_1 \times T_2 \times T_3 ..... \times T_n \tag{2.5}$$

with $T \in \mathbb{R}^{4\times4}$ for 3D [5, 22].

Now the T from the equation 2.5 can be multiplied with the point as followed:

$$P' = P \times T$$

2.4 Helmert Transformation

The Helmert transformation or seven-parameter Helmert transformation is often used in goedesy[1] to convert coordinates from one right-handed CS to another right-handed CS in $\mathbb{R}^3$ [19].

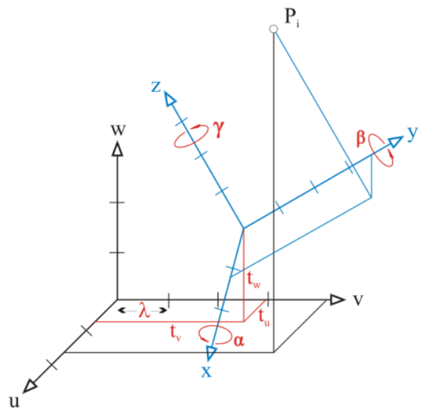


Figure 2.4: Seven-parameter Helmert transformation of the point P_i (see equation 2.6) with (x,y,z) as origin CS and (u,v,w) as the new CS with the parameters $(t_u, t_v, t_w, \lambda, \alpha, \beta, \gamma)$. Retrieved from Jaud et al.[19]

To perform this transformation three translations and three rotations are needed as well as a scaling factor. In this case λ is the scaling factor. For the purposes of this thesis, the scaling factor will remain $\lambda = 1$. This means that no scaling is required. For 3D three translations are possible (one for each axis of the CS). As for the rotations, one around each axis of the CS is described by the matrix R below (2.7).

[1]" *The science of the measurement and mapping of the earth's surface*" [5]

$$\begin{bmatrix} u_i \\ v_i \\ w_i \end{bmatrix} = \begin{bmatrix} t_u \\ t_v \\ t_w \end{bmatrix} + \lambda R(\alpha, \beta, \gamma) \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}, [19] \tag{2.6}$$

in which the Matrix R is:

$$\begin{bmatrix} \cos\gamma\cos\beta & \cos\gamma\sin\beta\sin\alpha + \sin\gamma\sin\alpha & -\cos\gamma\sin\beta\cos\alpha + \sin\gamma\sin\alpha \\ -\sin\gamma\cos\beta & -\sin\gamma\sin\beta\sin\alpha + \cos\gamma\cos\alpha & \sin\gamma\sin\beta\cos\alpha + \cos\gamma\sin\alpha \\ \sin\beta & -\cos\beta\sin\alpha & \cos\beta\cos\alpha \end{bmatrix}, [19] \tag{2.7}$$

To express the point $P_i = (x_i, y_i, z_i)^T$ shown in Figure 2.4 in the new CS as $P_i = (u_i, v_i, w_i)^T$ the Helmert transformation shown in equation 2.6 has to be performed [19].

Chapter 3

Method

This chapter presents the informal propositions (Section 3.1) for the implementation of *IfcSectionedSolidHorizontal* in Chapter 4. Section 3.2 gives some insight into the design of the implementation and therefore will list the steps needed to implement the entity. The last Section 3.3 refers to the method used to test the implementation.

3.1 Informal Propositions

The scope of the implementation is limited by the following conditions:

1. Each CrossSection must have the same number of points in it as the next CrossSection, so that the interpolation works correctly.
2. The directrix will be represented by an *IfcAlignmentCurve*.
3. Only profiles understood by the OIP profile converter can be visualized.

3.2 Design

The design of the implementation of *IfcSectionedSolidHorizontal* can be divided into different steps or methods that will lead to the visualization of this entity in an IFC 4x3 (IFC-Road) format.

1. The first step is to interpret the directrix attribute by calculating points along the curve to which the profiles can later be attached in order to create a sweeping geometry along the *IfcAlignmentCurve*.

2. In the second step the information of each *IfcProfileDef* is interpreted and saved.
3. Since the profile information is in 2D, the position along the curve as well as its rotations must be calculated such that every profile point can then be transformated into the curve CS and placed in the correct 3D position.
4. Next, the two lists(one with the curve points and the other with the points where the *CrossSections* are defined) need to be merged into one ordered list, from the first CrossSection to the last CrossSection on the *IfcAlignmentCurve*.
5. Step 5 is to calculate the profile for the curve points, since the profile is defined for the *CrossSections* and not for each curve point.
6. Once the list from the step 4. is completed with the direction and profile for each point on the curve, the vertices must be defined and converted from the CS of the curve into the global CS.
7. The last step is to conect the vertices of each profile with those of the next profile building a triangle mesh that represents the body of the object until it is a closed one.

These steps will be explained in more detail in the following Chapter 4 as well as the C++ code behind them.

3.3 Testing

The method used for testing the implementation will be based on the visualization of examples. As stated in the first chapter (Section 1.4) the examples will begin with a low degree of complexity and slowly build upon each other to create more complex scenarios. For example, the first example could be extruded along a straight line so there is no need for interpolation. Then the next could have two different geometries extruded along the same line. The more complex examples are then extruded along a curve (*IfcAlignmentCurve*) so that the interpolation may also be tested.

Chapter 4

Implementation

Chapter 4 describes the implementation of *IfcSectionedSolidHorizontal*. Therefore, Section 4.1 starts by presenting the helper functions used in the code. Then in Section 4.2 the steps stated in Chapter 3.2 are explained with more detail as well as the algorithms behind them.

4.1 Helper functions

Altough the implementation of *IfcSectionedSolidHorizontal* is written in the solid model converter, some of the functions needed to interpret the geometry correctly can be found in other converters inside of the OIP's geometry converter. For example in the CurveConverter, PlacementConverter or ProfileConverter.

getStationsForTessellationOfIfcAlignmentCurve

This helper function can be found in the CurveConverter and takes the *IfcAlignmentCurve* as input. It splits the curve into different sections that are important for the correct visualization of the curve. The output is a vector of points (stations) where the tessellation needs to be calculated.

convertBoundedCurveDistAlongToPoint3D

Since the *IfcAlignmentCurve* is a subtype of the *IfcBoundedCurve* (Chapter 2.1) this function takes the curve, the stations from the helper function before, a boolean, and two 3D vectors (targetPoint3D, targetDirection3D) as input arguments. Then it calculates a point on the curve and the direction of that point for the station and saves this information in the two input vectors. This function is from the PlacementConverter.

convertIfcDistanceExpressionOffsets

Another helper function from the PlacementConverter is the *convertIfcDistanceExpressionOffsets* which takes an *IfcDistanceExpression* like *CrossSectionPositions* (Chapter 2.1) and interprets the lateral, vertical and longitudinal offset attributes. The output is a 3D vector with the offset information.

calculateCurveOrientationMatrix

Also in the PlacementConverter is the function *calculateCurveOrientationMatrix* which takes the direction of a point on the curve and also the attribute FixedAxisVertical from the *IfcSectionedSolidHorizontal* and based on those two parameters then calculates a matrix with the rotation of the *directrix* in that point.

distance

This output of this function is the distance between two points. For this, it calculates the Euclidean norm of the points in 2D or 3D. Therefore the distance is always positive. It can take the two points P_1 and P_2 or the coordinates of the points $(x_1, y_1, z_1, x_2, y_2, z_2)$ as input parameters.

getCoordinates

The function *getCoordinates* is in the ProfileConverter. This function returns "paths" . The ProfileConverter interprets the different *IfcProfileDef*'s and saves the coordinates of each profile in a vector. Since the profiles can have multiple profiles and the profile points need to be ordered from first to last to correctly interpret the profiles later, paths is a vector(vector(vector(2Dvector))). This means the first vector has the information of how many profiles there are in one *IfcProfileDef*, the second vector states the order in which the 2D points need to be conected to form the profile. The last vector saves the 2D position of the points.

carve

Although Carve is not a function, it is a library that provides most of the vector matrix multiplicsations and also vector arithmetics needed for the implementation to work. In addition it automatically converts the coordinates of a vector into homogeneous coordinates (Chapter 2.3). Carve also supports operations between polygonial meshes and is therefore often used in CSG.

4.2 IfcSectionedSolidHorizontal

As stated in the Chapter 3.2, the implementation of *IfcSectionedSolidHorizontal* is based on various steps and methods. The first step in the implementation is to interpret the *directrix* attribute which will be the curve used to perform the sweeping operation.

4.2.1 Directrix

Initially, the *directrix* curve must be split into segments. Therefore the helper function *getStationsForTesselationOfIfcAlignmentCurve* from the CurveConverter must be used. This returns the vector *stations* with the length of each segment saved as a double. The vector is then handed to the next helper function *convertBoundedCurveDistAlongToPoint3D* in the PlacementConverter. This function calculates the point on the *directrix* as well as the direction in that point. Given that *stations* is a vector, this process must be repeated for each element in the vector so that at the end the *directrix* is transformed into a list of points (BasisCurvePoints) along with a list of their respective directions (BasisPointDirection). This method is presented below in algorithm 1.

Algorithm 1 Calculate points and directions on the directrix [1].

```
Input: stations
Output: BasisCurvePoints, BasisPointDirection
1: for each element in stations do
2:     calculate the point on the curve
3:     calculate the direction of the point
4:     save the point and direction in vectors
5: end for                                   ▷ Refer to A.3 to see the C++ code.
```

4.2.2 CrossSections

After splitting the *directrix* into a list of points with their directions, the profiles must then be interpreted. The attribute *CrossSections* contains the information for profiles that will be extruded along the *directrix* (positioned at the BasisCurvePoints). First, the vector *paths* is defined as a vector(vector(vector(2Dvector))). The first vector indicates the CrossSection, the second vector the number of profiles in the CrossSection, the third one indicates the order of the profile points and the final vector contains the 2D coordinates for the point. For each CrossSection one ProfileConverter is needed. The Profile converter then calls the helper function *getCoordinates* and calculates the 2D points for each profile in the corresponding CrossSection. The iteration continues until all *CrossSections* are covered as shown inthe pseudocode below (algorithm 2).

Algorithm 2 Save CrossSections list with 2D coordinates for each profile point [1]

```
Input: vector of CrossSections
Output: paths
1: for $i \leftarrow 0$ to $i <$ size.vector(CrossSection) do
2:     getProfileConverter one for each CrossSection
3:     for $p \leftarrow 0$ to $p <$ size.Profiles in CrossSection) do
4:         getCoordinates of each profile
5:         save each CrossSection as vector of profiles with 2D coordinates
6:     end for
7:     save paths as vector of CrossSections
8: end for                                   ▷ Refer to A.4 to see the C++ code.
```

4.2.3 CrossSectionPositions

The next step is to interpret the *CrossSectionPositions* to enable the profiles to be placed in the correct position along the *IfcAlignmentCurve*. Since *CrossSectionPositions* is of type *IfcDistranceExpression* it has the attributes *DistanceAlong*, *OffsetLateral*, *OffsetVertical*, *OffsetLongitudinal* and the boolean *AlongHorizontal* as stated in Chapter 2.1 in Figue 2.2.

Algorithm 3 Interpret CrossSectionPositions [1]

```
Input: List of CrossSectionPositions
Output: offsets, CrossSectionPoints, directionsOfCurve, object_placement_matrix
1: for $pos \leftarrow 0$ to $pos <$ size.vector(CrossSectionPositions) do
2:     1. Calculate offsets from the curve
3:     2. Calculate the CrossSectionPoint and its direction
4:     3. Interpret FixedAxisVertical
5:     4. Calculate the rotations
6:     5. Calculate the object_placement_matrix
7: end for                                   ▷ Refer to A.5 to see the C++ code.
```

As shown in the algorithm 3, five steps must be performed to interpret the *CrossSectionPositions*:

Step 1: Calculate the offsets from the curve

The helper function *convertIfcDistanceExpression* from the PlacementConverter is called to interpret the lateral, vertical and horizontal offsets. For every CrossSectionPosition one 3D vector is saved in *offsetFromCurve*. *offsetFromCurve* is a vector(3D vector). Every element of the vector *offsetFromCurve* saves the offsets from the respective *CrossSectionPositions* element.

Step 2: Calculate the position and direction of the base curve

In this step, the point on the curve where the CrossSection is to be placed is calculated. The attribute*DistanceAlong* indicates the position from the *CrossSection* based on the distance along the *directrix*. To generate the point in the correct position, the function *CalculatePositionOnAndDirectionOfBaseCurve* is used (Method A.2). This function takes the *directrix* and the *IfcDistanceExpression* as input. In addition to this, it also sets the *relativeDistAlong* to 0 so that the distance is measured from the begining of the *IfcAlignmentCurve*. With the distance to the point now defined, it may be used in *convertBoundedDistAlongToPoint3D* (Section 4.1) to generate the output required output; pointOnCurve and directionOfCurve which corresponds to the point in the curve where the CrossSection will be moved to in the translation (step 5). These two vectors are saved in the vector(3Dvector) *CrossSectionPoints* and directionsOfCurve such that there is exactly one point per CrossSectionPosition.

Step 3: Interpret FixedAxisVertical

Furthermore, should *FixedAxisVertical* be set to true, the z-component of the directionsOfCurve must be corrected to 0. The vector is then normalized.

Step 4: Calculate the rotations

To calculate the rotations for the transformation matrix, the program relies on the PlacementConverter as well as the helper function *calculateCurveOrientationMatrix* which is presented in Section 4.1. This function takes the directionsOfcurve of the point from the CrossSection being analyzed and also the the value of *FixedAxisVertical*, it returns a transformation matrix (*localPlacementMatrix*) which multiplied with the offsets will then move the point to the correct position.

Step 5: Calculate the transformation matrix

Before calculating the final transformation matrix (*object_placement_matrix*) first the translation must first be calculated by performing the following equation 4.1:

$$Translate = CrossSectionPoints[pos] + localPlacementMatrix[pos] \times offsetFromCurve[pos] \tag{4.1}$$

The transformation matrix is a Helmert transformation with a scaling factor of 1. However unlike in Chapter 2.4, the Helmert transformation will be performed here with homogeneous coordinates as in Chapter 2.3, so that the vector translate from the equation 4.1 may be written as part of the transformation Matrix. Hence, the Helmert transformation will only require te multiplying pointOnCurve with the *object_placement_matrix*. The coordinates from the profile in 2D will then be transformed to 3D into the curves coordinate system and be moved to the right place on the curve.

Since this Helmert transformation must be performed for each point of each profile, it is converted into the function *convertCurveOrientation*. This function accepts the following three arguments: directionsOfCurve from the *CrossSectionPositions* at which it is currently positioned in the given iteration; the value from *FixedAxisVertical* and the tranlsate vector also from the *CrossSectionPositions* on which it is currently positioned. It returns a matrix like the one in Chapter 2.3 with four function blocks:

$$\begin{bmatrix} \text{Scaling and rotating} & \text{Translations} \\ \text{0 for the homogeneus representation} & 1 \end{bmatrix}, [5]$$

or in the C++ Code A.1 :

A.1: ConvertCurveOrientation

```
// produce a rotation matrix
return carve::math::Matrix(
local_x.x, local_y.x, local_z.x, translate.x,
local_x.y, local_y.y, local_z.y, translate.y,
local_x.z, local_y.z, local_z.z, translate.z,
0, 0, 0, 1);
```

This matrix performs the 3D rotations as well as the translations from equation 4.1.

The rotations local_x, local_y and local_z are being calculated from the vector directionsOfCurve as follows:

A.1: ConvertCurveOrientation

```
// produce a rotation matrix
carve::geom::vector<3> local_z(carve::geom::VECTOR(directionOfCurve.x,
    directionOfCurve.y, fixed_axis_vertical ? 0.0 : directionOfCurve.z));

// get the perpendicular to the left of the curve in the z-y plane (curve
    's coordinate system)
carve::geom::vector<3> local_y(carve::geom::VECTOR(-local_z.y, local_z.x,
     0.0)); // always lies in the z-y plane(Curve (and x-y 2D-Profile)

// get the vertical as cross product
carve::geom::vector<3> local_x = carve::geom::cross(local_z , local_y);
```

The three vectors are also normalized before being placed in the rotation matrix in order to keep them pointig in the same direction, only with a length of 1.

4.2.4 Merge Lists

The *CrossSections* (profiles) are only defined at the CrossSectionPoints (CSP) and not at the BasisCurvePoints (BCP). However, since the profile needs to be swept across all points, the profiles between each *CrossSections* must be calculated additionally calculated. The profile also begins at the first CrossSection and not at the first BCP. Before merging and sorting the BCP and CSP into one ordered list, first the startingpoint of the profile must be found. The algorithm 4 shown below presents the method used.

Algorithm 4 Find profile starting point

```
Input: BasisCurvePoints, CrossSectionPoints
Output: First point in points_for_tessellation
 1: set i = 0                                  ▷ i will be the counter for the BCP
 2: set j = 0                                  ▷ j will be the counter for the CSP
 3: if BCP[0] = CSP[0] then
 4:     increment i value
 5:     increment j value
 6: else
 7:     define dist_1 = distance(BCP[i], CSP[0]
 8:     define dist_2 = distance(BCP[i+1], CSP[0])
 9:     while dist_1 > dist_2 do
10:         increment i
11:     end while
12: end if
```

This algorithm ends when the value j has reached the BCP that is after the first CSP . This is the starting point for the profile; from this point on the directions of the points will be saved in the vector *directions_for_tessellation*. The points will be saved in the vector *points_for_tessellation*, and the corresponding information for the profiles will be saved in a vector like paths called *paths_for_tessellation* until the last CSP is reached.

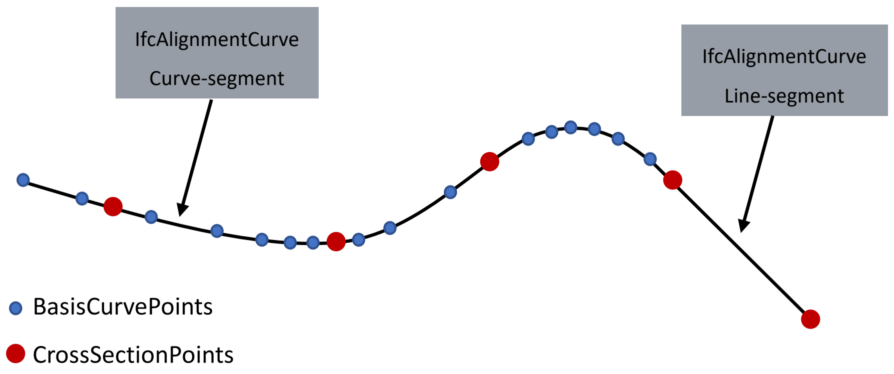


Figure 4.1: BasisCurvePoints (BCP) and CrossSectionPoints (CSP) on the directrix represented by an *IfcAlignmentCurve*

It is also worth mentioning that the CurveConverter does not calculate any point on a line segment as shown in the Figure 4.1. The first CrossSection, CSP and direction are then saved as the first elements in *paths_for_tessellation*, *points_for_tessellation* and *directions_for_tessellation*. Then the value from j is incremented.

Figure 4.1 also shows how the points are ordered along the *directrix*. The points on the list should consequently also be ordered like this as well. The method used to merge both lists of points into the new one is best shown by the following Nassi–Schneiderman diagram:

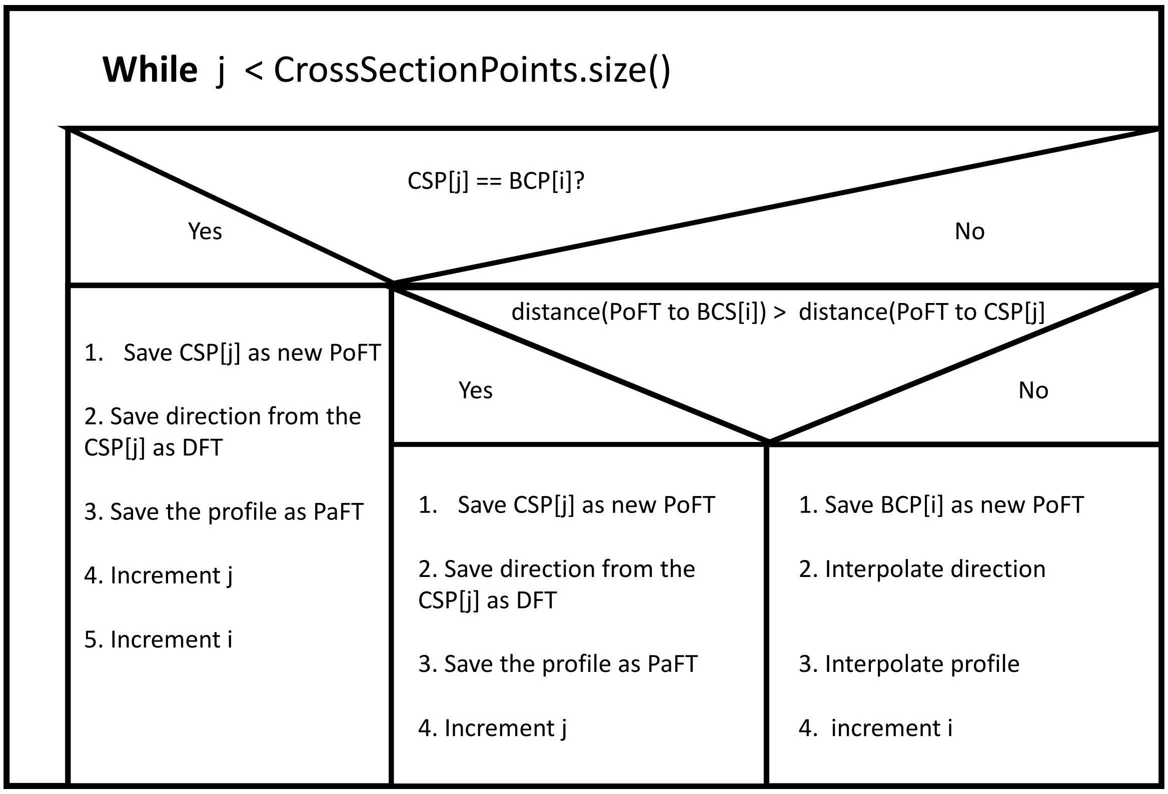

While j < CrossSectionPoints.size()
CSP[j] == BCP[i]?
Yes
No
1. Save CSP[j] as new PoFT
2. Save direction from the CSP[j] as DFT
3. Save the profile as PaFT
4. Increment j
5. Increment i
distance(PoFT to BCS[i]) > distance(PoFT to CSP[j]
Yes
No
1. Save CSP[j] as new PoFT
2. Save direction from the CSP[j] as DFT
3. Save the profile as PaFT
4. Increment j
1. Save BCP[i] as new PoFT
2. Interpolate direction
3. Interpolate profile
4. increment i

Figure 4.2: Nassi-Schneidermann diagramm for merging the lists with PoFT: *points_for_tessellation*, PaFT: *paths_for_tessellation*, DFT: *directions_for_tessellation* .

As presented in the diagram the process is repeated until the last CSP has been reached, also marking the end of the geometric form. At the end, *paths_for_tessellation* has one profile for each point regardless of it being a CSP or a BCP. The profile points for a BCP need to be interpolated as well as the directions. This method uses the CSP before and after the BCP. This process will be explained in more detail in the next Subsection 4.2.5.

The diagram in Figure 4.2 also gives an insight into how to proceed if the next point after the *points_for_tessellation* is a CSP. In this case the profile points are already saved in the vector *paths* but the geometry is still in 2D and in the CS of the profile. To correctly place the profile points in 3D and in the Curve's CS, a Helmert transformation has to be performed in homogeneous coordintes, this means multiplying every point with the *ob-*

ject_placement_matrix calculated in Section 4.2.3. The method used to transform all profile points from one CrossSection (path[j]) is presented in algorithm 5:

Algorithm 5 Transform profile from 2D to 3D on the curve [1].

```
Input: paths, BCP
Output: paths_for_tessellation
 1: compositeProfile = paths[j]                    ▷ paths[j] is the CrossSection of BCP[j]
 2: define vector Tcompositeprofile that will save the new crossSection in 3D
 3: for $w \leftarrow 0$ to $w <$ size.compositeProfile do
 4:     loop = compositeProfile[w]
 5:     define vector Tloop that will save the new profile points in 3D
 6:     for $k \leftarrow 0$ to $k <$ size.loop do
 7:         $(Oldpoint)$ = loop[k]
 8:         $newpoint$ = object_placement_matrix $\times$ $(Oldpoint)$
 9:         fill Tloop each iteration with the $newpoint$
10:     end for
11:     fill Tcompositeprofile with Tloop
12: end for
13: Save Tcompositeprofile as new $\in$ of paths_for_tessellation
14: increment j                                    ▷ Refer to A.6 to see the C++ code.
```

4.2.5 Interpolation

The process for the for the BCP is different because the profiles must to be interpolated given that they are not yet defined at theses points. As such, the factors for the interpolation in the first step must be calculated based on the distance from the BCP to the CSP, both before and after. For this purpose the three variables of type double are defined: totalDistance, factorBefore and factorAfter. The distance is calculated with the helper function *distance* (4.1).

totalDistance = distance(CSP[j-1], CSP[i])

factorBefore = distance(CSP[j-1], BCSP[i])÷ totalDistance

factorAfter = distance(CSP[j], BCP[i])÷ totalDistance

Before interpolating the point, a new *object_placement_matrix* must be calculated to later move and rotate the interpolated profile to the correct position on the curve. Therefore, the steps 1, 4 and 5 from Section 4.2.3 must be calculated. Step one, however does not use a helper function, as the offsets can simply be interpolated with the following equation and saved in the vector offsetFromCurve.

IoffsetFromCurve = offsetFromCurve[j - 1] $\times$ factorAfter + offsetFromCurve[j] $\times$ factorBefore

For each BCP also one *object_placement_matrix* must also be calculated. As a result, this process is also a part of the algorithm. The method used resebmbles algorithm 5 only differing by the fact that, difference being that in every step the iteration has to be performed with paths[j-1] and paths[j] to get the informations of the profile coordinates from the profile before and the profile after the corresponding BCP. To calulate the new profile point, the innermost for loop first calculates the difference between the x coordinates and y coordinates of the two profiles. For that it saves the difference in a 2D vector called *deltapoint* with the following coordinates:

$$\text{deltapoint} = (x, y) = (\text{pointAfter.x - pointBefore.x , pointAfter.y - pointBefore.y})$$

the *pointAfter* referes to the profile from the CSP[j] and *pointBefore* to that of CSP[j-1]. Lastly, the new profile points can be calculated in 2D by taking the *profileBefore* and adding the difference muliplied by the factor to it as shown below in the code.

Point interpolation

```
carve::geom::vector<2> Tpoint2D = carve::geom:: VECTOR
(deltapoint.x * factorBefore + pointBefore.x,
deltapoint.y * factorBefore + pointBefore.y);

carve::geom::vector<3> Tpoint = object_placement_matrix_pos *
(carve::geom::VECTOR(Tpoint2D.x,Tpoint2D.y, 0));
```

Afterwards, the point is multiplied with the *object_placement_matrix* so that it is rotated in 3D and converted into the curve's CS.

4.2.6 Vertices

Algorithm 6 Calculate vertices of the body [1]

Input: paths_for_tessellation along the directrix,
Output: list of vertices in the global coordinate system

```
1: body ← empty
2: for i ← 0 to i < paths_for_tessellation do
3:     for w ← 0 to w < profiles do
4:         for k ← 0 to k < profile points do
5:             point = loop[k]
6:             add the point to body and convert to global coordinates
7:         end for
8:     end for
9: end for                                ▷ Refer to A.9 to see the C++ code.
```

The algorithm 6 shows the method used to define the vertices after *paths_for_tessellations* obtained all the information from the points on the *directrix* and the profiles on each point.

The transformation from the curves CS to the global CS is achieved by multiplying the *pos* matrix with all points in all the profiles. The vertices are saved in *body_data*. Each point representing a vertex and *body_data* is also saved in the order of the *directrix*, with the first profile point from the first profile to the last profile point from the last profile.

4.2.7 Tessellation

The last step in the implementation of *IfcSectionedSolidHorizontal* is to create the body of the geometry by connecting the vertices calculated in the previous step and creating triangular faces that then combine to form a closed solid geometric form. This process is divided into three steps:

1. Add faces between the profiles to create a body along the *directrix*.
2. Close front cap of the body.
3. Close back cap of the body.

Step 1: Create the body along the directrix

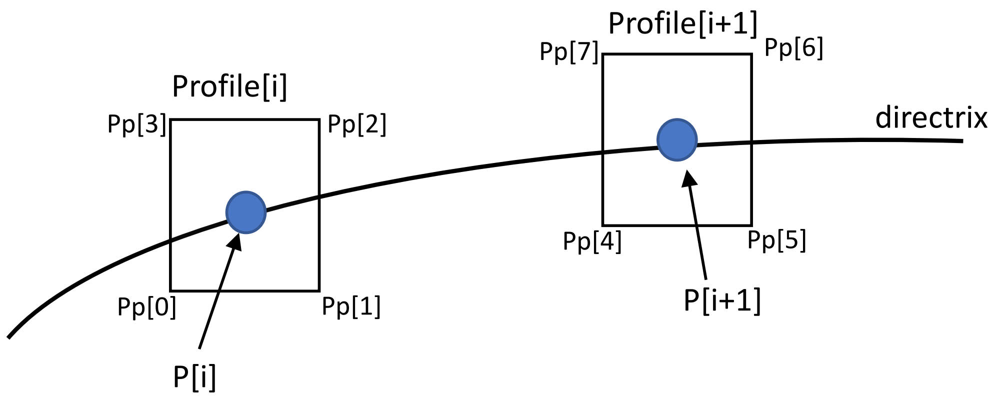


Figure 4.3: Two profiles placed at the *points_for_tessellation* along the the directrix with P: *points_for_tessellation* , Pp: Profile points.

As shown in the Figure 4.3 the profiles are positioned at the corresponding *points_for_tessellation*, but are not connected to each other. To create a body along the *directrix* it is necessary to add faces to *body_data*, so as to connect each profile point with one from the next profile. This gradually forms triangles that are then connected with each other, creating a closed outer hull for the body that can then be understood by the rendering engine (Chapter 2.2 [18]). Now that all the profile points are saved as one long list of vertices, numbered from 0 (first point of the first profile) to the last vertex index (last point of the last pofile) in the *directrix*. For the Figure 4.3, this would mean that the first vertex is the point 0 and the last vertex is the point 7.

Sides	**Points**	**Faces**
1	(0,1,4,5)	(0,1,5), (0,4,5)
2	(0,3,4,7)	(0,3,7), (7,0,4)
3	(2,3,6,7)	(3,2,7), (2,6,7)
4	(1,2,5,6)	(6,1,5), (6,2,1)

Table 4.1: Table of faces required to create the outer shell.

Table 4.1 shows the faces needed to close the body for the Figure 4.3. For more complex models with more faces or with more profiles this is written into the algorithm 7.

Algorithm 7 Calculate faces and body of IfcSectionedSolidHorizontal [1].

```
Input: Vertices, paths_for_tessellation (PFTS),
Output: Faces and closed body of IfcSectionedSolidHorizontal
 1: for $i \leftarrow 0$ to $i <$ PFTS-1 do
 2:     for $j \leftarrow 0$ to $j <$ ppoints(pp) -1 do
 3:         add face to body (i*pp+j, i*pp+j+1, (i+1)*pp+j )        ▷ i+1 = next profile
 4:         add face to body (i*pp+j+1, (i+1*pp+j, (i+1)*pp+j+1))
 5:     end for
 6: end for
 7: add face to body (0, pp-1, pp)
 8: for $ii \leftarrow 0$ to $ii <$ (vertices÷pp)-1 do
 9:     add face to body (ii*pp, ii*pp-1, (ii+1)*pp-1 )             ▷ ii+1 = next profile
10:     add face to body ((ii+1)i*pp, ii*pp, (ii+1)*pp-1))
11: end for                                    ▷ Refer to A.9 to see the C++ code.
```

Step 2: Close front cap of the body

To close the front cap, the points of the first profile must be connected to each other to form triangles and subsequently faces. Therefore, the method used to close the front cap takes the first vertex(0) and iterates to the vertex (number of profile points - 3) connecting the faces as followed:

Algorithm 8 Closing front and back profile of IfcSectionedSolidHorizontal [1].

```
Input: Vertices
Output: Closes front and back from the IfcSectionedSolidHorizontal
1: for $jj \leftarrow 0$ to $i <$ ppoints - 2 do                          ▷ closing front
2:     add faces to body (0, jj+1, jj+2)
3: end for
4: for $jj \leftarrow vertices - pp$ to $jj <$ vertices  do               ▷ closing back
5:     define the last vertex (vertices-1)
6:     add faces to body (last, jj-1, jj-2)
7: end for                                    ▷ Refer to A.9 to see the C++ code.
```

Step 3: Close back cap of the body

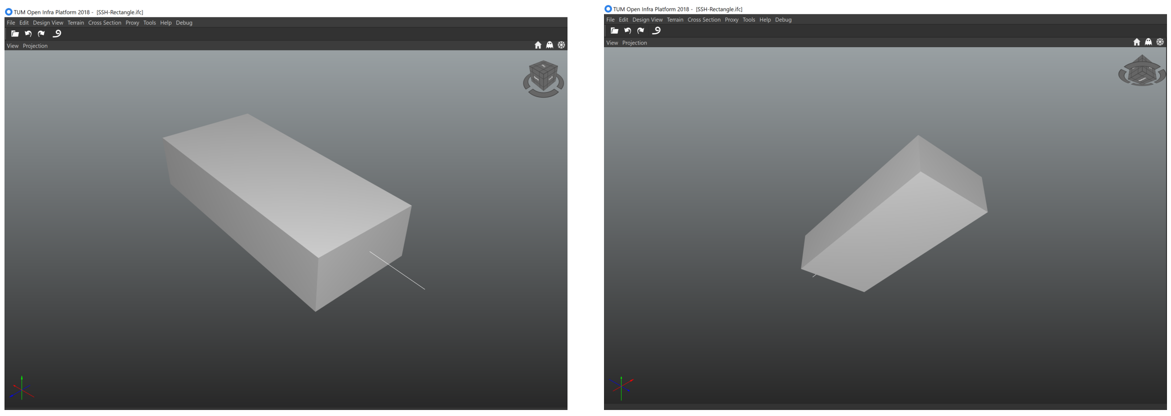
As shown, algorithm 8 is the method defined to close the body at the end. Since the final profile is at the end of the vertex list, the iteration must start at the first point of the final profile (vertices - profile points(pp)), and terminate at the last point defined as last (vertices - 1) therefore closing the back part of the body by connecting the points of the last profile to the last profile point.

Chapter 5

Use Cases

This chapter will test the implementation by visualizing IFC example files. The examples begin at a low level of complexity, such as extrusions along a line segment, and build up to more complex operations, such as sweeping the geometry along a curve. To also test the limits of the implementation, the last example file 5.5 is designed to surpass the limit and thus will not render.

5.1 Rectangle Profile


TUM Open Infra Platform 2018 - [SSH-Rectangle.ifc]
File Edit Design View Terrain Cross Section Proxy Tools Help Debug
View Projection
TUM Open Infra Platform 2018 - [SSH-Rectangle.ifc]
File Edit Design View Terrain Cross Section Proxy Tools Help Debug
View Projection

(a) Top view. **(b)** Bottom view.

Figure 5.1: Visualization of *IfcSectionedSolidHorizontal* with the profile extruded along a line-segment.

The first example for the *IfcSectionedSolidHorizontal* is a rectangle that is extruded along the line segment of the *IfcAlignmentCurve* which can be observed in Figure 5.1. The profiles are of the type *IfcRectangleProfileDef* with only two *CrossSections* in this case.

IfcSectionedSolidHorizontal example snippet of A.10

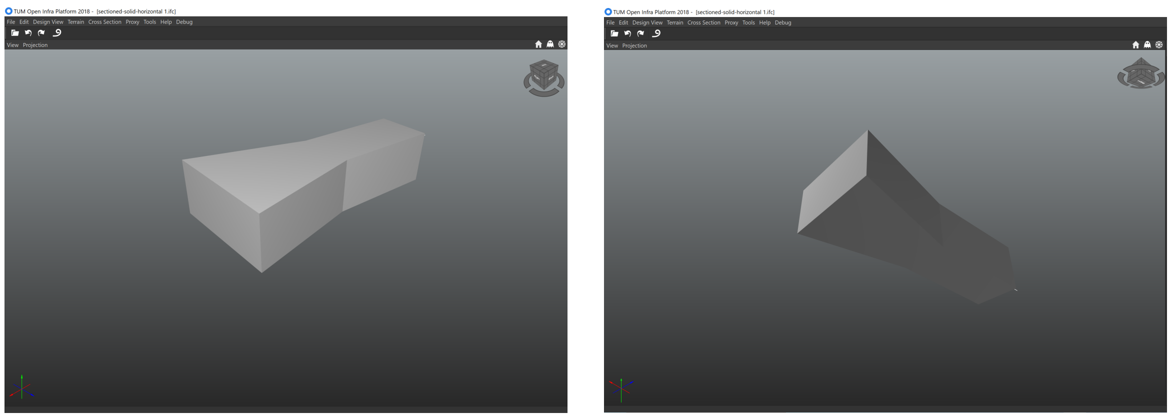
```
#15= IFCSHAPEREPRESENTATION(#9,'Body','Tessellation',(#18));

#18= IFCSECTIONEDSOLIDHORIZONTAL(#97,(#98,#98),(#103,#104),.T.);
#103= IFCDISTANCEEXPRESSION(0.,$,$,$,.T.);
#104= IFCDISTANCEEXPRESSION(4320.,$,$,$,.T.);
#98= IFCRECTANGLEPROFILEDEF(.AREA.,'1m x 2m rectangle',$,1000.,2000.);
```

#18 connects the profiles with the corresponding *IfcDistanceExpressions* and to the *IfcAlignmentCurve* (#97). Position #103 defines the position of the first CrossSection with the profile #98(1× 2 [m] rectangle) and the position #104 defines the position of the second and last CrossSection also with the same profile.

It is also worth mentioning that all examples have the same "FILE_SCHEMA", in this case "IFC4x3_RC1" as they are designed to be part of the IFC4x3 version.

5.2 Variable Profile



(a) Top view. **(b)** Bottom view.

Figure 5.2: Visualization of *IfcSectionedSolidHorizontal* with two profiles extruded along on a line-segment.

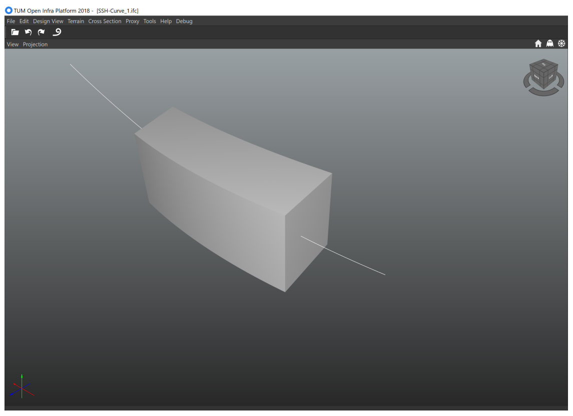
The second example builds upon the first and has one addtional CrossSection. Unlike example 5.1, the profile at the end is no longer the same anymore as shown in Figure 5.2. In this case, the profile widens in a similar fasion to a highway, for example. This example still uses the same #97 *IfcAlignmentCurve* for the extrusion along the line segment.

IfcSectionedSolidHorizontal example snippet of A.10

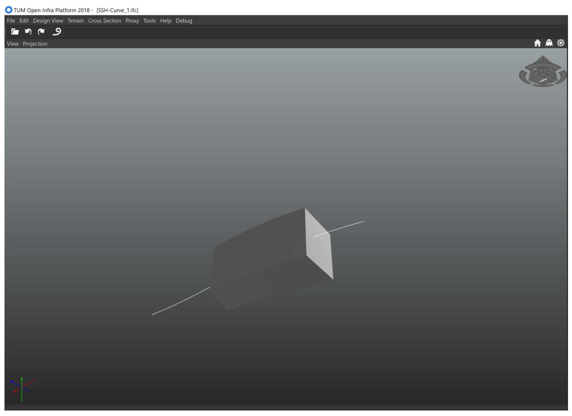
```
#15= IFCSHAPEREPRESENTATION(#9,'Body','Tessellation',(#18));
#18= IFCSECTIONEDSOLIDHORIZONTAL(#97,(#98,#99,#100),(#103,#104,#105),.T.)
    ;
#103= IFCDISTANCEEXPRESSION(0.,$,$,$,.T.);
#104= IFCDISTANCEEXPRESSION(2160.,$,$,$,.T.);
#105= IFCDISTANCEEXPRESSION(4320.,$,$,$,.T.);
#98= IFCRECTANGLEPROFILEDEF(.AREA.,'1m x 1m rectangle',$,1000.,1000.);
#99= IFCRECTANGLEPROFILEDEF(.AREA.,'1m x 1m rectangle',$,1000.,1000.);
#100= IFCRECTANGLEPROFILEDEF(.AREA.,'1m x 2m rectangle',$,1000.,2000.);
```

The new CrossSection (profile #100) is paired in #18 with the *IfcDistanceExpression* in #105 as shown in the code snippet above. There is still no need for a interpolation because the example does not generate BasisCurvePoints, just CrossSectionPoints.

5.3 Extrusion along a curve segment


TUM Open Infra Platform 2018 - [SSH-Curve_1.ifc]
File Edit Design View Terrain Cross Section Proxy Tools Help Debug
View Projection

(a) Top view.


TUM Open Infra Platform 2018 - [SSH-Curve_1.ifc]
File Edit Design View Terrain Cross Section Proxy Tools Help Debug
View Projection

(b) Bottom view.

Figure 5.3: Visualization of *IfcSectionedSolidHorizontal* with the profile extrudet along a curve-segment

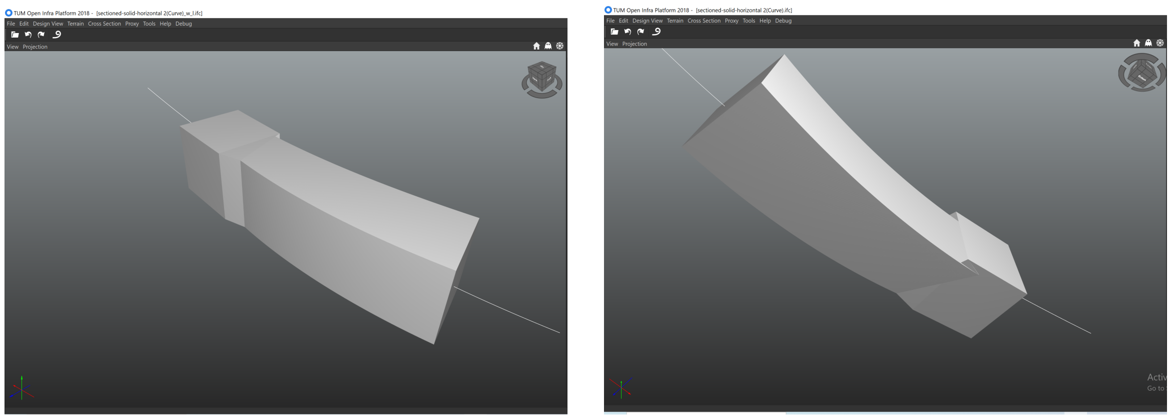
The third example is the first one to be extruded along a curve segment of the *IfcAlignmentCurve* as shown in Figure 5.3. In this case the CurveConverter mentioned in Chapter 4.2.1 calculates the BasisCurvePoints (BCP), and the profiles must be interpolated using the profiles form the CrossSectionPoints, both before and after the BCP.

Curvature of the IfcAlignmentCurve

```
#97= IFCALIGNMENTCURVE(#489,#490,$);
#489= IFCALIGNMENT2DHORIZONTAL($,(#541));
#490= IFCALIGNMENT2DVERTICAL((#543,#544));
#541= IFCALIGNMENT2DHORIZONTALSEGMENT(.T.,'61+25.00','63+72.14',#595);
#543= IFCALIGNMENT2DVERSEGLINE(.T.,$,$,0.,3600.,190.56,0.0579);
#544= IFCALIGNMENT2DVERSEGPARABOLICARC(.T.,$,$
    ,3600.,3685.68,399.,0.0579,36000.,.F.);
#595= IFCCIRCULARARCSEGMENT2D(#485,13.35833333,2965.68,9279.,.F.);
#485= IFCCARTESIANPOINT((0.,0.));
```

The curvature of the *IfcAlignmentCurve* in the example shown in Figure 5.3 is generated by the "IFCCIRCULARSEGMENT2D" in position #595 as presented in the code snippet above. Since the curved segment is referenced by the horizontal part of the Alignment (#489), the curvature appears not only in the horizontal but also in the vertical fraction of the curve by adding the line #544 (which is of type *IfcAlingment2DVerSegParabolicArc*).

5.4 Extrusion along a curve with a line segment



(a) Top view. **(b)** Bottom view.

Figure 5.4: Visualization of *IfcSectionedSolidHorizontal* with a profile extruded along a curve-segment and a line-segment

Figure 5.4 displays an *IfcRectangleProfileDef* swept along the *directrix*. In this case the *directrix IfcAlignmentCurve* has a curve segment and a line segment at the end. Therefore, the profile is simply interpolated at the BasisCurvePoints from the curve segment.

Additional line segment at the end of the IfcAlignmentCurve

```
#97= IFCALIGNMENTCURVE(#489,#490,$);
#489= IFCALIGNMENT2DHORIZONTAL($,(#541,#542));
#490= IFCALIGNMENT2DVERTICAL((#543,#544));
#541= IFCALIGNMENT2DHORIZONTALSEGMENT(.T.,'61+25.00','63+72.14',#595);
#542= IFCALIGNMENT2DHORIZONTALSEGMENT(.T.,'63+72.14','65+65.00',#596);
#543= IFCALIGNMENT2DVERSEGLINE(.T.,$,$,0.,3600.,190.56,0.0579);
#544= IFCALIGNMENT2DVERSEGPARABOLICARC(.T.,$,$
    ,3600.,3685.68,399.,0.0579,36000.,.F.);

#595= IFCCIRCULARARCSEGMENT2D(#485,13.35833333,2965.68,9279.,.F.);
#485= IFCCARTESIANPOINT((0.,0.));
#596= IFCLINESEGMENT2D(#597,-4.95408690777971,4320.);
#597= IFCCARTESIANPOINT((2945.13464253837,216.386895560319));
```

As shown in the code snippet above now the *IfcAlignment2dHorizontal*(#489) not only references the *IfcCircularsegment2D* (#595) but also the *IfcLineSegment2D* in the line #596 which generates the line seen at the end of the Figure 5.4.

5.5 Unsupported Profiles

Finally the Example 5 shows the limits of this implementation and consists of five profiles(CrossSections) of type *IfcArbitraryClosedProfileDef* which are then defined by the points that are connected by an*IfcIndexedPolyline* to create profiles that may have more edges(vertices) than the four edges in a *IfcRectangleProfileDef* per profile.

CrossSection described by an IndexedPolycurve

```
#9= IFCSHAPEREPRESENTATION(#2,'Body','AdvancedSweptSolid',(#38));
#38= IFCSECTIONEDSOLIDHORIZONTAL
    (#97,(#98,#98,#98,#98,#98,#99,#100,#101,#102)
    ,(#103,#104,#105,#106,#107,#108,#109,#110,#111),.T.);
#103= IFCDISTANCEEXPRESSION(0.,$,$,$,.T.);
#97= IFCALIGNMENTCURVE(#489,#490,$);
#98= IFCARBITRARYCLOSEDPROFILEDEF(.AREA.,$,#491);
#491= IFCINDEXEDPOLYCURVE(#545,(IFCLINEINDEX((1,2,3,4,1))),.F.);
#545= IFCCARTESIANPOINTLIST2D(((180.,14.4),(0.,0.),(0.,192.),(180.,206.4)
    ),$);
```

The code snippet above presents one of the many *CrossSections* of this example. Each CrossSection(profile) is defined by the *IfcArbitraryClosedProfileDef* which, as mentioned before is defined by the *IfcIndexedPolyline* that marks the vertices of the profile.

This presents a problem for the implementation, since the ProfileConverter does not as of yet support the *IfcIndexedPolyCurve*. In addition, the *IfcIndexedPolyCurve* can have as many points as needed to define the profile, whichwould also lead to a compiler error if the code could support it. This is due to the fact that the *CrossSections* must have the same number of profile points in order to correctly calculate the vertices and interpolation as well as connect the faces to form the body.

Chapter 6

Discussion

This chapter evaluates the implementation of *IfcSectionedSolidHorizontal* by discussing the results from the use cases in the previous chapter (Chapter 5). Section 6.1 will start by discussing those use cases that have a lower level of complexity, and then proceed to those that are more complex. Section 6.2 will then evaluate the limitations of the methods presented in the informal proposition in Chapter 3.1 as well as suggestions for improving the implementation.

6.1 Evaluation of the Use Cases

The first two use cases (see Sections 5.1 and 5.2) showed an extrusion along a *IfcLineSegment* with the first use case having the same profile throughout the rectangular geometry and the second use case having a variable geometry. In both cases, the implementation connected the CrossSections along the directrix correctly producing the visualization shown in Figures 5.1 and 5.2.

Since the *CrossSections* (*IfcProfileDef*) are extruded(swept) along a curve segment in the use cases 5.3 and 5.4, the implementation relies on methods such as the interpolation to calculate the profiles between the *CrossSections*. As shown in the Figure 5.3, the interpolation for the profiles on the *directrix* curve creates a profile for each BasisCurvePoint with such a small distance between BasisCurvePoints that the object appears curved and lacks sharp edges. However, in the next use case (5.4), the change from the curve segment to a line segment is clearly reflected in the geometry. The angle of the last CrossSection before the line segment creates a visible edge. Furthermore, one side of the CrossSection is pushed into the first CrossSection of the line segment, whilst the other side does not reach the point. In order for the implementation to work correctly, this angle must be corrected.

6.2 Evaluation of Methods

The implementation of *IfcSectionedSolidHorizontal* requires the *CrossSections* to be extruded along the *IfcAlignmentCurve* (sweeping operation [21]). To define profiles between the *Cross-Sections* they must be calculated by interpolating the known profiles. One of the limitations presented by the interpolation is that the *CrossSections* must have the same number of points. This enables each profile point to be calculated by taking the same point from the profile prior to as well as directly after the curve point. Should it be the case that the profiles have a different number of points, the calculation would have to use a modified algorithm. This modified algorithm would be such that so that the interpolated profile could retrieve the information needed from the two *CrossSections* without the vectors going out of scope in the iterations and thus producing an error. In addition, if the *CrossSections* were to have a different number of points, the method used to calculate the body of the function would also have to be improved.

Furthermore, to improve the implementation an algorithm should be developed to provide a smooth transition between the profile sweept along the curve segment and the connection to the extrusion along the line segment of the *IfcsAlingmentCurve*. This would avoid the sharp edges shown in Figure 5.4 by correcting the angle of this transition.

Chapter 7

Conclusion

The conclusion presents a summary of the findings in Section 7.1. Furthermore, the Section 7.2 provides an outlook on the future research that could be conducted in this field.

7.1 Summary of Findings

The release of IFC 4x1 introduced new entities such as *IfcSectionedSolidHorizontal* to help support infratstructure extension. Moreover, projects such as the IFC-Bridge, IFC-Road and IFC-Rail rely on the Alignment as a common ressource for the representation and visualization [3]. The *IfcAlignmentCurve* is also the main curve along which the profiles are swept along in the newly introduced entity *IfcSectionedSolidHorizontal*.

This entity is also of vital importance for the IFC-Road project, as it helps support roadway geometry by allowing multiple profiles, called *CrossSections* to be swept along the Alignment [6]. Therefore, it is also of vital importance to implement this entity into the workflow of the existing Open Source BIM visualization software of the TUM called OpenInfraPlatform (OIP), as mentioned in Chapter 2.2.

As stated in the propositions in Chapter 3.1, the implementation of *IfcSectionedSolidHorizontal* will visualizes this entity as long as every CrossSection has the same number of profile points and the profile is understood by other parts of the OIP such as the ProfileConverter.

First of all, in the implementation *directrix* (*IfcAlignmentCurve*) is split into characteristic points called BasisCurvePoints (BCP) by the OIP's CurveConverter. These points do not possess any information about the profiles since these are only defined at the *CrossSections* along the *directrix*.

Furthermore, the profiles (*CrossSections*) are defined in 2D. The method presented in Chapter 4.2.3 converts the information contained in the *CrossSectionPositions* in order to find the position and direction of the points where these profiles are defined along the *directrix*. It then saves the list of points as well as the transformation matrices discussed in Chapters 2.3 and 2.4 using the mtehod performed in Chapter 4.2.3.

Next, the two lists (one with the CSP and the profiles and the other with the BCP) are merged into one list containing the points along the *directrix* where the profiles are defined. This new list also simultaneously calculates the missing profiles at the BCP by interpolating them with the method with the profiles before and after the point, rotates the profiles in 3D and places them in the correct position along the *directrix*.

The last part then defines the vertices of each profile and connects them forming triangular faces that add up to a solid geometry, allowing the model to be rendert by the rendering engine.

As a result of the testing with the examples, it is clear that the next step for the implementation would be to improve the code so as to correct the angle of the profiles between a curve segment and a line segment, as well as to improve the interpolation so that it may represent profiles in which the number of points for each profile CrossSection) is not the same.

7.2 Future Works

Regarding the implementation presented in this thesis, there is still much work that can be done to improve the code, starting with the development of a smarter interpolation algorithm that could interpret *CrossSections* with a variable number of points without the iterations going out of scope. Also, the profile angles should be corrected in the transitions from a curve segment to a line segment of the *IfcAlignmentCurve* as previously mentioned in the Chapters 6.2 and 7.1.

As for the IFC-Road project the Alignment could also be expanded to support new scenarios such as crossing two roads at an intersection [6]; this could then support different angles of the two roads or for the IFC-Bridge, where two roads could cross each other but at different heights so that they pass without intersecting with each other.

For the purpose of this thesis, the *IfcSectionedSolidHorizonzal* was implemented for the IFC 4x3 format (IFC)-Road), but it is important to highlight the fact that it could be expanded to go further and also support other infrastructure projects such as the upcoming IFC-Rail, as this project since it also relies on the Alignment curve and may contain variable *CrossSections* [6]. To support the rail it could also define new entities to describe the superelevation on the railway alignment[6]. The harmonization between the Rail, Road and Bridge projects builds upon the next major release of the Industry Foundation Classes standard with the version IFC 5 [16].

In addition, the OIP could also be expanded to implement new entities to support more use cases in the road project, such as the crossing of roads and junctions or also support the next IFC versions such as the IFC Tunnel, IFC Ports and Waterways.

Appendix A

Code

The most relevant parts of the C++ code are listed in the appendix as well as one example file. The complete OIP code and the more complex examples developed for this thesis can be found on Github: https://github.com/tumcms/Open-Infra-Platform/blob/development/Core/src/IfcGeometryConverter/SolidModelConverter.h

https://github.com/tumcms/Open-Infra-Platform/tree/development/testdata/IFC4X3ExampleFiles

A.1 ConvertCurveOrientation

Listing A.1: ConvertCurveOrientation

```
carve::math::Matrix convertCurveOrientation(
const carve::geom::vector<3> directionOfCurve,
const bool fixed_axis_vertical,
const carve::geom::vector<3> translate = carve::geom::VECTOR(0., 0., 0.)
) const throw(...)
{

    carve::geom::vector<3> local_z(carve::geom::VECTOR(directionOfCurve.x
        , directionOfCurve.y, fixed_axis_vertical ? 0.0 : directionOfCurve
        .z));

    // get the perpendicular to the left of the curve in the z-y plane (
        curve's coordinate system)
    carve::geom::vector<3> local_y(carve::geom::VECTOR(-local_z.y,
        local_z.x, 0.0)); // always lies in the z-y plane(Curve (and x-y 2
        D-Profile)

    // get the vertical as cross product
    carve::geom::vector<3> local_x = carve::geom::cross(local_z, local_y)
        ;

    // normalize the direction vectors
    local_x.normalize();
    local_y.normalize();
    local_z.normalize();

    // produce a rotation matrix
    return carve::math::Matrix(
    local_x.x, local_y.x, local_z.x, translate.x,
    local_x.y, local_y.y, local_z.y, translate.y,
    local_x.z, local_y.z, local_z.z, translate.z,
    0, 0, 0, 1);
}
```

A.2 CalculatePositionOnAndDirectionOfBaseCurve

Listing A.2: CalculatePostionOnAndDirectionOfBaseCurve

```
std::tuple< carve::geom::vector<3>, carve::geom::vector<3>>
    calculatePositionOnAndDirectionOfBaseCurve(
const EXPRESSReference<typename IfcEntityTypesT::IfcCurve>& directrix,
const EXPRESSReference<typename IfcEntityTypesT::IfcDistanceExpression>&
    cross_section_positions,
const double relativeDistAlong = 0.
) const throw(...)
{
// check input
if (directrix.expired())
throw oip::ReferenceExpiredException(directrix);
if (cross_section_positions.expired())
throw oip::ReferenceExpiredException(cross_section_positions);

// defaults
carve::geom::vector<3> pointOnCurve = carve::geom::VECTOR(0.0, 0.0, 0.0);
carve::geom::vector<3> directionOfCurve = carve::geom::VECTOR(1.0, 0.0,
    0.0);

// account for relative placement
double dDistAlong = cross_section_positions->DistanceAlong * UnitConvert
    ()->getLengthInMeterFactor()
+ relativeDistAlong;

// convert the point
placementConverter->convertBoundedCurveDistAlongToPoint3D(
directrix.as<typename IfcEntityTypesT::IfcBoundedCurve>(),
dDistAlong,
cross_section_positions->AlongHorizontal.value_or(true),
pointOnCurve,
directionOfCurve
);
return { pointOnCurve, directionOfCurve };
}
```

A.3 Directrix

Listing A.3: Directrix

```
//Give directrix to Curve converter: for each station 1 Point and 1 Direction
// the stations at which a point of the tessellation has to be calculated - to be converted and fill the targetVec
std::vector<double> stations = curveConverter->getStationsForTessellationOfIfcAlignmentCurve(directrix.as<typename IfcEntityTypesT::IfcAlignmentCurve>());

carve::geom::vector<3> targetPoint3D;
carve::geom::vector<3> targetDirection3D;
std::vector<carve::geom::vector<3>> BasisCurvePoints;
std::vector<carve::geom::vector<3>> BasisPointDirection;

// attach the curve points
for (auto& it_station : stations)
{
// call the placement converter that handles the geometry and calculates the 3D point along a curve
placementConverter->convertBoundedCurveDistAlongToPoint3D(directrix.as<typename IfcEntityTypesT::IfcBoundedCurve>(), it_station, true, targetPoint3D, targetDirection3D);
BasisCurvePoints.push_back(targetPoint3D);
BasisPointDirection.push_back(targetDirection3D);
}

std::shared_ptr<carve::input::PolyhedronData> body_data = std::make_shared<carve::input::PolyhedronData>();
itemData->closed_polyhedrons.push_back(body_data);

int num_curve_points = BasisCurvePoints.size();
carve::math::Matrix matrix_sweep;

// Less than two points is a point
if (num_curve_points < 2)
{
throw oip::InconsistentModellingException(sectioned_solid_horizontal, "num curve points < 2");
}
```

A.4 CrossSections

Listing A.4: CrossSections

```cpp

//define vector to fill with the coordinates of the CrossSections
std::vector<std::vector<std::vector<carve::geom::vector<2> > > > paths;

for (int i = 0; i < vec_cross_sections.size(); ++i)
{
std::shared_ptr<ProfileConverterT<IfcEntityTypesT>> profile_converter =
    profileCache->getProfileConverter(vec_cross_sections[i]);
const std::vector<std::vector<carve::geom::vector<2> > >& profile_coords
    = profile_converter->getCoordinates();

// Save profile coords in paths
std::vector<std::vector<carve::geom::vector<2> > > profile_coords_2d;
for (int p = 0; p < profile_coords.size(); ++p)
{
const std::vector<carve::geom::vector<2> >& profile_loop = profile_coords
    [p];

profile_coords_2d.push_back(profile_loop);
}
paths.push_back(profile_coords_2d);

}
```

A.5 CrossSectionPositions

Listing A.5: CrossSectionPositions

```
for (int pos = 0; pos < cross_section_positions.size(); ++pos)
{
//1. get offset from curve
carve::geom::vector<3> offset = placementConverter->
    convertIfcDistanceExpressionOffsets(cross_section_positions[pos]);
offsetFromCurve.push_back(offset);

//2. calculate the position on and the direction of the base curve
//also applay the relative dist along
carve::geom::vector<3> pointOnCurve;
carve::geom::vector<3> directionOfCurve;
std::tie(pointOnCurve, directionOfCurve) =
    calculatePositionOnAndDirectionOfBaseCurve(directrix,
    cross_section_positions[pos]);
CrossSectionPoints.push_back(pointOnCurve);
directionsOfCurve.push_back(directionOfCurve);

//3. get information from FixedAxisVertical
if (fixed_axis_vertical == true)
{
directionsOfCurve[pos].z = 0;
directionsOfCurve[pos].normalize();
}

//4. calculate the rotations
//the direction of the curve's tangent = directionOfCurve
//now that localPLacement Matrix is a Vector ---> 1 Matrix for each
    CrossSectionPosition saved in the Vector localPlacementMatrix
carve::math::Matrix localm = placementConverter->
    calculateCurveOrientationMatrix(directionsOfCurve[pos],
    fixed_axis_vertical);
localPlacementMatrix.push_back(localm);

// 5. calculate the position
// the position on the curve = pointOnCurve & offsets = offsetFromCurve
carve::geom::vector<3> translate = CrossSectionPoints[pos] +
    localPlacementMatrix[pos] * offsetFromCurve[pos];
carve::math::Matrix object_placement_matrix_pos = convertCurveOrientation
    (directionsOfCurve[pos], fixed_axis_vertical, translate);
object_placement_matrix.push_back(object_placement_matrix_pos);
}
```

A.6 CrossSectionPoints

Listing A.6: CrossSectionPoints

```

//1. Save the point and direction in new vector
points_for_tessellation.push_back(CrossSectionPoints[j]);
direction_for_tessellation.push_back(directionsOfCurve[j]);

//2. Save CrossSections
std::vector<std::vector<carve::geom::vector<2> > >& compositeProfile =
    paths[j];
std::vector<std::vector<carve::geom::vector<3> > > Tcompositeprofile;
for (int w = 0; w < compositeProfile.size(); ++w)
{
std::vector<carve::geom::vector<2> >& loop = compositeProfile[w];
std::vector<carve::geom::vector<3>> Tloop;
for (int k = 0; k < loop.size(); ++k)
{
carve::geom::vector<2>& point = loop[k];

carve::geom::vector<3>  Tpoint = object_placement_matrix[j] * (carve::
    geom::VECTOR(point.x, point.y, 0));
Tloop.push_back(Tpoint);
}
Tcompositeprofile.push_back(Tloop);
}

paths_for_tessellation.push_back(Tcompositeprofile);

//3. increment based on its position on the directrix
++j;
```

A.7 Factors for Interpolation

Listing A.7: Factors for interpolation

```cpp
//Save the point in the curve in the new vector
points_for_tessellation.push_back(BasisCurvePoints[i]);
direction_for_tessellation.push_back(BasisPointDirection[i]);

//calculate the direction of the point (Interpolate with the pointOnCurve
    before and after that point)
int directionSize = direction_for_tessellation.size()-1;
int lastPoint = points_for_tessellation.size() -1;

double totalDistance =
distance(CrossSectionPoints[j - 1], CrossSectionPoints[j]);

double factorBefore =
 (distance(CrossSectionPoints[j - 1], BasisCurvePoints[i]))/totalDistance
    ;
double factorAfter =
 (distance(BasisCurvePoints[i], CrossSectionPoints[j]))/totalDistance;
```

A.8 Interpolation

Listing A.8: Interpolation

```
//1. interpolate OffsetsFromCurve
carve::geom::vector<3> IoffsetFromCurve = offsetFromCurve[j - 1] *
    factorAfter + offsetFromCurve[j] * factorBefore;

//Calculate Matrices
//2. local_placement
carve::math::Matrix localm = placementConverter->
    calculateCurveOrientationMatrix(BasisPointDirection[i],
    fixed_axis_vertical);

// 3. calculate the position
// the position on the curve = pointOnCurve
// the offsets = offsetFromCurve
carve::geom::vector<3> translate = BasisCurvePoints[i] + localm *
    IoffsetFromCurve;
carve::math::Matrix object_placement_matrix_pos = convertCurveOrientation
    (BasisPointDirection[i], fixed_axis_vertical, translate);


//interpolate profile
// Informal proposition: for the Interpolation to work the profiles of
    the CrossSection before and after need to have the same amount of
    points and loops.

std::vector<std::vector<carve::geom::vector<2> > >&
 compositeProfileBefore = paths[j - 1];

std::vector<std::vector<carve::geom::vector<2> > >&
 compositeProfileAfter = paths[j];

std::vector<std::vector<carve::geom::vector<3> > > Tcompositeprofile;

for (int w = 0; w < compositeProfileBefore.size(); ++w)
{
     std::vector<carve::geom::vector<2> >& loopBefore =
         compositeProfileBefore[w];
     std::vector<carve::geom::vector<2> >& loopAfter =
         compositeProfileAfter[w];

     std::vector<carve::geom::vector<3>> Tloop;
     for (int k = 0; k < loopBefore.size(); ++k)
    {
        carve::geom::vector<2>& pointBefore = loopBefore[k];
        carve::geom::vector<2>& pointAfter = loopAfter[k];

        carve::geom::vector<2> deltapoint = carve::geom::VECTOR(
            pointAfter.x - pointBefore.x,
        pointAfter.y - pointBefore.y);
        carve::geom::vector<2> Tpoint2D = carve::geom::VECTOR(deltapoint.
            x * factorBefore + pointBefore.x,
        deltapoint.y * factorBefore + pointBefore.y);

        carve::geom::vector<3> Tpoint = object_placement_matrix_pos *(
            carve::geom::VECTOR(Tpoint2D.x,
        Tpoint2D.y, 0));
        Tloop.push_back(Tpoint);
    }
    Tcompositeprofile.push_back(Tloop);
}

paths_for_tessellation.push_back(Tcompositeprofile);
```

A.9 Tessellation

Listing A.9: Tessellation

```

size_t num_vertices = body_data->getVertexCount();
int PFTS = paths_for_tessellation.size() -1;
for (int i = 0; i < PFTS ; ++i)
{
    for (int j = 0; j < ppoints - 1; ++j)
    {
        body_data->addFace(i*ppoints + j, i*ppoints + j + 1, (i + 1)*
            ppoints + j);
        body_data->addFace(i*ppoints + j + 1, (i + 1)*ppoints + j, (i +
            1)*ppoints + j + 1);
    }
}

//close the first positions of the body
body_data->addFace(0, ppoints - 1, ppoints);

//close the body
int closingBodyVertices = num_vertices / ppoints;
for (int ii = 1; ii < closingBodyVertices - 1; ++ii)
{
    body_data->addFace(ii*ppoints, ii*ppoints - 1, (ii+1)*ppoints - 1);
    body_data->addFace((ii + 1)*ppoints, ii*ppoints, (ii + 1)*ppoints -
        1);
}

// close front cap
for (int jj = 0; jj < ppoints - 2; ++jj)
{
    body_data->addFace(0, jj + 1, jj + 2);
}

// close back cap
int BackCapVertex = num_vertices - ppoints;
for (int jj = BackCapVertex; jj < num_vertices ; ++jj)
{
    int last = num_vertices - 1;
    body_data->addFace(last, jj - 1, jj - 2);
}
```

A.10 Example 1

Rectangle example

```
ISO-10303-21;
HEADER;
FILE_DESCRIPTION((''),'2;1');
FILE_NAME('','2019-03-20T15:56:21',(''),(''),'BuildingSmart IfcKit by
    Constructivity','IfcDoc 12.0.0.0','');
FILE_SCHEMA(('IFC4X3_RC1'));
ENDSEC;
DATA;

#1= IFCPROJECT('0xScRe4drECQ4DMSqUjd6d',#2,'proxy with tessellation',$,$,
    $,$,(#3),#4);
#2= IFCOWNERHISTORY(#6,#7,$,.ADDED.,1320688800,$,$,1320688800);
#3= IFCGEOMETRICREPRESENTATIONCONTEXT($,'Model',3,1.0E-05,#8,$);
#4= IFCUNITASSIGNMENT((#10,#11));
#6= IFCPERSONANDORGANIZATION(#12,#13,$);
#7= IFCAPPLICATION(#13,'1.0','IFC text editor','ifcTE');
#8= IFCAXIS2PLACEMENT3D(#14,$,$);
#9= IFCGEOMETRICREPRESENTATIONSUBCONTEXT('Body','Model',0,$,$,$,#3,$,.
    MODEL_VIEW.,$);
#10= IFCSIUNIT(*,.LENGTHUNIT.,.MILLI.,.METRE.);
#11= IFCCONVERSIONBASEDUNIT(#16,.PLANEANGLEUNIT.,'degree',#17);
#12= IFCPERSON($,'','',$,$,$,$,$);
#13= IFCORGANIZATION($,'buildingSMART International',$,$,$);
#14= IFCCARTESIANPOINT((0.,0.,0.));
#16= IFCDIMENSIONALEXPONENTS(0,0,0,0,0,0,0);
#17= IFCMEASUREWITHUNIT(IFCPLANEANGLEMEASURE(0.017453293),#20);
#20= IFCSIUNIT(*,.PLANEANGLEUNIT.,$,.RADIAN.);

#5= IFCRELAGGREGATES('2YBqaV_8L15eWJ9DA1sGmT',$,$,$,#1,(#23));
#22= IFCBUILDINGELEMENTPROXY('1kTvXnbbzCWw8lcMd1dR4o',$,'P-1','sample
    proxy',$,#24,#19,$,$);

#24= IFCLOCALPLACEMENT(#26,#27);
#26= IFCLOCALPLACEMENT($,#28);
#27= IFCAXIS2PLACEMENT3D(#29,$,$);
#28= IFCAXIS2PLACEMENT3D(#14,$,$);
#29= IFCCARTESIANPOINT((1000.,0.,0.));
#23= IFCBUILDING('2FCZDorxHDT8NI01kdXi8P',$,'Test Building',$,$,#26,$,$,.
    ELEMENT.,$,$,$);

```

```
#25= IFCRELCONTAINEDINSPATIALSTRUCTURE('2TnxZkTXT08eDuMuhUUFNy',$,'
    Physical model',$,(#22,#373),#23);

#19= IFCPRODUCTDEFINITIONSHAPE($,$,(#15));

#15= IFCSHAPEREPRESENTATION(#9,'Body','Tessellation',(#18));

#18= IFCSECTIONEDSOLIDHORIZONTAL(#97,(#98,#98),(#103,#104),.T.);
#103= IFCDISTANCEEXPRESSION(0.,$,$,$,.T.);
#104= IFCDISTANCEEXPRESSION(4320.,$,$,$,.T.);
#98= IFCRECTANGLEPROFILEDEF(.AREA.,'1m x 2m rectangle',$,1000.,2000.);

#373= IFCALIGNMENT('0oo95mAXjAB8ZG7EUGVOwd',$,'Alignment',$,$,#602,$
    ,#97,.NOTDEFINED.);
#602= IFCLOCALPLACEMENT($,#604);
#604= IFCAXIS2PLACEMENT3D(#14,#610,#611);
#610= IFCDIRECTION((0.,0.,1.));
#611= IFCDIRECTION((1.,0.,0.));
#97= IFCALIGNMENTCURVE(#489,#490,$);
#489= IFCALIGNMENT2DHORIZONTAL($,(#541));
#490= IFCALIGNMENT2DVERTICAL((#543));

#541= IFCALIGNMENT2DHORIZONTALSEGMENT(.T.,'61+25.00','63+72.14',#595);
#543=IFCALIGNMENT2DVERSEGLINE(.T.,$,$,0.,4320.,190.56,0.0579);
#595= IFCLINESEGMENT2D(#597,-4.95408690777971,4320.);
#597= IFCCARTESIANPOINT((0.,0.));

ENDSEC;

END-ISO-10303-21;
```

Bibliography

[1] based on the Open Infra Platform code. https://github.com/tumcms/Open-Infra-Platform/tree/development/Core/src/IfcGeometryConverter.

[2] *Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries.* ISO, Geneva, 2013.

[3] J. Amann and A. Borrmann. Creating a 3D-BIM-compliant road design based on IFC alignment originating from an OKSTRA-accordant 2D road design using the TUM Open Infra Platform and the OKSTRA class library. 2015.

[4] J. Amann, F. Schöttl, D. Singer, M. Kern, A. Widner, P. Geisler, D. Below, H. Hecht, L. Coetzee, C. andDuckerschein, R. Schill, M. Werner, M. Böckle, N. Gupta, Ahmed Mustafa, Štefan Markič, and André Borrmann. Tum open infra platform 2019, (2019).

[5] F. Bensaali, A. Amira, I.S. Uzun, and A. Ahmedsaid. An FPGA implementation of 3D affine transformations. In *10th IEEE International Conference on Electronics, Circuits and Systems, 2003. ICECS 2003. Proceedings of the 2003*, pages 715–718, Sharjah, United Arab Emirates, 2003. IEEE.

[6] André Borrmann, Julian Amann, Tim Chipman, Juha Hyvärinen, Thomas Liebich, Sergej Muhič, Laura Mol, Jim Plume, and Paul Scarponcini. IFC Infra Overall Architecture Project: Documentation and Guidelines. Technical report, buildingSMART International, 2017. (Accessed: 2020-05-05).

[7] André Borrmann. *Building information modeling: technology foundations and industry practice.* Springer Berlin Heidelberg, 2018.

[8] André Borrmann, Sergej Muhic, Juha Hyvärinen, Tim Chipman, Stefan Jaud, Christophe Castaing, Claude Dumoulin, Thomas Liebich, and Laura Mol. The IFC-Bridge project – Extending the IFC standard to enable high-quality exchange of bridge information models. pages 377–386, July 2019.

[9] buildingSMART International. IfcAlignment instance diagram. http://docs.buildingsmartalliance.org/IFC4x2_Bridge/link/alignment-attributes.htm. (Accesed: 2020-09-02).

[10] buildingSMART International. IfcAlignment2DHorizontalSegment. https://standards.buildingsmart.org/IFC/DEV/IFC4_2/FINAL/HTML/schema/ifcgeometricconstraintresource/lexical/ifcalignment2dhorizontalsegment.htm. (Accesed: 2020-09-04).

[11] buildingSMART International. IfcAlignment2DVertticalSegment. https://standards.buildingsmart.org/IFC/DEV/IFC4_2/FINAL/HTML/schema/ifcgeometricconstraintresource/lexical/ifcalignment2dverticalsegment.htm. (Accesed: 2020-09-04).

[12] buildingSMART International. IfcAlignmentCurve. https://standards.buildingsmart.org/IFC/DEV/IFC4_2/FINAL/HTML/schema/ifcgeometricconstraintresource/lexical/ifcalignmentcurve.htm. (Accesed: 2020-02-15).

[13] buildingSMART International. IfcSectionedSolidHorizontal. https://standards.buildingsmart.org/IFC/DEV/IFC4_2/FINAL/HTML/schema/ifcgeometricmodelresource/lexical/ifcsectionedsolidhorizontal.htm. (Accesed: 2020-02-15).

[14] buildingSMART International. IfcSolidModel. https://standards.buildingsmart.org/IFC/DEV/IFC4_2/FINAL/HTML/schema/ifcgeometricmodelresource/lexical/ifcsolidmodel.htm. (Accesed: 2020-09-03).

[15] C. A. Coetzee. Sketch-BasedAlignmentDesign, 2019. Technische Universität München, Lehrstuhl für Computergestütze Modellierung.

[16] Sebastian Esser and Korbinian Aicher. Ifcbridge model generation using visual programming. In *31. Forum Bauinformatik: 11.–13. September 2019 in Berlin. Proceedings*, page 101. Universitätsverlag der TU Berlin, 2019.

[17] Michael P. Gallaher, Alan C. O'Connor, John L. Dettbarn, Jr., and Linda T. Gilday. Cost Analysis of Inadequate Interoperability in the U.S. Capital Facilities Industry. Technical Report NIST GCR 04-867, National Institute of Standards and Technology, August 2004.

[18] Helge Hecht and Štefan Jaud. TUM OpenInfraPlatform: The Open-Source BIM Visualisation Software. 2019.

[19] Štefan Jaud, Andreas Donaubauer, Otto Heunecke, and André Borrmann. Georeferencing in the context of building information modelling. *Automation in Construction*, 118:103211, 2020.

[20] Mikael Laakso and Arto Kiviniemi. The IFC Standard - A Review Of History, Development, And Standardization.

[21] Štefan Markič. Ifc-bridge: Previous initiatives and their proposals. In *Proc. of the 29th Forum Bauinformatik, Dresden, Germany*, 2017.

[22] R. Mundani. *Bau- und Umweltinformatik 2 Vorlesung Teil 4: Lineare Transformationen.* Technische Universität München, Lehrstuhl für Bauinformatik, 2019.

[23] Matthias Weise, Peter Katranuschkov, Thomas Liebich, and Raimar J Scherer. Structural analysis extension of the ifc modelling framework. *Journal of Information Technology in Construction (ITcon)*, 8(14):181–200, 2003.

[24] Ayako Yasaka, Hiromi Kataoka, Kou Kasima, Makoto Takeda, Masato Usami, Nobuyoshi Yabuki, Norihide Matumoto, Satoru Furukawa, Yoshihiro Mogi, and Yoshinobu Adachi. The development of the reinforced-concrete structural model on ifc specification. In *Proc. of the joint international conference on computing and decision making in civil and building engineering*, pages 3116–25, 2006.

List of Figures

List of Tables

Declaration

With this statement I declare, that I have independently completed this Bachelor's thesis. The thoughts taken directly or indirectly from external sources are properly marked as such. This thesis was not previously submitted to another academic institution and has also not yet been published.

München, 15. September 2020

Mauricio P.

Carlos Mauricio Platteau

Carlos Mauricio Platteau
[redacted]