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
Supervisor: Prof. Dr.-Ing. André Borrmann
Advisor: Julian Amann
Submission Date: 26. September 2017
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 Historical Background .................................................. 1
   1.2 Why Industry Foundation Classes? .................................. 2

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

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.

![Horizontal road alignment including straight, transition, and arc sections (created in C3D)](image)

**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
2.2. Description of Alignment Elements

- 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

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:

\[ x^2 + y^2 = R^2 \]
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)

\[ R_{\text{min}} = \frac{V^2}{3.6^2 \times g \times (0.925 \times f_T \times n + q)} \, [m] \]

with:

- \( V = \text{curve design speed} \, [\text{km} / \text{h}] \)
- \( g = \text{acceleration of gravity} \, [\text{m} / \text{s}^2] \)
- \( n = \text{maximum utilisation of radial adhesive coefficient} \, [-] \)
- \( q = \text{slope} \, [-] \)
- \( f_T = \text{tangential adhesive coefficient based on tables} \, [-] \)

To guarantee good driving dynamics it is not only important that the curvature does not change within a curve: \( k = 1/r = \text{const.} \), 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.
2.2. Description of Alignment Elements

\[ A^2 = R \times L \quad [m^2] \]  \hspace{1cm} (2.3)

The variable \( A \) is called flatness or homothetic parameter and works as a stretching factor for the clothoid.
For the design of a road alignment the length of the transition curve is calculated with the following formula (Bird, 2015b, p. 15):

\[
\text{Transition Length} = \frac{V^3}{46.656 \times q \times R} \quad [m]
\]

with:

\[
V = \text{curve design speed} \quad [\frac{m}{s^2}]
\]

\[
q = \text{change in radial acceleration} \quad [\frac{m}{s^3}]
\]

\[
R = \text{radius} \quad [m]
\]

Due to safety reasons and to provide the driver with adequate reaction time the minimum value \(A_{\text{min}}\) should not be less than \(A_{\text{min}} = \frac{R}{3}\) but also not exceed \(A_{\text{max}} = R\). (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.
2.2. Description of Alignment Elements

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):

$$ g = \frac{\Delta h \times 100}{l} \ [%] $$

with:

$\Delta h = \text{difference in level between the two intersection points} \ [m]$

$l = \text{horizontal distance between the two intersection points} \ [m]$ (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)
### Table 2.1: Maximum gradient depending on the different Entwurfsklassen (road classes) in Germany adapted from Freudenstein (2016, p. II/23)

<table>
<thead>
<tr>
<th>Entwurfklasse</th>
<th>max. g (%)</th>
<th>Entwurfklasse</th>
<th>max. g (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EKA 1 A</td>
<td>4.0</td>
<td>EKL 1</td>
<td>4.5</td>
</tr>
<tr>
<td>EKA 1 B</td>
<td>4.5</td>
<td>EKL 2</td>
<td>5.5</td>
</tr>
<tr>
<td>EKA 2</td>
<td>4.5</td>
<td>EKL 3</td>
<td>6.5</td>
</tr>
<tr>
<td>EKA 3</td>
<td>6.0</td>
<td>EKL 4</td>
<td>8.0</td>
</tr>
</tbody>
</table>

'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:

$$ L = K \times (g_2 - g_1) \times 100 \ [m] $$

with:

$$ g_2 = \text{gradient of the leaving tangent} \ [-] $$

$$ g_1 = \text{gradient of the entering tangent} \ [-] $$

**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):

\[
h_x = h_s + g_1 \cdot x + \frac{x^2}{200 \cdot K} \quad [m]
\]

with:

- \( h_s = \text{starting height of the parabola} \quad [m] \)
- \( g_1 = \text{starting gradient} \quad [-] \)
- \( x = \text{horizontal distance along the curve} \quad [m] \)

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)

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)

![Diagram of alignment and its segments](image)

**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)](image)

### 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 **IfcAlignment2DHorizontal** Segment 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).

![Class diagram of the IFC horizontal alignment part](image)

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
3.4. Horizontal Alignment

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 \( \text{rad}, \frac{\pi}{m} \). 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

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
3.4. Horizontal Alignment

- 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):

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

From the \( A \) value the transition curve length \( L \) can be derived:

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

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 \( \alpha \) (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 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 \( R \times L = \text{const} \). 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):
3.4. Horizontal Alignment

- x-values:

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

- y-values:

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

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:

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

- y-values:

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

**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:

\[ x = l - 0.02190112582400869 \frac{l^3}{R^3} \ [m] \quad (3.7) \]
- y-values:

\[
y = l \times \left( 0.1413363707560822 \times \frac{l}{R} - 0.0026731818162654 \times \frac{l^3}{R^3} \right) \quad [m]
\] (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:

\[
x = l - 0.0226689447 \times \frac{l^3}{R^2} \quad [m]
\] (3.9)

- y-values:

\[
y = 0.14867881635766 \times \frac{l^2}{R} \quad [m]
\] (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:

\[
x = l \quad [m]
\] (3.11)

- y-values:

\[
y = \frac{l^3}{6RL} \quad [m]
\] (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:

$$x_I = l \quad [m]$$ (3.13)

- y-values:

$$y_I = \frac{l^4}{6RL_E^2} \quad [m]$$ (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 $2\alpha$ around the ending point P of parabola I. The turning angle $\alpha$ 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](image)

**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:
  \[
  x = l - 0.0226689447 \frac{l^3}{R^2} \quad [m]
  \]  
  (3.15)

- y-values:
  \[
  y = l \times (0.1486788163576622 \frac{l}{R} - 0.0027402322400286 \frac{l^3}{R^3}) \quad [m]
  \]  
  (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 Tangential-Continuity* 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 simply 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.

![IfcAlignment2DVerSegLine](image)

Figure 3.11: IfcAlignment2DVerSegLine

3.5.2 IFC Vertical Circular Arc Segment

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.

![IfcAlignment2DVerSegCircularArc](image)

Figure 3.12: IfcAlignment2DVerSegCircularArc

3.5.3 IFC Vertical Parabolic Arc Segment

The vertical parabolic arc segment (IfcAlignment2DVerSegParabolicArc) is similar to the circular arc segment but it also has a 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

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

```csharp
using Autodesk.AutoCAD.ApplicationServices;
using Autodesk.AutoCAD.DatabaseServices;
using Autodesk.AutoCAD.Runtime;
using Autodesk.Civil;
using Autodesk.Civil.ApplicationServices;
using Autodesk.Civil.DatabaseServices;
using Autodesk.Private.Windows;
using Autodesk.Windows;
using Surface = Autodesk.Civil.DatabaseServices.Surface;

//The IFC class library
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**

```csharp
namespace C3DIfcAlignmentPlugIn.PlugIn
{
    public class C3DIfcAlignmentAddIn
    {
        [CommandMethod("ifc", CommandFlags.Session)]
        public void Main()
        {
            ...
        }
    }
}
```

### 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](image)
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 \texttt{ifc} command is being called from the command line a new \textbf{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 \texttt{ifc} command more than once in one session.

\textbf{Listing 4.3 Creating the new button}

```csharp
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 }
```
//1. Create new Split-Button
RibbonSplitButton rbSplitButton = new RibbonSplitButton();

//2. Create Panel
RibbonPanel Panel = new RibbonPanel();

//3. Create Panel Source
RibbonPanelSource Source = new RibbonPanelSource();

//4. Create Ribbon Button 1
RibbonButton rbb1 = new RibbonButton();
//...Define Button 1 Attributes
rbb1.Name = "Export IFC..."; //Define Button 1 Attributes
rbb1.Text = "IFC Export";

//Create Tooltip Button 1
RibbonToolTip rbnTT = new RibbonToolTip();
rbnTT.Command = "IFC"; //Define Button 1 Tool-Tip

//5. Create Ribbon Button 2
RibbonButton rbb2 = new RibbonButton();

//...Define Button 2 Attributes and Tooltip

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.
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

```csharp
public class ExportCmdHandler : System.Windows.Input.ICommand
{
    public bool CanExecute(object parameter)
    {
        return true;
    }
    public event EventHandler CanExecuteChanged;
    public void Execute(object parameter)
    {
        //select active Document
        if (parameter is RibbonButton) //if Button is clicked...
            {
                RibbonButton button = parameter as RibbonButton;
                //Open SaveFile-DIALOG
                Microsoft.Win32.SaveFileDialog dlg = new Microsoft.Win32.SaveFileDialog();
                dlg.FileName = "ifcAlignment"; // Default File Name
                dlg.DefaultExt = ".ifc"; // Default File Extension
                dlg.Filter = "IFC Files (.ifc)|*.ifc"; //Filter Files by Extension
                if (result == true)
                    {
                        ...
                    }
    }
}
```

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.
4.4. The Civil 3D Plug-In

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

```csharp
var civilSurfaceCollection = civilDocument.GetSurfaceIds();
foreach (ObjectId surfaceId in civilSurfaceCollection)
{
    var civilSurface =
        (Surface) civilTransactionManager.GetObject(surfaceId, OpenMode.ForRead);
    if (civilSurface is TinSurface)
    {
        //IFC Geometric Representation Context
        IfcGeometricRepresentationContext ifcGeometricRepresentationContext =
            new IfcGeometricRepresentationContext(){...};
        ifcWriter.insertEntity(ifcGeometricRepresentationContext);

        //IFC Triangulated Face Set
        IfcTriangulatedFaceSet ifcTriangulatedFaceSet = new
            IfcTriangulatedFaceSet(){...};
        ifcWriter.insertEntity(ifcTriangulatedFaceSet);

        //IFC Cartesian Point List 3D
        IfcCartesianPointList3D vertexList = new IfcCartesianPointList3D(){...};
        ifcTriangulatedFaceSet.m_Coordinates = vertexList;
        ifcWriter.insertEntity(vertexList);

        var civilTinTriangleCollection = civilTinSurface.GetTriangles(false);
        int vertexCounter = 1;
        foreach (TinSurfaceTriangle civilTinTriangle in civilTinTriangleCollection )
        {
            var vertices = new List<IfcBase>();
            var triangle = new List<IfcBase>();

            var vertex1 = CreateIfcPoint3D(civilTinTriangle.Vertex1.Location.X,
                Location.Z);
            vertices.Add(vertex1);
            triangle.Add(new IfcPositiveInteger() { _value = new IfcInteger() {
                _value = vertexCounter } });
            vertexCounter++;
```

```csharp
```
Horizontal Alignment

<table>
<thead>
<tr>
<th>Entity-Nr.</th>
<th>Chainage</th>
<th>Entity Type</th>
<th>Tangential Continuity</th>
<th>Start Point X</th>
<th>Start Point Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>624.6796488</td>
<td>Spiral</td>
<td>true</td>
<td>5188.186881</td>
<td>7399.5789148</td>
</tr>
<tr>
<td>3</td>
<td>629.6796488</td>
<td>Circular Arc</td>
<td>true</td>
<td>5192.2522754</td>
<td>7402.4896829</td>
</tr>
<tr>
<td>4</td>
<td>1156.0649861</td>
<td>Spiral</td>
<td>true</td>
<td>5396.9789976</td>
<td>7861.2969108</td>
</tr>
<tr>
<td>5</td>
<td>1161.0649861</td>
<td>Line</td>
<td>false</td>
<td>5396.4300237</td>
<td>7866.2666766</td>
</tr>
</tbody>
</table>

**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 ObjectIds 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**

```csharp
foreach (ObjectId alignId in civilAlignmentCollection)
{
    var civilAlignment =
        (Alignment) civilTransactionManager
            .GetObject(alignId, OpenMode.ForRead); //Open File for Read

    //Ifc Alignment
    IfcAlignment ifcAlignment = new IfcAlignment(){...};
    ifcWriter.insertEntity(ifcAlignment);

    //IfcAlignmentCurve
    IfcAlignmentCurve ifcAlignmentCurve = new IfcAlignmentCurve();
    ifcWriter.insertEntity(ifcAlignmentCurve);

    //IfcAlignment2DHorizontal
    IfcAlignment2DHorizontal ifcAlignment2DHorizontal = new
        IfcAlignment2DHorizontal(){...};
    ifcWriter.insertEntity(ifcAlignment2DHorizontal);

    //Get Alignment Entities
    var civilAlignmentEntityList = SortC3DAlignmentEntities(civilAlignment);
```
foreach (var horizontalAlignmentEntity in civilAlignmentEntityList)
{
    var civilAlignmentSubentityCounter = horizontalAlignmentEntity.SubEntityCount;
    for (var i = 0; (i <= (civilAlignmentSubentityCounter - 1)); i++)
    {
        var civilAlignmentSubentity = horizontalAlignmentEntity[i];
        switch (civilAlignmentSubentity.SubEntityType)
        {
            case AlignmentSubEntityType.Arc:
                var alignSubEntArc = (AlignmentSubEntityArc) civilAlignmentSubentity;
                IfcAlignment2DHorizontalSegment ifcHorSegArc = new IfcAlignment2DHorizontalSegment(){...};
                IfcCircularArcSegment2D ifcCircularArc = new IfcCircularArcSegment2D()
                {...};
                //Assign Property Values:
                ...
                //Insert Entities:
                ifcWriter.insertEntity(ifcHorSegArc);
                ifcWriter.insertEntity(ifcCircularArc);
                ifcWriter.insertEntity(ifcArcStartPoint);
                //Start Next Element:
                break;
            case AlignmentSubEntityType.Line:
                //Create IfcLineSegment2D:
                ...
                //Assign Property Values:
                ...
                //Insert Entities:
                ...
                //Start Next Element:
                break;
            case AlignmentSubEntityType.Spiral:
                //Create IfcTransitionCurveSegment2D
                ...
                //Assign Property Values:
                ...
                //Insert Entities:
                ...
                //Start Next Element:
                break;
        }
    }
}
4.4. The Civil 3D Plug-In

**Vertical Alignment**

![Diagram of vertical alignment in Civil 3D]

**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

```csharp
int ver_counter = 1; // Set Vertical Alignment Counter
foreach (ObjectId profile_id in align.GetProfileIds())
{
    // Opens the Profile for read
    Profile profile = (Profile)acTrans.GetObject(profile_id, OpenMode.ForRead);
    if (profile.StyleName != "Existing Ground Profile")
    {
        // Profile is Vertical Alignment
        ...
    }
    ...
}
```
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)**

```csharp
string ver_align_name = profile.Name.ToString(); //Get Profile Name
int ver_ent_counter = 1; //Set Element Counter

foreach (ProfileEntity ver_align_Ent in profile.Entities)
{
    switch (ver_align_Ent.EntityType) //if Element-Type is a...
    {
        //...Tangent:
        case ProfileEntityType.Tangent:
            ProfileTangent ver_tangent = (ProfileTangent)ver_align_Ent;
            //IfcAlignment2DVerSegLine:
            IfcAlignment2DVerSegLine ifcVerSegLine = new IfcAlignment2DVerSegLine() { ... }; //Assign Property Values:
            ...
            ifcWriter.insertEntity(ifcVerSegLine);
            //Start Next Element:
            ver_ent_counter++;
            break;
        //... Asymmetric Parabola:
        case ProfileEntityType.ParabolaAsymmetric:
            ProfileParabolaAsymmetric ver_parabola_a = (ProfileParabolaAsymmetric) ver_align_Ent;
            //IfcAlignment2DVerSegParabolicArc:
            IfcAlignment2DVerSegParabolicArc ifcVerSegParabola1 = new IfcAlignment2DVerSegParabolicArc() { ... }; //Assign Property Values:
            //Create Parabola 1:
            ...
            ifcWriter.insertEntity(ifcVerSegParabola1);
            //Start Next Element:
            ver_ent_counter++;
            break;
        //... Symmetric Parabola:
        case ProfileEntityType.ParabolaSymmetric:
```
4.4. The Civil 3D Plug-In

ProfileParabolaSymmetric ver_parabola_s = (ProfileParabolaSymmetric) ver_align_Ent;

//IfcAlignment2DVerSegParabolicArc:
IfcAlignment2DVerSegParabolicArc ifcVerSegParabola =
new IfcAlignment2DVerSegParabolicArc(){...};

//Assign Property Values:
...

//Insert Entities:
...

//Start Next Element:
ver_ent_counter++;
break;

//...Circular Arc:
case ProfileEntityType.Circular:
ProfileCircular ver_circ = (ProfileCircular)ver_align_Ent;
//Ifc VerSegCircularArc:
IfcAlignment2DVerSegCircularArc ifcVerSegArc =
new IfcAlignment2DVerSegCircularArc(){...};

//Assign Property Values:
...

//Insert Entities:
...

//Start Next Element:
ver_ent_counter++;
break;
}

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 \textit{ParabolaConstant} property (which is the same for the entering and the exiting parabola) can be calculated with the C3D genuine property \( K \):

\[
\text{ParabolaConstant} = K \times 100 \text{ [m]} \tag{4.1}
\]

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)

\textbf{Listing 4.9} Getting the properties of the second parabola of an asymmetric parabola

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

The \textit{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.

\subsection*{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
4.4. The Civil 3D Plug-In

- 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](image)

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

```csharp
for (int i = 1; i <= ifcEntityCounter; i++)
{
    if (ifcEntities[i].GetType() == typeof(IfcAlignment))
    {
        IfcAlignment castedAlignment = (IfcAlignment)ifcEntities[i];
        //Create C3D Alignment:
```
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`**

```csharp
//Get the Alignment Curve:
IfcAlignmentCurve ifcCurve = (IfcAlignmentCurve)castedAlignment.m_Axis;

//Get Horizontal/Vertical Alignment:
IfcAlignment2DHorizontal ifcAlignmentHorizontal = ifcCurve.m_Horizontal;
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**

```csharp
foreach (var entity in ifcAlignmentHorizontal.m_Segments)
{
    if (entity.m_CurveGeometry.GetType() == typeof(IfcLineSegment2D))
    {
        IfcLineSegment2D castedLine = (IfcLineSegment2D)entity.m_CurveGeometry;

        //Create Civil Line:
        AlignmentLine civilLine = civilAlignment.Entities.AddFixedLine(
            entityCounter, ConvertIfcPoint2D(castedLine.m_StartPoint),
            CalculateLinePoint(castedLine, 1));
        entityCounter++;  
    }
    else if (entity.m_CurveGeometry.GetType() == typeof(IfcCircularArcSegment2D))
    {
        IfcCircularArcSegment2D castedArc = (IfcCircularArcSegment2D)entity.m_CurveGeometry;

        Point3d start = ConvertIfcPoint2D(castedArc.m_StartPoint);
        Point3d middle = GetArcPoint(castedArc, 0.5);
```
4.4. The Civil 3D Plug-In

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

```csharp
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   name = "Imported_TIN_Surface_" + surfaceCounter.ToString();
11    surfaceCounter++;
12
13   var castedTin = (IfcTriangulatedFaceSet)ifcEntities[i];
14   //Get PointList:
15   var pointList = castedTin.m_Coordinates.m_CoordList;
16
17   //Select a Surface style to use
18   using (Transaction civilTransactionManager = civilDatabase.
19       TransactionManager.StartTransaction())
20   {
```
4.4. The Civil 3D Plug-In

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

```csharp
for (int k = 0; k <= pointList.Count - 1; k++)
{
    //to speed up the import process of large surfaces:
    var simplifier = 0;
    if (pointList.Count >= 1000)
    {
        simplifier = 100;
    }
    else if (pointList.Count >= 10000)
    {
        simplifier = 500;
    }
    var point = pointList[k];
    double x = point[0].Value;
    double y = point[1].Value;
    double z = point[2].Value;
    var pt = new Point3d(x, y, z);
    k = k + simplifier;
    surface.AddVertex(pt);
}
```
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 counterclockwise, and decreases clockwise. Depending on the plane angle unit, either degree or radians, the sensible range is \(-360^\circ \leq n \leq 360^\circ\) (or \(-2\pi \leq n \leq 2\pi\)). Values larger than a full circle (> \(|360^\circ|\) 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)

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* ≥ 180° as well as values lower than *StartDirection* ≤ −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 \textit{StartDirection}:

The orientation of the directed ray at the start point. Direction value $\theta$ 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 $-360^\circ \leq n \leq 360^\circ$ (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 \textbf{bearing (}\(\Theta\)\textbf{)} 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](image)

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 $1.93 \times 10^{-12}$ $\%$, which equals a
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 (P2,1) it now uses Taylor’s series, which will result in a different endpoint to the one calculated within program A (P2) illustrated in figure 5.5. In comparison the clothoid’s endpoint P2 is (195.0698/32.6948) (calculated in program A) whilst P2,1 is (195.0575/32.7428) (calculated in program B). However, P2 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
the start point \((P_2\text{ and } #11\text{ in figure 5.4})\) 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

```
private static Point3d CheckForVicinty(double compareX, double compareY,
    IfcCartesianPoint ifcPoint)
{
    double xValue = ifcPoint.m_Coordinates[0]._value;
    double yValue = ifcPoint.m_Coordinates[1]._value;

        m_Coordinates[0]._value), 2) + Math.Pow(Math.Abs(compareY - ifcPoint.
        m_Coordinates[1]._value), 2));

    if (deviation <= 0.001)
    {
        xValue = compareX;
        yValue = compareY;
    }

    Point3d civilPoint = new Point3d(xValue, yValue, 0);
    return civilPoint;
}
```
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 plugins 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/c3dificalignmentplugin. 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
<table>
<thead>
<tr>
<th>Entity-Nr.</th>
<th>Chainage</th>
<th>EntityType</th>
<th>Tangential Continuity</th>
<th>Start Point X</th>
<th>Start Point Y</th>
<th>Length (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4679,6679893</td>
<td>Line</td>
<td>false</td>
<td>0,7628470</td>
<td>43,7079159</td>
<td>2937,669092</td>
</tr>
<tr>
<td>11</td>
<td>5117,5552953</td>
<td>Circular Arc</td>
<td>true</td>
<td>0,6197090</td>
<td>8362,0615026</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>5392,4485745</td>
<td>Spiral</td>
<td>true</td>
<td>-4,6007056</td>
<td>2811,6412839</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>7901,6929968</td>
<td>Circular Arc</td>
<td>true</td>
<td>0,6197090</td>
<td>8362,0615026</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>1061,0649861</td>
<td>Spiral</td>
<td>true</td>
<td>-263,3145351</td>
<td>4847,8512219</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>7399,5789148</td>
<td>Line</td>
<td>true</td>
<td>-263,6010140</td>
<td>43,7079159</td>
<td>35,5067118</td>
</tr>
<tr>
<td>16</td>
<td>2937,669092</td>
<td>Spiral</td>
<td>true</td>
<td>0,6247090</td>
<td>2811,6412839</td>
<td>8</td>
</tr>
<tr>
<td>17</td>
<td>1161,0649861</td>
<td>Spiral</td>
<td>true</td>
<td>-263,3145351</td>
<td>4847,8512219</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>7901,6929968</td>
<td>Circular Arc</td>
<td>true</td>
<td>0,6197090</td>
<td>8362,0615026</td>
<td>10</td>
</tr>
<tr>
<td>19</td>
<td>1061,0649861</td>
<td>Spiral</td>
<td>true</td>
<td>-263,3145351</td>
<td>4847,8512219</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>7399,5789148</td>
<td>Line</td>
<td>true</td>
<td>-263,6010140</td>
<td>43,7079159</td>
<td>35,5067118</td>
</tr>
<tr>
<td>21</td>
<td>2937,669092</td>
<td>Spiral</td>
<td>true</td>
<td>0,6247090</td>
<td>2811,6412839</td>
<td>8</td>
</tr>
<tr>
<td>22</td>
<td>1161,0649861</td>
<td>Spiral</td>
<td>true</td>
<td>-263,3145351</td>
<td>4847,8512219</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>7399,5789148</td>
<td>Line</td>
<td>true</td>
<td>-263,6010140</td>
<td>43,7079159</td>
<td>35,5067118</td>
</tr>
<tr>
<td>24</td>
<td>2937,669092</td>
<td>Spiral</td>
<td>true</td>
<td>0,6247090</td>
<td>2811,6412839</td>
<td>8</td>
</tr>
<tr>
<td>25</td>
<td>1161,0649861</td>
<td>Spiral</td>
<td>true</td>
<td>-263,3145351</td>
<td>4847,8512219</td>
<td>2</td>
</tr>
<tr>
<td>26</td>
<td>7399,5789148</td>
<td>Line</td>
<td>true</td>
<td>-263,6010140</td>
<td>43,7079159</td>
<td>35,5067118</td>
</tr>
<tr>
<td>27</td>
<td>2937,669092</td>
<td>Spiral</td>
<td>true</td>
<td>0,6247090</td>
<td>2811,6412839</td>
<td>8</td>
</tr>
</tbody>
</table>

**Entity-Nr.**: Identification number for each entity.
**Chainage**: Distance from a reference point.
**EntityType**: The type of geometric entity (Line, Circular Arc, Spiral).
**Tangential Continuity**: Indicates whether the entity is tangentially continuous.
**Start Point X, Y**: Coordinates of the start point of the entity.
**Length**: The length of the entity.
<table>
<thead>
<tr>
<th>Start Direction</th>
<th>FCS</th>
<th>Start Radius</th>
<th>Segment Length</th>
<th>Radius</th>
<th>isCCW</th>
<th>End Radius</th>
<th>isStartRadiusCCW</th>
<th>End Direction</th>
<th>Transition Curve Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>FCS</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>x</td>
<td>0.5</td>
<td>false</td>
<td>0.5</td>
<td>true</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A.1: “adding test” horizontal entities
<table>
<thead>
<tr>
<th>Entity-Nr.</th>
<th>Chainage (StartDistAlong)</th>
<th>Entity Type</th>
<th>Horizontal Length</th>
<th>Start Height</th>
<th>Start Gradient</th>
<th>isConvex (Curve Type)</th>
<th>Parabola Constant</th>
<th>Length:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1080,0000000</td>
<td>Line</td>
<td>1</td>
<td>137,6690924</td>
<td>0,0000000</td>
<td>false</td>
<td>1</td>
<td>2880,0000000</td>
</tr>
<tr>
<td>2</td>
<td>1279,8301238</td>
<td>Line</td>
<td>227,5475098</td>
<td>138,0000000</td>
<td>0,0042415</td>
<td>false</td>
<td>0,0000000</td>
<td>1151,8301238</td>
</tr>
<tr>
<td>3</td>
<td>2437,4879832</td>
<td>Line</td>
<td>42,4524902</td>
<td>134,9859633</td>
<td>-0,0572973</td>
<td>false</td>
<td>0,0000000</td>
<td>2543,4879832</td>
</tr>
<tr>
<td>4</td>
<td>514743,3562945</td>
<td>Line</td>
<td>65,0000000</td>
<td>123,9808190</td>
<td>-0,0570058</td>
<td>true</td>
<td>0,0000000</td>
<td>5172,4743629</td>
</tr>
<tr>
<td>5</td>
<td>785,0000000</td>
<td>Line</td>
<td>305,0000000</td>
<td>115,0109890</td>
<td>-0,0131868</td>
<td>true</td>
<td>0,0000000</td>
<td>1090,0000000</td>
</tr>
<tr>
<td>6</td>
<td>1090,0000000</td>
<td>Parabolic Arc</td>
<td>420,0000000</td>
<td>132,5535520</td>
<td>-0,0572973</td>
<td>true</td>
<td>0,0000000</td>
<td>1731,0000000</td>
</tr>
<tr>
<td>7</td>
<td>1240,0000000</td>
<td>Line</td>
<td>180,0000000</td>
<td>120,9890110</td>
<td>-0,0131868</td>
<td>true</td>
<td>0,0000000</td>
<td>2440,0000000</td>
</tr>
<tr>
<td>8</td>
<td>1670,0000000</td>
<td>Parabolic Arc</td>
<td>430,0000000</td>
<td>110,9890110</td>
<td>-0,0131868</td>
<td>true</td>
<td>0,0000000</td>
<td>3270,0000000</td>
</tr>
<tr>
<td>9</td>
<td>1820,0000000</td>
<td>Line</td>
<td>830,0000000</td>
<td>109,4536850</td>
<td>-0,0072842</td>
<td>true</td>
<td>0,0000000</td>
<td>2650,0000000</td>
</tr>
<tr>
<td>10</td>
<td>2650,0000000</td>
<td>Parabolic Arc</td>
<td>137,6690924</td>
<td>106,3214788</td>
<td>-0,0072842</td>
<td>true</td>
<td>0,0000000</td>
<td>3420,0000000</td>
</tr>
<tr>
<td>11</td>
<td>2800,0000000</td>
<td>Line</td>
<td>137,6690924</td>
<td>107,9457088</td>
<td>0,0149219</td>
<td>true</td>
<td>0,0000000</td>
<td>3570,0000000</td>
</tr>
<tr>
<td>12</td>
<td>2800,0000000</td>
<td>Parabolic Arc</td>
<td>137,6690924</td>
<td>106,7460984</td>
<td>0,0010729</td>
<td>true</td>
<td>0,0000000</td>
<td>3790,0000000</td>
</tr>
</tbody>
</table>

Figure A.2: "addin_Jest" vertical entities
Bibliography

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


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