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

Sketch-Based Alignment Design

Cara Anna Coetzee

Bachelorthesis
für den Bachelor of Science Studiengang Umweltingenieurwesen

Autor: Cara Anna Coetzee
Matrikelnummer: [redacted]
Betreuer: Štefan Jaud, M.Sc.

Ausgabedatum: 15. Mai 2019
Abgabedatum: 15. Oktober 2019

Abstract

This thesis presents the development of a sketch-based alignment design tool for the early planning phases of infrastructure projects. Designing an alignment, the main axis of any linear infrastructure project, is an iterative and time-consuming process. Once a preliminary design is found, it is manually converted to CAD software for the detail design - a laborious task. An interactive, sketch-based alignment design tool, however, could enable the automatic reconstruction of preliminary designs in domain-specific CAD software. For this tool, a sketching functionality and a sketch-interpreting algorithm were developed. The sketch-interpreting algorithm splits the alignment sketch into a sequence of alignment elements (lines, circular arcs and clothoids) by evaluating the curvature along the drawn curve. The findings lay the groundwork for automatically reconstructing the alignment.

Zusammenfassung

In dieser Arbeit wird die Entwicklung eines skizzenbasierten Trassenentwurfstools für frühe Planungsphasen vorgestellt. Der Entwurf der Trasse, die Hauptachse eines linearen Infrastrukturbauwerks, ist ein iterativer und zeitaufwändiger Prozess.
Wenn ein erster Entwurf gefunden ist, muss er manuell in CAD Software rekonstruiert werden, welches ebenfalls einen hohen Arbeitsaufwand einfordert. Ein interaktives, skizzenbasiertes Trassenentwurfstool hingegen könnte die automatische Rekonstruktion eines frühen Entwurfs in CAD Software ermöglichen. Für dieses Tool wurden eine Skizzierfunktionalität sowie ein Algorithmus, der die Skizze anschließend interpretiert, entwickelt. Der Skizzen-Interpretations-Algorithmus zerlegt die Trassenskizze in eine Abfolge von Trassenelementen (Linien, Kreisbögen und Klothoiden), indem er die Krümmung entlang der gezeichneten Kurve auswertet. Die Ergebnisse liefern die Grundlagen für die automatische Rekonstruktion einer Trassenskizze.

Contents

Abbreviations

2D	two-dimensional
3D	three-dimensional
bSI	buildingSmart International
CAD	Computer-Aided Design
SBIM	Sketch-Based Interfaces for Modeling
IFC	Industry Foundation Classes
CDP	Collaborative Design Platform
TUM	Technical University of Munich
UI	User interface
WIMP	Window Icon Menu Pointer
LCD	Liquid Crystal Display
AEC	Architecture, Engineering and Construction
FM	Facility Management
IAI	International Alliance for Interoperability
ha	Horizontal Alignment
va	Vertical Alignment

Chapter 1

Introduction

In the planning process of linear infrastructure projects, the alignment is the highest form of abstraction. It represents the main axis of the planned road, railway or tunnel. Determining a good course for the alignment from the origin to the destination is an iterative process and depends largely on the terrain. Once a preliminary alignment design is decided, the engineer makes use of domain-specific Computer-Aided Design (CAD) software such as *AutoCAD Civil3D* [1] or *ProVI* [2] for the detail design. To this end, the engineer has to manually convert the preliminary alignment design to a formal model in the CAD software.
This reconstruction process is a time-intensive and laborious task that can result in the loss of original intentions and annotations [1]. What if it were possible to automate the reconstruction process?

1.1 Motivation

Sketching is a natural way to communicate ideas quickly [2]. If it were possible to automatically recreate an alignment from a sketch, more time could be spent on evaluating several alignments in terms of their technical and economical viability, possibly leading to a better and more efficient choice of design.

What makes this form of conceptual design so appealing is that the sketching could also be done by someone without any knowledge of formal modeling syntax [1], or of particular alignment design requirements, for that matter. In practice, a client could visually communicate what they are looking for, and the outcome of their sketch could immediately be tested and improved upon by the engineer afterwards.

[1] autodesk.com
[2] provi-cad.de

Furthermore, Company *et al* [3] argue that the „design process needs non-sequential thought“. CAD systems, however, follow a sequential work flow. The engineer only starts using the CAD tool with a specific preliminary design in mind. A graphical, interactive tool on the other hand would allow users to play around with different ideas. Such a *sketch-based alignment design* tool would make that possible - and simultaneously speed up the reconstruction process.

1.2 Related work

Sketch-based modeling is a well-studied topic. Applications range from garment design to uses in CAD in the automotive, industrial and architectural industries [4, 5, 6, 7]. The main problem in sketch-based modeling is translating a 2D sketch into a 3D model of the sketched object [2]. The lack of information about the third dimension in the sketch makes this translation process very difficult. On top of that, sketches are inherently ambiguous: how they are interpreted depends on the view point and knowledge of the beholder - be it a person or computer.

Although there is a lot of research on sketch-based modeling in other industries, very little exists for sketch-based *alignment* design specifically. An approach by McCrae and Singh [9, 29] fits a sequence of clothoid segments to a sketched curve. The result is a fair curve that approximates the sketched curve within a specific error tolerance. However, their approach does not consider the fact that an alignment is typically composed of line, circular arc *and* transition curve segments such as the clothoid.

1.3 Problem statement

The aim of this thesis is to develop a sketch-based alignment design tool as a medium for early-stage alignment design. To this end, a sketching functionality and a sketch-interpreting algorithm that allows the automatic reconstruction of an alignment sketch as a sequence of alignment elements (lines, circular arcs and clothoids) are to be developed.

1.4 Objectives

In order to develop the sketch-based alignment design tool, the objectives of this thesis are:

1. to develop a sketching functionality,
2. to develop a sketch-interpreting algorithm, and
3. to test and to evaluate the sketching functionality as well as the sketch-interpreting algorithm.

1.5 Thesis structure

Chapter 2 establishes the theoretical framework for the development of a sketch-based alignment design tool. It studies alignment design (Section 2.1), existing sketch-based modeling approaches (Section 2.2) and alignment modeling using Industry Foundation Classes (IFC) (Section 2.3). Chapter 3 details the method chosen for the development of the sketch-based alignment design tool. The design of the implementation is described in Chapter 4. The implementation of the sketch-based alignment design tool is described in Chapter 5 and evaluated in Chapter 6. Finally, Chapter 7 summarises the findings and discusses the suitability of the sketch-based approach for alignment design specifically.

Chapter 2

Theoretical Framework

Before discussing the method, design and implementation of the new sketch-based alignment design tool, it is necessary to establish the theoretical framework. The first section of this chapter gives more insight into how an alignment is typically designed by engineers today (Section 2.1). Section 2.2 deals with the basics behind developing user interfaces for sketch-based modeling (Sketch-Based Interfaces for Modeling (SBIM)). Finally, Section 2.3 describes the IFC that can be used to model an alignment and thus might be useful for the implementation of the sketch-based alignment design tool.

2.1 Alignment Design

buildingSmart International bSI defines an alignment as "*a reference system to position elements, mainly for linear construction works, such as roads, rails, bridges, and other.*" [12]. The traditional approach in alignment design is based on the superposition of two 2D curves: the horizontal and the vertical alignment. The horizontal alignment consists of a sequence of lines, circular arcs and transition curves whereas the vertical alignment is composed of lines, circular arcs and parabolic arcs. Figure 2.1 shows the superposition of the horizontal alignment (bottom bold line) and the vertical alignment (upper bold line). More detail on the notation will be given in Chapter 2.3.

When developing an alignment, the engineer considers restrictions imposed by the terrain such as existing human settlements and nature reserves that cannot be crossed, as well as other roads or railways that have to be intersected. Furthermore, there are geometric requirements to be met. For example, the longitudinal slope of road must guarantee a minimum sight distance. At the same time, there has to be minimum cross slope to ensure the drainage of the pathway. In Germany, these requirements are specified separately for highways and

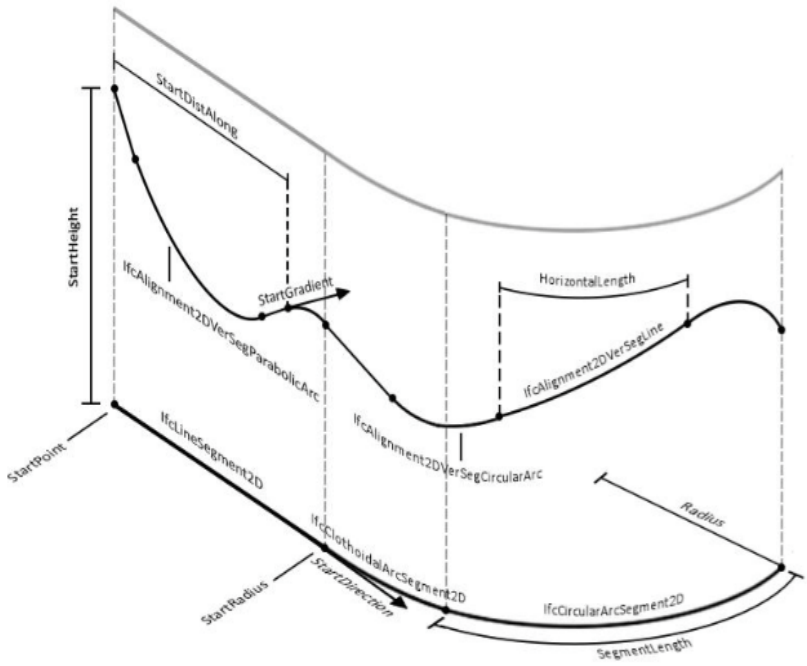

Figure 2.1: Superposition of horizontal (bottom bold line) and vertical (upper bold line) alignment. More information on the notation follows in Chapter 2.3. Retrieved from Wijnholts [8].

roads [13, 14]. The engineer's task is to harmonise these requirements while simultaneously limiting costly earth works.

The components of an alignment curve, the horizontal and the vertical alignment, can be parametrised by points of intersection or segments. The first representation takes all sections of the curve with zero curvature into account to find cross points. These cross points are the points of intersection that, when given as a sequential array of points, entirely describe the curve. The segment representation on the other hand defines each alignment element as its own segment with its own parameters [20]. Figure 2.2 shows these two representation possibilities for a horizontal alignment consisting of lines (black), circular arcs (green) and transition curves (red). The parameters included denote the following:

- b_j : bearing of the tangent at the beginning of the j-th segment (azimuth angle).
- l_j : length of the j-th segment.
- R_j / R_i : radius of the curve in [m] of the j-th segment or i-th point of intersection.
- A_j / A_i^a, A_i^b : transition curve parameter of the j-th segment or i-th point of intersection, before (b) or after (a) the circular arc.

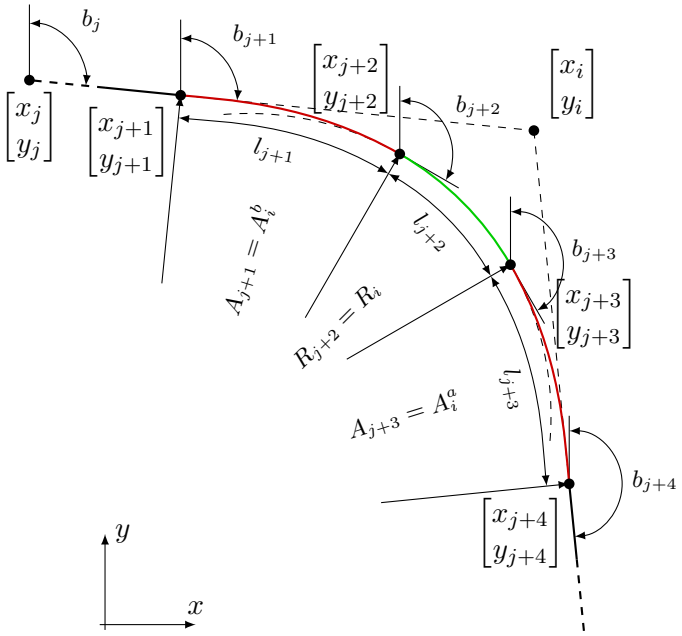

Figure 2.2: Representation of a horizontal alignment, by points of intersection ($[x_i, y_i]^T$) or by segments and their start points ($[x_j, y_j]^T$), together with additional parameters, depending on the type of segment [20].

Clothoids

The clothoid is a spiral whose curvature changes linearly with the distance travelled along its length [19]. As Table 2.1 illustrates, the clothoid is one of the most employed transition curves in horizontal alignment design [20].

The clothoid is formally described by the clothoid parameter A:

$$A^2 = R \cdot L \tag{2.1}$$

The parameter A describes the rate at which the curvature changes along the curve. The smaller A is, the faster the rate at which the curvature increases or decreases [19].

Fresnel Integrals parametrise the clothoid spiral using the arc length parameter t. B is a positive scaling parameter for the slope of linear curvature variation for a family of spirals [9]:

$$B\pi \begin{pmatrix} C(t) \\ S(t) \end{pmatrix} \tag{2.2}$$

Type	Alignment	Curve	Transition curve
Roadway	HA	circular arc	clothoid
	VA	circular arc, parabolic arc	-
Railway	HA	circular arc	clothoid, Bloss curve, Vienna curve, sinusoid, cosinusoid, cubic parabola, biquadratic parabola, Schramm curve
	VA	circular arc, parabolic arc	-
Magnetic levitation tracks	HA	circular arc	clothoid, sinusoid
	VA	circular arc	clothoid

Table 2.1: Curves used for the alignment in different infrastructure types. All types and alignments include straight elements which are not shown in the table. Reproduced with permission from Markič *et al* [20].

with

$$C(t) = \int_0^t \cos\frac{\pi}{2}u^2 du \tag{2.3}$$

and

$$S(t) = \int_0^t \sin\frac{\pi}{2}u^2 du \ . \tag{2.4}$$

The integrals are problematic for the computation of the coordinates. However, they can be approximated by a series or numerical integration. A computationally efficient rational approximation for clothoids is as follows [21]:

$$C(t) \approx \frac{1}{2} - R(t)sin(\frac{1}{2}\pi(A(t) - t^2)) \tag{2.5}$$

and

$$S(t) \approx \frac{1}{2} - R(t)cos(\frac{1}{2}\pi(A(t) - t^2)) \tag{2.6}$$

with

$$R(t) = \frac{0.0506t + 1}{1.79t^2 + 2.054t + \sqrt{2}} \tag{2.7}$$

and

$$A(t) = \frac{1}{0.803t^3 + 1.886t^2 + 2.524t + 2} \ . \tag{2.8}$$

Figure 2.3 depicts a clothoid spiral computed with this approximation.

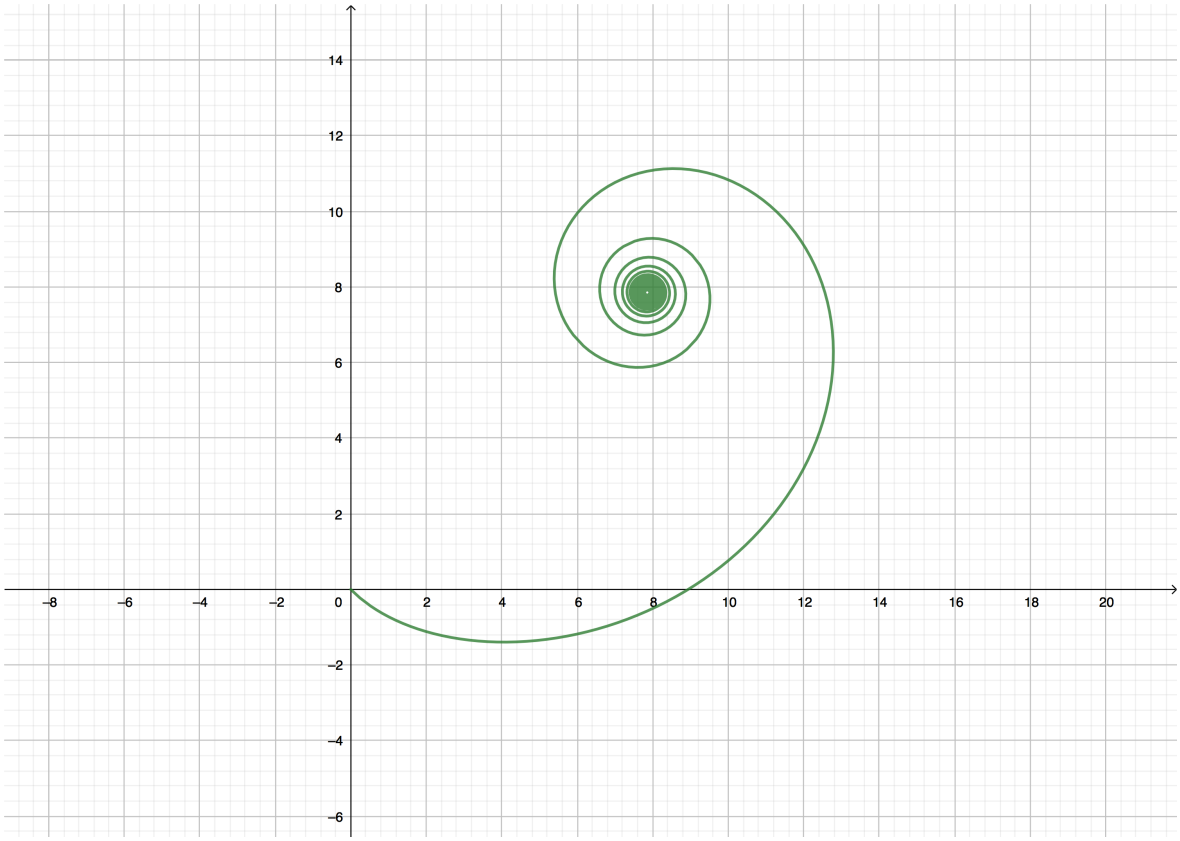

Figure 2.3: Approximated clothoid spiral for t ∈ [0,15] and B = 5 using Eq. (2.2) with Eqs.(2.5), (2.6).

2.2 Sketch-Based Interfaces for Modeling (SBIM)

A sketch made with pen and paper can be a very effective way of communicating an idea quickly. Replace pen and paper with a tablet and your finger and you have a digital modeling interface - a Sketch-Based Interface for Modeling (SBIM) [2].

2.2.1 Historical development of SBIM

Traditionally, CAD systems are designed according to the Window Icon Menu Pointer (WIMP) paradigm, such as one would find in a web browser. This paradigm may be useful if the complete design has already been determined, but proves inadequate for non-ordered, inceptive ideas that have not yet reached fruition. In the early 1990s, Microsoft developed *Windows for Pen Computers* [3]. The system, consisting of a Liquid Crystal Display (LCD)

tablet and a stylus, was a forerunner to the tablets widely in use today, but failed commercially due to the limited processing power available at the time.

Existing SBIM approaches can be grouped into gestural and reconstructional approaches. Gestural interfaces basically replace the selection of icons and menus by graphic gestures, resulting in an interactive, more intuitive user experience. Reconstructional approaches, on the other hand, extract and rebuild the sketched object's geometry from the pen-and-paper drawing automatically. Chronologically, reconstructional interfaces were developed first, since they required less processing power. One of the earliest examples of this is the attempt to extract information from analogous engineering plans and blueprints [3]. These are drawn up in complex views such as cuts or particular views and include many annotations to denote tolerances and dimensions. The multiple layers and angles the information is presented in make it very difficult for the computer to extract the object's geometry. Apart from insufficient processing power, this perception problem has proved to be one of the biggest challenges that developers face when designing SBIM products and solutions.

2.2.2 The SBIM Pipeline

According to Olsen *et al* [2], the SBIM pipeline consists of three steps: sketch acquisition, filtering and interpretation.

(1) Sketch acquisition: A sketch consists of one or more strokes, a "*stroke*" being a temporal sequence of points. A stroke begins once the user's finger touches the input device and ends once it is lifted again. The resemblance between the actual sketch and the recorded points hinges on the drawing speed. The faster the user sketches, the fewer points can be sampled along the sketch.

(2) Filtering: Filtering the input sketch serves the purpose of reducing noise produced by the user and the input device. Naturally, the user's hand is not completely steady when drawing, which results in deviations from straight lines and smooth curves. Further, the input device comes with built-in digitisation noise as the recognition of the input depends on the sensitivity of the mechanical hardware. Amongst others, reducing this noise can be achieved by minimising the maximum distance of any point to a straight line approximation [10], or, simpler, by discarding points within a certain threshold distance or time interval from one another.

(3) Interpretation: A sketch can be interpreted in numerous ways, depending on the view point and knowledge of the beholder - the computer, in this case. This makes assigning meaning to the sketch the most difficult step. As the name suggests, the value of SBIM

lies in their role as an intersection between conceptual sketch and formal model. Instead of manually converting an analogous sketch to a formal model in CAD software, SBIM makes it possible to automate this step, saving time and reducing the risk of losing information in the transition. The quality of the result - the CAD model - naturally depends on a good interpretation of the sketch.

2.3 Industry Foundation Classes (IFC)

The Industry Foundation Classes (IFC) standard [1] is an open data model standard "*intended to enable interoperability between building information modeling software applications in the AEC/FM industry*" [22]. Interoperability in this context is "*the loss-free exchange of data between software products by different vendors*" [23]. In the AEC/FM industry, there are several companies at play from the planning to the construction of a typical project. Moreover, the companies involved are typically small and medium-sized enterprises that collaborate only for the duration of a specific project [22, 23]. The fact that the sector is so fragmented makes it equally difficult and important to improve interoperability between different software applications used by different companies throughout the project. The uninhibited digital flow of project-related design, cost and production information could significantly reduce redundancy and uncertainty, and thereby increase the overall efficiency of the project delivery. For example, a study conducted in the USA in 2007 revealed that costs linked to manually re-entering data from one software application to another and checking document versions accounted for approximately 3 % of total project budgets [24]. This figure affirms the cost-saving potential of a common standard such as IFC that facilitates the data exchange between software applications from separate vendors.

buildingSmart International bSI, formerly the International Alliance for Interoperability (IAI), has been developing IFC for buildings since 1995. The object-oriented data model describes both the geometric and semantic aspects of buildings [25]. In recent years, there has been an increased interest to extend the schema to other domains as well. The BuildingSmart Infrastructure Room is dedicated to delivering an extension of the schema for roads, bridges, tunnels, ports and waterways. *IfcAlignment*, the extension relevant to this thesis, was added as part of the IFC 4x1 release in 2018 [12]. The most important extension is the description of the alignment's geometry, i.e. the *IfcAlignmentCurve* entity. It can be used to model the alignment of any linear infrastructure project, such as a road or bridge.

Figure 2.4 shows the alignment attributes inherent to *IfcAlignmentCurve*. The alignment is modelled as an *IfcAlignmentCurve*, which can be split into a horizontal alignment *IfcAlignment2DHorizontal* and a vertical alignment *IfcAlignment2DVertical*, as also discussed in

[1] ISO 16739:2018

Chapter 1. Each includes more entities that contain further attributes based on the requirements of the specific segment type.

In the case of the horizontal alignment, *IfcAlignment2DHorizontalSegment* is an intermediary entity that contains the attributes identical to all segments irrespective of segment type. These are specified within the "CurveGeometry" attribute and include the "StartPoint", "StartDirection" and "SegmentLength" of a horizontal segment. The subsequent entities *IfcCircularArcSegment2D*, *IfcLineSegment2D* and *IfcTransitionCurveSegment2D* inherit these attributes from their parent entity *IfcAlignment2DHorizontalSegment*. While *IfcLineSegment2D* requires no further attributes, *IfcCircularArcSegment2D* additionally lists the radius ("Radius") and orientation of the arc ("IsCCW"). *IfcTransitionCurveSegment3D* specifies radius and arc orientation for both start and end of the segment ("StartRadius", "EndRadius", "IsStartRadiusCCW", "IsEndRadiusCCW") and the kind of transition curve used ("TransitionCurveType") [26].

As to the vertical alignment, *IfcAlignment2DVertical* contains a list of all segments ("Segments"). Each segment is a *IfcAlignment2DVerticalSegment* with a starting point and segment length both defined and measured along the horizontal alignment ("StartDistAlong", "HorizontalLength"). Further, "StartHeight" supplies the z-coordinate to the x- and y-coordinates specified by the starting point "StartDistAlong". Finally, the tangent of the starting point is specified as "StartGradient". If the segment is a line, an *IfcAlignment2DVerSegLine*, no further attributes are required for its description. For an *IfcAlignment2DVerSegCircularArc*, as in the horizontal alignment, the radius ("Radius") and orientation of the arc ("IsConvex") are given. Similarly, *IfcAlignment2DVerSegParabolicArc* specifies the the parabola constant and orientation of the parabola ("ParabolaConstant" and"IsConvex") [27].

Figure 2.1 shown in Section 2.1 nicely shows how these entities and their attributes come together to describe an alignment curve.

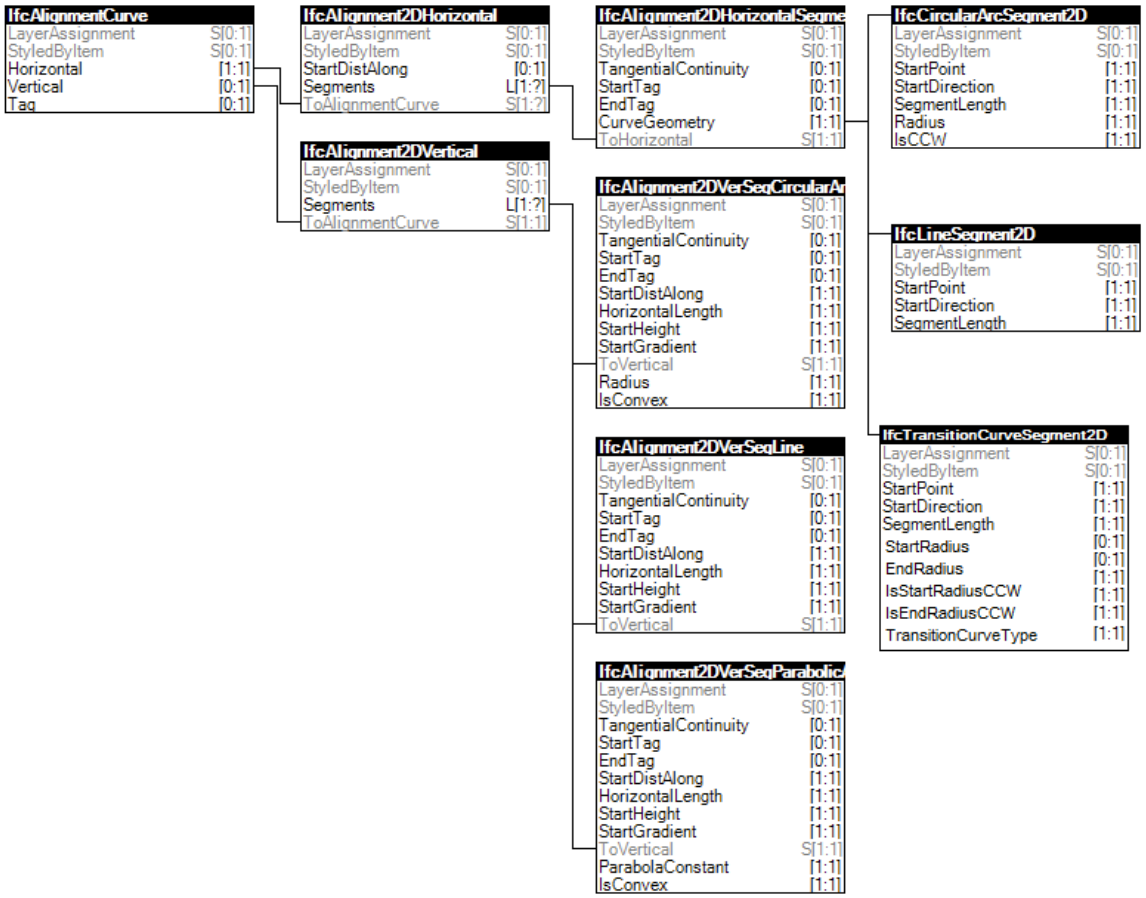

Figure 2.4: Entity inheritance diagram for *IfcAlignmentCurve*. Based on the *IfcAlignment* instance diagram [12].

Chapter 3

Method

This chapter defines the requirements according to which the sketch-based alignment design tool will be implemented (Section 3.1) and details the approach that will be used to split the alignment sketch into a viable sequence of alignment elements (Section 3.2).

3.1 Requirements

In the following section, the requirements according to which the sketching functionality will be implemented are stipulated. The requirements are split into functional (f) and non-functional requirements (nf).

1. The use of the tool should be intuitive and easy (nf)
To this end, there will be a *sketch* button. This should clarify that once this button is selected, the user enters the *sketch* mode. As when sketching with pen and paper, the alignment sketch will comprise all points from the moment the finger touches the screen until it is lifted again. Multiple touch events supplementing each other are not foreseen.

2. It is not allowed for a sketched line to cross itself (f)
This requirement is set to simplify the subsequent sketch interpretation. If a line were to cross itself, one would have to guess whether the intersection is intended at separate levels or not. Solving this problem goes beyond the scope of this thesis.

3. The new functionality should be integrated into the existing process and structure as far as possible (nf)
As the sketching functionality will be an extension to an existing system, it would be sensible to integrate it into the existing process and class structure as far as possible. This means making use of the existing UI, class structure and methods where possible.

4. The sketch-interpreting algorithm should split the alignment sketch into a viable sequence of alignment elements (f)
The purpose of the sketch-interpreting alignment is to split the alignment curve into a viable sequence of alignment elements. In general, a circular arc may be followed by a line or another circular arc; a line may only be succeeded by a circular arc (see Chapter 1).

5. The clothoid will be implemented as a first transition curve (f)
Due to its ubiquity in alignment design (as shown in Table 2.1), this implementation limits itself to the clothoid as a first transition curve. However, the class structure should be such that it is easy to add more transition curves at a later stage.

6. The new sketching functionality will only be implemented for the horizontal alignment design (f)
As this implementation tries to support the drawing of clothoids (see requirement 5), which are employed in the design of horizontal alignments, the sketching functionality and sketch-interpreting algorithm will only be tested for the horizontal alignment. If the sketch-interpreting algorithm works, a variation of it could be added for the vertical alignment design at a later stage.

3.2 Curvature approach

The horizontal alignment is composed of a sequence of lines, circular arcs and transition curves. These elements are distinguishable by their curvature. Curvature is a mathematical concept that describes the changing direction of a curve [15]. Overall, the curvature κ is inversely proportional to the radius of the curve R:

$$\kappa = \frac{1}{R} \tag{3.1}$$

Let's consider equation (3.1) for three basic alignment elements: lines, circular arcs and transition curves.

For lines, essentially curves with radius $R = \infty$, the curvature is close to zero ($\kappa \approx 0$).

Regarding circular arcs, the curvature κ is a constant value inversely proportional to its radius R ($\kappa = const.$). This implies that the bigger the radius of a circle, the smaller its curvature.

As to transition curves, there are multiple kinds in use, such as the clothoid curve, biquadratic parabola, bloss curve, cosine curve, cubic parabola and sine curve [16]. However, the one property they all have in common is smooth curvature that changes continuously with the distance travelled along their length. For this reason, they are placed in between lines and circular arcs. Specifically, they either connect a line with a circular arc segment or two

circular arc segments with different radii [17, 18]. On a road, this allows the driver to adjust the steering wheel gradually while driving in or out of a curve, which in turn results in a gradual change of centrifugal acceleration and thus a more comfortable driver experience. Also, transition curves absorb the torsion resulting from the different lateral gradients and ultimately create a visually pleasing alignment [19].

As curvature is the curve property that distinguishes these different alignment elements from one another, it could be used to identify separate line, circular arc and clothoid segments within the sketch. Such a sketch-interpreting algorithm could exploit the curvature property to find *segments* within the alignment sketch where the curvature κ remains close to zero (= line), is a constant value other than zero (= circular arc), or changes linearly (= clothoid).

When interpreting a sketched curve, Baran *et al* [11] argue that the approximating curve should satisfy two demands: fairness and fidelity.

(1) Fairness:
The sketch-interpreting algorithm should strictly enforce C^2 continuity between two consecutive curve segments in order to obtain a final composite curve with smooth curvature.

(2) Fidelity:
The resulting alignment curve should not deviate too far from the alignment sketch as this would defy the whole purpose of having a user sketch in the first place.

The sketch-interpreting algorithm can thus only be a compromise between these two conflicting demands.

The sketch-interpreting approach proposed by McCrae and Singh [9, 29] fits a sequence of clothoid segments to the sketched curve based on an initial discrete estimation of curvature. The idea is now to extend their approach to identifying line and circular arc segments as well.

The discrete curvature at a point P_i is calculated using the *Frenet-Serret* formula [32]. The circum-circle determined by three sequential points P_{i-1}, P_i and P_{i+1} is used to approximate the discrete curvature κ at the middle point P_i with V_1 and V_2 denoting the vectors $V_1 = P_{i-1}$ - P_i and $V_2 = P_i$ - P_{i+1}, respectively [9]:

$$\kappa(P_i) = 2 \cdot \frac{\sin(\frac{\alpha}{2})}{\sqrt{||V_1|| \cdot ||V_2||}} \tag{3.2}$$

with

$$\alpha = 2 \arccos(\frac{V_1}{||V_1||} \cdot \frac{V_2}{||V_2||}) \ . \tag{3.3}$$

Chapter 4

Design

The Interactive Alignment Design Tool tool developed by Schlenger [28] provides a good framework within which the problem of fitting a curve composed of line, circular arc and clothoid elements into a sketched curve can be addressed. The Interactive Alignment Design Tool was developed within the Collaborative Design Platform (CDP). For this reason, Section 4.1 describes the structure of the CDP. Section 4.2 gives insight into the workings of the Interactive Alignment Design Tool. The implementation of the sketching functionality is broken down into the steps of the process described in Section 4.3.

4.1 Collaborative Design Platform (CDP)

The CDP is a digital design platform developed at TUM as a response to the few software solutions available for the early design stages of architectural projects [33]. Figure 4.1 shows the hardware setup of the CDP. Essentially, it consists of two projection surfaces, the table-top and the projection plane. While the table-top (A) depicts a bird's eye view of the terrain, the projection plane (H) shows a lateral view. The surface of the interactive table is illuminated by infrared sensors (D) and reacts to touch and objects placed on its surface. Touch is recognised by the infrared camera (E) within the table, while placed objects are recorded as a 3D point cloud by the depth camera (I). The infrared camera takes a picture of the underside of the surface as reflected by the mirror (C). The computer (F) subsequently creates projection images for both projectors (B and G). The platform thus closes the gap between analogous, traditional design and digital, interactive simulations: the real-time feedback - visual and computational - on design variants has a positive influence on the decision-making process and thus improves the quality of the final design [34].

The software consists of the middleware (C++), the core structure of the platform, and plugins (C#). The middleware handles input (touch and object recognition), output (visu-

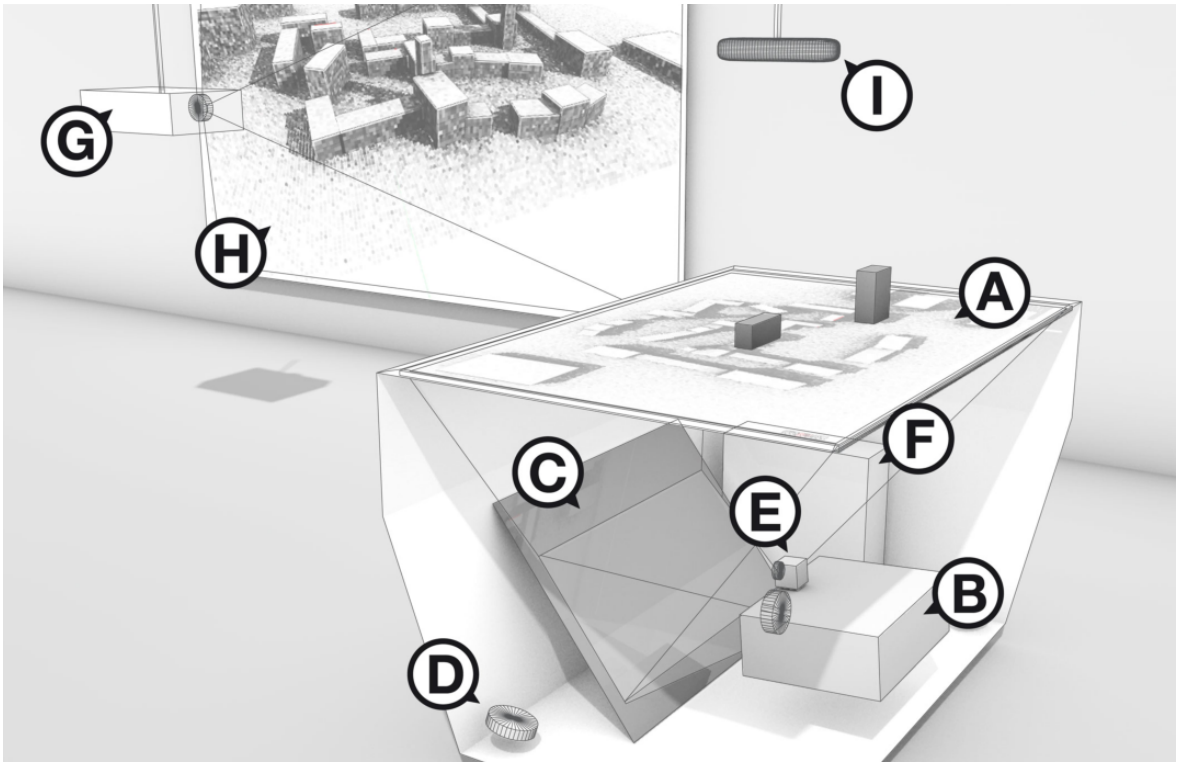

Figure 4.1: The hardware setup of the CDP: the interactive projection table (A), the projector (B), the mirror between A and B (C), the infrared sensors (D), the infrared camera (E), the computing unit (F), the second projector (G), the projection plane (H) and the depth camera (I). Reproduced with permission from Schubert [34].

alisation of results) and data exchange with the plugins. The plugins are linked to the middleware by means of a .dll, which allows them to access the middleware's central user interface library. The plugin architecture allows for flexible integration of different design support tools. Volume calculations, shade and wind analyses, energy simulations and the Interactive Alignment Design Tool are all examples for extensions developed for the "*toolbox system*" that is CDP [34].

4.2 Interactive Alignment Design Tool

The Interactive Alignment Design Tool was developed within the CDP. By accelerating the conceptual design process, the tool facilitates comparing different alignments at the outset of a project.

Figure 4.2 depicts the user interface (UI) of the tool. The tool projects a contour map onto the table-top screen. Points of intersection (red points) for both the horizontal and vertical alignment are marked by touching the screen. These points of intersection are then connected by straight lines. The resulting polygonal chain (dotted line) is rounded out wherever two

straight lines meet. In this manner, the user interactively creates an initial design of the horizontal alignment (black line). The same initial design process can be repeated for the vertical alignment using the calculated height profile of the horizontal alignment. The result, the alignment model consisting of horizontal and vertical alignment, is then exported as an .ifc file.

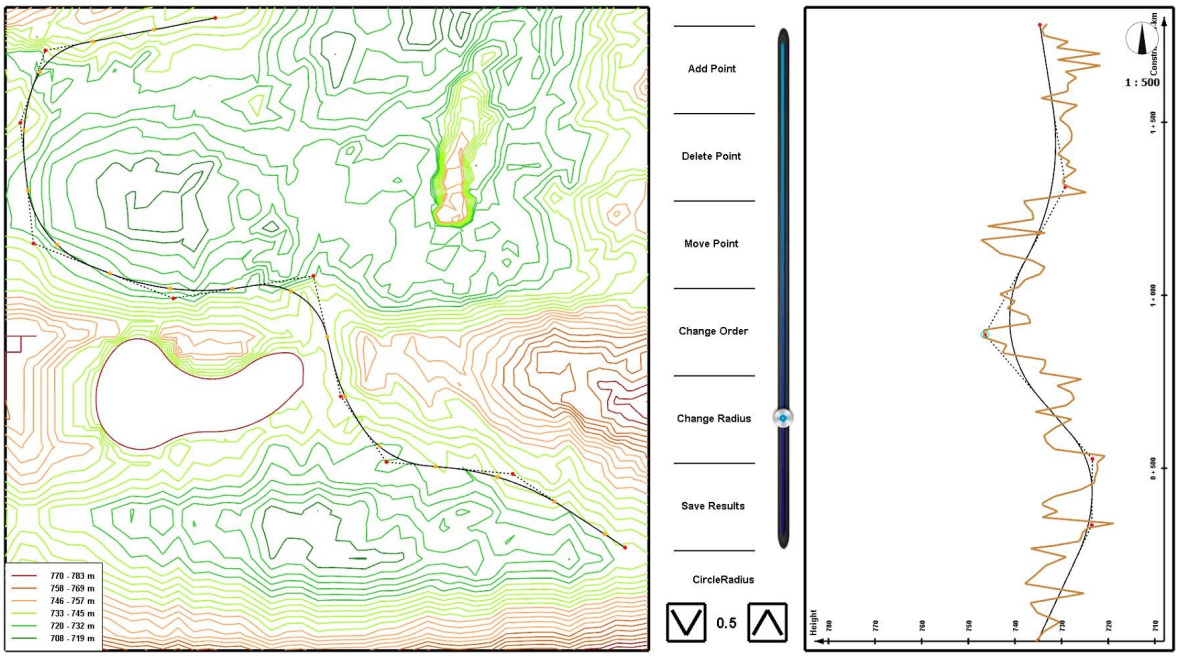

Figure 4.2: User interface of the Interactive Alignment Design Tool within the CDP. Reproduced with permisson from Schlenger [28].

The resulting alignment curve is effectively a sequence of lines and circular arcs, without any transition curves between them. Furthermore, the fillet radius, the "rounding out" of the polygonal chain, is based on a simple approximation: the start and end points for the circular arcs are placed in the middle of each original straight line.

4.3 Proposed process

The proposed software process follows the three steps of an SBIM pipeline:

(1) Sketch acquisition:
The sketching functionality that allows the user to create an alignment sketch is to be implemented in the Interactive Alignment Design Tool of the CDP.

(2) Filtering:
For the sake of reducing user and device noise, points will only be recorded within a threshold distance or time interval of one another, depending on the hardware or middleware.

(3) Interpretation:
The sketch-interpreting algorithm should assign meaning to the alignment sketch by splitting it into a viable sequence of alignment elements. The alignment property that will be used to make this classification is curvature, as described Section 3.2.

The proposed software process behind sketch-based alignment design is shown in Figure 4.3.

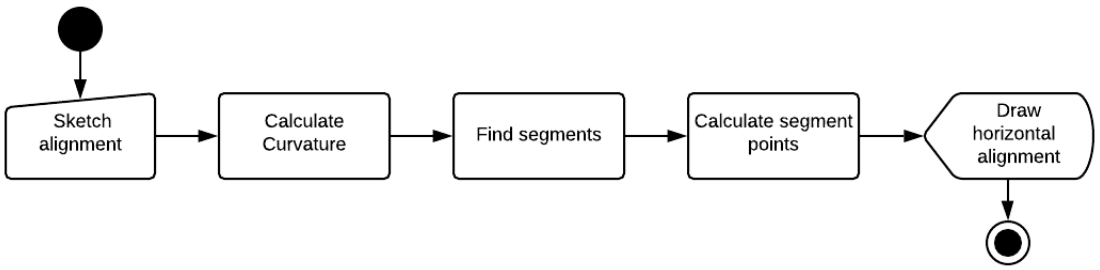

Figure 4.3: Sketch-based alignment design process

Step 1: Sketch alignment
The user can begin to sketch the alignment once the *Sketch* button is selected. If users want to modify the sketch, they have to delete the entire curve (i.e. all recorded points) and start over. Once users select the *ProcessSketch* button, the next step is initiated.

Step 2: Calculate curvature
In a first processing step, the curvature for each point recorded from the alignment sketch is calculated using Eq. (2.2).

Step 3: Find segments
Next, the curvatures calculated in the previous step are used to split the curve into line, circular arc and clothoid segments.

Step 4: Calculate segment points
In this step, sufficient curve points for each curve segment to appear as a continuous line on the screen are calculated. How the points are calculated depends on the segment type (line, circular arc or clothoid) determined in step 3.

Step 5: Draw horizontal alignment
The approximated horizontal alignment is drawn using the curve points calculated in the previous step.

Chapter 5

Implementation

The current alignment design process within the Interactive Alignment Design Tool is depicted in Figure 5.1. The digital terrain model, the contour map, is fed into the tool. The design process of horizontal alignment ("HA") and vertical alignment ("VA") is repeated until the desired result is obtained. The result is the alignment model, which is exported as an .ifc file. This chapter deals with the implementation of the sketching functionality and the sketch-interpreting algorithm within this Interactive Alignment Design Tool. As specified by the requirements in Chapter 3.1, the sketch-based alignment design will be implemented for the "HA design" only. Within this chapter, Section 5.1 describes the class structure that was extended and the methods that were added to carry out the process. Section 5.2 gives more insight into the devised sketch-based alignment design process.

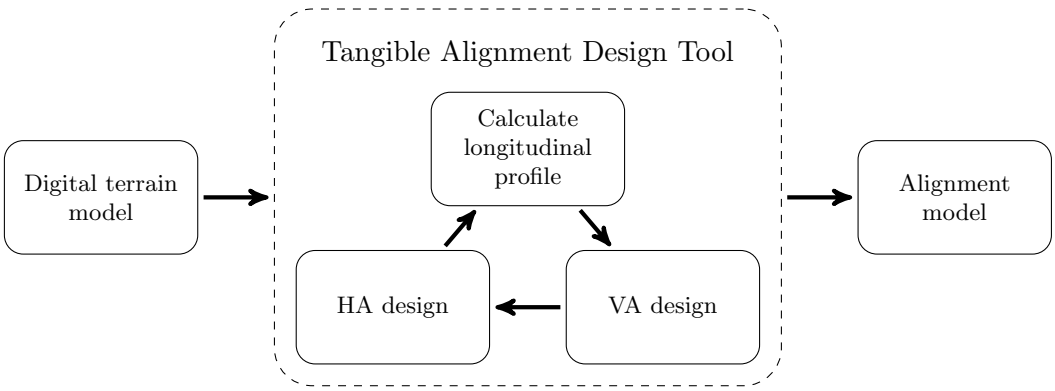

Figure 5.1: Current design process. Reproduced with permission from Markič *et al* [20].

5.1 Class model

The existing class model was extended by *CurvaturePoint* and *TransitionCurveSegment* classes that are necessary for the sketching functionality and the drawing of clothoids.

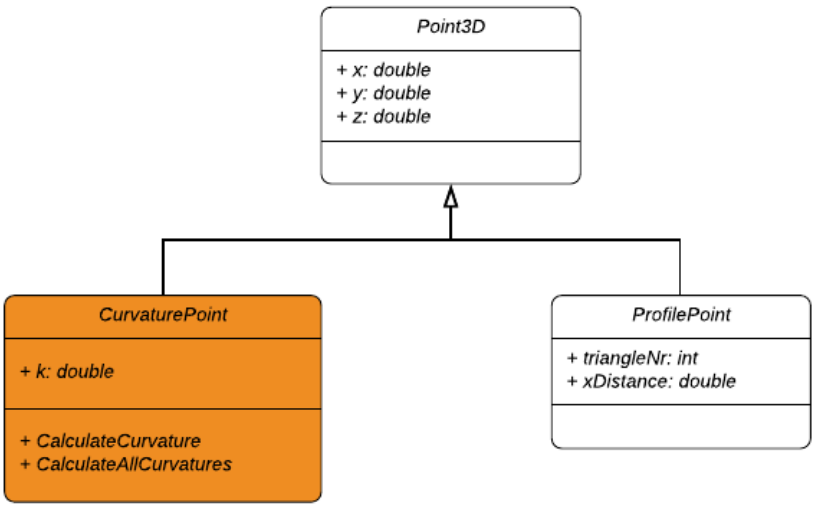

Figure 5.2: Point Classes. New *CurvaturePoint* class marked in orange.

5.1.1 Point classes

Figure 5.2 shows the inheritance structure of the *Point3D* class and its sub-classes *CurvaturePoint* and *ProfilePoint*. In addition to the coordinates of a point inherited from the parent *Point3D* class, the *CurvaturePoint* class includes an attribute specifying its curvature. Furthermore, the *CalculateCurvature* and *CalculateAllCurvatures* methods are located here.

As discussed in Section 3.2, the discrete curvature of a point can be calculated using the circum-circle it spans with its predecessor and successor point. Listing 5.1 shows the implementation of the curvature calculation using Eq. (2.2).

Listing 5.1: CalculateCurvature

```
1  public double CalculateCurvature(Point3D Point, Point3D previousPoint, Point3D nextPoint)
2          {
3              Point3D vec1 = new Point3D();
4              Point3D vec2 = new Point3D();
5              Point3D div1 = new Point3D();
6              Point3D div2 = new Point3D();
7              vec1.X = Point.X - previousPoint.X;
8              vec1.Y = Point.Y - previousPoint.Y;
9              vec2.X = nextPoint.X - Point.X;
10             vec2.Y = nextPoint.Y - Point.Y;
11
12             var dist1 = HelpFunction.GetDistance(Point, previousPoint);
13             var dist2 = HelpFunction.GetDistance(Point, nextPoint);
14             div1.X = vec1.X / dist1;
```

```
15              div1.Y = vec1.Y / dist1;
16              div2.X = vec2.X / dist2;
17              div2.Y = vec2.Y / dist2;
18
19              var angle = Math.Acos((div1.X * div2.X) + (div1.Y * div2.Y));
20              double k = 2 * Math.Sin(angle / 2) / (Math.Sqrt(dist1 * dist2));
21              return k;
22          }
```

CalculateAllCurvatures

The method executes the curvature calculation for every recorded point, with exception of the first and last point as these do not have the predecessor or successor point necessary for the circum-circle determination. *CalculateAllCurvatures* takes the list of points recorded from the alignment sketch and returns *curvaturePoints*, a list of *CurvaturePoint*s that specifies the coordinates as well as the curvature of each recorded point along the alignment sketch. The code for this method is shown in Listing 5.2.

Listing 5.2: CalculateAllCurvatures

```
1  public List<CurvaturePoint> CalculateAllCurvatures(List<Point3D> Points)
2  {
3      List<CurvaturePoint> curvaturePoints = new List<CurvaturePoint>();
4      for (int i = 1; i < (Points.Count)−1; i++)
5              {
6          double curvature = CalculateCurvature(Points.ElementAt(i), Points.ElementAt(i − 1),
               Points.ElementAt(i + 1));
7              CurvaturePoint curvaturePoint = new CurvaturePoint(curvature, Points.ElementAt(i));
8              curvaturePoints.Add(curvaturePoint);
9              }
10         return curvaturePoints;
11 }
```

5.1.2 Segment classes

Figure 5.3 shows the *Segment* class and its sub-classes *LineSegment*, *CircularArcSegment* and *TransitionCurveSegment*. The new transition curve segment class includes all attributes necessary for its description, except the “TransitionCurveType”, as the only transition curve type currently implemented is the clothoid. Still, the IFC structure (see Chapter 2.3) was adhered to in order to facilitate the implementation of other transition curve types in the

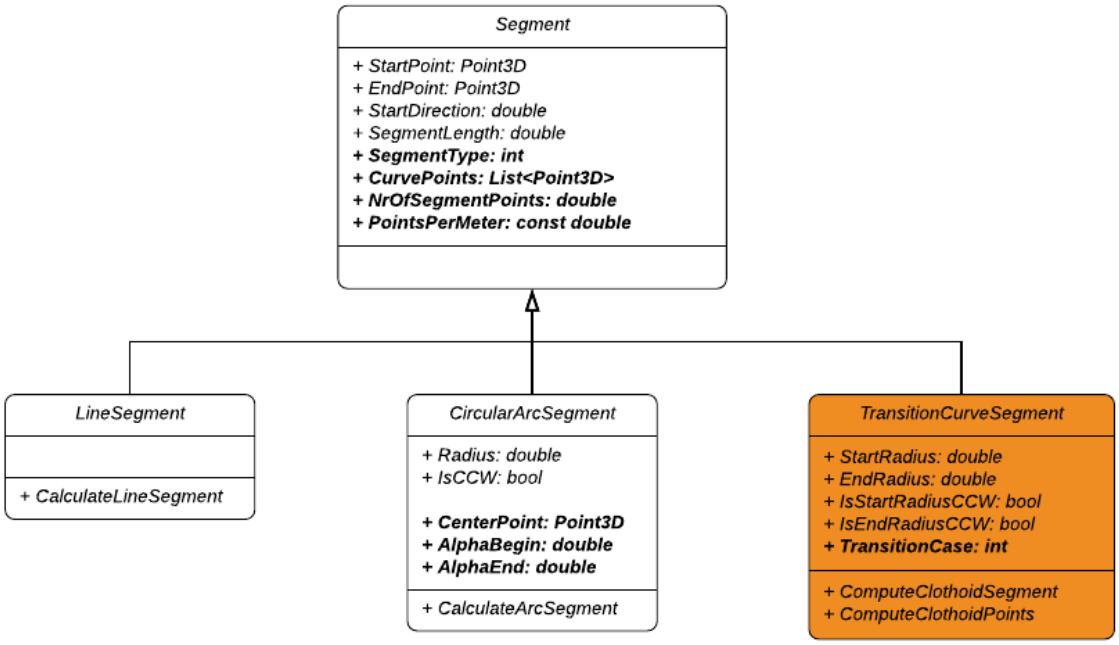

Figure 5.3: Segment Classes. New *TransitionCurveSegment* class shown in orange. Non-IFC attributes displayed in bold font.

future. Furthermore, the class includes the methods responsible for calculating the curve points within a clothoid segment, *ComputeClothoidSegment* and *ComputeClothoidPoints*.

ComputeClothoidSegment

ComputeClothoidSegment is called every time a new *TransitionCurveSegment* is created. The method is based on the numerical computation for clothoids by Vázquez Méndez and Casal Urcera [31]. Their approach differentiates the clothoid calculation for the different transition scenarios possible, namely *Line - Arc* and *Arc - Line*. The differentiation is necessary, because for each transition, the parameters for the calculation of the clothoid points have to be set accordingly. These parameters are:

- "radius": the radius of the circular arc
- "lambda": the orientation of the circular arc
- "alphaStart": the angle at the tangent in the start point of the clothoid
- "alphaEnd": the angle at the tangent in the end point of the clothoid
- "theta": the change in direction between start and end point of the clothoid. This is given as the difference between the angles "alphaStart" and "alphaEnd".

The code for this method is shown in Listing A.2.

ComputeClothoidPoints

Once the parameters "radius", 'theta", "lambda" (λ) and "alphaStart" (ϕ_0) are determined, *ComputeClothoidPoints* is called. First, the segment length s^n, the calculation step Δs and the clothoid parameter A are calculated:

$$s^n = 2 \cdot radius \cdot theta \tag{5.1}$$

$$\Delta s = \frac{s^n}{NrOfSegmentPoints} \tag{5.2}$$

$$A = \sqrt{radius \cdot s^n} \tag{5.3}$$

The method then calculates as many points between the start point and end point of the segment as specified by "NrOfSegmentPoints":

$$x^{n+1} = x^n + \Delta s \cos(\lambda \frac{(s^n)^2}{2A^2} + \phi_0) \tag{5.4}$$

$$y^{n+1} = y^n + \Delta s \sin(\lambda \frac{(s^n)^2}{2A^2} + \phi_0) \tag{5.5}$$

The code for the method is shown in Listing A.3.

5.2 Sketch-Based Alignment Design Process

Figure 5.4 shows the process proposed in Section 4.3 in more detail.

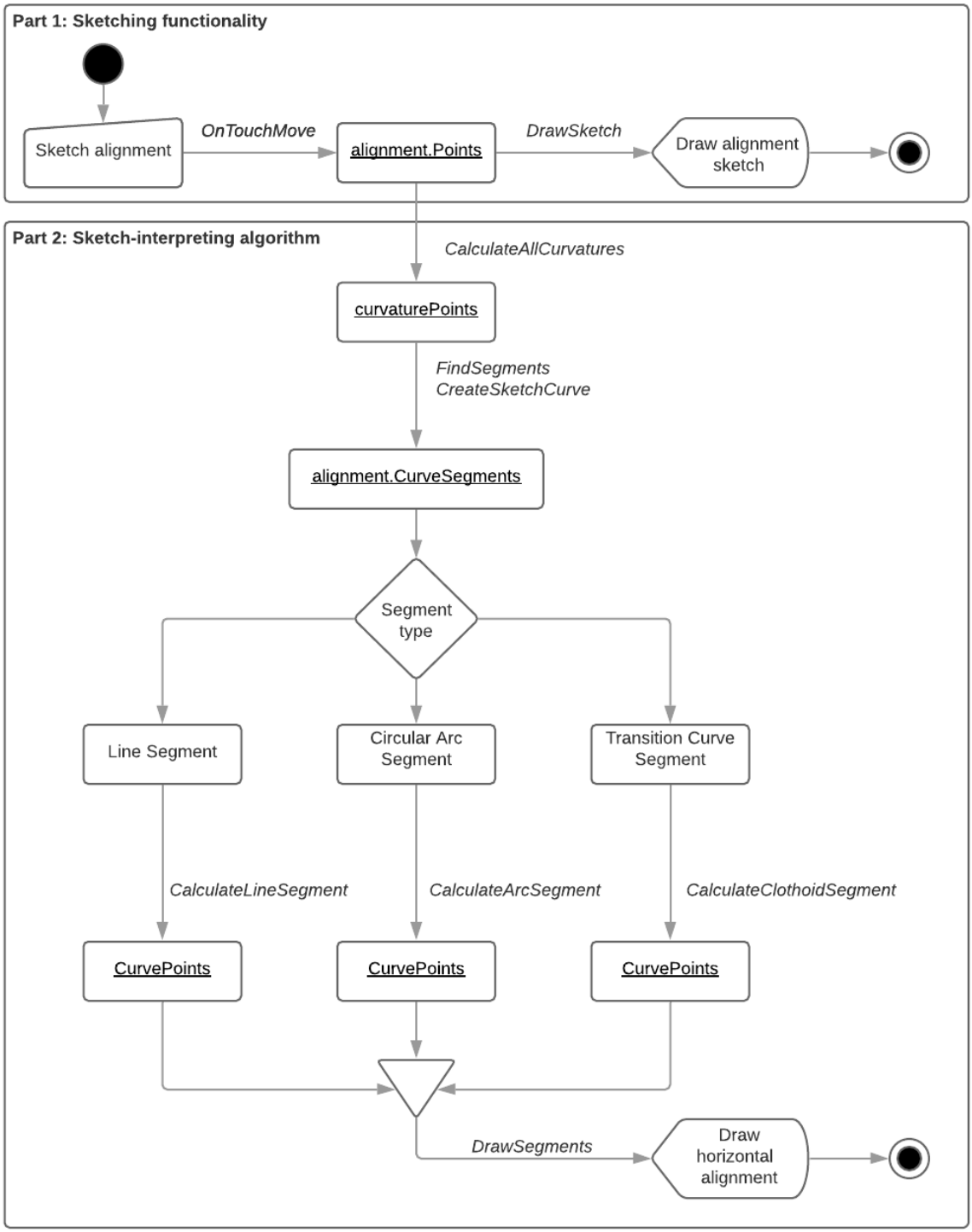

Figure 5.4: Sketch-based alignment design process

The following section carefully describes each step of the sketch-based alignment design process. Part 1 (Section 5.2.1) covers the sketching functionality as such while Part 2 deals with the sketch-interpreting algorithm (Section 5.2.2).

5.2.1 Part 1: Sketching functionality

The sketching functionality takes the user sketch and displays the result on the table-top screen. *OnTouchMove* is responsible for recording the alignment sketch points from the user-screen interaction. *DrawSketch* displays the recorded alignment sketch on the screen as a series of points. An exemplary sketch, the result of the sketching functionality, is shown in Figure 5.5. The corresponding height profile is shown in Figure 5.6.

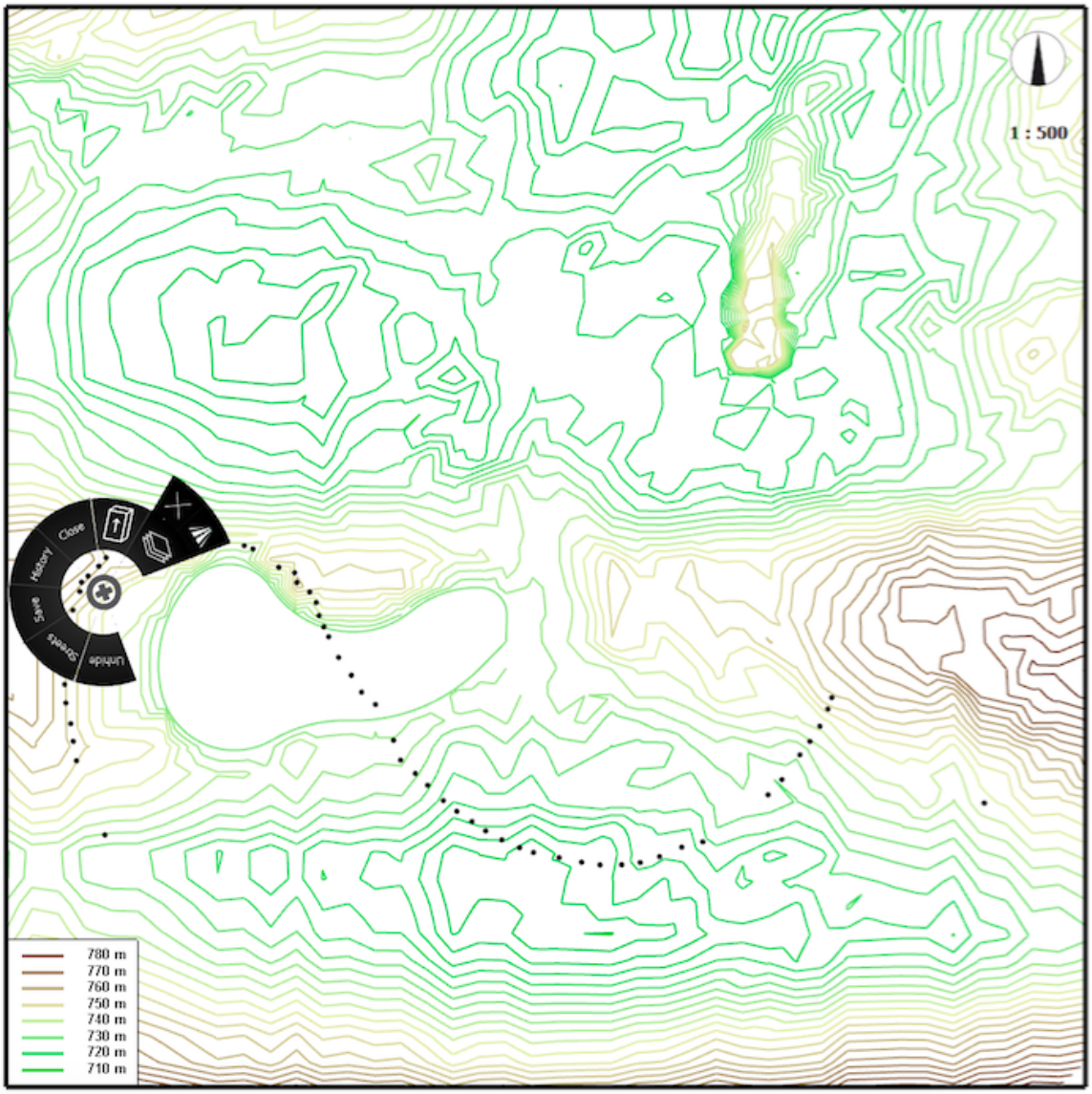

Figure 5.5: Exemplary alignment sketch.

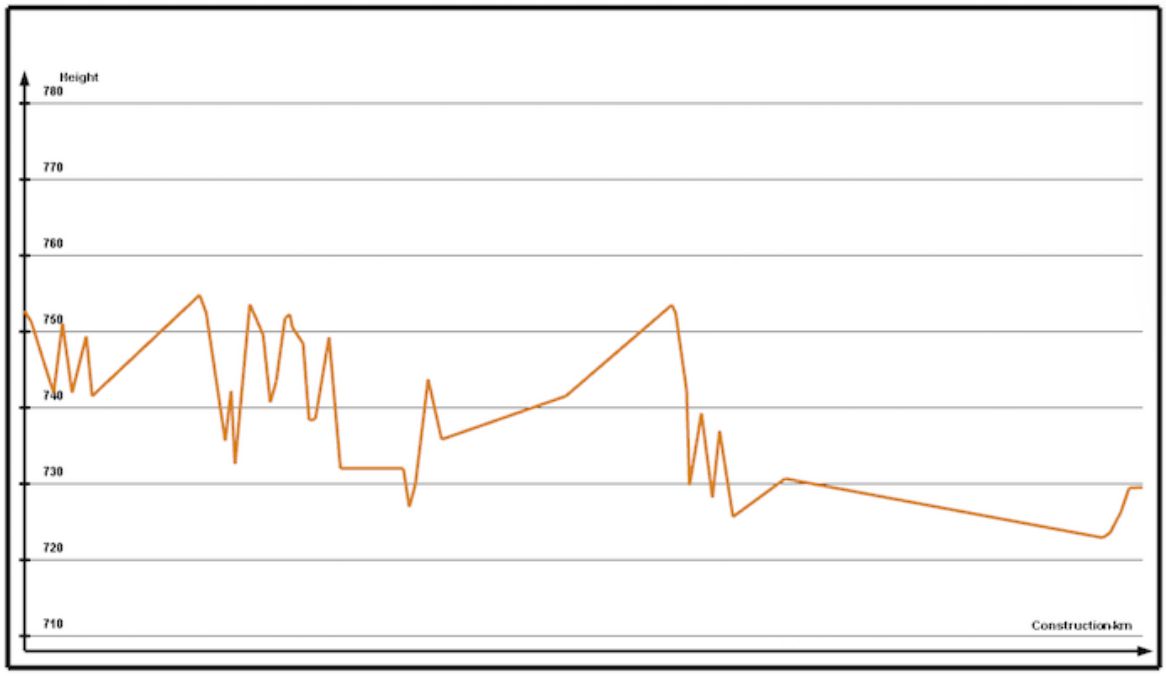

Figure 5.6: Calculated height profile for the exemplary alignment sketch.

Sketch alignment

Once the user selects the *Sketch* button, the sketch mode is started. While the user's finger touches the contour map and sketches, a list of time points is generated. A "TimePointF" consists of the *2D* coordinate of the point as well as a time stamp stating how much time has passed since the beginning of the sketch. Algorithm 1 shows the *OnTouchMove* method responsible for this step. It is part of the general user interface library of the CDP and was overridden for this tool. While the user is sketching, the x- and y- coordinate of points touched on the screen are recorded every five milliseconds. These points are stored in a list called *alignment.Points*. The time interval definition allows for some control over the amount of points generated. While too many samples would lead to poor performance, discarding too many would lead to poor reproduction of the original user input [11].

Draw alignment sketch

The alignment sketch created by the user is drawn using the *OpenGL* library as shown in Listing 5.3. The condition stated in line 9 ensures that the *DrawSketch* only tries to draw points if any where recorded. Each of these points is drawn as a black point. As can be seen in Figure 5.5, there can be large gaps along the sketch. This is due to the fact that the drawing speed that is not consistent throughout the sketch move.

Algorithm 1: OnTouchMove

Require:
C cursor points
t time stamp of the cursor point
Ensure:
P recorded points
procedure OnTouchMove(C)
  for all $c_i \in C$ **do**
    if $(t(c_0) - t(c_i)) > 5$ **then** ▷ time interval greater than 5ms
      $p_j \leftarrow c_i$
    end if
    $t(c_0) \leftarrow t(c_i)$
  end for
end procedure ▷ Refer to Listing A.1 for C# code.

While testing the functionality, changes in drawing speed were noticed on two occasions specifically. Firstly, one tends to draw slower when changing the direction of the alignment sketch. Secondly, when drawing longer, straight stretches, the drawing speed increases. Furthermore, *OnTouchMove* is very sensitive to touch. Changes in direction making the finger sketching tilt a little more might be interpreted as the finger lifting, even though that is not really the case. This interrupts the function call and the recording of points only resumes once the finger is flat on the screen again.

Listing 5.3: DrawSketch

```
1  public void DrawSketch(Alignment alignment)
2          {
3              GL.Color3(Color.Black);
4              GL.PointSize(6);
5
6              GL.Begin(PrimitiveType.Points);
7              for (int i = 0; i < alignment.Points.Count; i++)
8              {
9                  if (alignment.Points.ElementAt(i).X != 0)
10                 {
11                     GL.Vertex3(alignment.Points.ElementAt(i).X,
                           alignment.Points.ElementAt(i).Y, 0);
12                 }
13             }
14             GL.End();
15         }
```

5.2.2 Part 2: Sketch-interpreting algorithm

Once Part 1 of the Sketch-Based Alignment Design Process is completed, Part 2, the processing of the recorded sketch points is initiated. The goal of this part of the process is to interpret the alignment sketch in such a manner that it can be drawn as a horizontal alignment composed of a sequence of lines, circular arcs and clothoids and converted to an alignment model.

Calculate curvature

In this step, the curvature of each recorded point is calculated using the *CalculateAllCurvatures* method. The result is a list of curvature points, *curvaturePoints*, that contains the coordinates and curvature of every point along the alignment sketch (refer to 5.1.1). Figure 5.7 shows the result of the curvature calculation for all sketch points of an exemplary alignment sketch. The curvature κ is plotted against the distance s along the alignment sketch.

Based on the approach by McCrae and Singh [9, 29] it was expected that there would be distinct sections within the curvature plot that could be clearly identified as parts of lines, circular arcs or clothoids. Sections where the curvature fluctuates around zero ($\kappa \approx 0$) could then be classified as line segments; sections where the curvature fluctuates around some other constant value $\kappa = const$ could be marked as circular arc segments; and the sections between line and circular arc segments as clothoids. However, the curvature plot is spikier than expected. Therefore the sketch-interpreting algorithm should focus on finding characteristic points instead that give a clear indication of the start or end of a specific segment based on the curvature of that point.

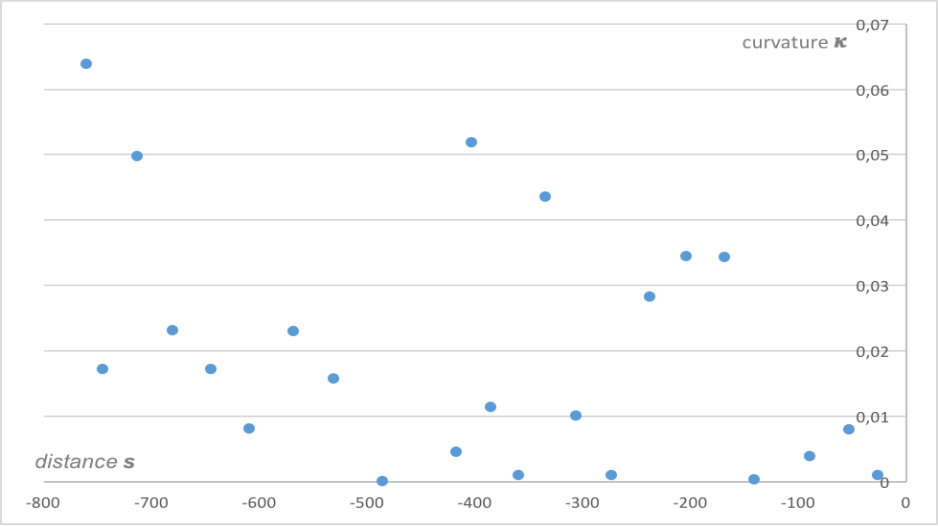

Figure 5.7: Curvature plot for the exemplary alignment sketch.

Find segments

FindSegments is the key method of the sketch-interpreting algorithm. It is responsible for splitting the alignment sketch into a sequence of segments by locating their start and end points. In a nutshell, the method goes over every point in *curvaturePoints* P and checks if it meets the condition for the start or end of an arc or line. Using the curvature κ calculated for every point $p_i \in P$ in the previous step, the determined conditions are as follows:

(1) $|\kappa_i| \geq \epsilon_{arcStart}$
The curvature of the point P_i is greater than or equal to the tolerance for the start of an arc. This means that the start of a circular arc is found.

(2) $|\kappa_i| \leq \epsilon_{end}$
Once the curvature of the point P_i falls below the defined tolerance, the end of the arc segment is reached.

(3) $|\kappa_i| \leq \epsilon_{lineStart}$
The curvature of the point P_i is smaller than or equal to tolerance for the start of a line. This means that the curvature is close enough to zero to be regarded as the curvature of a line.

(4) $|\kappa_i| \geq \epsilon_{end}$
Once the curvature of the point P_i exceeds the defined tolerance, the end of the line segment is reached.

The actual values of the tolerances were chosen after evaluating the curvature plots of several test sketches. They are:
- $\epsilon_{arcStart} = 0.0400$
- $\epsilon_{lineStart} = 0.0025$
- $\epsilon_{end} = 0.0050$

Figure 5.8 shows the segment start and end points found for an exemplary sketch using *FindSegments*.

The method, shown in Algorithm 2, starts by evaluating the curvature of the first point. If the value is smaller than the tolerance for the start of an arc, $\epsilon_{startArc}$, the first segment is a line, otherwise it is a circular arc. As mentioned in the requirements in Section 3.1, within this implementation, a circular arc may be followed by a line or another circular arc; a line may only be succeeded by a circular arc. Maintaining this sequence is guaranteed by setting the variable $IsLine$ If the start point of a line segment is found, $IsLine$ is set to true. Else - if the start point of a circular arc segment is found - $IsLine$ is set to false.

Depending on the value of $IsLine$, the method jumps to the respective section within the method and searches for the end point of the segment. Every time the end of a segment is

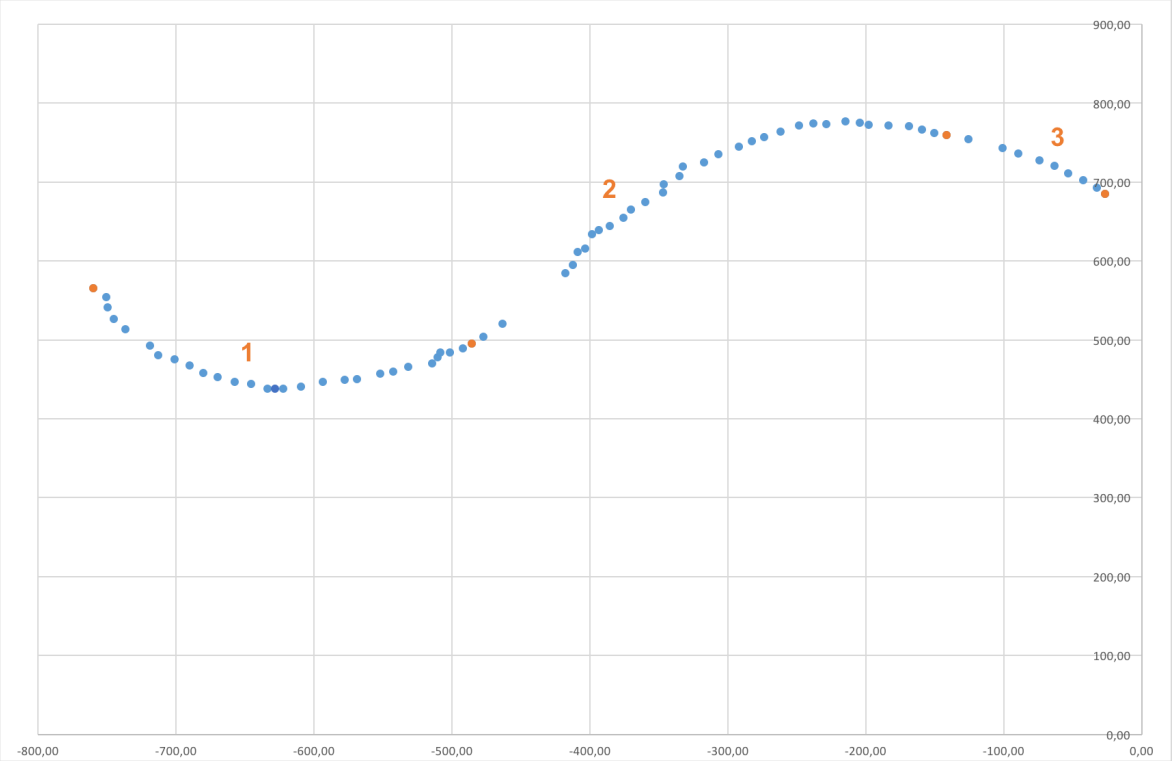

Figure 5.8: Segments start and end points (orange) found among the alignment sketch points (blue): 1 - circular arc, 2 - transition curve, 3 - line.

found, a new segment is created. It is saved to the list of segments called *Segments*. Before the search for the next segment start point is initiated, a check to see if the end of the sketch is near is performed. If so, the last segment is appended and the method finishes.

In all this, the variable *minCount* guarantees a minimum segment length in points. Every time the start or end point of a new segment is found, the iterator is updated by *minCount*. For example, if *minCount* is set to 3, it thus guarantees that the segment is at least 4 points long. As a bonus, skipping as many points as specified by *minCount* speeds up the evaluation process.

Algorithm 2: Find Segments

Require:
$p_i \in P$ CurvaturePoints
n number of CurvaturePoints
$minCount$ is the minimum length of a segment in points
Ensure:
$s_j \in S$ CurveSegments

procedure FINDSEGMENTS(P)
  if $|\kappa(p_0)| \leq \epsilon_{arcStart}$ **then** ▷ first segment
    $s_0 \leftarrow p_0$ ▷ start line
    $IsLine \leftarrow true$
  else
    $s_0 \leftarrow p_0$ ▷ start arc
    $IsLine \leftarrow false$
  end if
  $i = minCount$
  for $i < n$ **do**
    if $IsLine$ **then**
      if $|\kappa(p_i)| \leq \epsilon_{end}$ **then**
        $s_j \leftarrow p_i$ ▷ end line
        $i \leftarrow i + minCount$
        if $n - i \leq minCount$ **then** ▷ near end check
          $s_j \leftarrow p_n$ ▷ end last line
        else if $|\kappa(P_i)| \geq \epsilon_{arcStart}$ **then**
          $s_j \leftarrow p_i$ ▷ start arc
          $IsLine \leftarrow false$
          $i \leftarrow i + minCount$
        end if
      end if
    else
      if $|\kappa(p_i)| \leq \epsilon_{end}$ **then**
        $s_j \leftarrow p_i$ ▷ end arc
        $i \leftarrow i + minCount$
        if $n - i \leq minCount$ **then** ▷ near end check
          $s_j \leftarrow p_n$ ▷ end last arc
        else if $|\kappa(p_i)| \leq \epsilon_{lineStart}$ **then**
          $s_j \leftarrow p_i$ ▷ start line
          $IsLine \leftarrow true$
          $i \leftarrow i + minCount$
        end if
      end if
    end if
  end for
  return S
end procedure ▷ Refer to Listing A.4 for C# code.

CreateSketchCurve

Where *FindSegments* identifies the start and end points of possible line and circular arc segments, *CreateSketchCurve* fills the gaps in between with transition curve segments and calls the respective constructors with the necessary parameters. Each constructor then calls the corresponding method for the segment point calculation (*CalculateLineSegment*, *CalculateArcSegment* or *CalculateClothoidSegment*). The created segments are stored in a list of segments belonging to the alignment called *CurveSegments*. The C# code for this method is shown in Listing A.5.

Calculate segment points

In this step, sufficient curve points for each segment of the horizontal alignment to appear as a continuous line on the screen are calculated. The segment points for each segment are calculated based on the segment type (line, circular arc or clothoid) determined in the previous step.

Every time a new segment is created within *FindSegments*, the respective constructor calls the method calculating the segment points. For a *TransitionCurveSegment* the constructor is shown in Listing 5.4. The constructors for a *LineSegment* and *CircularArcSegment* are similar, but include other parameters, depending on what is necessary for the curve point calculation. In the case of the *TransitionCurveSegment*, the start and end points found, the predecessor and successor segment and the transition case ("ArcLine" or "LineArc") are needed. Within the constructor, the *ComputeClothoidSegment* is called to calculate the points along the clothoid. The detail of this method was explained in Section 5.1.2.

Listing 5.4: TransitionCurveSegment constructor

```
1 public TransitionCurveSegment(Point3D start, Point3D end, LineSegment line,
      CircularArcSegment arc, string transitionCase)
2         {
3             this.StartPoint = start;
4             this.EndPoint = end;
5             this.TransitionCase = transitionCase;
6             this.CurvePoints = new List<Point3D>();
7
8             ComputeClothoidSegment(line, arc);
9         }
```

Draw horizontal alignment

Drawing the horizontal alignment requires the segment points calculated in the previous step. In order to obtain a horizontal alignment with smooth curvature, the segment end points have to match. This is where the sketch-interpreting algorithm fails. Although it succeeds in splitting the alignment sketch into a viable sequence of alignment elements, it is impossible to obtain composite curve with smooth curvature from only a set of start and end points of the found elements. Under these conditions, the curvature in the points where two successive segments meet are not identical. This means that the resulting curve would be close to the original sketch ("fidelity"), but would not have the required C^2 continuity ("fairness").

Chapter 6

Evaluation

The effectiveness of the implementation of the sketch-based alignment design tool is measured by the requirements defined in Chapter 3.1. The following evaluates these requirements together with the sketching functionality (Section 6.1) and the sketch-interpreting algorithm (Section 6.2).

6.1 Evaluation of the sketching functionality

The use of the tool should be intuitive and easy. To this end, the *Sketch* button should clarify that once this button is selected, the user enters the *Sketch* mode. As when sketching with pen and paper, the alignment sketch comprises all points from the moment the finger touches the screen until it is lifted again. Once the user moves the slider to *ProcessSketch*, the processing of the alignment sketch is initiated.

There is nothing in the user interface to indicate that it is not allowed for a sketched line to cross itself. Meeting this requirement depends on the common sense of the user drawing the alignment.

The new sketching functionality is integrated into the existing process and structure as far as possible. Figure 6.1 demonstrates that even though the user input comes from a different source (alignment sketch versus points of intersection), it is still fed into the longitudinal profile calculation correctly.

6.2 Evaluation of the sketch-interpreting algorithm

The purpose of the sketch-interpreting alignment is to split the alignment curve into a viable sequence of alignment elements. Whenever a line segment is found, the method searches for

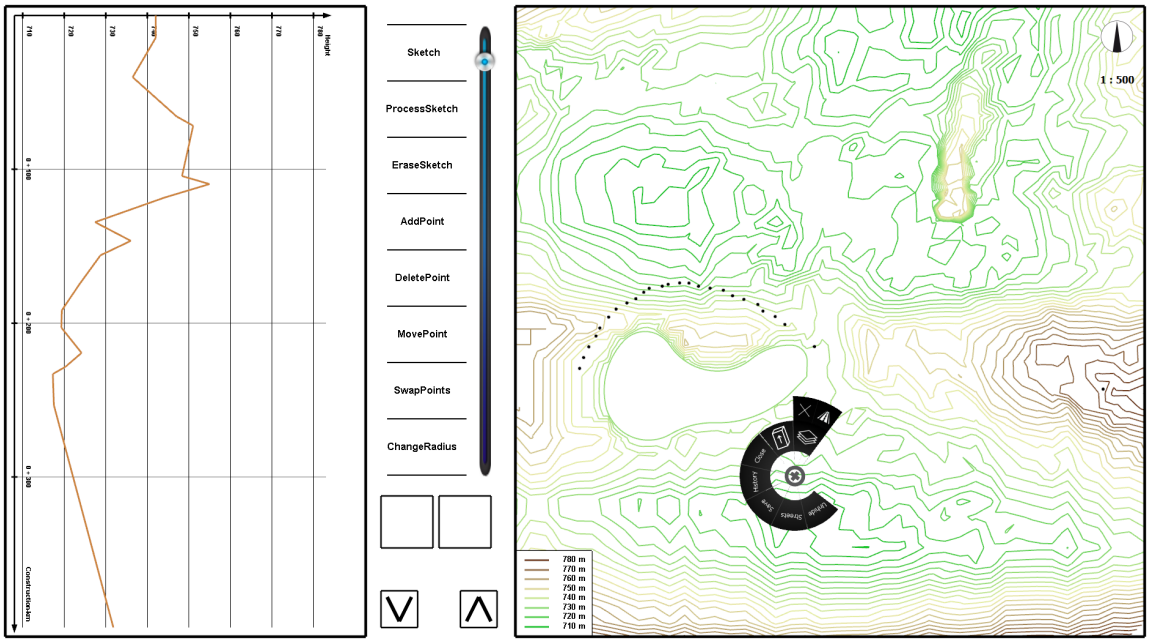

Figure 6.1: Horizontal alignment sketch (right) and resulting longitudinal profile (left)

the next circular arc segment; and vice versa. Although the sketch-interpreting algorithm identifies possible segments, more work has to be done to draw the horizontal alignment as a composite curve with smooth curvature (C^2 continuity) from the alignment sketch.

As stated in Chapter 3.2, the sketch-interpreting algorithm is a compromise between the fidelity to the original sketch and the fairness of the approximated alignment. The former demand is met by the segment start and end points that are retrieved from the user sketch. The latter demand requires further attention as it is not possible to create a composite alignment curve with smooth curvature based solely on the found segment start and end points. An additional algorithm to *merge* the found segments to form a composite alignment curve could be the answer. This algorithm should ensure smooth curvature (C^2 continuity) within the alignment curve, by calculating the segment points for a segment and then updating the start point of the next. Only then would it be possible to draw the horizontal alignment with lines, circular arcs and clothoids. Also, the fidelity to the original sketch would have to be re-examined.

Chapter 7

Conclusion

7.1 Summary of findings

Sketch-based alignment design comes with one great advantage compared to other sketch-based applications: the sketch is comparatively simple. As discussed in Section 2.1, an alignment is split into the horizontal alignment and the vertical alignment, essentially creating two 2D curves. This means that the perception problem inherent to modeling 3D shapes is irrelevant in sketch-based alignment design specifically.

The sketch-interpreting algorithm is heavily reliant on the quality of the user sketch. The amount and spacing of points generated depends on the drawing speed. Moreover, a change in direction making the finger sketching tilt a little more might be interpreted as the finger lifting, although that is not really the case. This interruption leads to a gap in the point recording which reduces the quality of the alignment sketch.

To a certain degree, if the user draws nonsense, the sketch-interpreting algorithm will also produce a nonsensical result. The idea is to have a sketch-based alignment design tool that makes it possible to communicate and try different ideas, even by users who don't necessarily have any understanding of alignment design. In this case, it is better to have an approximated alignment that is coherent than one that is very close to the user sketch. In other words, in the "*fidelity-fairness-tradeoff*" [11], in this user scenario at least, fairness wins.

The curve property exploited throughout this implementation is curvature. All alignment elements - lines, circular arcs and transition curves - have a characteristic curvature profile. This curvature property is used within the sketch-interpreting algorithm, *FindSegments*, splits the alignment sketch into a viable sequence of lines, circular arcs, and clothoids.

If one considers the problem as a split-and-merge procedure, the sketch-interpreting algorithm covers the first part of splitting the sketch into a viable sequence of alignment elements. The

next step would now be to merge the found segments to form a composite horizontal alignment with smooth curvature, ensuring C^2 continuity. The start and end points of segments found could be used to calculate an initial segment length for each segment. Then, starting from the first point, the segment points could be calculated correctly according to segment type. For each subsequent segment, the start point could be updated to the previous segment's calculated end point. Then, as before, points for the next segment would be calculated until the segment length is reached.

7.2 Future work

The sketch-interpreting algorithm splits the alignment sketch into a viable sequence of lines, circular arcs, and clothoids. The next step would be to *merge* the found segments to form a composite horizontal alignment. Once a reliable sketch-based reconstruction for the horizontal alignment is found, the approach could be transferred to vertical alignment design.

Although this approach paves the way for a good approximation of the alignment sketch, it does not allow the alignment to hit specific points along the alignment itself.
For example, if it were necessary for the alignment to pass through a river or existing roads and railways at a specific location, the algorithm would quite simply ignore these points and treat them the same as all other points in the alignment sketch, with exception of the first and last point. However, as it is often necessary to meet specific points along the alignment in practice (see Section 2.1), it would be worth it to pursue further deliberations in this regard.

Within *FindSegments*, every time the start or end point of a new segment is found, the respective iterator is updated by "minCount". This guarantees a minimum segment length and creates space for a clothoid segment between a line and circular arc segment. Currently, the value of "minCount" is set to four, meaning that a segment is always at least five points long. This minimum segment length could be adapted to different minimum segment lengths specified by regulation [14, 13] and based on a distance in pixels, depending on the scale of the map.

The maximum gradient and radii allowed for certain road classes provide further parameters that could be included in the Interactive Alignment Design Tool. For instance, the road class could be specified before beginning the sketch. If the gradient or radius allowed is exceeded, the user is given a warning and asked to modify the sketch accordingly. However, one could also argue that that would exceed the purpose of this tool - early-stage alignment design - and that this should be left to the detail design phase in CAD software. This addition is also recommended by Schlenger [28].

The tolerances used within the *FindSegments* method could also be adapted to specific design scenarios.

For example, in railway design, linear alignments are preferred over winding ones. To incorporate this within the algorithm, the tolerance for a line could be set higher, which means that lines would be found more often.

The clothoid is the only transition curve implemented within the scope of this thesis. One could, however, consider implementing multiple other transition curve types apart from the clothoid. For example, biquadratic parabola, bloss curve, cosine curve, cubic parabola and sine curve are also specified by the IFC 4x1 standard [16]. The property exploited throughout the implementation is the characetristic curvature of specific alignment elements. Thus, as long as a transition curve type has a characteristic curvature profile that it can be identified by, it should be possible to implement these as well.

Furthermore, it is worthwhile spending more time on the development of Sketch-Based Alignment Design, as it could be adapted to any other domain that requires the representation of an alignment, e.g. railways, nature trails, pipes or power lines or waterways [29].

Another functionality that could be implemented in a SBIM tools is a curve augmentation method such as *oversketching* [2]. Instead of erasing the sketched curve in its entirety and starting over, *oversketching* allows the user to improve parts of the curve by sketching a new line on top of the existing one. The computer then tries to fit the new curve piece by linking it to the parts not changed to its left and right. A different version of curve augmentation could be allowing the curve to be modified by dragging. Yet another idea would be to include an option to *re-import* the alignment curve from CAD software back to the sketching interface for further manipulation.

The decision to implement the sketch-based alignment design tool within the CDP was made on the grounds that there was an existing framework within to do so. Future work could include transferring the sketch-based alignment design concept to other user interfaces that are already available in engineering offices, such as tablets or possibly even virtual reality headsets for an enhanced 3D-feeling. This hardware would still enable collaborative and intuitive design, but is often already available in engineering offices and thus more commercially viable, which would mean that it could be used by many.

Appendix A

Code

This appendix lists important parts of the code refered to within the thesis. The complete code developed for this thesis can be found in the CDP project of the TUM Department of Architecture on Gitlab: https://gitlab.lrz.de/cdp/cdp/tree/feature/plugin_interactiveAlignmentDesign/Plugins/Plugin_InteractiveAlignmentDesign.

A.1 OnTouchMove

OnTouchMove is the method responsible for recording the alignment sketch as a list of points. For its description, refer to section 5.2.1.

Listing A.1: onTouchMove

```
public override void onTouchMove(IPluginContext context)
        {
            base.onTouchMove(context);
            {
                lock (context)
                {
                    foreach (Cursor cursor in context.cursors)
                    {
                        TimePointF point = cursor.points.First();
                        alignment.AddPoint(point.X, point.Y);

                        if (emode == ControlWindow.eMode.Sketch)
                        {
                            foreach (TimePointF cursorPoint in cursor.points)
                            {
                                if ((point.Time - cursorPoint.Time) > 5)
                                {
                                    alignment.AddPoint(cursorPoint.X, cursorPoint.Y);
                                }
                                point = cursorPoint;
                            }
                        }
                    }
                }
            }
        }
```

A.2 ComputeClothoidSegment

The *ComputeClothoidSegment* method assigns the parameters for the clothoid points calculation based on the transition case (*Arc-Line* or *Line-Arc*) and calls *ComputeClothoidPoints*.

Listing A.2: ComputeClothoidSegment

```
public void ComputeClothoidSegment(LineSegment line, CircularArcSegment arc)
        {
            // (1)
            // Add start point of clothoid segment.
            CurvePoints.Add(StartPoint);

            // (2)
            // Compute clothoid points depending on transition case.

            if (TransitionCase == "LineArc") // Line to arc
            {
                // Radii.
                this.StartRadius = Double.PositiveInfinity;
                this.EndRadius = helpFunction.GetDistance(EndPoint, arc.CenterPoint);

                // Arc orientation.
                int lambda;
                if (arc.isCCW == true)
                {
                    this.IsStartRadiusCcw = true;
                    this.IsEndRadiusCcw = true;
                    lambda = 1;
                }
                else
                {
                    this.IsStartRadiusCcw = false;
                    this.IsEndRadiusCcw = false;
                    lambda = -1;
                }

                // Angles.
                double alphaArc = helpFunction.GetCuttingAngle(arc.CenterPoint, EndPoint,
                    lambda);
                double alphaLine = helpFunction.GetCuttingAngle(line.StartPoint, StartPoint,
                    arc.CenterPoint, EndPoint);
                double theta;
                double angleTest = lambda * (alphaArc - alphaLine);

```

```
37                  if (angleTest >= 0)
38                  {
39                      theta = angleTest;
40                  }
41                  else
42                  {
43                      theta = 2 + Math.PI + angleTest;
44                  }
45
46                  // Function calculating segment points called.
47                  ComputeClothoidPoints(EndRadius, alphaLine, theta, lambda);
48              }
49              else      // Arc to line
50              {
51                  // Radii.
52                  this.EndRadius = Double.PositiveInfinity;
53                  this.StartRadius = helpFunction.GetDistance(StartPoint, arc.CenterPoint);
54
55                  // Arc orientation.
56                  int lambda;
57                  if (arc.isCCW == true)
58                  {
59                      this.IsStartRadiusCcw = true;
60                      this.IsEndRadiusCcw = true;
61                      lambda = 1;
62                  }
63                  else
64                  {
65                      this.IsStartRadiusCcw = false;
66                      this.IsEndRadiusCcw = false;
67                      lambda = -1;
68                  }
69
70                  // Angles.
71                  double alphaArc = helpFunction.GetCuttingAngle(arc.CenterPoint, StartPoint,
                        lambda);
72                  double alphaLine = helpFunction.GetCuttingAngle(arc.CenterPoint, StartPoint,
                        EndPoint, line.EndPoint);
73                  double theta;
74                  double angleTest = lambda * (alphaArc - alphaLine);
75
76                  if (angleTest >= 0)
77                  {
78                      theta = angleTest;
79                  }
```

```
80                  else
81                  {
82                      theta = 2 + Math.PI + angleTest;
83                  }
84
85                  // Function calculating segment points called.
86                  ComputeClothoidPoints(StartRadius, alphaLine, theta, lambda);
87
88              }
89              // (3)
90              // Add end point of clothoid segment.
91              CurvePoints.Add(EndPoint);
92          }
```

A.3 ComputeClothoidPoints

The *ComputeClothoidPoints* method calculates the curve points for a clothoid segment.

Listing A.3: ComputeClothoidPoints

```
public void ComputeClothoidPoints(double radius, double angle, double theta, double lambda)
        {
            // Clothoid length and parameter.
            double totalLength = 2 * radius * theta;
            double A = Math.Sqrt(radius * totalLength);

            // Auxiliary variables for segment point calculation.
            base.NrOfSegmentPoints = (int)helpFunction.GetDistance(StartPoint, EndPoint) *
                PointsPerMeter;
            double step = totalLength / NrOfSegmentPoints;
            double parS = Math.Pow(totalLength, 2);
            double parA = 2 * Math.Pow(A, 2);
            double term = lambda * (parS / parA) + angle;

            Point3D previousPoint = new Point3D(StartPoint.X, StartPoint.Y);
            double dist = 0;

            for (int i = 0; i <= NrOfSegmentPoints; i++)
            {
                // Add next point.
                Point3D nextPoint = new Point3D
                {
                    X = previousPoint.X - step * Math.Cos(term),
                    Y = previousPoint.Y - step * Math.Sin(term)
                };
                CurvePoints.Add(nextPoint);

                // Update segment length.
                dist = dist + helpFunction.GetDistance(previousPoint, nextPoint);

                // Update previousPoint for next iteration step.
                previousPoint.X = nextPoint.X;
                previousPoint.Y = nextPoint.Y;
            }
            // Keep segment length.
            this.SegmentLength = dist;
        }
    }
```

A.4 FindSegments

FindSegments is the method that splits the alignment sketch into a sequence of alignment elements based on the curvature of the points recorded within the alignment sketch.

Listing A.4: FindSegments

```
public List<Segment> FindSegments(List<CurvaturePoint> s)
        {
            // List of segments.
            Segment segment = new Segment();
            List<Segment> Segments = new List<Segment>();

            if (s.Count >= 20)
            {
                // Start and end point of segment.
                Point3D startSegment = new Point3D();
                Point3D endSegment = new Point3D();

                // Tolerances.
                double tolLine = 0.0025; // tried and tested
                double tolEnd = 0.0050;
                double tolArc = 0.0400;
                int minCount = 3; // used as updates step for iterators; ensures that segment is
                    at least 5 points long
                bool isLine; // to guarantee sequence
                int start, end, mid;
                start = 0;

                // Start and end point of last segment.
                Point3D endLast = new Point3D(s.ElementAt(s.Count - 1).X,
                    s.ElementAt(s.Count - 1).Y);
                Point3D midPointArc = new Point3D();

                if (s.ElementAt(0).k <= tolArc) // Very first point.
                {
                    startSegment = new Point3D(s.ElementAt(0).X, s.ElementAt(0).Y);
                    isLine = true; // First segment is a line.
                }
                else // First segment is an arc.
                {
                    startSegment = new Point3D(s.ElementAt(0).X, s.ElementAt(0).Y);
                    isLine = false;
                }

```

```
37              for (int b = minCount; b < s.Count - minCount; b++) // Find end point.
38              {
39                  if (isLine) // Search for end of line.
40                  {
41                      if (Math.Abs(s.ElementAt(b).k) >= tolEnd) // End of a line.
42                      {
43                          endSegment = new Point3D(s.ElementAt(b).X, s.ElementAt(b).Y);
44                          segment = new Segment(startSegment, endSegment);
45                          Segments.Add(segment);
46
47                          if ((s.Count - b) < minCount) // Check if near end of sketch, if yes,
                                append last segment.
48                          {
49                              Point3D startLast = new Point3D(s.ElementAt(s.Count - b).X,
                                    s.ElementAt(s.Count - b).Y);
50                              segment = new Segment(startLast, endLast);
51                              Segments.Add(segment);
52                              return Segments;
53                          }
54
55                          for (int c = b + minCount; c < s.Count - minCount; c++) // Find
                                next arc.
56                          {
57                              if ((s.Count - c) < minCount
58                              ) // Check if near end of sketch, if yes, append last segment.
59                              {
60                                  start = s.Count - c;
61                                  end = s.Count - 1;
62                                  Point3D startLast = new Point3D(s.ElementAt(start).X,
                                        s.ElementAt(start).Y);
63                                  mid = (end - start) / 2;
64                                  midPointArc = new Point3D(s.ElementAt(mid).X,
                                        s.ElementAt(mid).Y);
65
66                                  segment = new Segment(startLast, midPointArc, endLast, 1);
67                                  Segments.Add(segment);
68
69                                  Segments.Add(segment);
70                                  return Segments;
71                              }
72
73                              if (Math.Abs(s.ElementAt(c).k) >= tolArc) // Start of an arc.
74                              {
75                                  startSegment = new Point3D(s.ElementAt(c).X,
                                        s.ElementAt(c).Y);
```

```
76                              start = c;
77                              b = c + minCount;
78                              isLine = false;
79                              break;
80                          }
81                      }
82                  }
83              }
84              else if (!isLine) // Search for end of arc.
85              {
86                  if (Math.Abs(s.ElementAt(b).k) <= tolEnd) // End of an arc.
87                  {
88
89                      endSegment = new Point3D(s.ElementAt(b).X, s.ElementAt(b).Y);
90                      end = b;
91                      mid = (end - start) / 2;
92                      midPointArc = new Point3D(s.ElementAt(mid).X,
                            s.ElementAt(mid).Y);
93
94                      CircularArcSegment arc = new CircularArcSegment(startSegment,
                            midPointArc, endLast);
95                      Segments.Add(arc);
96
97
98                      segment = new Segment(startSegment, midPointArc, endSegment, 1);
99                      Segments.Add(segment);
100
101                     if ((s.Count - b) < minCount
102                     ) // Check if near end of sketch, if yes, append very last segment.
103                     {
104                         if (s.ElementAt(s.Count - minCount).X <= tolLine) // Last
                                segment is a line.
105                         {
106                             Point3D startLast = new Point3D(s.ElementAt(s.Count -
                                    b).X,
107                                 s.ElementAt(s.Count - b).Y);
108                             segment = new Segment(startLast, endLast);
109
110                         }
111
112                         return Segments;
113                     }
114
115                     for (int c = b + minCount; c < s.Count - minCount; c++) // Find
                            next line.
```

```
116                         {
117                             if ((s.Count - c) < minCount) // Append very last segment if
                                    close to end of sketch.
118                             {
119                                 if (s.ElementAt(s.Count - minCount).k <= tolLine) // Last
                                        segment is a line.
120                                 {
121                                     Point3D startLast = new Point3D(s.ElementAt(s.Count -
                                            c).X,
122                                         s.ElementAt(s.Count - c).Y);
123                                     segment = new Segment(startLast, endLast);
124                                     Segments.Add(segment);
125                                 }
126
127                                 return Segments;
128                             }
129
130                             if (Math.Abs(s.ElementAt(c).k) <= tolLine) // Start of a line.
131                             {
132                                 startSegment = new Point3D(s.ElementAt(c).X,
                                        s.ElementAt(c).Y);
133                                 b = c + minCount;
134                                 isLine = true;
135                                 break;
136                             }
137                         }
138                     }
139                 }
140             }
141         }
142
143         return Segments;
144     }
```

A.5 CreateSketchCurve

This method takes the start and end points found for in the previous step and creates a sequence of lines and circular arcs with clothoids in between.

Listing A.5: CreateSketchCurve

```
public List<Segment> CreateSketchCurve(List<Segment> s)
        {
            // Only one segment.
            if (s.Count == 1)
            {
                if (s.ElementAt(0).SegmentType == 0)
                {
                    LineSegment line = new LineSegment(s.ElementAt(0).StartPoint,
                        s.ElementAt(0).EndPoint);
                    CurveSegments.Add(line);
                }
                else
                {
                    CircularArcSegment arc = new CircularArcSegment(s.ElementAt(0).StartPoint,
                        s.ElementAt(0).MidPoint, s.ElementAt(0).EndPoint);
                    CurveSegments.Add(arc);
                }
            }

            // Exactly two segments.
            if (s.Count == 2)
            {
                // First segment is a line.
                if (s.ElementAt(0).SegmentType == 0)
                {
                    LineSegment line = new LineSegment(s.ElementAt(0).StartPoint,
                        s.ElementAt(0).EndPoint);
                    CurveSegments.Add(line);

                    // (1) Line-Arc transition
                    CircularArcSegment arc = new CircularArcSegment(s.ElementAt(1).StartPoint,
                        s.ElementAt(0).MidPoint, s.ElementAt(1).EndPoint);
                    TransitionCurveSegment trans = new
                        TransitionCurveSegment(s.ElementAt(0).EndPoint,
                        s.ElementAt(1).StartPoint, line, arc, "LineArc");

                    CurveSegments.Add(trans);
                    CurveSegments.Add(arc);
```

```
36                  }
37                  // First segment is an arc.
38                  else
39                  {
40                      CircularArcSegment arc1 = new
                            CircularArcSegment(s.ElementAt(0).StartPoint, s.ElementAt(0).MidPoint,
41                          s.ElementAt(0).EndPoint);
42                      CurveSegments.Add(arc1);
43
44                      // (2) Arc-Line transition:
45                      if (s.ElementAt(1).SegmentType == 0) // Next element is a line.
46                      {
47                          LineSegment line = new LineSegment(s.ElementAt(1).StartPoint,
                                s.ElementAt(1).EndPoint);
48                          TransitionCurveSegment trans = new
                                TransitionCurveSegment(s.ElementAt(0).EndPoint,
49                              s.ElementAt(1).StartPoint, line, arc1, "ArcLine");
50
51                          CurveSegments.Add(trans);
52                          CurveSegments.Add(line);
53                      }
54                  }
55              }
56
57              // More than 2 segments.
58              if (s.Count > 2)
59              {
60                  for (int i = 0; i < s.Count - 1; i++)
61                  {
62                      // First segment is a line.
63                      if (s.ElementAt(i).SegmentType == 0)
64                      {
65                          LineSegment line = new LineSegment(s.ElementAt(i).StartPoint,
                                s.ElementAt(i).EndPoint);
66                          CurveSegments.Add(line);
67
68                          // (1) Line-Arc transition:
69                          CircularArcSegment arc = new CircularArcSegment(s.ElementAt(i +
                                1).StartPoint,
70                              s.ElementAt(i + 1).MidPoint,
71                              s.ElementAt(i + 1).EndPoint);
72                          TransitionCurveSegment trans = new
                                TransitionCurveSegment(s.ElementAt(i).EndPoint,
73                              s.ElementAt(i + 1).StartPoint, line, arc, "LineArc");
74
```

```
75                      CurveSegments.Add(trans);
76                      CurveSegments.Add(arc);
77
78                  }
79                  // First segment is an arc.
80                  else
81                  {
82                      CircularArcSegment arc1 = new
                            CircularArcSegment(s.ElementAt(i).StartPoint,
83                          s.ElementAt(i + 1).MidPoint, s.ElementAt(i).EndPoint);
84                      CurveSegments.Add(arc1);
85
86                      // (2) Arc-Line transition:
87                      if (s.ElementAt(i + 1).SegmentType == 0) // Next element is a line.
88                      {
89                          LineSegment line = new LineSegment(s.ElementAt(i + 1).StartPoint,
90                              s.ElementAt(i + 1).EndPoint);
91                          TransitionCurveSegment trans = new
                                TransitionCurveSegment(s.ElementAt(i).EndPoint,
92                              s.ElementAt(i + 1).StartPoint, line, arc1, "ArcLine");
93
94                          CurveSegments.Add(trans);
95                          CurveSegments.Add(line);
96                      }
97                  }
98              }
99
100         }
101         return CurveSegments;
102     }
```

Bibliography

[1] D. Wüest and M. Glinz. Flexible sketch-based requirements modeling. In *17th International Working Conference for Requirements Engineering*, pages 100–105, Essen, Germany, 28 March - 30 March 2011. Foundation for Software Quality.

[2] L. Olsen, F. Samavati, M. Costa Sousa, and J. Jorge. Sketch-based modeling: A survey. *Computers & Graphics*, 33:85–103, 2009.

[3] P. Company, A. Piquer, M. Contero, and F. Naya. A survey on geometrical reconstruction as a core technology to sketch-based modeling. *Computers & Graphics*, 29:892–904, 2004.

[4] E. Turquin, M. Cani, and J. Hughes. Sketching garments for virtual characters. *Proceedings of first eurographics workshop on sketch-based interfaces and modeling (SBIM '04)*, 2004.

[5] K. Daz, P. Diaz-Gutierrez, and M. Gopi. Example-based conceptual styling framework for automotive shapes. *Proceedings of eurographics workshop on sketch-based interfaces and modeling (SBIM '07)*, 2007.

[6] L. Kara and K. Shimada. Sketch-based 3d shape creation for industrial styling design. *IEEE Computer Graphics & Applications*, 27(1):60–71, 2007.

[7] R. Juchmes, P. Leclerq, and S. Azar. A freehand-sketch environment for architectural design supported by a mulit-agent system. *Computers and Graphics*, 29(6):905–915, 2005.

[8] L. Wijnholts. Automated geometry checking for infrastructure projects. Master's thesis, Eindhoven University of Technology, 2016.

[9] J. McCrae and K. Singh. Sketching piecewise clothoid curves. *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, 2008.

[10] Y. Kurozumi and W. Davis. Polygonal Approximation by the Minimax Method. *Computer Graphics and Image Processing*, 19:248–264, 1982.

[11] I. Baran, J. Lehtinen, and J. Popović. Sketching clothoid splines using shortest paths. *Eurographics*, 2009.

[12] buildingSmart International. IfcAlignment. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifcproductextension/lexical/ifcalignment.htm (12 July 2019).

[13] *Richtlinie für die Anlage von Autobahnen.* Forschungsgesellschaft für Straßen- und Verkehrswesen, FGSV Verlag, Wesselinger Str. 17, 50999 Köln, 2008.

[14] *Richtlinie für die Anlage von Landstraßen.* Forschungsgesellschaft für Straßen- und Verkehrswesen, FGSV Verlag, Wesselinger Str. 17, 50999 Köln, 2012.

[15] J. Casey. *Exploring Curvature.* Friedrich Vieweg und Sohn Verlagsgesellschaft mbH, Braunschweig Wiesbaden, 1996.

[16] buildingSmart International. IfcTransitionCurveType. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifcgeometryresource/lexical/ifctransitioncurvetype.htm (15 July 2019).

[17] J. Amann and A. Borrmann. Embedding procedural knowledge into building information models: The IFC procedural language and its application for flexible transition curve representation. *Journal of Computing in Civil Engineering*, 2016.

[18] L. Yang, J. Zheng, and R. Zhang. Implementation of road horizontal alignment as a whole for CAD. *Journal of Central South University*, 21:3411–3418, August 2014.

[19] S. Freudenstein. *Verkehrswegebau Grundmodul Vorlesung.* Technische Universität München, Lehrstuhl Verkehrswegebau, 2017.

[20] Š. Markič, J. Schlenger, and I. Bratoev. Tangible alignment design. *Forum Bauinformatik 2018, Weimar*, 2018.

[21] M. Heald. Rational approximations for the fresnel integrals. *Mathematics of Computation 44*, 170:459–461, 1985.

[22] M. Laakso and A. Kiviniemi. The IFC standard - a Review of History, Development, and Standardization. *Electronic Journal of Information Technology in Construction*, May 2012.

[23] A. Borrman, M. König, C. Koch, and J. Beetz. *Building Information Modeling - Technology Foundations and Industry Practice.* Springer, 2018.

[24] N. Young, S. Jones, H. Bernstein, and J. Gudgel. Interoperability in the Construction Industry. Technical report, McCraw Hill Construction SmartMarket Report, 2007.

[25] V. Bazjanac and D. Crawley. The Implementation of Industry Foundation Classes in Simulation Tools or the Building Industry. *Building Simulation Conference in Prague, Czech Republic*, September 1997.

[26] buildingSmart International. IfcAlignment2DHorizontalSegment. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifcgeometricconstraintresource/lexical/ifcalignment2dhorizontalsegment.htm (22 July 2019).

[27] buildingSmart International. IfcAlignment2DVerticalSegment. https://standards.buildingsmart.org/IFC/RELEASE/IFC4_1/FINAL/HTML/schema/ifcgeometricconstraintresource/lexical/ifcalignment2dverticalsegment.htm (22 July 2019).

[28] J. Schlenger. Entwicklung eines Tools für interaktiven Achsenentwurf. Bachelorthesis, Technical University of Munich, 2018.

[29] J. McCrae and K. Singh. Sketch-based path design. *Graphics Interface*, 2009.

[30] R. Mundani. *Bau- und Umweltinformatik 2 Vorlesung Teil 5: Kurvendarstellung*. Technische Universität München, Lehrstuhl Bauinformatik, 2019.

[31] M. Vazquez Méndez and G. Casal Urcera. The clothoid computation: A simple and efficient numerical algorithm. *Journal of Surveying Engineering*, August 2016.

[32] G. Mullineux and S. Robinson. Fairing point sets using curvature. *Computer-Aided Design*, 39(1):27–34, 2007.

[33] G. Schubert, E. Artinger, F. Petzold, and G. Klinker. Bridging the Gap - a (Collaborative) Design Platform for Early Design Stages. *Proceedings of eCAADe in Llubljana, Slovenia*, 2011.

[34] G. Schubert. Early Design Support: Interaktive Simulationen in frühen Entwurfsphasen. *Forschungsbericht, Technische Universität München, Lehrstuhl Architekturinformatik*, 2012.

List of Figures

List of Tables

Declaration

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

München, 15 October 2019

Cara Anna Coetzee

Cara Anna Coetzee
[redacted]