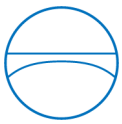
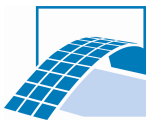




Chair of Computational Modeling and Simulation
Technical University of Munich
Department of Civil, Geo and Environmental Engineering

Development of an IFC Alignment Import/ Export Plug-In for Autodesk AutoCAD Civil 3D

Felix Rampf

Bachelorthesis
for the Bachelor of Science Program of Study Civil Engineering

Author:	Felix Rampf
Student Number:	03646809
Supervisor:	Prof. Dr.-Ing. André Borrmann
Advisor:	Julian Amann

Submission Date: 26. September 2017

CHAIR OF COMPUTATIONAL MODELING AND SIMULATION
TECHNICAL UNIVERSITY OF MUNICH
Department of Civil, Geo and Environmental Engineering

Development of an IFC Alignment Import/Export Plug-in for Autodesk AutoCAD Civil 3D

Entwicklung eines IFC Alignment Import/Export Plug-in's für Autodesk AutoCAD Civil 3D

Felix Rampf

Bachelorthesis

for the Bachelor of Science Program of Study Civil Engineering

Abstract

This thesis aims to clarify the different steps and thoughts involved in the programming process of an IFC Import/Export Plug-in exemplified within the Autodesk AutoCAD Civil 3D environment.

It begins with the basic principles of designing a horizontal and vertical road alignment including various transition curve types. Thereafter, this thesis is dedicated to describing the core data scheme of the IFC Alignment 1.1 extension that will be implemented in the coming IFC standard.

Based on a fully working plug-in for Civil 3D exemplified with several code snippets the main part gives an example of how to write a plug-in that allows the user to import and export IFC files within a given software environment.

Chapter 5 is dedicated to outline a few possible implementation errors that result from some of the definitions given by bSI on their official website.

Finally this paper will give a brief prospect of further developments and opportunities for the implementation of the IFC data schema.

Contents

1 Introduction **1**
1.1 Historical Background 1
1.2 Why Industry Foundation Classes? 2

2 Theoretical Practice **3**
2.1 Classification of Road Alignment in the Planning Process of a Road Project . 3
2.2 Description of Alignment Elements 3
2.2.1 Surface Model 3
2.2.2 Horizontal Alignment 6
2.2.3 Vertical Alignment 10

3 Mapping of Alignments onto IFC **15**
3.1 Background and current Development State 15
3.2 IFC Alignment - Conceptual Model 15
3.3 Surface Model 16
3.4 Horizontal Alignment 17
3.4.1 IFC Line Segment 18
3.4.2 IFC Circular Arc Segment 19
3.4.3 IFC Transition Curve Segment 20
3.4.4 IFC Clothoidal Arc Segment 26
3.5 Vertical Alignment 27
3.5.1 IFC Gradient Segment 28
3.5.2 IFC Vertical Circular Arc Segment 28
3.5.3 IFC Vertical Parabolic Arc Segment 28

4 Programming an IFC Import/Export Plug-in **31**
4.1 Development Environment 31
4.2 Platform and Limitations 31
4.3 Current Development State 33
4.4 The Civil 3D Plug-In 33
4.4.1 Overview 33

4.4.2 Preparation 34
4.4.3 The Example Alignment 35
4.4.4 Creation of the Plug-In Button 36
4.4.5 Export Function 38
4.4.6 Import Function 46
4.4.7 Results 52

5 Programming Challenges and Critics **55**
5.1 The StartDirection Property 55
5.2 Import Function 57

6 Conclusion **63**
6.1 Prospect 63
6.2 Resolution 63

A Tables **65**

List of Abbreviations

TUM	Technical University of Munich
CSTB	Centre Scientifique et Technique du Bâtiment
bSI	buildingSMART International
AEC3	AEC3 Deutschland GmbH
OGC	Open Geospatial Consortium
ISO	International Organization for Standardization
IFC	Industry Foundation Classes
BIM	Building Information Modelling
C3D	Autodesk AutoCAD Civil 3D (2018)
OIP	TUM Open Infra Platform
DTM	Digital Terrain Model
TIN	Triangulated Irregular Network
API	Application Programming Interface
STEP	Standard for the Exchange of Product Model Data
C#	C Sharp Programming Language designed for the Common Language Infrastructure
***.dwg**	File Extension specifying a Autodesk genuine file (.dwg from *Drawing*)
***.ifc**	File Extension specifying an IFC file
***.dll**	File Extension specifying an Dynamic Link Library file
NSW	New South Wales

Acknowledgement

This bachelor's thesis formed with the support of my tutor MSc. Julian Amann who assisted me with answers and advice on occurring questions and challenges throughout the writing process.
A special thanks shall be addressed to Peter Bonsma from RDF ltd. who was always reachable via email and gave me highly valuable advice on software development.
Further gratitude goes to Senior Lecturer MSc. Roger Bird from Newcastle University, answering emails and giving further explanations on transition curves, even though I was only an Erasmus student at Newcastle for one year.
Finally the community in the Autodesk AutoCAD Civil 3D developers forum, and Jeff Mishler in particular, were a great help supporting me with developing the plug-in.

Chapter 1

Introduction

1.1 Historical Background

The wide range of building softwares using different file formats, which are quite frequently not compatible with one another, creates a challenge for civil engineers. Unlike other industries there are no legal regulations controlling the use of specific software in order to achieve optimum interoperability between the different sections such as planning offices, structural analysts, building services engineers and maintenance companies working on a big construction project. This results in every section using their own programs that are highly functional for the sector's own purpose and specifically developed for the particular industry. (Borrmann *et al.*, 2015, p. 79) Therefore, a lot of the data has to be manually entered many times for each development state constituting a serious source of errors.
To avoid these problems and to guarantee a better workflow the Building Information Modelling (BIM) principle has been established, which is more and more used for construction programs.
The scope of BIM is a continuous usage of a digital building model that serves as the foundation for all development states and sections involved in the construction process (Borrmann *et al.*, 2015, pp. 83). With the spreading of BIM and its growing importance and application, it is now more important than ever to develop, improve and establish an international standard for a working file format that is highly regulated and accessible for everyone.
On the initiative of Autodesk, a private alliance of twelve companies has been formed in 1995 to prove the benefits of interoperability between different software programs being used in the building industry. They came to the three critical conclusions that

- *"interoperability [is] viable and [has] great commercial potential(...)[,]*

- *any standards must be open and international, not private or proprietary(...)[and]*

- *that the alliance must open its membership to interested parties around the globe."* (buildingSMART International, 2017)

In 2008 the alliance was renamed to buildingSMART International (bSI) in order "to better reflect the nature and goals of the organization" (buildingSMART International, 2017). Furthermore, they complied with the industry's need for a standardized file format and developed the Industry Foundation Classes (IFC), which is a comprehensive format for sharing and exchanging BIM data independently from the software manufacturer. The first version of IFC was published in 1997 as *"IFC 1.0"* reaching to the current *"IFC 4.0"* standard (published in 2013) after many development steps (figure 1.1).

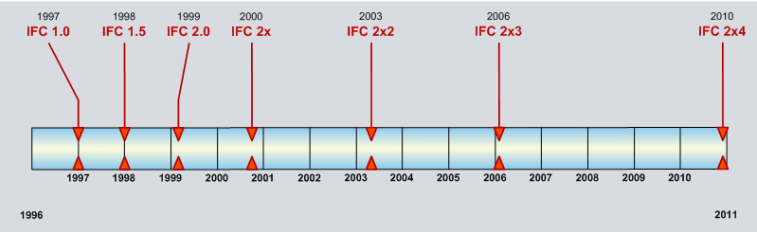


Figure 1.1: Releases of different IFC versions until 2011 (Borrmann *et al.*, 2015, p. 85)

1.2 Why Industry Foundation Classes?

Before the existence of bSI, groups of engineers and scientists attempted to develop file formats similar to IFC but their approaches were usually limited to mapping geometric data. The result was the neglect of the second important part of the application of BIM, the semantics – these are essential for the data exchange between different sections – making the result unattractive for exchange scenarios. IFC files first combine both, the geometric and the semantic, aspects of BIM data. This makes it a highly complex data schema, which requires a detailed implementation approach.
However, due to an "additional standardization as an International Organization for Standardization (ISO) standard" (Borrmann *et al.*, 2015, p. 85) IFC is more attractive than ever before. Many countries have already made the IFC format compulsory for public construction projects with Singapore, Finland, the US and Great Britain being pioneers. Although IFC has so far been focusing on mapping data for buildings, the IFC Alignment 1.0, as well as IFC Alignment 1.1 extension and future implementation of further infrastructure data such as cross sections, bridges and tunnels, indicate the considerable potential of IFC.

Chapter 2

Theoretical Practice

2.1 Classification of Road Alignment in the Planning Process of a Road Project

A new road project is divided into several technical and legal stages that are interconnected. In general, the planning process starts with the identification of needs and the issuing of a planning assignment (Natzschka, 1997, p. 28). After that, it is important to provide all the necessary documents and traffic data to determine the circumstances under which the road will be designed.
The next stage is the actual road design resulting in a horizontal and vertical alignment. After several legal stages (i.e. consultation of the public) the road will be built and finally opened to the public.

2.2 Description of Alignment Elements

In order to obtain a basic understanding of the tasks and challenges involved to establish a standard format for road alignment projects such as the IFC Alignment extension it is most important to understand the different geometric shapes used in modern road alignment design. The following paragraphs, therefore, describe these basic elements.

2.2.1 Surface Model

The base of every future road alignment – but also the greatest determining factor – is the terrain on which the road will be built.
With the use of modern computer programs, starting in the middle of the 20th century, civil engineers base their design on a Digital Terrain Model (DTM). (Li *et al.*, 2005, p. 4)

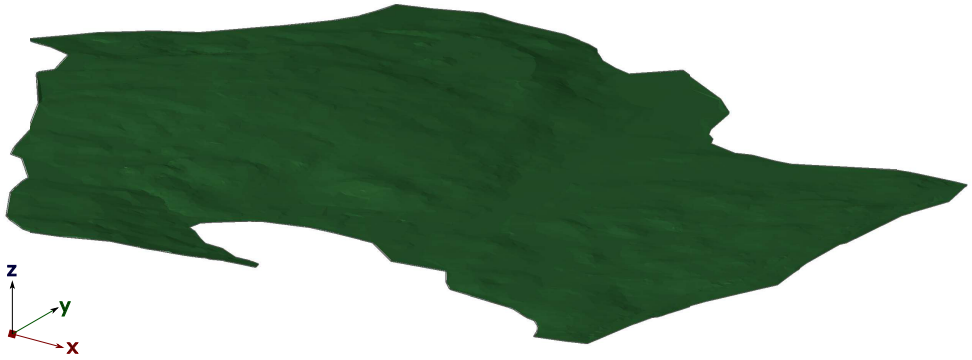


Figure 2.1: A TIN surface created with C3D shown in a realistically rendered view

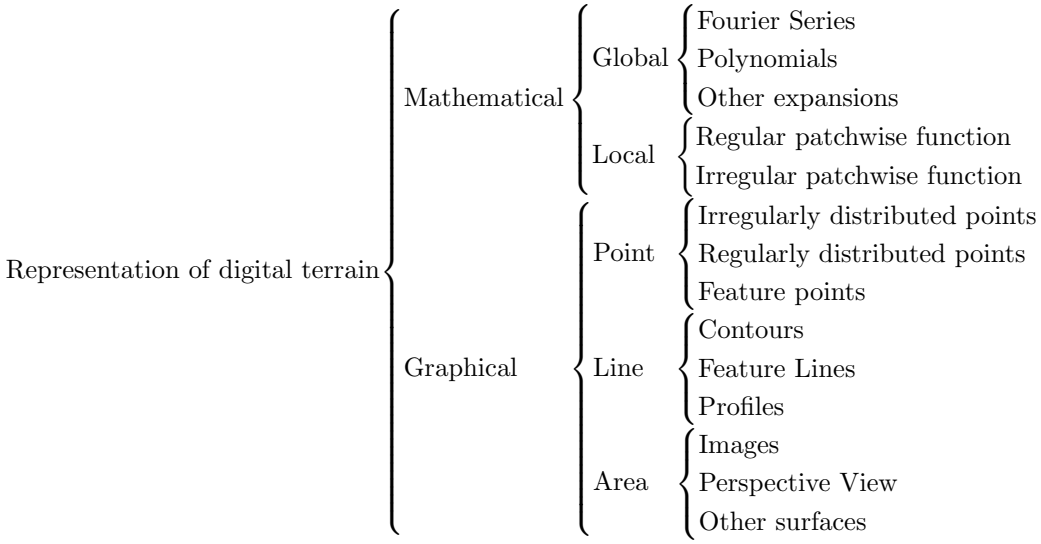


Figure 2.2: Different representation schemes of digital terrain surfaces adapted from Natzschka (1997, p. 4)

Such surface models can be represented mathematically and graphically. Figure 2.2 shows the different representations of a digital terrain model. (Li *et al.*, 2005, p. 4)
In the construction of a DTM surface "points are sampled from the terrain to be modelled with a certain observation accuracy, density, and distribution; the terrain surface is then represented by the set of sample points". (Li *et al.*, 2005, p. 9)

The final DTM surface is then created from the sampled data points by using interpolation. Figure 2.3 shows the different stages within the digital terrain modelling process. (Li *et al.*, 2005, p. 10)

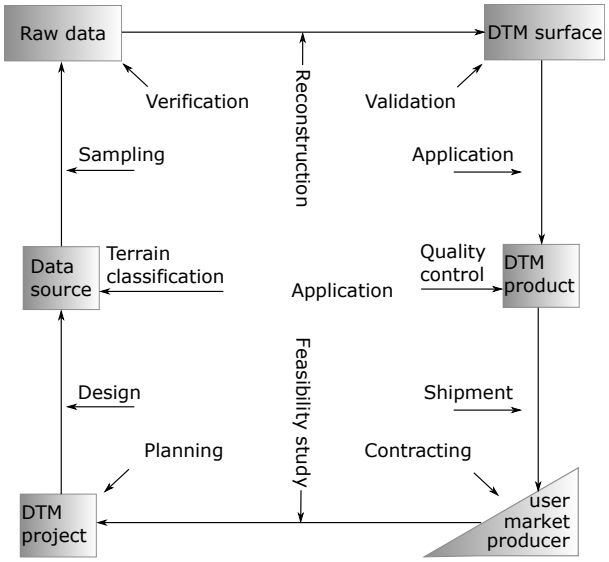


Figure 2.3: Steps involved in the process of creating a DTM surface model adapted from Li *et al.* (2005, p. 10)

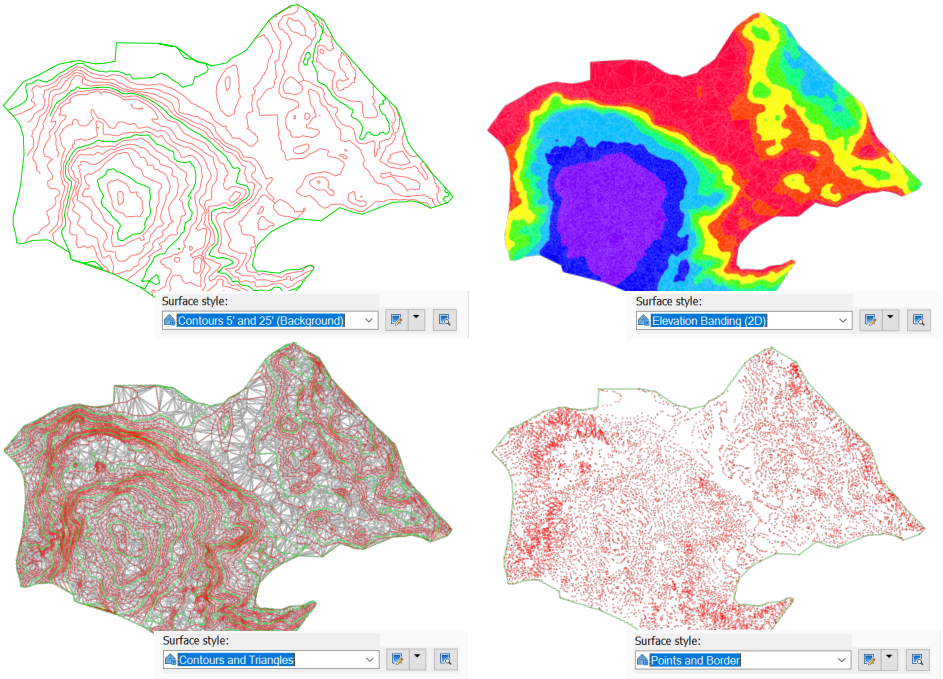


Figure 2.4: Four different types of illustrating a TIN surface in C3D

2.2.2 Horizontal Alignment

The horizontal alignment results in a road layout from the 'bird's eye view'. It consists of the following elements: **Straight Section**, **Circular Arcs** and **Transition Curves**, which are described in more detail in the following.

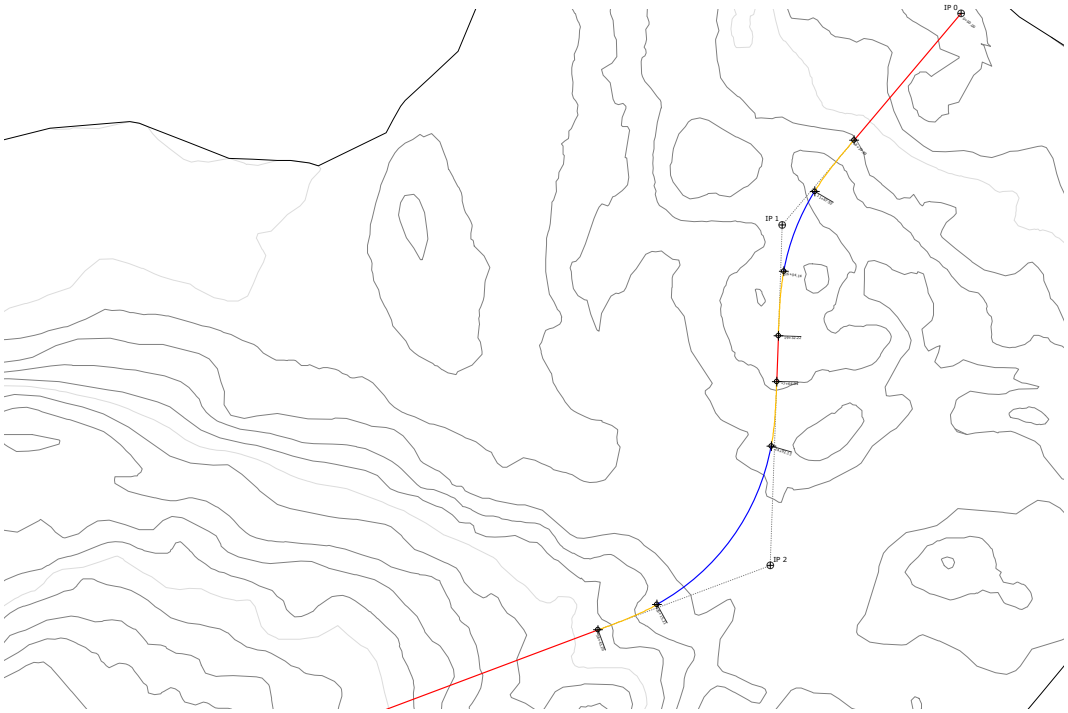


Figure 2.5: Horizontal road alignment including straight, transition, and arc sections (created in C3D)

Straight Line Section

The **Straight Line Section** defines the shortest connection between two points. Advantages of straight lines are:

- Good sight conditions for right-angle crossings
- Easier adaptation to local conditions (plains, valleys, train tracks, buildings)

However, they are hardly used in road design due to their many disadvantages such as:

- Aggravation of the estimation of oncoming traffic's speed and distance
- Temptation to speed
- Increased monotony and tiredness

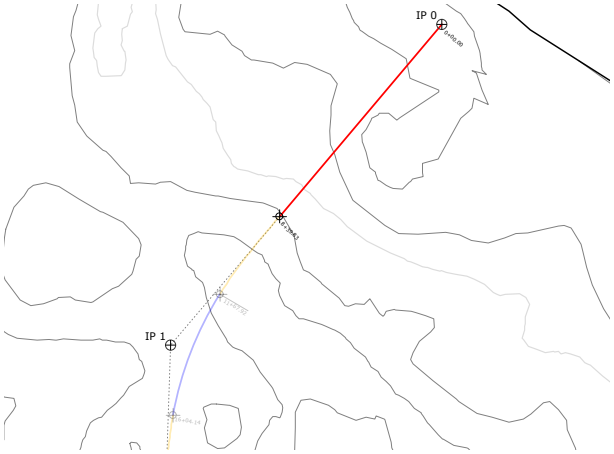


Figure 2.6: Straight section of a horizontal alignment

- Increased blinding at night due to oncoming traffic

Accordingly, it is recommended to limit the maximum length of straight lines to 2000 m on motorways and 1500 m on rural roads whenever possible. (Freudenstein, 2016, p. II/13)

Circular Arc Section

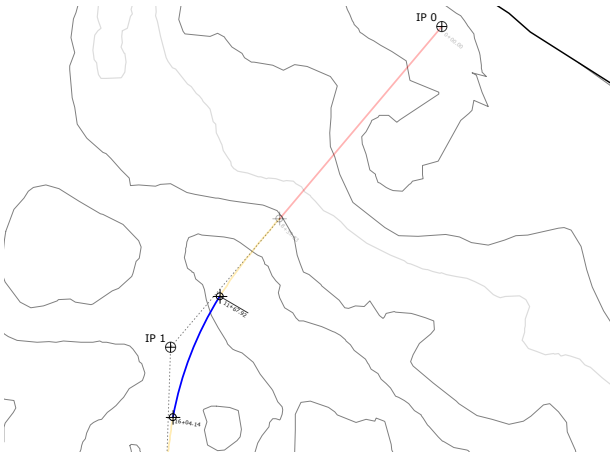


Figure 2.7: Circular arc section of a horizontal alignment

The **Circular Arc Section** allows a change in direction within the alignment. It also provides a possibility to adapt to the environmental situation. Besides, an arc is easy to handle from the mathematical perspective with its basic form:

$$x^2 + y^2 = R^2 \tag{2.1}$$

The minimum radius derives from the frictional force transferred from the car tires onto the road, which is explained by figure 2.8 and equation 2.2. (Freudenstein, 2016, p. II/14)

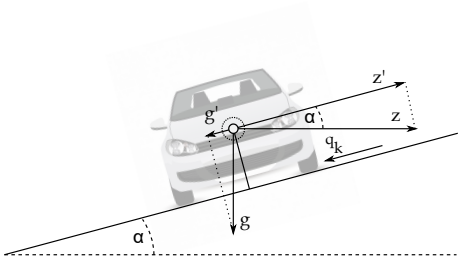


Figure 2.8: Derivation of the minimum radius R_{min} recommended for a circular arc curve adapted from Freudenstein (2016, p. II/14)

$$
\begin{aligned}
&R_{min} = \frac{V^2}{3.6^2 * g * (0.925 * f_T * n + q)} \quad [m] \\
&with: \\
&V = curve\ design\ speed \quad [\frac{km}{h}] \\
&g = acceleration\ of\ gravity \quad [\frac{m}{s^2}] \\
&n = maximum\ utilisation\ of\ radial\ adhesive\ coefficient \quad [-] \\
&q = slope \quad [-] \\
&f_T = tangential\ adhesive\ coefficient\ based\ on\ tables \quad [-]
\end{aligned}
\tag{2.2}
$$

To guarantee good driving dynamics it is not only important that the curvature does not change within a curve: $k = 1/r = const.$, but also that the radii R_1 and R_2 of two following curves do not differ too much. Figure 2.9 shows the recommended radii relations. (Freudenstein, 2016, p. II/16)

Transition Curves

A **Transition Curve** is used between a straight section and an arc or connecting two following arcs to prevent a sudden change in lateral acceleration, which provides higher driving comfort and reduces the risk of load shifting.
In road alignment clothoids are mostly used; these are spiral curves with a constant rate of change of curvature and radial acceleration (the radial acceleration is a function of the radius). Railway design engineers often make use of bloss curves (3.4.3) instead.

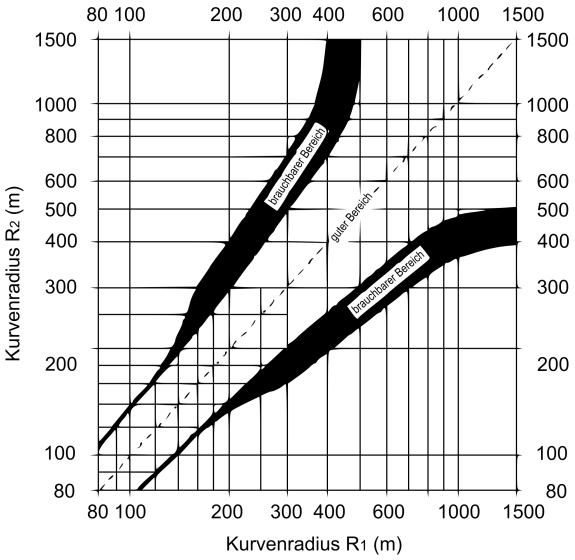


Figure 2.9: Relation of successive curve radii reprinted from (Freudenstein, 2016, p. II/16)

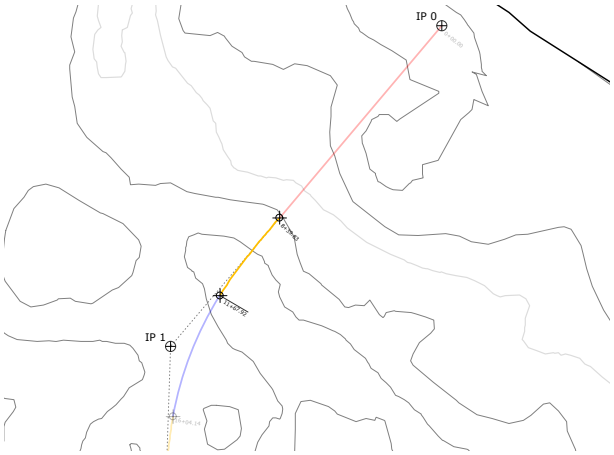


Figure 2.10: Transition curve section of a horizontal alignment

The product of a clothoid's length L and the radius R is always constant and described by the variable A (figure 2.11).

$$A^2 = R * L \quad [m^2] \tag{2.3}$$

The variable A is called *flatness* or *homothetic parameter* and works as a stretching factor for the clothoid.

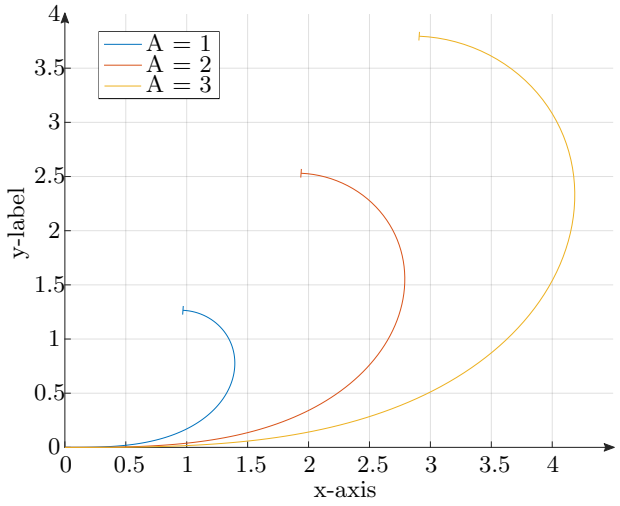


Figure 2.11: Three clothoids with different clothoid parameters A

For the design of a road alignment the length of the transition curve is calculated with the following formula (Bird, 2015b, p. 15):

$$
\begin{aligned}
&Transition\ Length = \frac{V^3}{46.656 * q * R} \quad [m] \\
&with: \\
&V = curve\ design\ speed \quad [\frac{m}{s^2}] \\
&q = change\ in\ radial\ acceleration \quad [\frac{m}{s^3}] \\
&R = radius \quad [m]
\end{aligned}
\tag{2.4}
$$

Due to safety reasons and to provide the driver with adequate reaction time the minimum A *value* A_{min} should not be less than $A_{min} = \frac{R}{3}$ but also not exceed $A_{max} = R$. (Freudenstein, 2016, p. II/21)

2.2.3 Vertical Alignment

In order to avoid excessive environmental intrusions, it is also important to take the resulting levels of the road alignment into account throughout the horizontal planning process. The vertical alignment planning generally follows the same principles as the horizontal alignment. Therefore, it is recommended to design the vertical and horizontal alignment parallel in sections. (Natzschka, 1997, p. 125)
The result of the vertical alignment is a 'profile view' of the road alignment that consists of **Gradient Sections**, **'Crest'** and **'Sag' Curves**.

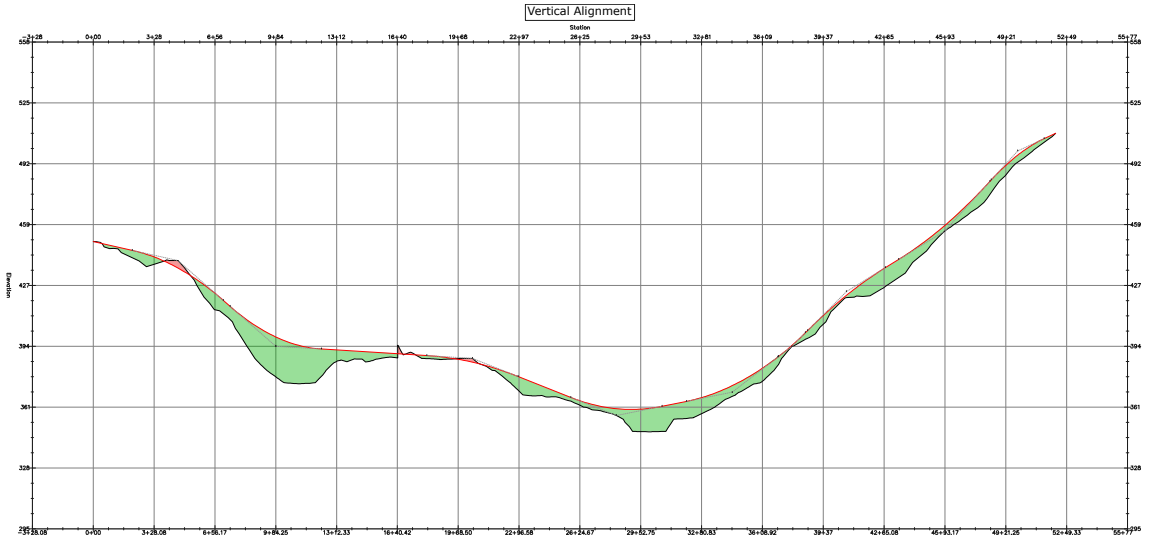


Figure 2.12: Vertical road alignment – The scale of the chainage is the same as it is in the horizontal alignment plan whilst the elevation levels are shown with an exaggeration factor of 10 (created in C3D)

Gradient Section

The **Gradient Section** is a piece of road alignment with a constant gradient. A low gradient improves the driver's security by providing a higher sight distance. Additionally, it also results in better traffic quality and road capacity. Another advantage is the lower road operating cost and costs for the drivers, which also lower the emissions. (Freudenstein, 2016, p. II/22) A higher gradient, therefore, improves the ability to adapt to the terrain, which again results in lower construction costs.
The gradient g between two vertical intersection points is given by the following formula (Natzschka, 1997, p. 127):

$$
\begin{aligned}
& g = \frac{\Delta h * 100}{l} \quad [\%] \\
& with: \\
& \Delta\, h = difference\ in\ level\ between\ the\ two\ intersection\ points \quad [m] \\
& l = horizontal\ distance\ between\ the\ two\ intersection\ points \quad [m]
\end{aligned}
\tag{2.5}
$$

Due to traffic safety reasons (in Germany) the gradient should not exceed the values given in table 2.1. To avoid sections of weak outflow it is advised to always plan with a minimum gradient of $g \geq\ 1.0\%$ ($better\ 1.5\%$) in areas of higher winds. (Freudenstein, 2016, pp. II/22)

Autobahnen		Landstrassen	
Entwurfsklasse	max. g (%)	Entwurfsklasse	max. g (%)
EKA 1 A	4.0	EKL 1	4.5
EKA 1 B	4.5	EKL 2	5.5
EKA 2	4.5	EKL 3	6.5
EKA 3	6.0	EKL 4	8.0

Table 2.1: Maximum gradient depending on the different Entwurfsklassen (road classes) in Germany adapted from Freudenstein (2016, p. II/23)

'Crest' and 'Sag' Curves

The necessary transition curve between the vertical intersection points is created by a basic square parabola.
Each vertical curve is defined by its $K value$, which gives the length in metres for a 1% change in gradient. Hence the length of the vertical curve is calculated by:

$$\begin{aligned} &L = K * (g_2 - g_1) * 100 \quad [m] \\ &with: \\ &g_2 = gradient\ of\ the\ leaving\ tangent \quad [-] \\ &g_1 = gradient\ of\ the\ entering\ tangent \quad [-] \end{aligned} \tag{2.6}$$

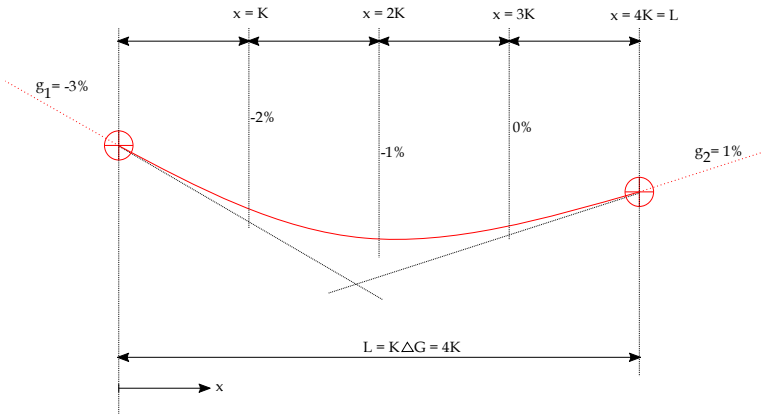


Figure 2.13: 'Sag' curve with a K *value* of $L/4$ adapted from Bird (2015c, p. 22)

The level h_x at any point of the vertical curve is given by the following formula (Bird, 2015c, p. 25):

$$
\begin{aligned}
&h_x = h_s + g_1 * x + \frac{x^2}{200 * K} \quad [m] \\
&with: \\
&h_s = starting\ height\ of\ the\ parabola \quad [m] \\
&g_1 = starting\ gradient \quad [-] \\
&x = horizontal\ distance\ along\ the\ curve \quad [m]
\end{aligned}
\tag{2.7}
$$

For a 'crest' curve the *K value* (figure 2.14) is negative, whilst for a 'sag' curve it takes a positive value (figure 2.15). (Bird, 2015c, p. 30)

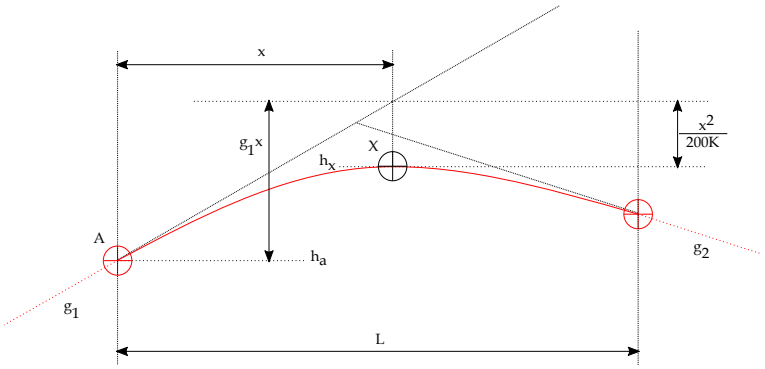


Figure 2.14: Derivation of the equation to calculate the vertical level at any point on the vertical alignment parabola of a 'crest' curve adapted from Bird (2015c, p. 22)

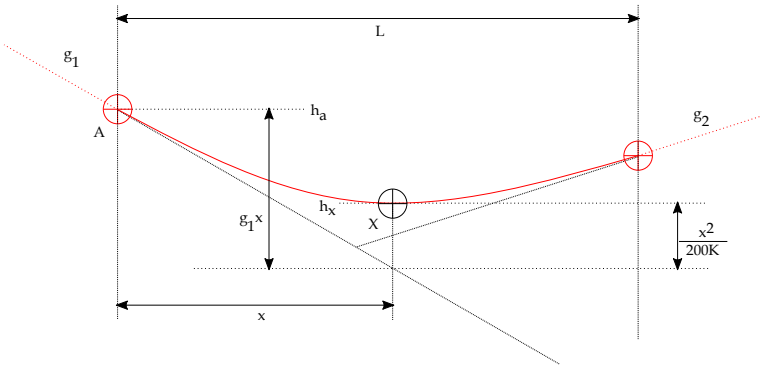


Figure 2.15: Derivation of the equation to calculate the vertical level at any point on the vertical alignment parabola of a 'sag' curve adapted from Bird (2015c, p. 31)

The *K value* has to be chosen in a way to provide the driver with adequate sight distance. It also should not create optical illusions within the curves. Therefore there are tables giving

the minimum *K values* and vertical tangent lengths for different road types. (Bird, 2015c, p. 33)

Chapter 3

Mapping of Alignments onto IFC

3.1 Background and current Development State

Although the IFC standard has been developed and widely improved over the last couple of years it has not yet provided an option to map alignments such as those used for roads, bridges, and tunnels. In a first step, Rijkwaterstaat (Netherlands) and Trafikverket (Sweden) started the IFC Alignment project to develop an IFC Alignment 1.0 data scheme in order to map alignments onto the IFC data format. The project team consisted of several international institutions and companies such as the AEC3 Deutschland GmbH (AEC3), Bentley Systems, the French Centre Scientifique et Technique du Bâtiment (CSTB), Open Geospatial Consortium (OGC) and the Technical University of Munich (TUM). (Liebich *et al.*, 2015a, p. 4)
In February 2015 the data model officially reached the 'Candidate Standard' status and successfully passed the public reviewing process. After several updates, the IFC Alignment 1.1 version has now reached the rank of an official standard – the bSI Final Standard – and will therefore be integrated into the coming release of the IFC data scheme. (buildingSMART International, 2017)
All of IFC's class and property definitions were taken from the bSI website (buildingSMART International, 2015). For a better understanding classes will be printed bold whilst properties italic in the following.

3.2 IFC Alignment - Conceptual Model

The basic concept of the **IFC Alignment 1.1** data model follows the widely used principle of separating the road alignment creation process into a horizontal and vertical alignment part. These parts are then specified further by multiple alignment segments which are described in

the following. Thus the generic entity **IfcAlignment** will be implemented in the IFC data scheme. (Liebich, 2017, p. 2)

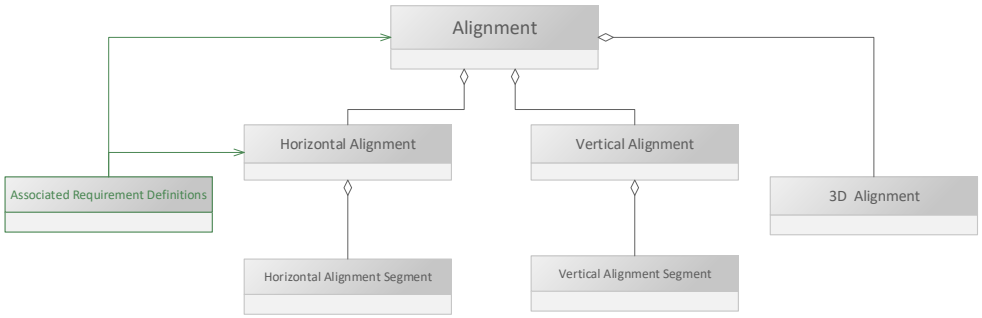


Figure 3.1: Basic data scheme for the IFC Alignment that will be integrated in the next IFC standard adapted from Liebich *et al.* (2015a, p. 9)

Liebich states:

> *The single alignment is a non-branching, continuous, single location and single alternative alignment but it can be grouped into an alignment system, and two different alignments can be related to each other. (Liebich et al., 2015b, p. 9)*

Accordingly, one Alignment may consist of (Liebich *et al.*, 2015a, p. 11):

- a horizontal alignment, one vertical alignment and a resulting 3D alignment
- a horizontal alignment and one vertical alignment
- a horizontal alignment only
- or a 3D alignment only

The various alignment segments of the horizontal and vertical alignment share some properties such as starting position (*StartPoint*), start direction (*StartDirection*) and segment length (*SegmentLength*) hence the parent class **IfcCurveSegment2D** has been added from which the alignment segments inherit named properties.

3.3 Surface Model

To map the digital surface model onto an IFC file the **IfcTriangulatedFaceSet** is used and connected to the **IfcGeographicElement** entity that represents surface data. The creation

principle is a simple triangulation between given points.
Within the interface of C3D the user can specify **breaklines** on a TIN surface (the breaklines show in the **IfcTriangulatedIrregularNetwork** as the property *flags*).
A breakline is used to define features, such as retaining walls, curbs, tops of ridges, and streams. The Autodesk website further explains, "breaklines force surface triangulation along the breakline preventing triangulation across the breakline". (Autodesk, 2016)

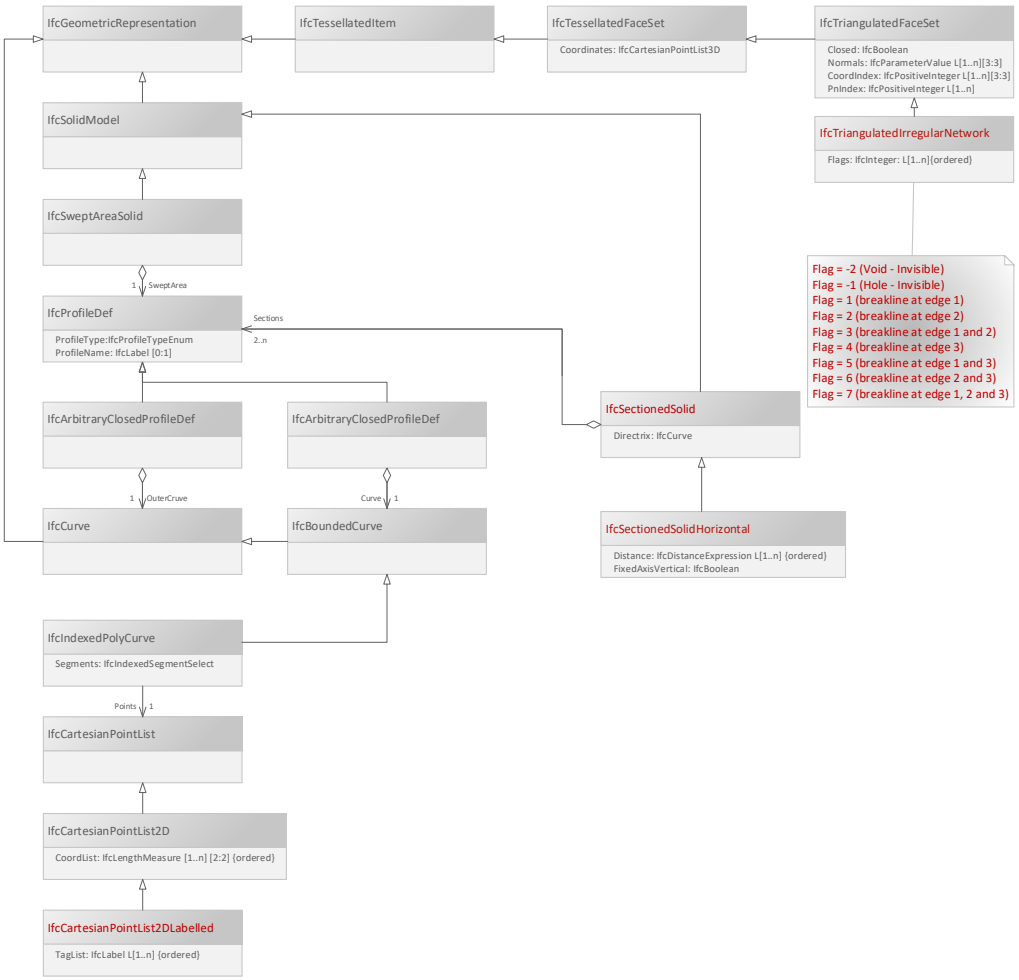


Figure 3.2: Class diagram of the IFC surface model part (red entities were added in course of the IFC Alignment 1.1 update) adapted from Liebich (2017, p. 14)

3.4 Horizontal Alignment

The horizontal alignment class (**IfcAlignment2DHorizontal**) exists of an ordered list of alignment segments (**IfcAlignment2DHorizontalSegment**) included in the horizontal alignment. These segments can be:

- **IfcLineSegment2D**
- **IfcCircularArcSegment2D**
- **IfcClothoidalArcSegment2D**
- **IfcTransitionCurveSegment2D**

To describe the connection between two consecutive alignment segments the **IfcAlignment2DHorizontalSegment** additionally has the property *TangentialContinuity* to the segments. The *TangentialContinuity* determines whether the following segment should be tangential to the previous element or not (*TangentialContinuity* = *true*: The following segment is tangential, *TangentialContinuity* = *false*: The following segment is not tangential).

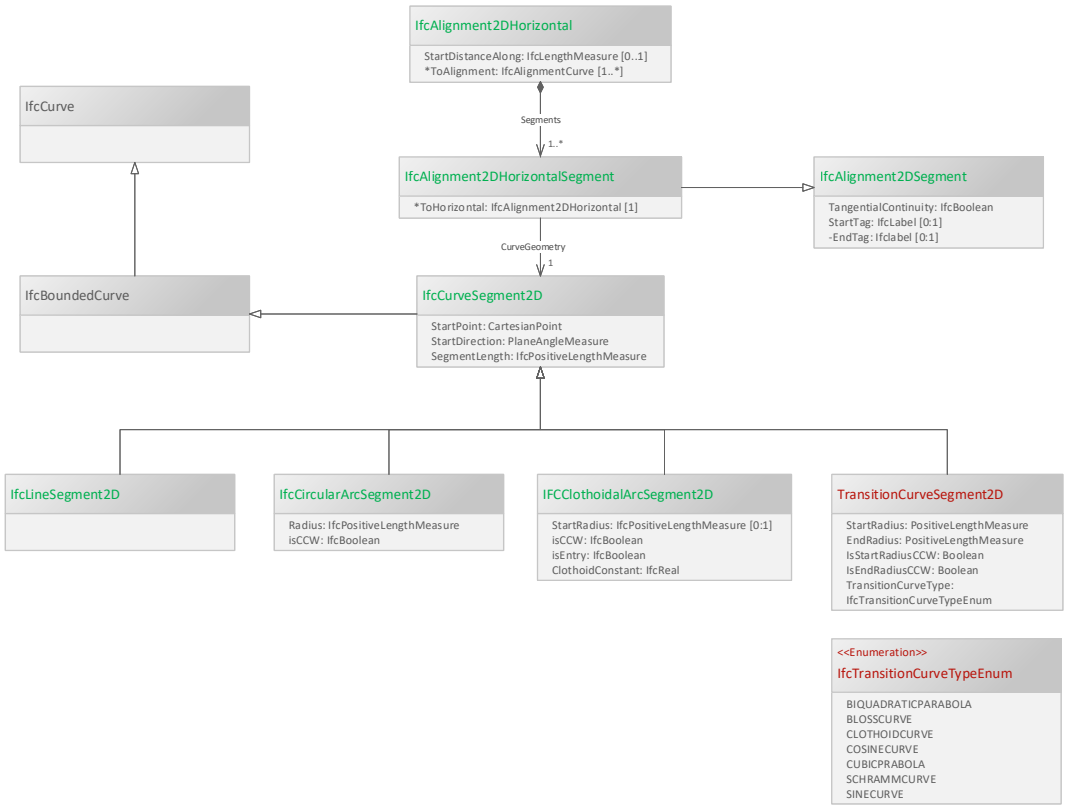


Figure 3.3: Class diagram of the IFC horizontal alignment part (red entities were added in course of the IFC Alignment 1.1 update, green entities represent the IFC Alignment 1.0 extension) adapted from Liebich (2017, p. 14)

3.4.1 IFC Line Segment

The **IfcLineSegment2D** has the properties *StartPoint*, *StartDirection*, and *SegmentLength* all inherited from the **IfcCurveSegment2D** as mentioned earlier. The *StartPoint* is specified by the entity **IfcCartesianPoint**, which is a 2D point defined by Cartesian coordinates. The

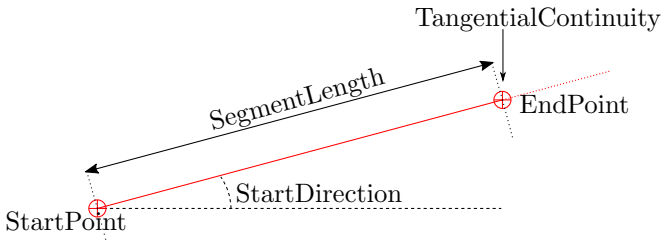


Figure 3.4: IFCLineSegment2D

StartDirection is a single value defining the line's rotation from the x-Axis (1,0). It is a plane angle described in a counter-clockwise order usually measured in radian (rad, $\frac{m}{m}$). Section 5.1 gives further explanation on how the property is determined. Because the *SegmentLength* specifies the length of the segment it should therefore not be negative to avoid problems (the entity *IfcPositiveInteger*, which is the type of the *SegmentLength* property ensures that).

3.4.2 IFC Circular Arc Segment

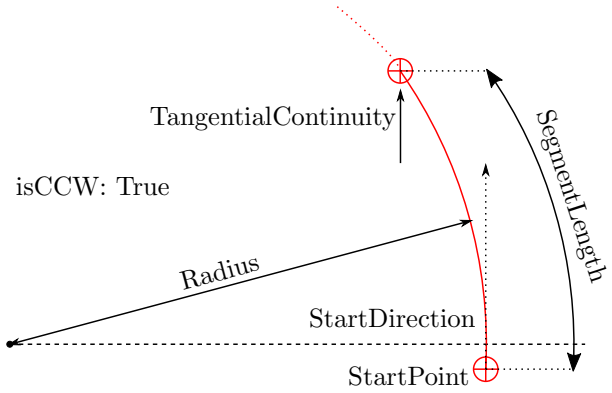


Figure 3.5: IfcCircularArcSegment2D

Just like the **IFCLineSegment2D** the **IFCCircularArcSegment2D** inherits all the properties from **IFCAlignment2DHorizontalSegment** but also adds the properties:

- *Radius*
- *IsCCW*

Radius is a positive length measure (*IfcPositiveLengthMeasure*) giving the radius of the arc segment. The *IsCCW* property describes the direction of the curvature with a boolean expression being either *'true'* or *'false'* (*'true'*: counter-clockwise or 'to the left', *'false'*: clockwise or 'to the right').

3.4.3 IFC Transition Curve Segment

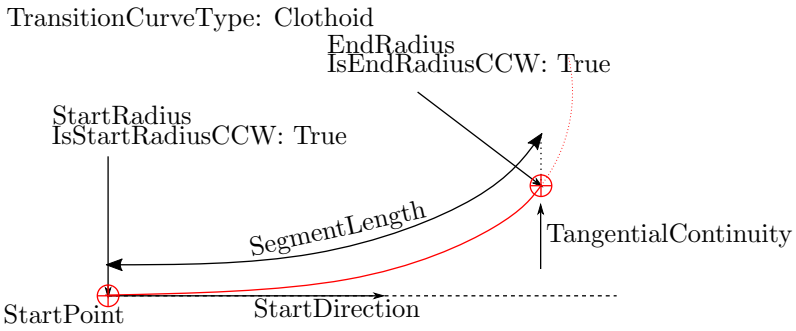


Figure 3.6: IfcTransitionCurve2DSegment

Unlike the previous version, the new IFC Alignment 1.1 extension now also allows for the implementation of different types of transition curves to be used for the horizontal alignment. (Liebich, 2017, p. 10) Therefore besides the previously used **IfcClothoidalArcSegment2D** class a new overall **IfcTransitionCurveSegment2D** class has been implemented that is further described by the property *TransitionCurveType*. This property is filled with the variable *IfcTransitionCurveTypeEnum* of the type enumeration, allowing the following types of transitional curves:

- *Biquadratic Parabola*
- *Blosscurve*
- *Clothoidcurve*
- *Cosinecurve*
- *Cubic Parabola*
- *Schramm Curve*
- *Sinecurve*

Because this paper is mainly dedicated to the creation of an IFC plug-in for C3D it is important to point out that C3D only offers:

- *Clothoid*
- *Bloss Spiral*
- *Sinusoidal Curve*
- *Sine Half-Wavelength Diminishing Tangent Curve*

- *Cubic Parabola*

- *Cubic (JP)*

- *NSW Cubic Parabola*

- *Biquadratic Parabola (Schramm Curve)*

The **IfcClothoidalArcSegment2D** class is still used to ensure downward compatibility but it might be removed in a future version of the IFC alignment extension. (Liebich, 2017, p. 14) To complete the documentation of the IFC Alignment 1.1 extension the different items in the *IfcTransitionCurveTypeEnum* list will be briefly described in the following. NSW Cubic Parabola, used in New South Wales Australia, or Cubic (JP) from Japan are not yet included in the IFC Alignment 1.1 extension. (Autodesk, 2016)

Getting the Design Parameters

Before designing any of the following transition curves the desired curve radius R, the curve design speed V, and the $q\,value$ (Change of lateral acceleration while driving the curve) have to be defined. In the following step, the spiral parameter A is being calculated. A good formula to get a suitable $A\,value$ is (Bird, 2015c, p. 2) (also see section 2.2.2):

$$A = \frac{V^3}{46.656 * q} \quad [m^2] \tag{3.1}$$

From the $A\,value$ the transition curve length L can be derived:

$$A^2 = R * L \rightarrow L = \frac{A^2}{R} \quad [m] \tag{3.2}$$

For figure 3.7 1000 evaluation points (*numberOfEvaluationPoints = 1000*) have been used, so that the length measure along the transition curve l reaches from *0* to L in 999 (*numberOfEvaluationPoints-1*) equidistant steps. At the use of around *numberOfEvaluationPoints = 700* the value of the turning angle α (used in 3.4.3) does not change significantly anymore, so that *numberOfEvaluationPoints = 1000* gives sufficient accuracy for the transformation used to calculate the Bi-quadratic Parabola.

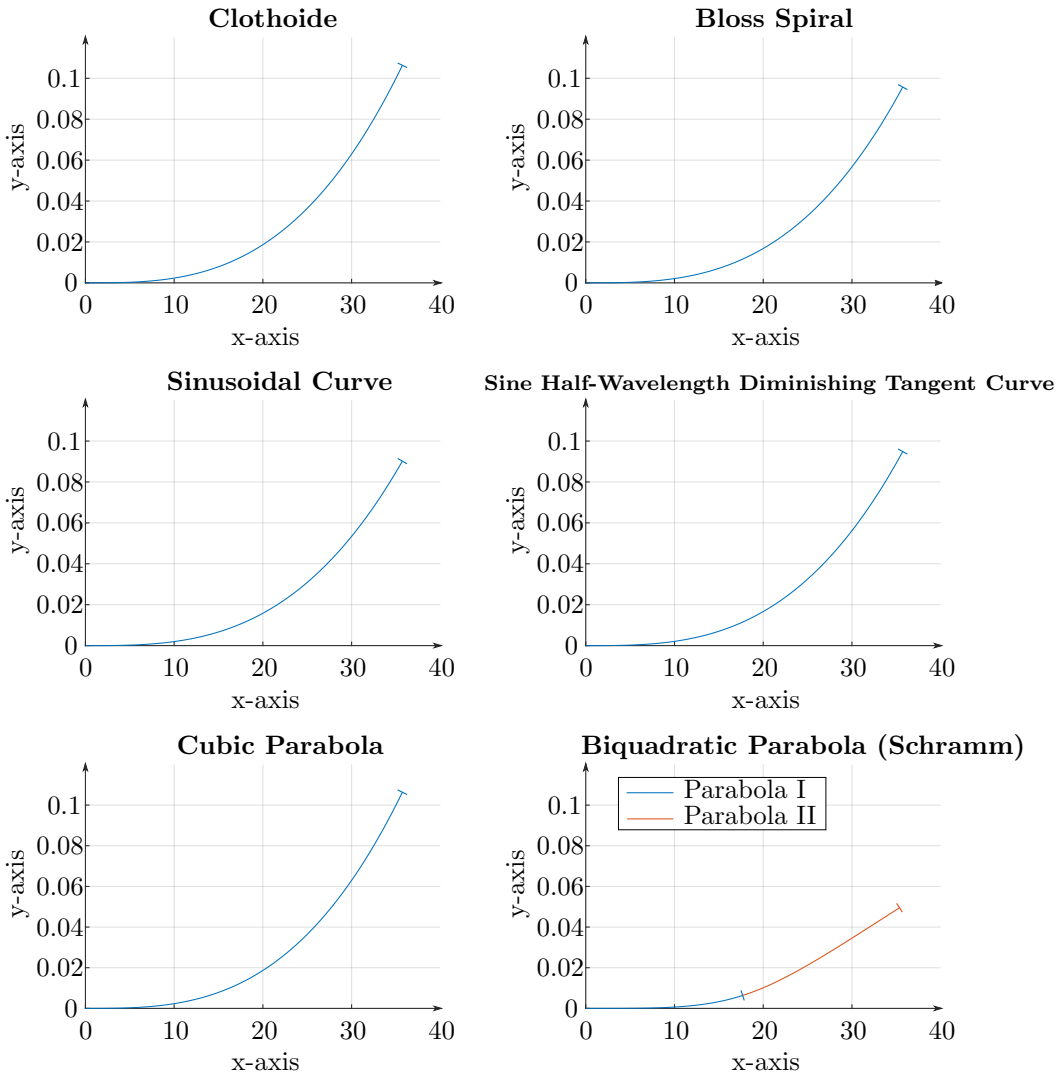


Figure 3.7: Different types of transition curves for an alignment with a design speed of 100 km/h and a target curve radius of 2000 m; The design q-value is 0.3 $\frac{m}{s^3}$

Clothoid Curve

Generally it is possible to use any kind of spiral curve with a varying radius as a transition. However, regardless of the spiral used, engineers should aim for continuity of bearing (direction) and change of curvature (radius) at any point and where spirals join other elements (arc or straight). That is the advantage of the clothoid, it is (at its origin) straight, and the curvature is inversely proportional to the length ($R * L = const$). It can, therefore, be used to join any other highway element.

To get the x- and y-values for the clothoid shown in figure 3.7 the following formulas have been used (Autodesk, 2016):

- x-values:

$$x = l(1 - \frac{l^2}{40R^2} + \frac{l^4}{3456R^4} - ...) \quad [m] \tag{3.3}$$

- y-values:

$$y = \frac{l^2}{6R} * (1 - \frac{l^2}{56R^2} + \frac{l^4}{7040R^4} - ...) \quad [m] \tag{3.4}$$

Refer to section 2.2.2 on page 8 and section 3.4.4 on page 26 for more information.

Bloss Curve

The Bloss Curve's advantage over the clothoid is the smaller shift, resulting in a longer transition with a larger spiral extension. This factor makes the Bloss Curve more suitable than the clothoid in rail design. (Autodesk, 2016)
To get the x- and y-values for the Graph shown in Figure 3.7 the following formulas have been used (Autodesk, 2016):

- x-values:

$$x = l - \frac{l^3}{43.8261R^2} + \frac{l^5}{3696.63R^4} \quad [m] \tag{3.5}$$

- y-values:

$$y = \frac{3l^2}{20R} - \frac{l^4}{363.175R^3} \quad [m] \tag{3.6}$$

Sinusoidal Curve

Sinusoidal Curves have a consistent course of curvature but are only applicable to transition curves from 0 through 90 degrees of tangent reflection.
However, these curves are steeper than other transition curves and therefore not widely used for alignment designs. (Autodesk, 2016)
To get the x- and y-values for the graph shown in figure 3.7 the following formulas have been used (LandXML, 2016):

- x-values:

$$x = l - 0.02190112582400869\frac{l^3}{R^3} \quad [m] \tag{3.7}$$

- y-values:

$$y = l * (0.1413363707560822\frac{l}{R} - 0.0026731818162654 * \frac{l^3}{R^3}) \quad [m] \tag{3.8}$$

Sine Half-Wavelength Diminishing Tangent Curve

Autodesk states, "this form of equation is commonly used in Japan for railway design. The curve is useful in situations where you need an efficient transition in the change of curvature for low deflection angles (in regard of vehicle dynamics)." (Autodesk, 2016)
The x- and y-values for the graph shown in figure 3.7 have been calculated using the following formulas (Autodesk, 2016):

- x-values:

$$x = l - 0.0226689447\frac{l^3}{R^2} \quad [m] \tag{3.9}$$

- y-values:

$$y = 0.14867881635766\frac{l^2}{R} \quad [m] \tag{3.10}$$

Cubic Parabola

Cubic Parabolas are popular in railway and highway design because they converge less rapidly than cubic spirals. (Autodesk, 2016)
The x- and y-values for the graph shown in figure 3.7 have been calculated using the following formulas (LandXML, 2016):

- x-values:

$$x = l \; [m] \tag{3.11}$$

- y-values:

$$y = \frac{l^3}{6RL} \; [m] \tag{3.12}$$

Bi-quadratic Parabola (Schramm Curve)

Bi-quadratic or so-called 'Schramm Curves' consist of two second-degree parabolas whose radii vary as a function of curve length. The change of radial acceleration increases until the

end of the first parabola and then decreases within the length of the second parabola until it reaches zero at the beginning of the circular arc segment. The calculation of this type of transition curve is a bit more complicated than the other transition curve types.
The transition curve length is split into two elements L_E of equal length. The x- and y-values for parabola I (figure 3.7) are calculated with (A. Pirti, M. Ali Yücel, 2012, p. 4):

- x-values:

$$x_I = l \quad [m] \tag{3.13}$$

- y-values:

$$y_I = \frac{l^4}{6RL_E^2} \quad [m] \tag{3.14}$$

The formulas to get parabola II are the same as for parabola I, only the coordinate system is different. The x-axis for parabola II is the tangent t at the ending point pf parabola I. Therefore, parabola II is created by mirroring parabola I at axis m and then turned by the angle 2α around the ending point P of parabola I. The turning angle α is derived by calculating the tangent t at P (figure 3.8).

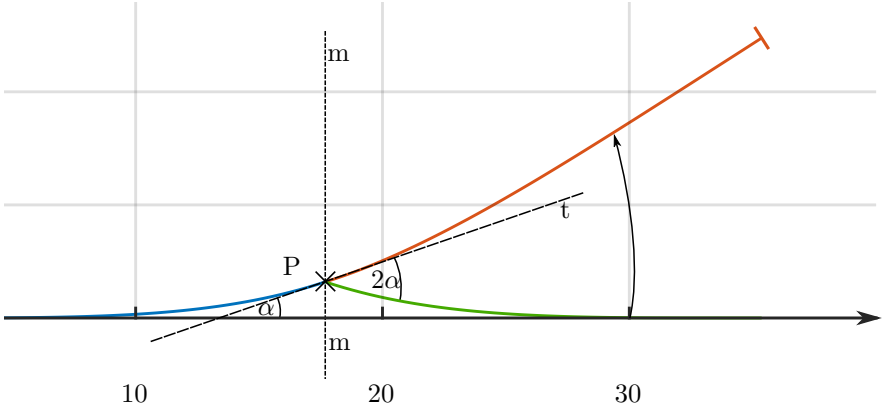


Figure 3.8: Creation of the transition curve using the Bi-quadratic Parabola

Cosinusoidal Curve

The Cosinusoidal Curve is similar to the Sinusoidal Curve and also not widely used in alignment design.
The x- and y-values for the graph shown in figure 3.9 have been calculated using the following formulas (LandXML, 2016, p. 20):

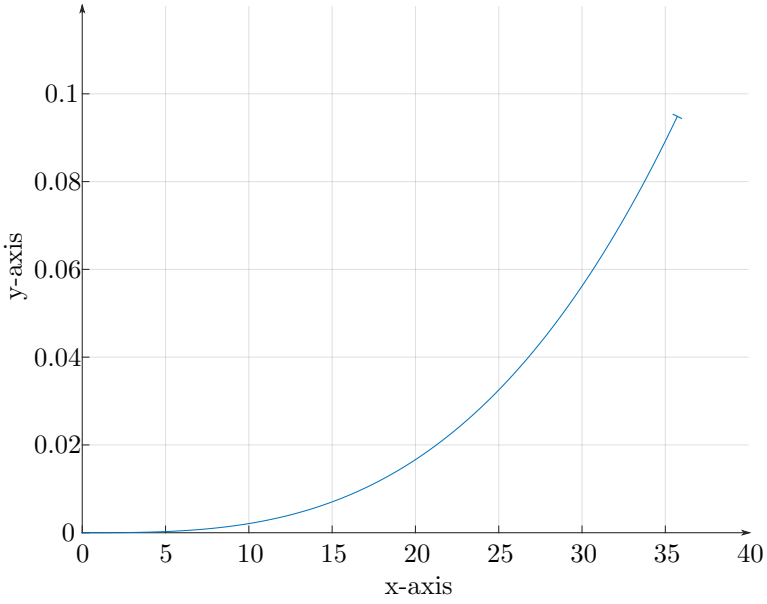


Figure 3.9: Cosinusoidal Transition Curve with the same design parameters as used in 3.7

- x-values:

$$x = l - 0.0226689447\frac{l^3}{R^2} \quad [m] \tag{3.15}$$

- y-values:

$$y = l * (0.1486788163576622\frac{l}{R} - 0.0027402322400286\frac{l^3}{R^3}) \quad [m] \tag{3.16}$$

3.4.4 IFC Clothoidal Arc Segment

Although replaced by the **IfcTransitionCurveSegment2D**, the **IfcClothoidalArcSegment2D** is still included in the IFC Alignment 1.1 data scheme and should not be missing on this list due to completeness. *StartPoint*, *StartDirection*, *SegmentLength*, and *TangentialContinuity* properties are again inherited from the **IfcAlignment2DHorizontalSegment** class. The **IfcClothoidalArcSegment2D** has the following additional properties:

- *IsCCW*
- *IsEntry*
- *ClothoidConstant*

IsCCW is specified by the boolean argument *'true'* or *'false'* and follows the same principle as the *IsCCW* property within the **IfcCircularArcSegment2D**.

IsEntry defines the curvature of the transition curve segment. Boolean *'true'* means a decreasing curvature describing an entering transition curve whilst the boolean *'false'* means an 'increasing' curvature and therefore the 'exiting' transition curve.
ClothoidConstant simple gives the clothoid constant A further described in section 2.3.

3.5 Vertical Alignment

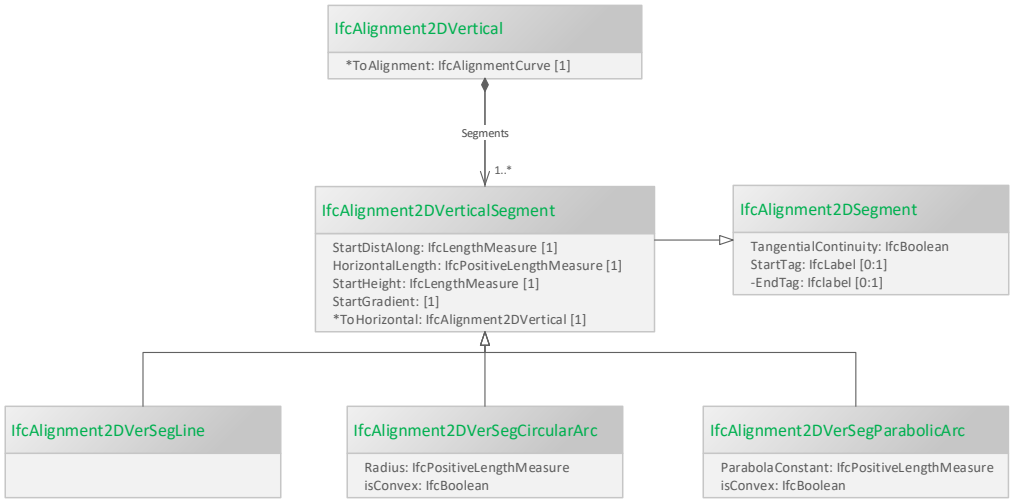


Figure 3.10: Class diagram of the IFC vertical alignment part adapted from Liebich (2017, p. 14)

Another entity within the IFC Alignment 1.1 extension is the **IfcAlignment2DVertical** class, that consists of an ordered list of vertical alignment segments (**IfcAlignment2DVerticalSegment**). Each segment inherits all the properties from the **IfcAlignment2DSegment** (figure 3.10). Possible types of vertical alignment elements are:

- **IfcAlignment2DVerSegLine**
- **IfcAlignment2DVerSegCircularArc**
- **IfcAlignment2DVerSegParabolicArc**

The circular arc and parabolic arc segments also have the property *IsConvex*, that can be defined as either *'true'* or *'false'* to indicate whether the parabolic arc defines a 'sag' ('false') or a 'crest' ('true').

3.5.1 IFC Gradient Segment

The gradient segment (**IfcAlignment2DVerSegLine**) of the vertical alignment does not have any additional properties and thus only gets all the properties inherited from **IfcAlignment2DVerticalSegment**.
An endpoint can then be calculated by the computer using all the given parameters.

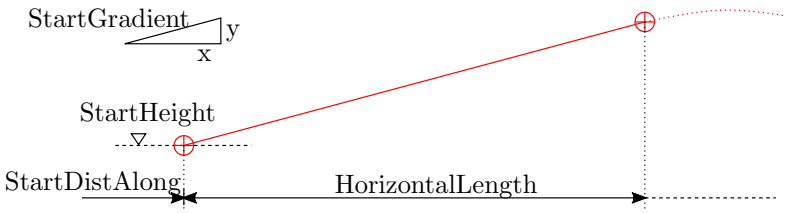


Figure 3.11: IfcAlignment2DVerSegLine

3.5.2 IFC Vertical Circular Arc Segment

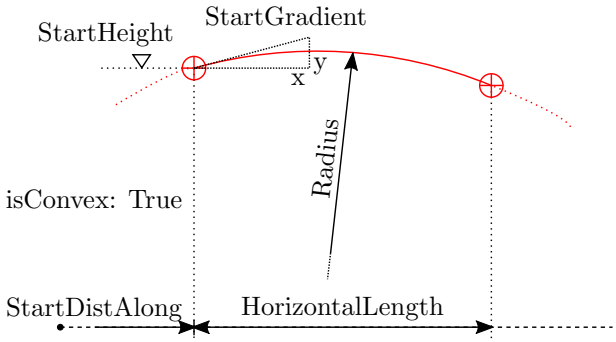


Figure 3.12: IfcAlignment2DVerSegCircularArc

Unlike the gradient segment, the circular arc segment **IfcAlignment2DVerSegCircularArc**, also adds the properties *IsConvex* (described in section 3.5) and *Radius*. The radius property is specified by an *IfcPositiveLengthMeasure*. An endpoint of the circular arc segment can then be derived from the given parameters.

3.5.3 IFC Vertical Parabolic Arc Segment

The vertical parabolic arc segment (**IfcAlignment2DVerSegParabolicArc**) is similiar to the circular arc segment but it also has an property called *ParabolaConstant* that is specified by an *IfcPositiveLengthMeasure* and describes the shape of the vertical parabola (section 2.2.3). The ending point again can be calculated with the given parameters.

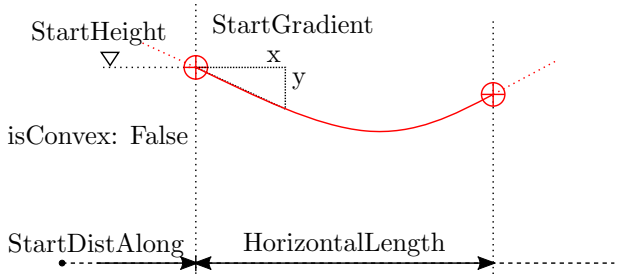


Figure 3.13: IfcAlignment2DVerSegParabolicArc

Chapter 4

Programming an IFC Import/Export Plug-in

4.1 Development Environment

In this chapter the development of a basic IFC Import/Export Plug-in will be documented, exemplified within the *Autodesk AutoCAD Civil 3D ObjectARX API Programming Environment*. The programming language used is C#, which offers all the tools and commands needed for this purpose and also presents easy access to the field of software development even without prior experience.
To perform the actual programming *Microsoft Visual Studio 2017* such as *Microsoft Visual Studio 2015 Express* were used. Colour coding in the given example listings also follows the Visual Studio colour coding to achieve maximum unification. This plug-in works exclusively for C3D. However, importing or exporting *.ifc files within different environments and programmes will work very similar.
All screenshots such as the TIN surface used to create the alignment are being used with the permission of Autodesk Inc.

4.2 Platform and Limitations

The plug-in was written on a machine running *Microsoft Windows 10 Education* using *Autodesk AutoCAD Civil 3D 2018*. However, the plug-in should be downward compatible to older versions of C3D.
Due to the IFC data schema (see section 3.1) one horizontal alignment can only have one or no vertical alignment assigned to it (See section 3.2).
To generate a correct *.ifc file from the original file (*.dwg file) the units within C3D need to

be set to metric units (the option to choose different unit systems such as imperial units will be implemented in the future), which is done under *Drawing Utilities* → *Drawing Settings* → *Units and Zone* → *Drawing Units* (see figure 4.1 and figure 4.2).

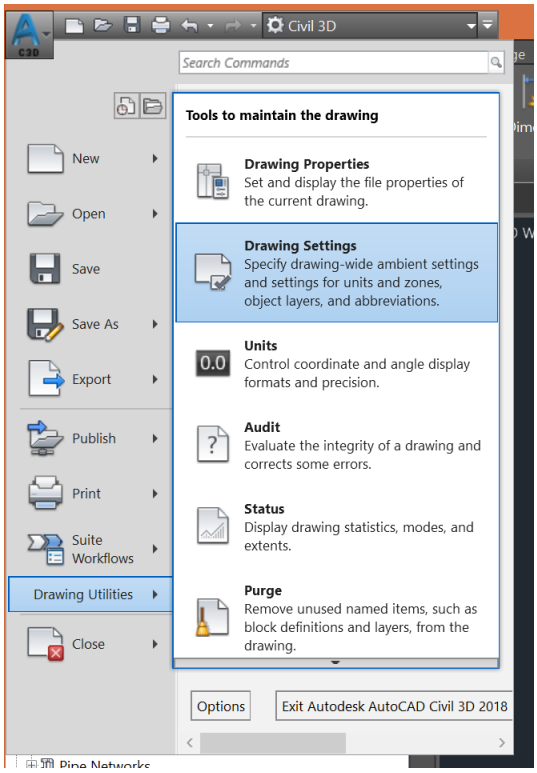


Figure 4.1: Accessing the unit drawing settings in C3D

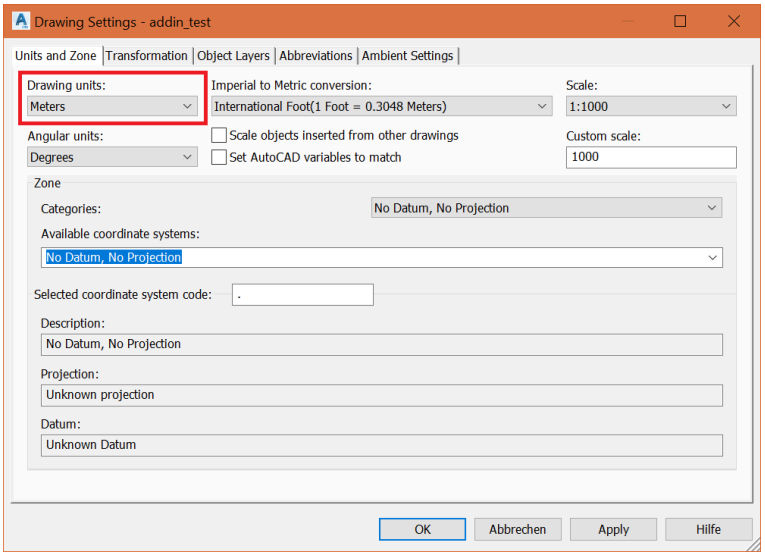


Figure 4.2: Setting the correct units in C3D

4.3 Current Development State

Currently the plug-in only fully supports the export (*.dwg to *.ifc). However, an import function is in the process of development and already works for importing horizontal alignments and TIN surfaces.
Although the plug-in was tested throughout the developing process to ensure a bug free work flow, bugs are still possible due to the early release of the plug-in.

4.4 The Civil 3D Plug-In

4.4.1 Overview

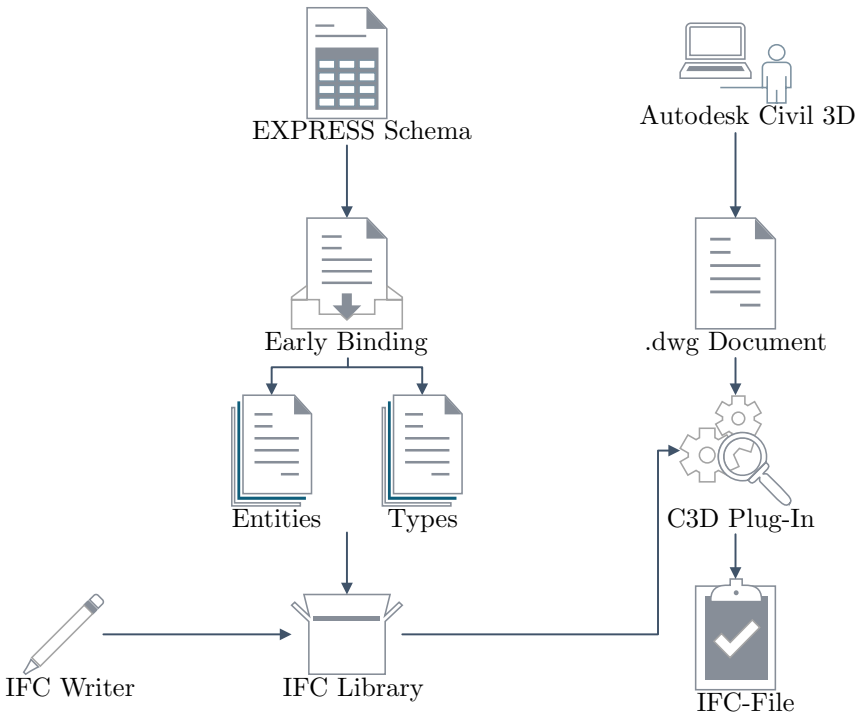


Figure 4.3: Conceptual developing stages involved in creating the final *.ifc file (Current development state)

Many different steps were involved in creating the plug-in (see figure 4.3). Based on the newest IFC express schema a C# IFC class library had to be generated. This was done through an external early binding generator (*IFCSharpEarlyBinding Generator*), which scans the respective EXPRESS schema and creates entities and types accordingly.
Additionally **IfcWriter** and **IfcAlignmentReader** (IfcAlignmentReader is still in develop-

ment and currently only works for IFC Alignments, see 4.4.6) classes are included in the IFC class library. The **IfcWriter** has a property called *entityList* that serves as a container for all the different objects within the then created *.ifc file. Further, every entity and type (all implemented as classes) have the method *GetIdOrValue()* to create their STEP string, that accommodates the name and the properties following the *ISO 10303 - 21 standard* (International Organization for Standardization, 2016).
Using the **IfcWriter's** method *insertEntity(IfcEntity)*, every IFC entity has to be added to the *entityList* container. In a first step the **IfcWriter** writes the header section to the file, assigns a successive *ID* to every entity, then calls the entity's *GetIdOrValue()* method to create the Standard for the Exchange of Product Model Data (STEP) string, and, in a final step, creates the entire *.ifc file (This is only a rough description of the file creation process; the documentation covering the early binding is not part of this thesis).

4.4.2 Preparation

To be able to access the C3D class library and all the necessary tools, besides some Microsoft Windows genuine references, certain C3D references (*.dll files) have to be imported applying the using command within the code. On top the following namespaces are included in the code:

Listing 4.1 Plug-in Namespaces

```
1  using Autodesk.AutoCAD.ApplicationServices;
2  using Autodesk.AutoCAD.DatabaseServices;
3  using Autodesk.AutoCAD.Runtime;
4  using Autodesk.Civil;
5  using Autodesk.Civil.ApplicationServices;
6  using Autodesk.Civil.DatabaseServices;
7  using Autodesk.Private.Windows;
8  using Autodesk.Windows;
9  using Application = Autodesk.AutoCAD.ApplicationServices.Core.Application;
10 using Surface = Autodesk.Civil.DatabaseServices.Surface;
11
12 //The IFC class library
13 using Ifc4x1;
```

If necessary, further documentation on the *.dll files can be found on the Autodesk Inc.'s documentation website (Autodesk, 2017).
With these namespaces included all the classes and methods used for the plug-in are then accessible.
In a next step a new namespace and class is being defined. Afterwards, a command method needs to be implemented so that the plug-in can be called within C3D. The chosen command to be typed into the command line here is '**ifc**' (in order to access the plug-in/the 'ifc'

command, it is necessary that the user calls the C3D genuine **netload** command that allows for plug-ins (*.dll's) to be loaded into the main program). This is a method specific to C3D and might be done differently within different programs. The *Main* method finally starts the actual function of the plug-in.

Listing 4.2 Initialisation of the plug-in

```
1 namespace C3DIfcAlignmentPlugIn.PlugIn
2 {
3     public class C3DIfcAlignmentAddIn
4     {
5         [CommandMethod("ifc", CommandFlags.Session)]
6
7         public void Main()
8         {
9             ...
```

4.4.3 The Example Alignment

To demonstrate the plug-in's procedure an example alignment, created within C3D, shall form the base of the IFC export. It consists of one horizontal alignment called "*addin_test*" (appendix A.1) and an assigned profile view (vertical alignment, appendix A.2). Further, a TIN surface is included in the *.dwg file to actually facilitate the creation of a vertical alignment. Simple clothoids were used as transition curves for the horizontal, and parabolas for the vertical alignment. Figure 4.4 and 4.5 show the alignment including the surface within C3D.

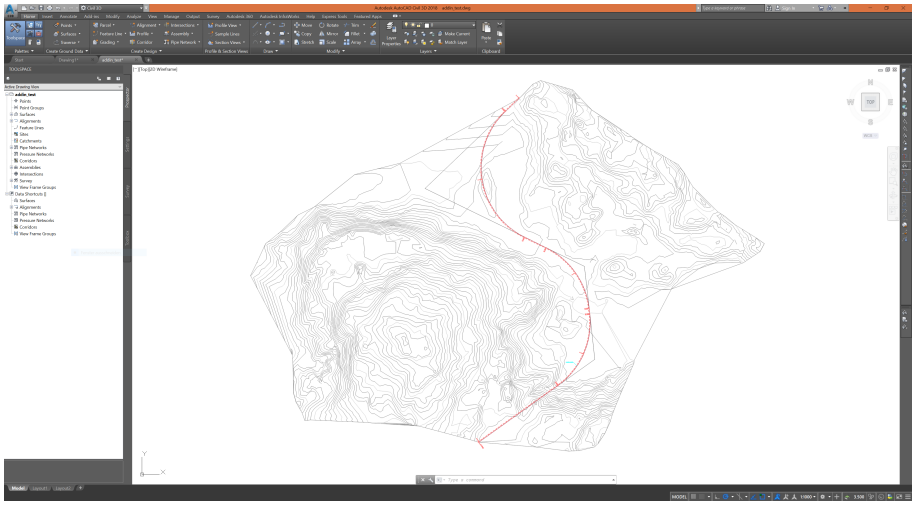


Figure 4.4: "*addin_test*" alignment in C3D

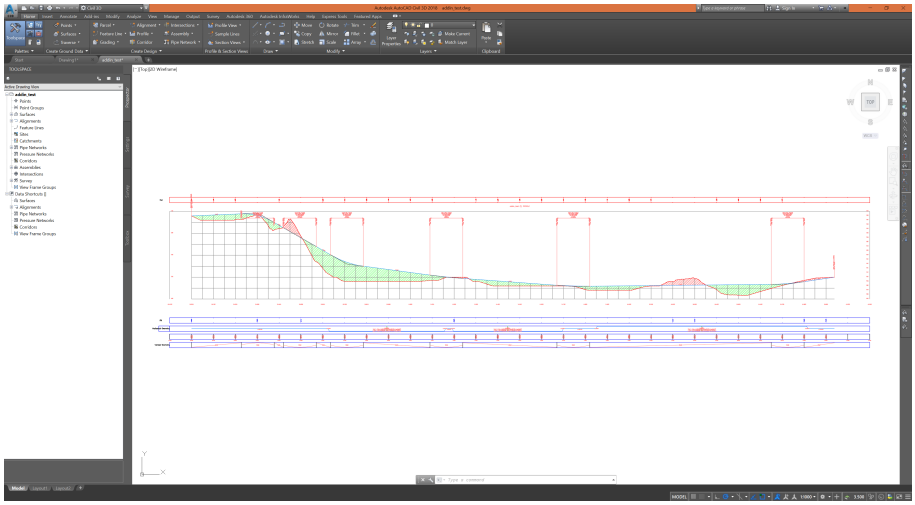


Figure 4.5: Profile of "*addin_test*" and the assigned vertical alignment in C3D

4.4.4 Creation of the Plug-In Button

Calling functions and commands within C3D can be done in many different ways but creating an additional button is the most user friendly way.
When the **ifc** command is being called from the command line a new **Split Button** will show up in the Add-ins Tab. This can be reached using the code shown in listing 4.3.
In preparation the plug-in checks if the panel, which stores the IFC buttons, already exists and deletes it in case before the new button is created. This ensures that the user does not end up with more than one button for the same function when he calls the **ifc** command more than once in one session.

Listing 4.3 Creating the new button

```
1  //Get C3D UI-Information
2  RibbonControl ribbon = ComponentManager.Ribbon;
3  //Find the correct Ribbon
4  RibbonTab btab = ribbon.FindTab("CIVIL.ID_Civil3DAddins");
5  //Check if the Panel already exists
6  if (btab != null)
7  {
8      if (btab.Panels != null)
9      {
10         foreach (RibbonPanel panel in btab.Panels)
11         {
12             if (panel.UID == "UID_IFCPanel")     //if the Panel already exists...
13             {
14                 RibbonPanel remove_pnl = panel;
15                 btab.Panels.Remove(remove_pnl);      //...delete the Panel
16                 break;
17             }
```

```
18         }
19     }
20 }
21 //Create the new Button
22 //1. Create new Split-Button
23 RibbonSplitButton rbSplitButton = new RibbonSplitButton();
24 rbSplitButton.Text = "IFC Add-in";
25 rbSplitButton.ShowText = true;
26 rbSplitButton.Size = RibbonItemSize.Large;
27 rbSplitButton.LargeImage = getBitmap("ifclogo.png");
28 rbSplitButton.ShowImage = true;
29 rbSplitButton.Name = "IFC Add-in";
30 //2. Create Panel
31 RibbonPanel Panel = new RibbonPanel();
32 Panel.UID = "UID_IFCPanel";
33 Panel.Id = "ID_IFCPanel";
34 //3. Create Panel Source
35 RibbonPanelSource Source = new RibbonPanelSource();
36 Source.Id = "ID_IFCPanelSource";
37 Source.Name = "IFCPanelSource";
38 Source.Title = "IFC Alignment v1.1";
39 Panel.Source = Source;  //... and bind it to the Panel
40 //4. Create Ribbon Button 1
41 RibbonButton rbb1 = new RibbonButton();
42 rbb1.Name = "Export IFC...";    //Define Button 1 Attributes
43 rbb1.Text = "IFC Export";
44 rbb1.CommandHandler = new ExportCmdHandler();
45 //Create Tooltip Button 1
46 RibbonToolTip rbnTT = new RibbonToolTip();
47 rbnTT.Command = "IFC";   //Define Button 1 Tool-Tip
48 rbnTT.Title = "IFC-Export Add-in";
49 //5. Create Ribbon Button 2
50 RibbonButton rbb2 = new RibbonButton();
51 //...Define Button 2 Attributes and Tooltip
52 //Add Buttons to the Split-Button
53 rbSplitButton.Items.Add(rbb1);  //Add Button 1
54 rbSplitButton.Items.Add(rbb2);  //Add Button 2
55 //Add Panel to the Add-ins Tab
56 btab.Panels.Add(Panel);
57 btab.IsActive = true;   //...and set as active
```

Line 44 in listing 4.3 binds the command handle to the button that will then convert the *.dwg file to an *.ifc file. **Button 1** will export the current *.dwg file into an *.ifc file and **Button 2** handles the import function of the plug-in that will convert a *.ifc file into a *.dwg file (function only partly implemented yet). Within C3D the outcome looks like figure 4.6.

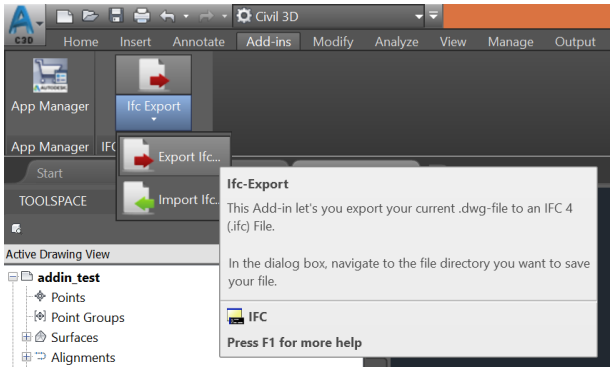


Figure 4.6: Split Button created by the plug-in in C3D

4.4.5 Export Function

Upon pressing **Button 1** (Export button) a *Windows SaveFile Dialog* shows up to let the user specify a file name and a location to which the *.ifc file will be saved.

Listing 4.4 Creating button command

```
1 public class ExportCmdHandler : System.Windows.Input.ICommand
2 {
3     public bool CanExecute(object parameter)
4     {
5         return true;
6     }
7     public event EventHandler CanExecuteChanged;
8     public void Execute(object parameter)
9     {
10        //select active Document
11        Document docc = acApp.DocumentManager.MdiActiveDocument;
12        if (parameter is RibbonButton)  //if Button is clicked...
13        {
14            RibbonButton button = parameter as RibbonButton;
15            //Open SaveFile-Dialog
16            Microsoft.Win32.SaveFileDialog dlg = new Microsoft.Win32.
                  SaveFileDialog();
17            dlg.FileName = "ifcAlignment";    // Default File Name
18            dlg.DefaultExt = ".ifc";     // Default File Extension
19            dlg.Filter = "IFC Files (.ifc)|*.ifc"; //Filter Files by Extension
20            if (result == true)
21            {
22                ...
```

After the last if clause the export of all the objects in the drawing and creation of the actual *.ifc file is being started. This process will be explained in more detail below.

Surface Model

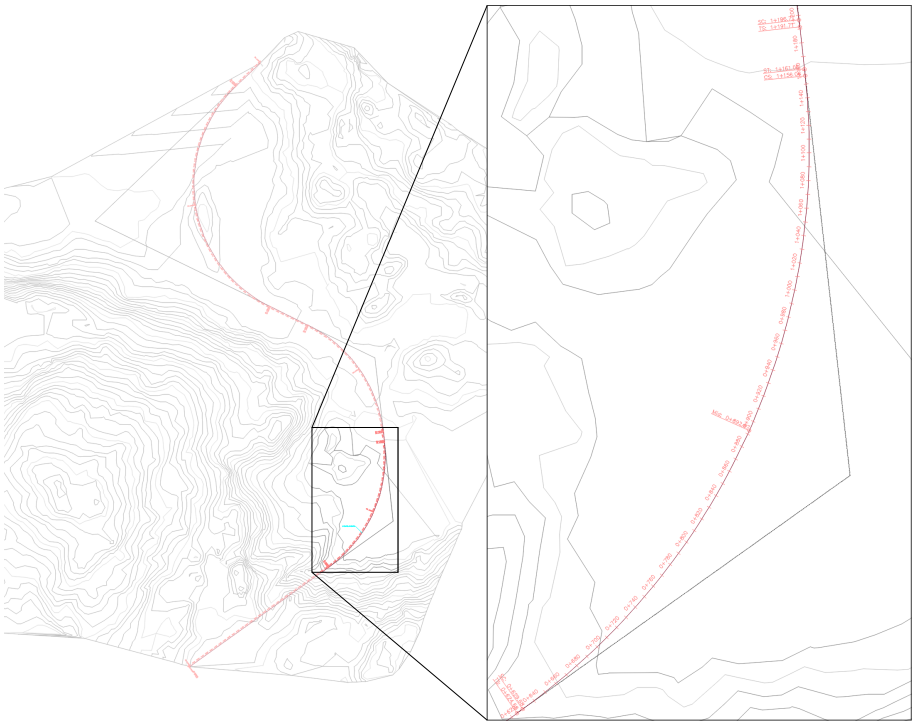
At the current stage of development the plug-in only allows the export of TIN surfaces even though different surface types can be created within C3D. The implementation follows the same principle, such as the implementation of the horizontal and vertical alignment. The plug-in simply runs through all the surfaces in C3D, checks whether the current surface is a TIN surface, creates the equivalent IFC objects, and assigns the property values accordingly. Listing 4.5 shows the code necessary for a working surface export.

Listing 4.5 Converting the surface model

```
1  var civilSurfaceCollection = civilDocument.GetSurfaceIds();
2  foreach (ObjectId surfaceId in civilSurfaceCollection)
3  {
4     var civilSurface =
5     (Surface) civilTransactionManager.GetObject(surfaceId, OpenMode.ForRead);
6     if (civilSurface is TinSurface)
7     {
8        //IFC Geometric Representation Context
9        IfcGeometricRepresentationContext ifcGeometricRepresentationContext =
             new IfcGeometricRepresentationContext(){...};
10       ifcWriter.insertEntity(ifcGeometricRepresentationContext);
11
12       //IFC Triangulated Face Set
13       IfcTriangulatedFaceSet ifcTriangulatedFaceSet = new
             IfcTriangulatedFaceSet(){...};
14       ifcWriter.insertEntity(ifcTriangulatedFaceSet);
15
16       //IFC Cartesian Point List 3D
17       IfcCartesianPointList3D vertexList = new IfcCartesianPointList3D(){...};
18       ifcTriangulatedFaceSet.m_Coordinates = vertexList;
19       ifcWriter.insertEntity(vertexList);
20
21       var civilTinTriangleCollection = civilTinSurface.GetTriangles(false);
22       int vertexCounter = 1;
23       foreach (TinSurfaceTriangle civilTinTriangle in
             civilTinTriangleCollection )
24       {
25          var vertices = new List<IfcBase>();
26          var triangle = new List<IfcBase>();
27
28          var vertex1 = CreateIfcPoint3D(civilTinTriangle.Vertex1.Location.X,
29          civilTinTriangle.Vertex1.Location.Y, civilTinTriangle.Vertex1.
                Location.Z);
30          vertices.Add(vertex1);
31          triangle.Add(new IfcPositiveInteger() { _value = new IfcInteger() {
                _value = vertexCounter } });
32          vertexCounter++;
```

```
33
34              var vertex2 = ...
35              vertexCounter++;
36
37              var vertex3 = ...
38              vertexCounter++;
39
40              //Add to IFC Entity
41              vertexList.m_CoordList.Add(vertices);
42              ifcTriangulatedFaceSet.m_CoordIndex.Add(triangle);
43          }
44          ...
45      }
46  }
```

Horizontal Alignment



Entity-Nr.	Chainage	Entity Type	Tangential Continuity	Start Point X	Start Point Y
2	624,6796488	Spiral	true	5188,1868881	7399,5789148
3	629,6796488	Circular Arc	true	5192,2522754	7402,4896829
4	1156,0649861	Spiral	true	5396,9789976	7861,2969108
5	1161,0649861	Line	false	5396,4300237	7866,2666766

Figure 4.7: Part of the horizontal alignment in C3D including a table containing the respective property values

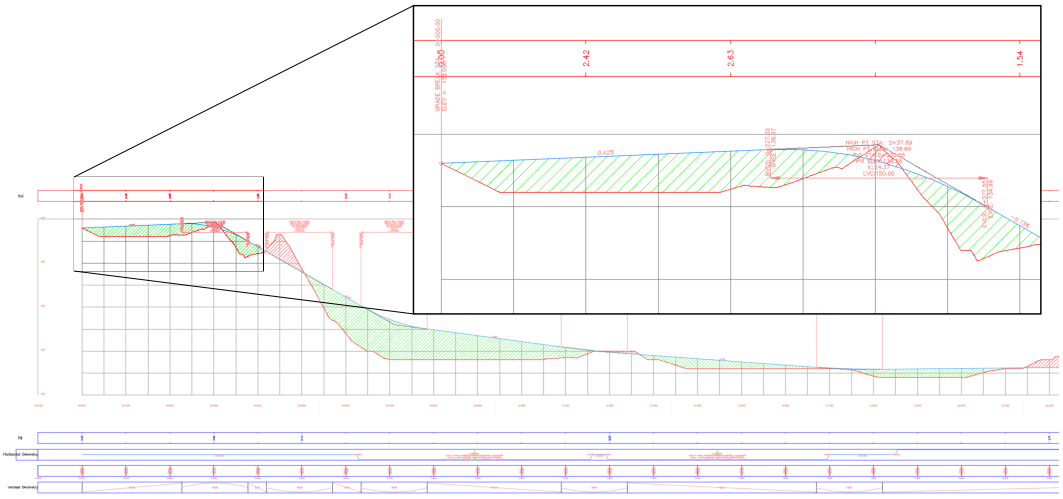
Within one C3D document multiple **horizontal alignments** can be created. To ensure all the horizontal alignments are implemented into the created *.ifc file the plug-in checks for every horizontal alignment in the active document. These alignments are stored with their *ObjectId's* in the **ObjectIdCollection** class. Every **alignment entity** and the included **alignment sub-entities** (C3D always stores a horizontal curve as one entity with entering transition, circular curve, exiting transition as sub-entities) are being checked on their type and the properties will be assigned to the equivalent IFC class properties below. Different to the rest of the entity properties the *isTangential* property is stored within the **AlignmentEntity** class and has to be extracted one hierarchy step higher than the other necessary properties.
In the case of the horizontal alignment, as well as the associated profiles, the alignment/profile entities are not stored in the order they actually occur in the alignment. To cover this additional *SortC3DAlignmentEntities* and *SortC3DProfileEntities* methods had to be implemented to change the order of the entities and store them in the right order for the creation of the *.ifc file.
Upon finding an alignment in the current drawing the plug-in will run through all the entities and create the equivalent IFC objects. The code looks like listing 4.6.
Note that certain property definitions such as *StartDirection* are defined in a different way in C3D and had to be converted using other implemented methods to suit the IFC property definitions. Further, some IFC properties are not stored as such in C3D so they had to be derived through additional methods.

Listing 4.6 Converting the horizontal alignment

```
1  foreach (ObjectId alignId in civilAlignmentCollection)
2  {
3      var civilAlignment =
4      (Alignment) civilTransactionManager
5      .GetObject(alignId, OpenMode.ForRead); //Open File for Read
6
7      //Ifc Alignment
8      IfcAlignment ifcAlignment = new IfcAlignment(){...};
9      ifcWriter.insertEntity(ifcAlignment);
10
11     //IfcAlignmentCurve
12     IfcAlignmentCurve ifcAlignmentCurve = new IfcAlignmentCurve();
13     ifcWriter.insertEntity(ifcAlignmentCurve);
14
15     //IfcAlignment2DHorizontal
16     IfcAlignment2DHorizontal ifcAlignment2DHorizontal = new
           IfcAlignment2DHorizontal(){...};
17
18     ifcWriter.insertEntity(ifcAlignment2DHorizontal);
19     //Get Alignment Entities
20     var civilAlignmentEntityList = SortC3DAlignmentEntities(civilAlignment);
```

```
21    foreach (var horizontalAlignmentEntity in civilAlignmentEntityList)
22    {
23       var civilAlignmentSubentityCounter = horizontalAlignmentEntity.
             SubEntityCount;
24       for (var i = 0; (i <= (civilAlignmentSubentityCounter - 1)); i++)
25       {
26          var civilAlignmentSubentity = horizontalAlignmentEntity[i];
27          switch (civilAlignmentSubentity.SubEntityType)
28          {
29          //...Circular Arc
30          case AlignmentSubEntityType.Arc:
31          var alignSubEntArc = (AlignmentSubEntityArc) civilAlignmentSubentity;
32          //Ifc Alignment2DHorizontalSegment
33          IfcAlignment2DHorizontalSegment ifcHorSegArc =
34          new IfcAlignment2DHorizontalSegment(){...};
35
36          IfcCircularArcSegment2D ifcCircularArc = new IfcCircularArcSegment2D
                ({...};
37          //Assign Property Values:
38          ...
39          //Insert Entities:
40          ifcWriter.insertEntity(ifcHorSegArc);
41          ifcWriter.insertEntity(ifcCircularArc);
42          ifcWriter.insertEntity(ifcArcStartPoint);
43          //Start Next Element:
44          break;
45
46          //...Line
47          case AlignmentSubEntityType.Line:
48          //Create IfcLineSegment2D:
49          ...
50          //Assign Property Values:
51          ...
52          //Insert Entities:
53          ...
54          //Start Next Element:
55          break;
56
57          //...Spiral
58          case AlignmentSubEntityType.Spiral:
59          //Create IfcTransitionCurveSegment2D
60          ...
61          //Assign Property Values:
62          ...
63          //Insert Entities:
64          ...
65          //Start Next Element:
66          break;
```

Vertical Alignment



Entity-Nr.	Chainage (StartDistAlong)	Entity Type	Horizontal Length	Start Height	Start Gradient
1	0,0000000	Line	227,5475098	138,0000000	0,0042415
2	227,5475098	Parabolic Arc	150,0000000	138,9651445	0,0042415

Figure 4.8: Part of the vertical alignment in C3D including a table containing the respective property values

Although within C3D each horizontal alignment can have one or more **vertical profiles**, IFC only supports the usage of a single profile assigned to a horizontal alignment. This profile can be obtained using the code in listing 4.7 inside the foreach clause covering all the horizontal alignments (line 1, listing 4.6). Listing 4.7 further checks if the current profile is the surface profile, which does not need to be stored at this point as it only represents the actual surface profile along the current horizontal alignment.

Listing 4.7 Getting all the profiles associated to the current alignment

```
1  int ver_counter = 1; //Set Vertical Alignment Counter
2  foreach (ObjectId profile_id in align.GetProfileIds())
3  {
4      //Opens the Profile for read
5      Profile profile = (Profile)acTrans.GetObject(profile_id, OpenMode.ForRead);
6
7      if (profile.StyleName != "Existing Ground Profile")
8      {
9          //Profile is Vertical Alignment
10         ...
11     }
12     ...
```

In the following step (listing 4.8) the plug-in runs through all the **ProfileEntities** within the current profile, checks whether the entity is a **tangent**, a **circular arc** or a **parabola**, and assigns the properties to the *.ifc file accordingly.

Listing 4.8 Converting the profile(s)

```
1  string ver_align_name = profile.Name.ToString();    //Get Profile Name
2  int ver_ent_counter = 1;    //Set Element Counter
3
4  foreach (ProfileEntity ver_align_Ent in profile.Entities)
5  {
6     switch (ver_align_Ent.EntityType)   //if Element-Type is a...
7     {
8        //...Tangent:
9        case ProfileEntityType.Tangent:
10       ProfileTangent ver_tangent = (ProfileTangent)ver_align_Ent;
11       //IfcAlignment2DVerSegLine:
12       IfcAlignment2DVerSegLine ifcVerSegLine = new IfcAlignment2DVerSegLine()
             {...};
13       //Assign Property Values:
14       ...
15       //Insert Entities:
16       ifcWriter.insertEntity(ifcVerSegLine);
17       //Start Next Element:
18       ver_ent_counter++;
19       break;
20
21       //... Asymmetric Parabola:
22       case ProfileEntityType.ParabolaAsymmetric:
23       ProfileParabolaAsymmetric ver_parabola_a = (ProfileParabolaAsymmetric)
             ver_align_Ent;
24       //IfcAlignment2DVerSegParabolicArc:
25       //Create Parabola 1:
26       IfcAlignment2DVerSegParabolicArc ifcVerSegParabola1 =
27       new IfcAlignment2DVerSegParabolicArc(){...};
28       //Assign Property Values:
29       //Create Parabola 2:
30       IfcAlignment2DVerSegParabolicArc ifcVerSegParabola2 =
31       new IfcAlignment2DVerSegParabolicArc(){...};
32       //Assign Property Values:
33       ...
34       //Insert Entities:
35       ...
36       //Start Next Element:
37       ver_ent_counter++;
38       break;
39
40       //...Symmetric Parabola:
41       case ProfileEntityType.ParabolaSymmetric:
```

```
42        ProfileParabolaSymmetric ver_parabola_s = (ProfileParabolaSymmetric)
              ver_align_Ent;
43        //IfcAlignment2DVerSegParabolicArc:
44        IfcAlignment2DVerSegParabolicArc ifcVerSegParabola =
45        new IfcAlignment2DVerSegParabolicArc(){...};
46        //Assign Property Values:
47        ...
48        //Insert Entities:
49        ...
50        //Start Next Element:
51        ver_ent_counter++;
52        break;
53
54         //...Circular Arc:
55        case ProfileEntityType.Circular:
56        ProfileCircular ver_circ = (ProfileCircular)ver_align_Ent;
57        //Ifc VerSegCircularArc:
58        IfcAlignment2DVerSegCircularArc ifcVerSegArc =
59        new IfcAlignment2DVerSegCircularArc(){...};
60        //Assign Property Values:
61        ...
62        //Insert Entities:
63        ...
64        //Start Next Element:
65        ver_ent_counter++;
66        break;
67    }
68 }
```

Different to the IFC data scheme, the C3D database offers two different classes of vertical parabolas:

- **Symmetric Parabola**
- **Asymmetric Parabola**

Due to the fact that the asymmetric parabola is not covered by the IFC data scheme it has to be divided into two different parabolas ending up in two **IFCAlignment2DVerSegParabolicArc** objects. The horizontal lengths for the different parabolas within the asymmetric parabola are stored within the **ProfileParabolaAsymmetric** class, such as the following properties:

- *StartDistAlong (entering parabola)*
- *StartHeight (entering parabola)*
- *StartGradient (entering parabola)*

The IFC alignment *ParabolaConstant* property (which is the same for the entering and the exiting parabola) can be calculated with the C3D genuine property K:

$$ParabolaConstant = K * 100 \quad [m] \tag{4.1}$$

For the exiting parabola the properties are derived using listing 4.9. Depending on the relation of the entering and the exiting gradient of the asymmetric parabola, the entering gradient for the second parabola is calculated in a different way (line 9 and following, listing 4.9)

Listing 4.9 Getting the properties of the second parabola of an asymmetric parabola

```
1 ProfileParabolaAsymmetric ver_parabola_a = (ProfileParabolaAsymmetric)
      ver_align_Ent;
2
3 //Get Parabola 1 Parameters
4 ...
5 //Get Parabola 2 Parameters:
6 double parabola2_StartDistAlong = ver_parabola_a.StartStation +
      parabola1_HorizontalLength;
7 double parabola2_HorizontalLength = ver_parabola_a.AsymmetricLength2;
8 double parabola2_StartHeight = parabola1_StartHeight + parabola1_StartGradient
       * parabola1_HorizontalLength + parabola1_HorizontalLength *
      parabola1_HorizontalLength / (200 * parabola1_ParabolaConstant);
9 double parabola2_StartGradient = 0;
10 if (ver_parabola_a.GradeIn > ver_parabola_a.GradeOut)
11 {
12    parabola2_StartGradient = - parabola1_HorizontalLength /
          parabola1_ParabolaConstant + parabola1_StartGradient;
13 }
14 else
15 {
16    parabola2_StartGradient = parabola1_HorizontalLength /
          parabola1_ParabolaConstant + parabola1_StartGradient;
17 }
```

The *IsConvex* property can be defined as either *'true'* or *'false'* to indicate whether the parabolic arc defines a 'sag' ('false') or a 'crest' ('true'). This attribute is already stored within the C3D entities.

4.4.6 Import Function

As mentioned above, the import function is only partly implemented in the plug-in at this point of time. However, it already supports the following elements:

- Import of TIN surfaces

- Import of horizontal alignments

Not implemented yet is the following:

- Import of vertical alignment
- Import of alignment sites

The basic principle is similar to the export function. The entity **IfcAlignmentReader** is added to the early binding, which is able to read an *.ifc file with its *ReadFile()* method and store all the alignment entities with all the respective attributes such as *SegmentLength*, *StartDirection*, etc. within its *entityList* property. The result is shown within figure 4.9.

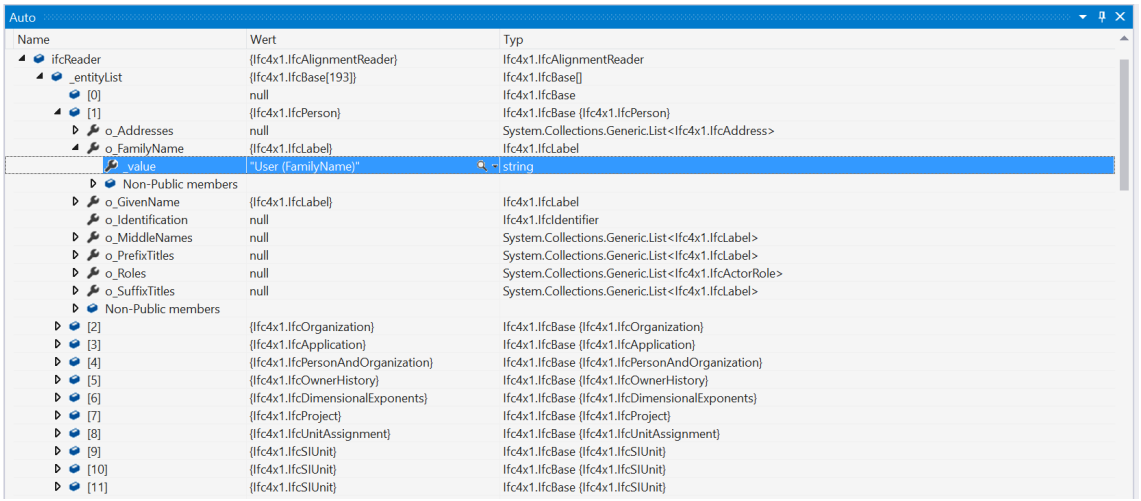


Figure 4.9: Filled "*entityList*" property of the **IfcAlignmentReader** entity

Horizontal Alignment

In the main plug-in the code then runs through all the entities stored in the *entityList* and creates the C3D equivalent objects (Creating the new button function works exactly the same way as it does for the export function and is therefore not explained again).

Listing 4.10 Creating C3D alignment based on the imported *.ifc file

```
1 for (int i = 1; i <= ifcEntityCounter; i++)
2 {
3     if (ifcEntities[i].GetType() == typeof(IfcAlignment))
4     {
5         IfcAlignment castedAlignment = (IfcAlignment)ifcEntities[i];
6         //Create C3D Alignment:
7
```

```
8       var civilAlignmentId = Alignment.Create(civilDocument, alignmentName, ""
            , "C-ROAD", "Proposed", "All Labels");
9   ...
```

Within listing 4.10 "Proposed" defines the label style for the labels along the alignment and "All Labels" makes sure all labels are being displayed within C3D. "C-ROAD" gives the layer on which the new alignment will be created. All options can easily be changed after the import.
After the alignment is created the plug-in checks for the horizontal and vertical part attached to **castedAlignment** shown in listing 4.11.

Listing 4.11 Getting the horizontal and vertical part of **castedAlignment**

```
1 //Get the Alignment Curve:
2 IfcAlignmentCurve ifcCurve = (IfcAlignmentCurve)castedAlignment.m_Axis;
3
4 //Get Horizontal/Vertical Alignment:
5 IfcAlignment2DHorizontal ifcAlignmentHorizontal = ifcCurve.m_Horizontal;
6 IfcAlignment2DVertical ifcAlignmentVertical = ifcCurve.o_Vertical;
```

In a last step the code runs through all the entities stored in *ifcAlignmentHorizontal.m_Segments* and creates the actual alignment entities such as lines, circular arcs, and spirals (see figure 4.12). At this point only clothoids are implemented and therefore possible to be imported.

Listing 4.12 Creating the C3D objects based on the imported *.ifc file

```
1 foreach (var entity in ifcAlignmentHorizontal.m_Segments)
2 {
3    if (entity.m_CurveGeometry.GetType() == typeof(IfcLineSegment2D))
4    {
5       IfcLineSegment2D castedLine = (IfcLineSegment2D)entity.m_CurveGeometry;
6
7       //Create Civil Line:
8       AlignmentLine civilLine = civilAlignment.Entities.AddFixedLine(
            entityCounter, ConvertIfcPoint2D(castedLine.m_StartPoint),
            CalculateLinePoint(castedLine, 1));
9       entityCounter++;
10   }
11   else if (entity.m_CurveGeometry.GetType() == typeof(IfcCircularArcSegment2D
         ))
12   {
13      IfcCircularArcSegment2D castedArc = (IfcCircularArcSegment2D)entity.
            m_CurveGeometry;
14
15      Point3d start = ConvertIfcPoint2D(castedArc.m_StartPoint);
16      Point3d middle = GetArcPoint(castedArc, 0.5);
```

```
17
18          //Create Civil Arc:
19          AlignmentArc civilArc = civilAlignment.Entities.AddFixedCurve(
                entityCounter, start, middle, end);
20          entityCounter++;
21      }
22      else if (entity.m_CurveGeometry.GetType() == typeof(
            IfcTransitionCurveSegment2D))
23      {
24          IfcTransitionCurveSegment2D castedTransition = (
                IfcTransitionCurveSegment2D)entity.m_CurveGeometry;
25
26          //Check Transition Type:
27          if (castedTransition.m_TransitionCurveType._value ==
                IfcTransitionCurveType.Values.CLOTHOIDCURVE)
28          {
29              float zero = 0;
30              double endRadius;
31
32              if (castedTransition.o_EndRadius == null)  //radius is infinite
33              {
34                  //create infinity
35                  endRadius = 1 / zero;
36              }
37              else
38              {
39                  endRadius = castedTransition.o_EndRadius._value;
40              }
41
42              AlignmentSpiral civilSpiral = civilAlignment.Entities.AddFloatSpiral(
                    entityCounter, endRadius, castedTransition.m_SegmentLength._value
                    ._value, CheckTheCivilTime(castedTransition.m_IsStartRadiusCCW.
                    _value), SpiralType.Clothoid);
43              entityCounter++;
44          }
45          ...
46      }
47      ...
48  }
```

C3D offers many different methods to create alignment entities based on different parameters. All these methods are further explained in the Autodesk C3D .Net API Reference Guide (Autodesk, 2017).
The following methods are also added to the plug-in to convert some IFC properties to their equivalent C3D values. These methods are:

- `Point3d ConvertIfcPoint2D(IfcCartesianPoint ifcPoint)`
  Converts a two dimensional **IfcCartesianPoint** such as the start point of an entity to a three dimensional C3D point with a fixed height of 0

- `Point3d CalculateLinePoint(IfcLineSegment2D ifcLine, double dist)`
  Calculates a C3D point along a line at a distance specified by the user (i.e. $dist = 0.5$ gives the point half way along the line's length)

- `Point3d GetArcPoint(IfcCircularArcSegment2D ifcArc, double dist)`
  Calculates a C3D point along a circular arc at a distance specified by the user (i.e. $dist = 0.5$ gives the point half way along the arc's length)

- `double ConvertIfcDirection(double ifcDir)`
  Converts the IFC direction to the according C3D direction

- `bool CheckTheCivilTime(bool isCCW)`
  Changes the IFC *isCCW* property to the equivalent C3D property

TIN Surface

Besides the horizontal alignment the plug-in also allows for the import of a TIN surface. Listing 4.13 illustrates the process.

Listing 4.13 Creating the TIN surface

```
1  int surfaceCounter = 1;
2  if (ifcEntities[i].GetType() == typeof(IfcAlignment))
3  {
4    ...
5  }
6  else if (ifcEntities[i].GetType() == typeof(IfcTriangulatedFaceSet))
7  {
8     //Get name
9     string name;
10
11    name = "Imported_TIN_Surface_" + surfaceCounter.ToString();
12    surfaceCounter++;
13
14    var castedTin = (IfcTriangulatedFaceSet)ifcEntities[i];
15    //Get PointList:
16    var pointList = castedTin.m_Coordinates.m_CoordList;
17
18    //Select a Surface style to use
19    using (Transaction civilTransactionManager = civilDatabase.
          TransactionManager.StartTransaction())
20    {
```

```
21        ObjectId styleId = civilDocument.Styles.SurfaceStyles["Cut and Fill
              Banding 0.5m Interval (2D)"];
22        ObjectIdCollection pointEntitiesIdColl = new ObjectIdCollection();
23        if (styleId != null && pointEntitiesIdColl != null)
24        {
25           //Create an empty TIN Surface
26           ObjectId surfaceId = TinSurface.Create(name, styleId);
27           ...
28        }
29     ...
30  }
```

All imported surfaces are called "Imported_TIN_Surface_ + *surfaceCounter*". After the initial and yet empty TIN surface has been created all the vertices are added to the surface (see listing 4.14). Just like the export of large surfaces, the import of large surfaces takes a long time. Therefore, the *simplifier* variable ensures that, in case the imported surface contains a lot of vertices, certain surface points are skipped during the import to speed up the process. This, of course, results in a less detailed version of the TIN surface. To actually import the whole surface mapped in the *.ifc file this variable can simply be removed.

Listing 4.14 Adding the vertices to the TIN surface

```
1  for (int k = 0; k <= pointList.Count - 1; k++)
2  {
3     //to speed up the import process of large surfaces:
4     var simplifier = 0;
5     if (pointList.Count >= 1000)
6     {
7        simplifier = 100;
8     }
9     else if (pointList.Count >= 10000)
10    {
11       simplifier = 500;
12    }
13    var point = pointList[k];
14    double x = point[0]._value;
15    double y = point[1]._value;
16    double z = point[2]._value;
17    var pt = new Point3d(x, y, z);
18    k = k + simplifier;
19    surface.AddVertex(pt);
20 }
```

4.4.7 Results

Export

Figure 4.10 and 4.11 show the exported IFC alignment "*addin_test*" when opened within *IFC Engine Viewer* (RDF Ltd., 2017). Screenshot 4.11 clearly shows the correct vertical alignment assigned to the horizontal alignment. The structure of the created *.ifc file can be further examined using *IFC Engine Editor*. (RDF Ltd., 2017) Both screenshots were used with the permission of Peter Bonsma.

Figure 4.10: "*addin_test*" within *IFC Engine Viewer* (RDF Ltd., 2017)

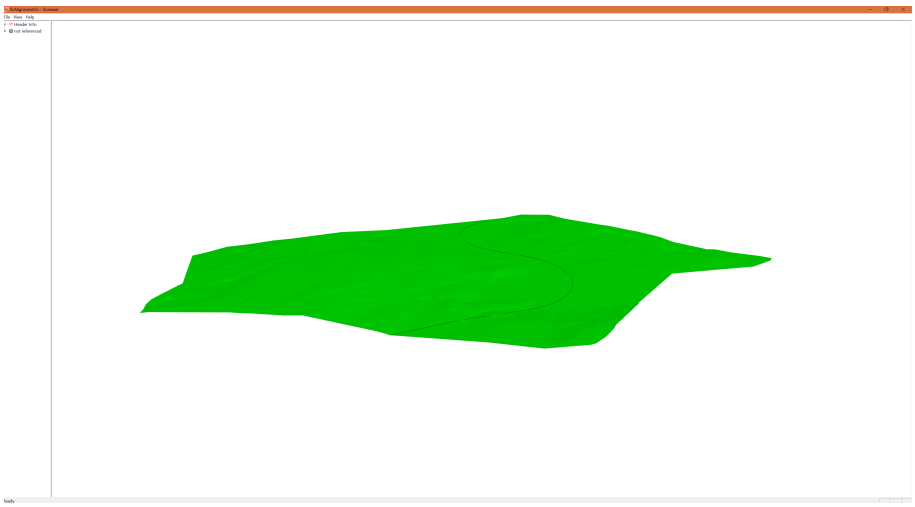


Figure 4.11: "*addin_test*" within *IFC Engine Viewer* (RDF Ltd., 2017)

Import

Figure 4.12 shows the example alignment ("*tum_example*") used by the Technical University of Munich that has priorly been exported from TUM Open Infra Platform (OIP) to an *.ifc file. The resulting file was then imported to C3D using the import function of the plug-in.

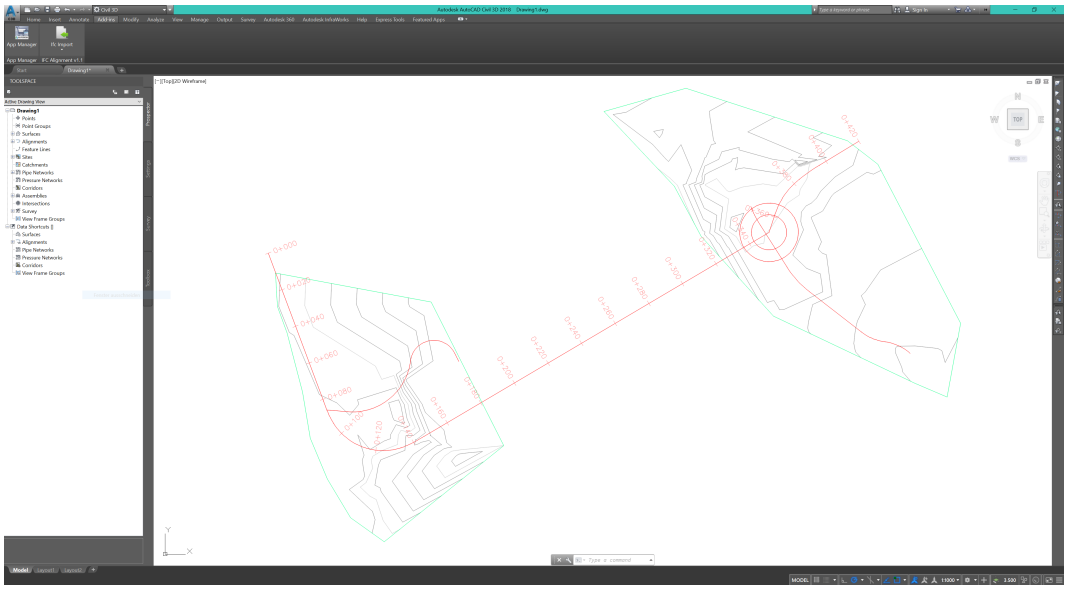


Figure 4.12: "*tum_example*" shown within C3D

Chapter 5

Programming Challenges and Critics

During the development of the plug-in a few challenges had to be overcome to get the export fully working. Although *IFC Engine* is now able to open and show the *.ifc file created by the plug-in that contains the correct and hence identical alignment to the original *.dwg drawing some changes regarding certain property definitions had to be made.
The most significant challenges that could cause the most implementation bugs will be described in this chapter.

5.1 The StartDirection Property

Below (as well as figure 5.1) I used a straight section to make my point because it is the most simple entity due to the fact that the start gradient does not change within the entity, which makes the explanation more simple and the respective figure clearer. However, the same applies for spirals and circular arcs, because only the 'start' gradient matters in this case.
On the bSI website under 'Core data schemas' (buildingSMART International, 2015), the *StartDirection* property of the **IfcCurveSegment2D** is defined as follows:

> *The direction of the tangent at the start point. Direction value 0. indicates a curve with a start tangent along the positive x-axis. Values increases counter-clockwise, and decreases clockwise. Depending on the plane angle unit, either degree or radians, the sensible range is* $-360° \leq n \leq 360°$ *(or* $-2\pi \leq n \leq 2\pi$*). Values larger then a full circle (*$> |360°|$ *or* $> |2\pi|$*) shall not be used.*

However, to enable *IFC Engine* to display the IFC alignment correctly this definition could not be matched completely within the plug-in. To understand the misleading definition given by bSI it is important to clarify the mathematical entities used. Therefore, *WolframAlpha* gives the following definition for a tangent:

> *A straight line is tangent to a given curve $f(x)$ at a point x_0 on the curve if the line passes through the point $(x_0, f(x_0))$ on the curve and has slope $f'(x_0)$, where $f'(x)$ is the derivative of $f(x)$. This line is called a tangent line, or sometimes simply a tangent.* (Weisstein, 2017)

A line is further defined by the following:

> *"A line is a straight one-dimensional figure having no thickness and extending infinitely in both directions."* (Stover & Weisstein, 2017)

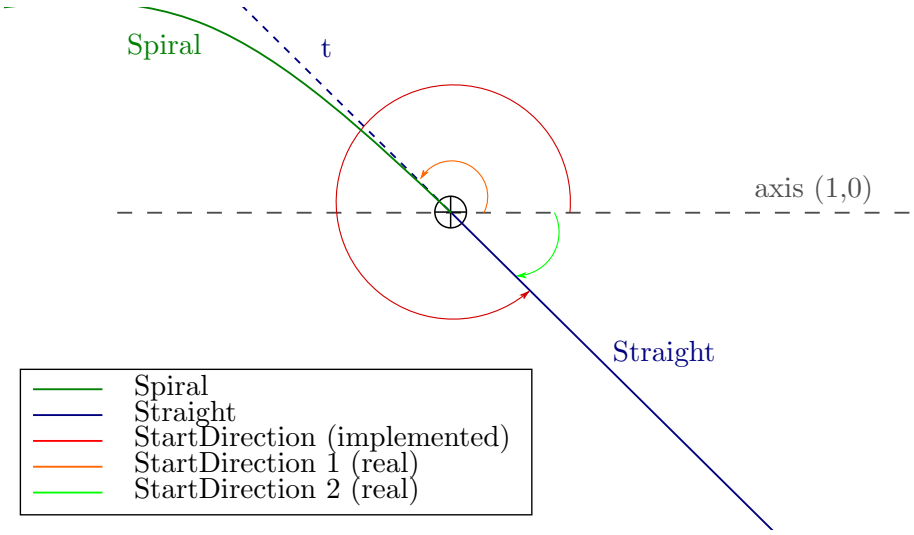


Figure 5.1: Demonstrating the different *StartDirections* possible

A tangent is always of an infinite length; consequently, the actual start direction of the straight should be either a positive value measuring clockwise – *StartDirection 1* – or a negative value measuring anti-clockwise – *StartDirection 2* (figure 5.1). *OIP* as well as *IFC Engine Viewer* implemented the *StartDirection* always measuring until the actual alignment element and not the tangent at the start point, which, in my opinion, makes the most sense. Otherwise an identification, whether the straight element would be pointing north-west or south-east (figure 5.1), would not be possible. Besides, sticking to the given definition by bSI *StartDirection* values greater than *StartDirection* $\geq 180°$ as well as values lower than *StartDirection* $\leq -180°$ would not be possible.
However, to reach maximum compatibility the plug-in implements the *StartDirection* in the

same way.
To cause less confusion and also to provide a mathematically correct definition I suggest the following changes being applied to the *StartDirection*:

> *The **orientation of the directed ray** at the start point. Direction value 0 indicates a curve with a start tangent along the positive x-axis. Values increase anti-clockwise, decrease clockwise. Depending on the plane angle unit, either degree or radians, the sensible range is* $-360° \leq n \leq 360°$ *(or* $-2\pi \leq n \leq 2\pi$*).*

However, for road alignments this definition is still not the one that should be used. When laying out a new alignment engineers use **bearing (Θ)** to describe the *direction* of an alignment segment rather than any of the definitions mentioned above. The bearing is the "direction, clockwise from North sometimes stated in degrees, minutes and seconds" (Bird, 2015a, p. 26).

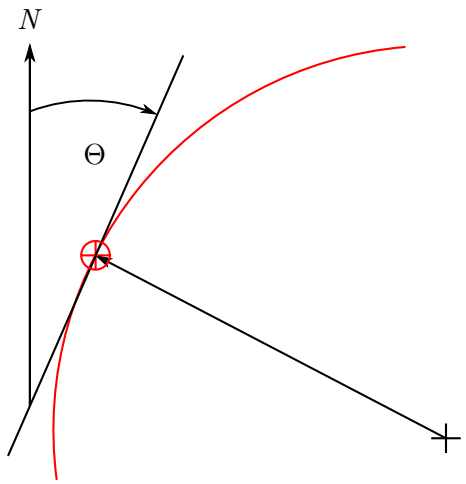


Figure 5.2: Demonstrating the bearing definition

5.2 Import Function

The import function was way more challenging to code than the export function. This was due to a number of reasons. One problem was the selection of the most suitable creation method for alignment entities. Although C3D offers a wide variety of methods there was no method that only needs input parameters, which are already given within the IFC entities. Therefore, most of the needed parameters had to be calculated using more or less complicated geometric procedures that may and did result in minor deviations to the actual course of the alignment.
I exported and imported "*addin_test*" multiple times to see whether the deviation was increasing with every import. The maximum deviation was $1.93 * 10^{-12}$ ‰, which equals a

	Import 1	Import 2	Import 3	Import 4
Segment 2: Spiral				
Start Direction	0	0	0	0
Segment Length	$-1.00 * 10^{-11}$	$-1.00 * 10^{-11}$	$-1.00 * 10^{-11}$	$-1.00 * 10^{-11}$
Start X	0	0	0	0
Start Y	0	0	0	0
Segment 4: Spiral				
Start Direction	0	0	0	0
Segment Length	$1.00 * 10^{-11}$	$1.00 * 10^{-11}$	$1.00 * 10^{-11}$	$1.00 * 10^{-11}$
Start X	0	0	0	0
Start Y	0	0	0	0
Segment 7: Arc				
Start Direction	0	$-9.66 * 10^{-13}$	$-1.99 * 10^{-12}$	$-1.99 * 10^{-12}$
Segment Length	0	0	0	0
Start X	0	0	0	0
Start Y	0	0	0	0

Table 5.1: Deviations after four export/import cycles of "*addin_test*" (all values are given in metres)

value of 10.00 picometres and did not increase anymore after the third export/import cycle. It can, therefore, be neglected. Note that the number of available IFC Alignment files is very limited at this early development stage so that this hypothesis has only been tested with the *.ifc file provided by *OIP's* export.
Table 5.1 shows the absolute deviations for the different alignment entities of "*addin_test*" after four export/import cycles. Included are only the entities that showed a deviation from the original file.

Another problem is the different mathematical representations and approximations for the various transition curves, which all result in slightly different coordinate values along the spirals.
Following situation is given: **Program *A*** uses numeric integration to approximate the **Fresnel integrals**, and calculate the clothoid's endpoint. **Program *B*** uses a **Taylor's series** approximation. The alignment is created and exported with *A* (figure 5.3) and then imported into *B*. The exported *.ifc file and the respective STEP structure is shown in figure 5.4.

To display a clothoid **program *B*** only gets the clothoid's start point (#9 in figure 5.3), its start direction, and the clothoid's length mapped in the *.ifc file. To calculate the endpoint ($P_{2.1}$) it now uses Taylor's series, which will result in a different endpoint to the one calculated within **program *A*** (P_2) illustrated in figure 5.5. In comparison the clothoid's endpoint P_2 is (195.0698/32.6948) (calculated in **program *A***) whilst $P_{2.1}$ is (195.0575/32.7428) (calculated in **program *B***). However, P_2 is also the start point of the next entity (the circular arc, #11 figure 5.4) given in the *.ifc file. To import this entity **program *B*** then uses

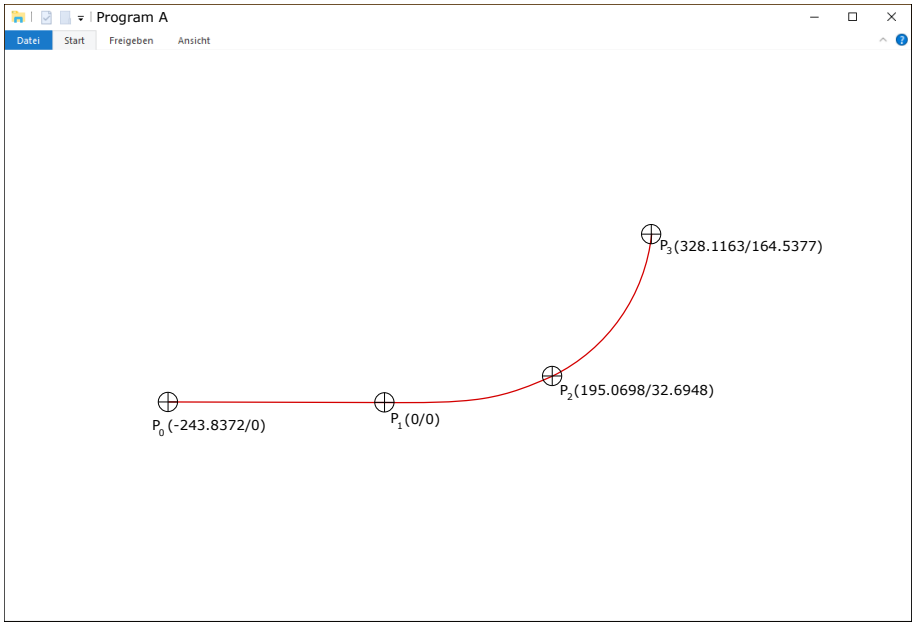


Figure 5.3: The example alignment created with **program *A***

```
#1=IFCALIGNMENT('EPRA27BVUkS4SL1sJ1FZSQ',$,'import_test',$,$,$,$,#2,$);
#2=IFCALIGNMENTCURVE(#3,$,$);
#3=IFCALIGNMENT2DHORIZONTAL(0.,(#4,#7,#10));
#4=IFCALIGNMENT2DHORIZONTALSEGMENT(.T.,$,$,#5);
#5=IFCLINESEGMENT2D(#6,0.000000000000000,243.8374442931);
#6=IFCCARTESIANPOINT((-243.8374442931,0.00000000000));
#7=IFCALIGNMENT2DHORIZONTALSEGMENT(.T.,$,$,#8);
#8=IFCTRANSITIONCURVESEGMENT2D(#9,0.000000000000000,200.,$.,200.,.T.,.T.,.CLOTHOIDCURVE.);
#9=IFCCARTESIANPOINT((0.00000000000,0.00000000000));
#10=IFCALIGNMENT2DHORIZONTALSEGMENT(.T.,$,$,#11);
#11=IFCCIRCULARARCSEGMENT2D(#12,0.498999500000500,226.385337287196,200.,.T.);
#12=IFCCARTESIANPOINT((195.069783631448,32.6948335864087));
```

Figure 5.4: "*import_test*" mapped onto the *.ifc file using **program *A***

the start point (P_2 and #11 in figure 5.4) calculated by A and mapped in the *.ifc file, which will not match $P_{2.1}$ calculated by B using Taylor's series. The same applies for the clothoid's end direction and, therefore, the start direction of the circular arc that changes from 0.498999500000500 *rad* (**program *A***) to 0.499499661700735 *rad* (**program *B***). This results in a fragmentary alignment within **program *B*** (figure 5.5).

I suspect that due to this problem the imported "*tum_example.ifc*" showed in C3D as a non-continuous alignment with the endpoints of entering transitions and the start points of the following circular arcs not being exactly identical. There are different ways on how to tackle this problem:

- All software developers agree on one mathematical approximation method for each spiral type (This is practically not possible)

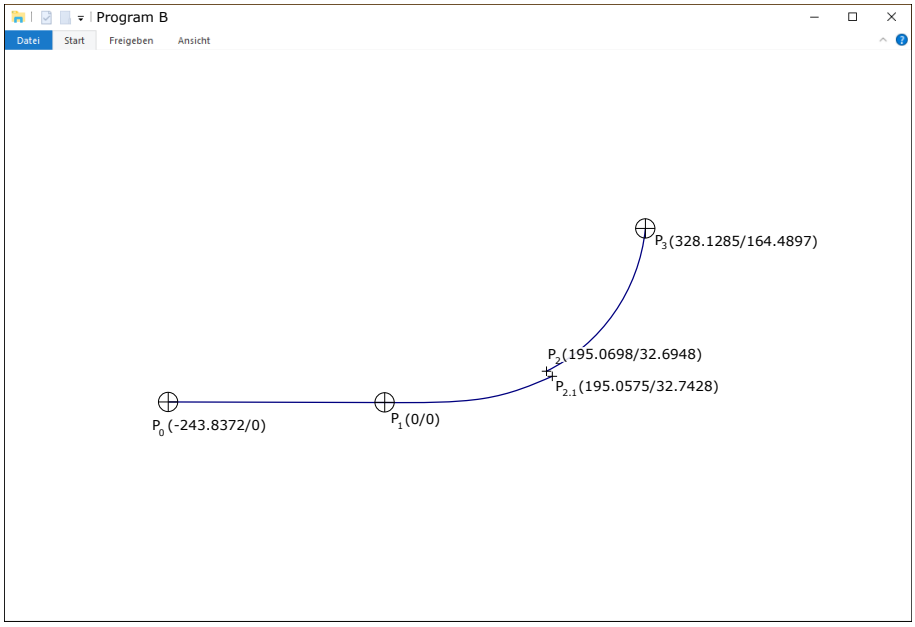


Figure 5.5: The example alignment imported into **program *B***

- Another method that recognizes a certain deviation and rescales the alignment entities accordingly (This is used within IFC Engine Viewer)

In the plug-in this method is called *CheckForVicinity(double, double, IfcCartesianPoint)* shown in listing 5.1. Additionally two double variables have been added to the beginning of the import called *compareX* and *compareY*. After adding a horizontal alignment entity *compareX* becomes the x-value and *compareY* the y-value of the respective entity. When adding the next entity the *CheckForVicinity* method is being called to check whether the deviation of its start point given in the imported *.ifc file and the endpoint (calculated by C3D) of the previous entity is less than 0.001 m. If that is the case the plug-in then replaces the start point given in the *.ifc file by the endpoint of the previous entity to guarantee a continuous alignment.

Listing 5.1 CheckForVicinty method included in the plug-in

```
1 private static Point3d CheckForVicintiy(double compareX, double compareY,
      IfcCartesianPoint ifcPoint)
2 {
3    double xValue = ifcPoint.m_Coordinates[0]._value;
4    double yValue = ifcPoint.m_Coordinates[1]._value;
5
6    double deviation = Math.Sqrt(Math.Pow(Math.Abs(compareX - ifcPoint.
         m_Coordinates[0]._value), 2) + Math.Pow(Math.Abs(compareY - ifcPoint.
         m_Coordinates[1]._value), 2));
7
8    if (deviation <= 0.001)
9    {
10      xValue = compareX;
11      yValue = compareY;
12   }
13
14   Point3d civilPoint = new Point3d(xValue, yValue, 0);
15   return civilPoint;
16 }
```

Chapter 6

Conclusion

6.1 Prospect

By providing the opportunity to now map road alignments onto IFC within the IFC Alignment 1.0 extension, another very important goal in developing a "strong standard for a comprehensive description of infrastructure constructions" (Amann & Borrmann, 2015, p. 10) has been reached. The proposed IFC Alignment 1.1 already expands the application range by implementing different types of transition curves.
However, this is only a first step – in order to fully describe infrastructure projects the IFC format has to be able to map cross sections, tunnels, and bridges as well, which is one of the future developing goals of bSI (buildingSMART International, 2017). In 2015 the China Railway BIM Alliance made a first proposal on how to implement railway design in IFC. At the International bSI Summit in Barcelona in April 2017, the results of the IFC RAIL project were presented and discussed and might be available in the process of a future extension. (buildingSMART International, 2017)
Although the bSI brand is already known all over the world, bSI is still doing extensive work to "increase [their] membership among organisations" (buildingSMART International, 2016) in order to widen the usage and development of IFC.

6.2 Resolution

In order to reach the named goals, a lot of challenges still have to be overcome. As described above, many possible implementation errors and confusions still exist and sometimes even assumptions were made to create a working IFC export function.
Another aspect, which resulted in a much harder implementation and occasionally constitutes the biggest hurdle for a uniform IFC implementation, and maximum interoperability, is the

fact that a free IFC binding (early or late), that provided and also recommended for use by bSI on their website for everyone to use who wants to create or read IFC files, does not exist. bSI does make certain suggestions on which programs to use but a lot of them are commercial or out of date. An IFC binding for the most common programming languages, which is easily accessible and downloadable from the bSI official website, could, therefore, significantly improve the interoperability of different IFC implementations (such as plug-ins etc.).

The C# early binding developed within the creation process of this thesis will be available in a future release at: https://bitbucket.org/FlixFix/c3difcalignmentplugin.

Due to my lack of experience and the early release of the early binding, I am aware of the fact that it will not be bug free at this stage. However, it does work correctly for the C3D plug-in and thus should work for implementation within other programs as well. It is published in the hope that more experienced software developers might use and improve the early binding to make it more usable and 'bullet proof'.

This could mark a further step towards making IFC a lot more accessible for a wider range of people. I consider that this should be the main goal for everyone using and spreading IFC – as a file format that focuses on interoperability rather than individual development and profit for a single company.

> *We believe significant improvements in cost, value and environmental performance can be achieved through the use of open sharable asset information in the creation and operation of civil infrastructure and buildings worldwide.* (buildingSMART International, 2017)

Appendix A

Tables

Horizontal Alignment							
Entity-Nr.	Chainage	Entity Type	Tangential Continuity	Start Point X	Start Point Y	Start Direction (rad)	Start Direction (°)
1	0,0000000	Line	true	4679,6679893	7036,7660248	0,6197090	35,5067118
2	624,6796488	Spiral	true	5188,1868881	7399,5789148	0,6197090	35,5067118
3	629,6796488	Circular Arc	true	5192,2522754	7402,4896829	0,6247090	35,7931907
4	1156,0649861	Spiral	true	5396,9789976	7861,2969108	-4,6057056	-263,8874929
5	1161,0649861	Line	false	5396,4300237	7866,2666766	-4,6007056	-263,6010140
6	1191,7143469	Spiral	true	5393,0141112	7896,7250885	-4,6007056	-263,6010140
7	1196,7143469	Circular Arc	true	5392,4485745	7901,6929968	-4,5957056	-263,3145351
8	1694,0690037	Spiral	true	5117,5552953	8291,6421524	-3,6009963	-206,3218896
9	1699,0690037	Line	false	5113,0663495	8293,8442673	-3,5959963	-206,0354107
10	1849,4702970	Spiral	true	4977,9273363	8359,8593877	-3,5959963	-206,0354107
11	1854,4702970	Circular Arc	true	4973,4383905	8362,0615026	-3,6009963	-206,3218896
12	2811,6412839	Spiral	true	4847,8512219	9169,9239261	0,7678470	43,9943948
13	2816,6412839	Line	false	4851,4598131	9173,3848528	0,7628470	43,7079159
Length:	2937,669092	m					

IFCStart Direction (°)	Segment Length	Radius	isCCW	Start Radius	End Radius	IsStartRadiusCCW	IsEndRadiusCCW	TransitionCurveType
54,4932882	624,6796488	×	×	×	×	×	×	×
54,4932882	5,000000	×	×	∞	500	true	true	Clothoid
54,2668093	526,3853373	500	true	×	×	×	×	×
353,8874929	5,000000	×	×	500	∞	true	true	Clothoid
353,6010140	30,6493608	×	×	×	×	×	×	×
353,6010140	5,000000	×	×	∞	500	true	true	Clothoid
353,3145351	497,3546568	500	true	×	×	×	×	×
296,3218896	5,000000	×	×	500	∞	true	true	Clothoid
296,0354107	150,4012933	×	×	×	×	×	×	×
296,0354107	5,000000	×	×	∞	500	false	false	Clothoid
296,3218896	957,1709870	500	false	×	×	×	×	×
46,0056052	5,000000	×	×	500	∞	false	false	Clothoid
46,2920841	121,0278084	×	×	×	×	×	×	×

Figure A.1: "addin_test" horizontal entities

Vertical Alignment								
Entity-Nr.	Chainage (StartDistAlong)	Entity Type	Horizontal Length	Start Height	Start Gradient	isConvex (Curve Type)	Parabola Constant
1	0,000000	Line	227,5475098	138,0000000	0,0042415	x	x
2	227,5475098	Parabolic Arc	150,0000000	138,9651445	0,0042415	false	2437,4879832
3	377,5475098	Line	42,4524902	134,9859633	-0,0572973	x	x
4	420,0000000	Parabolic Arc	150,0000000	132,5535520	-0,0572973	true	514743,3562945
5	570,0000000	Line	65,0000000	123,9808190	-0,0570058	x	x
6	635,0000000	Parabolic Arc	150,0000000	120,2754387	-0,0570058	true	3423,1697270
7	785,0000000	Line	305,0000000	115,0109890	-0,0131868	x	x
8	1090,0000000	Parabolic Arc	150,0000000	110,9890110	-0,0131868	true	25412,4754183
9	1240,0000000	Line	430,0000000	109,4536850	-0,0072842	x	x
10	1670,0000000	Parabolic Arc	150,0000000	106,3214788	-0,0072842	true	17948,9035238
11	1820,0000000	Line	830,0000000	105,8556279	0,0010729	x	x
12	2650,0000000	Parabolic Arc	150,0000000	106,7460984	0,0010729	true	10831,0338258
13	2800,0000000	Line	137,6690924	107,9457088	0,0149219	x	x
Length:	2937,6690924	m					

Figure A.2: "addin_test" vertical entities

Bibliography

A. Pirti, M. Ali Yücel (2012). The Fourth Degree Parabola (Bi-Quadratic Parabola) as a transition curve.

Amann, J. & Borrmann, A. (2015). Open BIM for Infrastructure – mit OKSTRA und IFC Alignment zur internationalen Standardisierung des Datenaustauschs.

Autodesk (2016). About Spiral Definitions. https://knowledge.autodesk.com/support/autocad-civil-3d/learn-explore/caas/CloudHelp/cloudhelp/2017/ENU/Civil3D-UserGuide/files/GUID-DD7C0EA1-8465-45BA-9A39-FC05106FD822-htm.html. Accessed: 2016-05-22.

Autodesk (2017). Autodesk Civil 3D .Net API Reference Guide. http://docs.autodesk.com/CIV3D/2018/ENU/API_Reference_Guide/index.html. Accessed: 2016-08-11.

Bird, R. (2015a). Land, Traffic and Highways: WS 2015/16 - Arcs.

Bird, R. (2015b). Land, Traffic and Highways: WS 2015/16 - Transitions.

Bird, R. (2015c). Land, Traffic and Highways: WS 2015/16 - Vertical.

Borrmann, A., König, M., Koch, C. & Beetz, J. (2015). *Building Information Modeling: Technologische Grundlagen und industrielle Praxis*. VDI-Buch. Wiesbaden: Springer Vieweg.

buildingSMART International (2015). Core data schemas. http://www.buildingsmart-tech.org/mvd/IFC4x1/Alignment/1.0.1/html/. Accessed: 2016-08-11.

buildingSMART International (2016). Annual Report 2016. http://buildingsmart.org/annual-report-2016/. Accessed: 2016-08-12.

buildingSMART International (2017). buildingSMART International Homepage. http://buildingsmart.org/about/about-buildingsmart/. Accessed: 2016-05-09.

Freudenstein, S. (2016). Verkehrswegebau Grundkurs.

International Organization for Standardization (2016). ISO 10303-21:2016.

LandXML (2016). Transition curves in Road Design.

Li, Z., Gold, C. & Zhu, Q. (2005). *Digital terrain modeling: Principles and methodology.* New York: CRC Press.

Liebich, T. (2017). IFC Alignment 1.1 project, IFC Schema extension proposal.

Liebich, T., Amann, J., Borrmann, A., Lebegue, E., Marache, M. & Scarponcini, P. (2015a). IFC Alignment.

Liebich, T., Amann, J., Borrmann, A., Lebegue, E., Marache, M. & Scarponcini, P. (2015b). IFC Alignment Project, Conceptual Model: (informative).

Natzschka, H. (1997). *Straßenbau: Entwurf und Bautechnik ; mit 178 Tabellen.* Stuttgart: Teubner.

RDF Ltd. (2017). IFC Engine. Accessed: 2016-08-11.

Stover, C. & Weisstein, E. W. (2017). Line, From MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/Line.html. Accessed: 2016-08-11.

Weisstein, E. W. (2017). Tangent Line, From MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/TangentLine.html. Accessed: 2016-08-11.

Confidentiality Statement

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, 20. September 2017

Felix Rampf

Felix Rampf
Gollierstraße 35
D-80339 München
e-Mail: f.rampf@tum.de