Binary data serialization of CAD 2D drawings

Optimized file format with FlatBuffers serialization library for presenting data

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor’s degree

Information and communication technology, Engineering

Spring 2023

Noora Turunen

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information and communication technology, Engineering		Abstract
Author	Noora Turunen	Year 2023
Subject	Binary data serialization of CAD 2D drawings: Optimized file format with FlatBuffers serialization library for presenting data	
Supervisors	Toni Laitinen (HAMK), Mika Salonen (Trimble)	

Tekla Structures uses TrimBim file format for transferring serialized data. This file format is optimized for the 3D model data. However, the same file format is used for 2D drawings, and, while the format functions sufficiently with the drawings, the 2D drawing optimized file format would increase the efficiency.

The aim of this thesis is to create an enabler for the possible development of 2D engineering drawing file format. The enabler is a FlatBuffers schema, which the binary data serialization with FlatBuffers can be implemented with. Furthermore, the thesis aims to validate the hypothesis that the file size of binary file format written with FlatBuffers serialized data is more compact compared to other vector graphics file formats, such as SVG and DWG.

In this thesis, the main topics to support the realization of the goals are 2D drawings and file formats, as well as data serialization and tools to implement serialization. Furthermore, these topics include subjects, such as vector graphics and data types.

The outcome of the thesis was a schema file for the thesis commissioner, Trimble Inc. Additionally, the thesis validated that the schema is functional by implementing data serialization with FlatBuffers and writing this data into binary file format. This was done separately from Tekla Structures as a prototype project. Furthermore, the SVG file was created and converted into DWG file to compare file sizes. By comparing all three file formats, it was concluded that, with a substantial amount of data, the file size is smaller in the binary file format consisting of data serialized with FlatBuffers.

Keywords	2D engineering drawing, binary file format, Computer-Aided Design, CAD, C++, data serialization, FlatBuffers, SVG, Tekla Structures, vector graphics
Pages	48 pages and Appendices 6 pages

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikan koulutus		Tiivistelmä
Tekijä	Noora Turunen	Vuosi 2023
Työn nimi	CAD-piirustusten binääridatan serialisointi: FlatBuffers serialisointi kirjastolla optimoitu tiedostoformaatti datan esittämiseen.	
Ohjaaja	Toni Laitinen (HAMK), Mika Salonen (Trimble)	

Tekla Stuctures -ohjelmisto käyttää tiedonsiirtoon TrimBim-tiedostoformaattia, joka on optimoitu 3D-malleille. Tätä formaattia käytetään myös 2D-piirustuksiin. Vaikka tämä formaatti on riittävä, tiedostoformaatti, joka on optimoitu 2D-piirrustuksille, lisäisi tehokkuutta tiedostojen käsittelyyn.

Opinnäytetyön tavoite on tuottaa FlatBuffers-skeema, joka voisi mahdollistaa 2D-piirustukselle optimoidun tiedostoformaatin jatkokehityksen. Skeemaa tarvitaan binääridatan serialisointiin, joka toteutetaan FlatBuffers serialisointikirjastolla. Tämän lisäksi tavoitteena on todentaa hypoteesi, jonka oletuksena on, että FlatBuffersin avulla serialisoidusta datasta tehty binääritiedosto on kooltaan pienempi kuin vertailussa olevat vektorigrafiikkatiedostot, kuten SVG ja DWG.

Teoriaosuudessa käsitellään 2D-piirrustuksia, tiedostoformaatteja sekä data serialisointia ja tapoja toteuttaa serialisointi. Muut aiheet, jotka kuuluvat näihin oleellisesti, ovat vektorigrafiikka ja datatyypit.

Toteutuksen lopputuloksena tuotettiin skeema toimeksiantajan Trimble Inc.:n käyttöön. Opinnäytetyössä varmistettiin, että skeeman toiminta täyttää vaatimukset. Tämä toteutettiin serialisoimalla malliaineisto FlatBuffersilla ja tuomalla data binääritiedostoon. Tuotos tehtiin erillisenä projektina, eikä sen koodi tai aineisto ole liitetty Tekla Structuresiin. Lisäksi tiedostokokojen vertailua varten tuotettiin SVG-tiedosto, josta konvertoitiin DWG-tiedosto. Malliaineiston pohjalta tehdyn binääri-, SVG- ja DWG-tiedoston vertailussa todennettiin hypoteesi todeksi, kun datan määrä on runsas ja monimutkainen.

Avainsanat	2D-piirrustukset, binääritiedosto, CAD, C++, data serialisointi, FlatBuffers, SVG, Tekla Structures, vektorigrafiikka
Sivut	48 sivua ja liitteitä 6 sivua

Content

1 Introduction 1

1.1 Research Problem 2

1.2 Goal 3

1.3 Prerequisites and scope 4

2 Handling of 2D drawing data and file formats 5

2.1 2D drawings and graphics 5

2.1.1 Coordinate system, geometric primitives and shapes in drawings 6

2.1.2 Drawing primitives onto the screen with 2D graphics library 7

2.2 Different formats to store and transfer 2D drawings 8

2.2.1 Binary versus text file formats 9

2.2.2 Comparing 2D drawing vector file formats 10

2.3 Data serialization 12

2.3.1 Data serialization with FlatBuffers in C++ 14

2.3.2 Data serialization with binary versus with text 15

2.3.3 Data types in C++ 17

3 Plans and instructions for realization of the thesis goals 19

3.1 Steps to reach goals of making schema and comparing file formats 19

3.2 The structure and syntax of schema 21

3.3 Generating the header file and adding libraries to FlatBuffers project 23

3.4 Writing and reading FlatBuffers 24

4 Process of making FlatBuffers schema and writing files for comparison 26

4.1 Writing primitives into schema 26

4.2 Writing and serializing dummy data with FlatBuffers 31

4.3 Writing and reading files in C++ 34

4.4 Reading and deserializing data with FlatBuffers in program 35

4.5 Generating SVG file from deserialized data 37

4.6 Comparing file sizes of SVG, DWG and new binary file format 39

5 Conclusions 43

Sources 44

Figures and Tables

Figure 1. Data Serialization and Deserialization process (Hazelcast, n.d.). 13

Figure 2. Struct for Point in FlatBuffers Schema. 27

Figure 3. Struct for Line in FlatBuffers Schema. 27

Figure 4. Model table in FlatBuffers Schema. 31

Figure 5. Start of FlatBuffers project file with headers and creating line. 32

Figure 6. Creating Polygon in FlatBuffers project file. 33

Figure 7. Finishing serializing data to FlatBuffers. 34

Figure 8. Writing and reading binary file with C++. 35

Figure 9. Verifying read data and passing it to buffer. 36

Figure 10. Accessing line and polygon data in FlatBuffers. 36

Figure 11. Template function for iterating through FlatBuffer vectors. 37

Figure 12. Function for the beginning for SVG file. 37

Figure 13. Write line content for SVG file. 38

Figure 14. Function to generate, or write, SVG file. 39

Figure 15. Comparing visualization between DWG and SVG. 40

Figure 16. Loop for creating points for polyline and writing polyline to FlatBuffers. 41

Table 1. Advantages and disadvantages Cons of DWG and DXF file (Andy, 2019). 11

Table 2. Comparing SVG and DXF size and loading time (Fahiem & Farhan, 2007, pp. 4–5). 12

Table 3. Comparing serialization between JSON and FlatBuffers (Zaks, 2016)..............16

Table 4. Data types and sizes in FlatBuffers Schema (FlatBuffers, n.d. -e). 22

Table 5. Representing primitives in FlatBuffers Schema. 28

Table 6. Comparing FlatBuffers binary file, SVG and DWG file sizes. 42

Appendices

Appendix 1.	Table of C++ data types, their sizes in bytes and their range. (GeeksForGeeks, 2023 -a; Codecademy, n.d.)
Appendix 2.	PrimitivesSchema.fbs, the FlatBuffers Schema for point, line, polyline, polygon, circle, text, and model
Appendix 3.	Material2d.fbs, the FlatBuffers Schema for pen with colour, line type and line width
Appendix 4.	Data serializing circle, pen, line with pen and text to FlatBuffers
Appendix 5.	Deserializing data for reading pen, circle and text with FlatBuffers
Appendix 6.	Write polygon, polyline, circle, and text to SVG file

1 Introduction

Trimble Inc. product portfolio includes solutions for agriculture, construction, geospatial information, and infrastructures among many others (Trimble n.d. -a). This thesis work will focus on Tekla products and especially Tekla Structures. Tekla Structures is Building Information Modelling (BIM) based Computer-Aided Design (CAD) software for construction industries needs and is used by structural design engineers, detailers, contractors, and project managers among others (Trimble n.d. -b).

The technology this thesis focuses on, CAD, was invented in 1980 (English, 2020), and Tekla Structures, which has roots in Xsteel software, was launched in 1990s (Trimble, 2013 -a, pp. 15-17, 29). At the heart of Tekla Structures are three-dimensional (3D) BIM models for constructions. The information in BIM models does not only include information about objects geometrics but also other levels of information, such as relations and properties between model objects (Czmocha and Pekala, 2014).

However, even in this millennium three-dimensional (3D) models need to be converted into two-dimensional (2D) engineering drawings that are used for example, on construction sites. With Tekla Structures, construction engineer can produce 2D drawings from 3D models almost automatically.

Compared back to the 14th century, when 2D drawings were invented in result of perspective drawing, and to the year 1765 when the descriptive geometry was invented, the drawings in the 2000s include more information. With 3D models, designing has become more convenient, but with the increasing number of information connected to the model objects, the file size has increased as well. This affects the performance of the CAD software and transferring the drawings. Moreover, different file formats challenge the interchangeability of the information between software. Furthermore, transferring and viewing drawings through World Wide Web is an appealing development direction, which would require more efficient file formats in size. (English, 2020; Maciej et al., 2017, p. 32–33)

1.1 Research Problem

Previously, Trimble has developed file format called TrimBim for 3D models. This enhanced transferring of file data in Tekla products. However, this 3D optimized formatting is currently also used for 2D drawings. Therefore, Trimble is investigating how to utilize a similar solution that is optimized for 2D drawings allowing organizing and reading data more efficiently. Moreover, in the future, Trimble seeks possibilities to utilize web applications more, where the size of drawings affects performance significantly.

To answer the questions about optimization and performance, Trimble would need a file format that transfers 2D drawing data efficiently. This would mean reducing the file size without compromising the data and drawing accuracy. The hypothesis is that the binary file format optimized for 2D drawings would be smaller in size and more efficient than already used vector graphic file formats.

Based on these research questions, two tasks have been set for this thesis:

1. to provide an enabler for the development of 2D drawings, which can be enhanced further to complete software specific cases better and maybe be implemented into Tekla Structures software
2. to validate hypotheses by comparing binary file format to chosen vector graphic file formats.

The process for implementing own binary file format for 2D drawing includes creating a schema, which enables data serialization and finally writing data into binary files. The chosen tool for data serialization is FlatBuffers by Google, which is Open Source cross platform serialization library for several programming languages, including C++, C# and JavaScript that are used in Tekla products. The serialization with FlatBuffers is done by creating schema for objects with different data types.

1.2 Goal

To complete the two tasks set for this practice-based thesis, the following goals have been established:

1. Create schema for data serialization with FlatBuffers. The schema would work as the enabler mentioned in the first task. The schema would be used as part of Trimble's product development
2. Validate the hypotheses by creating binary file with serialized data in FlatBuffers that uses the schema created in the first goal, and
3. Generate Scalar Vector Graphics (SVG) file with the previously used data for the comparison

The aim is to create an applicable, low-level schema that might be developed further in the future to include more objects such as fonts. Working means that the compiling of schema succeeds, and data is serialized to FlatBuffers. By generating SVG file from the same data in FlatBuffers, it possible also to validate that the schema works, since with the SVG it is possible to visualize used data on the screen. When making different file formats, the focus is to make publishing format, meaning that the file can only be viewed and not edited.

In the knowledge and the theory section of the thesis, the topic of what 2D drawing is, delving more into the file formats for vector graphics and focusing on SVG and DWG file formats, will be covered. Moreover, the data serialization and how serialization is done with FlatBuffers for C++ programming language will be defined. In the section that covers the objectives of the development work, the steps for the practical portion of this thesis and the instructions for using FlatBuffers in practice are introduced. In the realization of the output, the process of creating FlatBuffers schema with 2D drawing primitives, how the FlatBuffers was used to serialize and deserialize data, how the generation of SVG file was done, and compare file sizes of binary file format, SVG file format and DWG file format is described.

1.3 Prerequisites and scope

The 2D drawing primitive classes of Tekla Structures are used to describe the data structure for schema. Furthermore, having access to the FlatBuffers schema for TrimBim allows to use them as a reference when constructing separate schema for 2D drawings. With the chosen tool FlatBuffers, it is possible to compile the schema into many languages but, in this thesis, the focus is on presenting examples in C++ language only, unless mentioned otherwise.

The schema, the program for using FlatBuffers, and the generating of SVG are done in a separate project repository as a prototype. Even though the project is done from the bases of Tekla Structures primitives, during this thesis, the prototype is not connected to the actual Tekla Structures software or uses its data. All data used in data serialization is dummy data created within the project. The possible follow up could include the tuning of high-level schema and implementation to Tekla Structure, but technology and code presented is this thesis is not guaranteed or promised to be implemented in any Trimble product.

Tekla Structure includes many specific objects, and although primitives for those are available, the thesis focus is on to the so-called basic primitives such as lines, polyline, circle, and polygons as well as their styles. Within the thesis scope, these primitives are sufficient since they function as a bases for creating shapes on screen. However, text primitive is mentioned, and text is also shown in the visualization, but they are excluded from extensive examination, since the text is included in wider discussion about fonts and their vectorization, which has not been concluded during the making of this thesis.

Moreover, there are many more shapes that are mentioned in literature, such as splines, but it should be acknowledged, that this thesis includes only those primitives and possibilities that are available currently in Tekla Structures. Although the thesis is not able to cover every aspect of Tekla Structures, nothing will be introduced that Tekla Structure is not capable of doing.

2 Handling of 2D drawing data and file formats

In this thesis context, when mentioning 2D drawings, it refers to technical drawings, or rather to its subcategory, 2D engineering drawings. In Tekla Structures, a drawing is constructed from views in 3D model. There are several views that can be created in Tekla Structures, for example, views of the entire 3D model, of selected parts and of selected components (Trimble n.d. -c). Moreover, in Tekla Structures, there are 5 different drawing types that can be made for different use purposes: general arrangement, single-part, assembly, cast unit and multidrawings (Trimble n.d. -d).

Computer-Aided Design (CAD) programs have enabled that the 2D drawings could be manufactured from 3D models (Acaddrafting, 2016). For engineers, this means that there is no need to refine detailed drawings, which improves their workflow. If changes are made to the 3D model, the drawings are updated accordingly. This saves time and accuracy, because there is no need to redraw manually after every change. Although 3D improved design, 2D drawings are still the main means to view design in manufacturing and construction sites. (Acaddrafting, 2016; Maciej et al., 2017, p. 29)

In the following chapters, it is often referred to the term vector in different context. It should be noted that when the term “vector” is used as stand alone, it is a reference to sequence container used in C++ programming language (see chap. 2.3.3). When referring to file format or graphics, vector is a descriptive word preceding these terms, such as vector graphics (see chap. 2.1), vector file or vector file format (see chap. 2.2) and so forth.

2.1 2D drawings and graphics

Images that we see on computer screens are built from a collection of pixels. For example, a straight line is an alignment of pixel dots that form a single object that could be edited. When discussing images on the screen, there are two types that are used: a raster image and a vector image. With the raster images, formats such as PNG and JPEG, the pixels are displayed according to the information of pixels’ place and colour. With the vector images, the way the image is rendered to the screen is based on its shape and path definition.

Programs turn on and off the pixels in the viewed area based on the vector data calculation. (Grabowski, 2022; MDN, n.d. -a; Aalto, n.d.)

Consequently, CAD programs use vector file format. Vector files store information about every object in the drawing such as the type, the size, and the location. For example, when drawing a circle, CAD program would only need to know the centre point and the radius of the circle to implement the drawing. If the values change, the program will redraw the circle with new data. (Grabowski, 2022)

Every CAD program has their own algorithm and a way to produce a drawing from the model. However, there are formatting rules regarding the information that is contained in them, such as line types, scales, descriptions, dimensions, and reference line. Primary information, that should be included the drawings automatically, describes the represented element in the model and their properties. Properties are, for example, the name, the type, the geometric data, and the material. Moreover, additional information, such as dimensions, resizing or other previously undefined information, can be added to the model element. (Maciej et al., 2017, p. 29)

2.1.1 Coordinate system, geometric primitives and shapes in drawings

The locations of the drawing's elements in vector graphics, and therefore in CAD programs, are often defined with **Cartesian coordinate system**, which was first conducted by Descartes in his Discourse on Methods. This would mean that the location of starting and ending points of the line, the centre of circle and location of text are measured from the intersection of axes (by convention the X-axis and the Y-axis). These axes are perpendicular to one another creating intersection point or origin; also marked as 0,0 (x, y). Positive points are on the right and above the origin and negatives are on the left and below. (Grabowski, 2022)

The location of the axis depends on rendering library. For example, in Tekla Structures the origin (0,0 point) is in the bottom left of bounding box, meaning that the direction of Y is upwards. Whereas in SVG and in Canvas API, both widely used to draw 2D and vector graphics on web, the point of origin is in the top left corner and the direction of Y is downwards (MDN, n.d. -b).

2D drawings, and overall vector computer graphics, use fundamental vector types to represent geometry: *points*, *lines,* and *polygons*. These are also called **geometric primitives**. They are the simplest geometric shapes that can be used to construct a drawing. (Chen & Clarke, 2017, p. 1; Campbell, J. & Shin, 2011, pp. 86–87)

A *point* is one coordinate pair (x, y) that has only information about the location. Points are used in creating a *line* that has a start and end point. A line that has multiple points, but does not close back to the starting point, is called *polyline*. A collection of lines that are connected and closed to the starting point is called *polygon*. The meeting point of the lines and polygon corners are called vertices. (Campbell, J. & Shin, 2011, pp. 86–87)

In addition to fundamental geometric primitives, there are other complex yet usually predefined geometric shapes, such as circles and arcs. More complex shapes that are not predefined, are constructed from the compilation of geometric primitives. (Chen & Clarke, 2017, p. 1; Campbell, J. & Shin, 2011, pp. 86–87)

2.1.2 Drawing primitives onto the screen with 2D graphics library

The visualization of the 2D drawing data is done with the 2D graphics library. These are program libraries that are designed to assist with the rendering of computer graphics onto the monitor screens.

Rendering, also called rendering pipeline, consists of three conceptual stages: application, geometry, and rasterization. These stages in themselves consist of pipeline stages that have functional stages or, in other words, tasks. The number of tasks varies depending on the implementation in graphics library; in some they might be combined and in others separated. (Akenine-Möller et al., 2000, pp. 13–14)

The stage, where the developer can have to the most impact on implementation and on performance effecting the following stages, is the application stage (Akenine-Möller et al., 2000, pp. 14–15). In this thesis, the work is done on this stage. The geometric primitives and other rendering primitives, meaning points, lines etc., are handed over to geometry stage at the end of application stage. The geometry stage is where the model is transformed to different coordinate systems and furthermore to its unique world, as well as where partially shown primitives are handled. Rasterizer stage uses data from the geometry stage to compute and set colours for pixels that will ultimately become an image on screen. (Akenine-Möller et al., 2000, pp. 15–26)

There are many different graphics libraries available, and libraries in offline software bring even more variety. Some well-known libraries for 2D graphics are, for example, DirectX, and, especially for web usage, WebGL and Canvas API. (MDN, n.d. -c) Tekla Structures has its own 2D graphics library called xkit that passes on data given from Core side of the software. Another way to draw on screen is SVG, a XML mark-up language, that uses specifications of vector graphics and is supported by major browsers such as Chrome and Mozilla Firefox (MDN, n.d. -d).

2.2 Different formats to store and transfer 2D drawings

Usually, different software has their own file format that stores data in the most suitable way for the programs' needs. This, however, causes challenges for the interoperability between different software. Therefore, it is common practice that files can be converted into different file formats. (Maciej et al., 2017, p. 32–33)

It has been established that CAD programs use vector files since they support different type of data. Therefore, the focus is on comparing vector file formats that are used in CAD programs and representing in binary format.

The most common file formats for 2D drawings that use vector data are DXF, DWG, DGN, PDF, and SVG. The difference between the file formats comes from the compactness and which line formats, colours, layering, or curve types are supported. Some of the formats are binary-based and some use XML. (Maciej et al., 2017, p. 35)

At the time of writing this thesis, DWG is used in Trimble Connect software that links Tekla Structure to other Tekla products to transfer drawing data. SVG, on the other hand, is one of the most popular formats to show vector graphics on web and therefore, it is interesting to compare these to formats (MDN, n.d. -e). One of the thesis goals is to compare binary file format to SVG and DWG. For that purpose, it is important to understand what these file formats are.

2.2.1 Binary versus text file formats

Before comparing file formats, it should be examined how data is stored in them. Data can be categorized into two types: numeric data and character data. These data types (see chapter 2.3.3 about data types in C++) are represented in binary and are stored in memory in terms of bytes. (Dot Net Tutorials, n.d. -a)

Generally, there are two file types: binary and text. In binary files, the series of sequential bytes (8 bits each) are arranged into a binary format. Binary file can have in the same file different types of data that construct graphics, such as images, and videos. Text files in turn can only contain textual data, and, since it is a standard format, multiple programs can read and edit the file. By contrast, with the binary file, the program or hardware processor needs to know exactly how the content is formatted and data read. Usually this requires separate schema. The clearest difference between the formats is that text files are human-readable unlike the binary. (Sheldon, R., 2022; Itskawal2000 et al. 2023; FileInfo.com, 2011)

Binary file saves memory by storing any data type in memory as per its memory size. As this format is not the most user-friendly, an error can corrupt the file and the error is not easily detected. Moreover, since it is not human-readably, the reading and modifying is not effortless either. However, binary format stores data more compactly and can have application defined extension. (Itskawal2000 et al. 2023)

In a nutshell, the benefits for using binary files compared to text files are:

1. Input and output are faster when data is in binary format.

2. File size is smaller, which decreases the storage space, the transmitting and processing time. For example, in system, where certain data values are not permitted and they need to be translated into text format, the file size can increase by 30 percent (Rouse, M., 2012)

3. Not all data can be represented with characters.
(Kjell, B., n.d., chapter 86 page 6)

Developers rarely work directly with the binary file and the logic is defined and applied to human-readable text file, schemas, and to project file of the chosen high-level programming language. During the development process, schemas and project files are compiled so that the source code is translated to machine code. When transferring file as binary, programs that receive them do not have to review or change them. (Sheldon, R., 2022)

2.2.2 Comparing 2D drawing vector file formats

One aspect of comparing 2D drawing file formats is the accessibility. Software's internal presentation of vector graphics is mostly proprietary and the code to recreate them is not available. Therefore, many CAD programs enable importing and exporting drawings in different formats gaining possibility for interchanging information. (Eisenberg & Bellamy-Royds, 2014, chapter 1)

The two commonly used vector file formats for CAD drawings are DWG, which is native file format of AutoCAD that supports 3D and 2D, and DXF for 2D, which is recognized by CAD tools only. Both these file formats are de facto standards and store same type of data with similar level of quality. DXF and DWG contain data to represent the contents of CAD files, such as design, geometric data, and maps and photos, and are mainly used by architects, engineers, and designers. (FileFormat, n.d.; Fahiem & Farhan, 2007, p. 1; Adobe n.d.)

Although DWG is the most used in AutoCAD, some other CAD software, as well as Tekla Structures, support this format. However, DXF file, as an open standard file format, is supported by more software compared to DWG and thus, is a standard for exchanging the design files. (FileFormat, n.d.; Fahiem, M. & Farhan, 2007, p. 1; Adobe n.d.) Therefore, the apparent difference between these formats is accessibility and compatibility. Another difference between these files is that the DWG is a binary file and DXF text file, and consequently, DWG is more compact compared to DXF. There are also some CAD elements that DXF does not support, such as dynamic blocks. (BlueEnt, 2022) Some distinctive differences between DWG and DXF are gathered in the Table 1.

Table 1. Advantages and disadvantages Cons of DWG and DXF file (Andy, 2019).

	Advantages	**Disadvantages**
DWG	Native file format for AutoCAD Binary file format, resulting in smaller file size Supports both 2D and 3D graphics	Not publicly documented Not supported in all CAD programs Not supported in web browser
DXF	Open format Publicly documented Supported almost in all CAD programs	2D graphics only Text file format, resulting in larger file size Not supported in web browsers

As seen from comparing DWG and DXF in Table 1, both files lack support for web browser. Most of the CAD drawing files are encoded in binary and therefore, browsers and other software or products might not be able to parse and create vector files out of them. Therefore, one aspect of considering different file formats for representing drawings is the use of web applications or using HTML document viewer to observe the drawing. SVG has gained reputation as the most precise for representing and accessing CAD 2D drawings from browsers and devices (Snowbound, 2015).

Scalar vector graphics, SVG, is a vector format for 2D vectors that is based on XML markup languages and was developed by World Wide Web Consortium (W3C) in 1999. Defined in XML means that the SVG images can be searched, indexed, scripted, and compressed as well as created and edited with text editor or other software. Essentially, SVG is same for images than HTML is for text. (Snowbound, 2015; SVG on the Web n.d.; MDN n.d. -f; Fahiem, M. &

Farhan, 2007) What makes SVG appropriate for Web is its small file size and that most browsers support it (Andy, 2019). However, it should be noted that when using more complex SVG file with bigger file size, this would also affect the processing time in the browser (Amadine, 2022).

In their research, Representation of Engineering Drawings in SVG and DXF for Information Interchange, Fahiem et al. concluded that when distributing engineering drawings, such as CAD drawings, through web, SVG format is more suitable in comparison to DXF. This comparison was done in basis of file size and loading time. The results of their research are collected in Table 2, where it is possible to see that, at best, SVG was one-third of the size of DXF and one-fourth faster in processing speed. (Fahiem & Farhan, 2007, pp. 4–5)

Table 2. Comparing SVG and DXF size and loading time (Fahiem & Farhan, 2007, pp. 4–5).

	SVG	**DXF**
File size	Less, 200kB	More, 800kB
Loading time	Less, approx. 150ms	More, 600ms

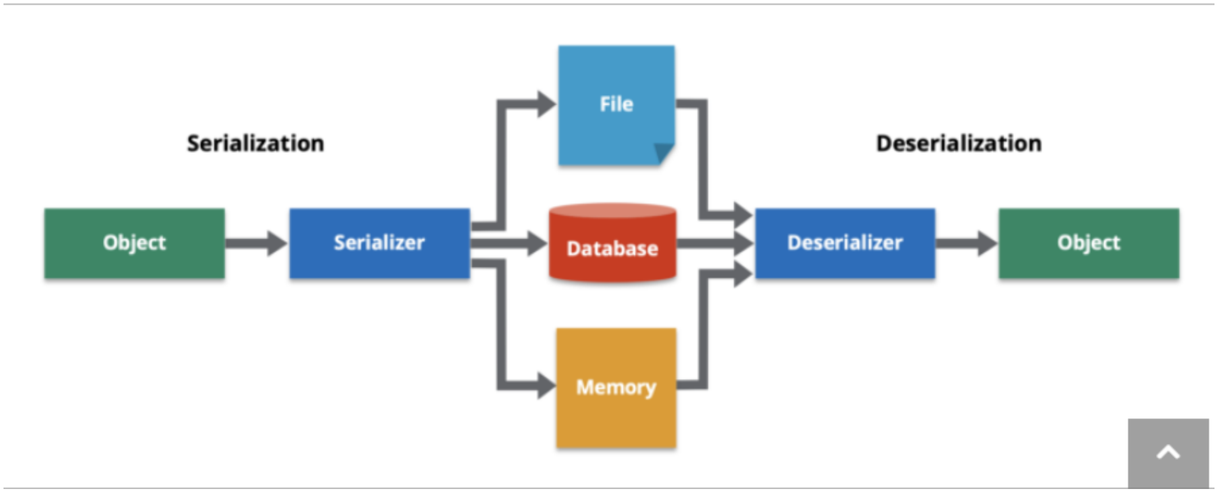
2.3 Data serialization

Now that the topics of file format types and their benefits and challenges in CAD file formats have been discussed, the next step is to define how the data structures are stored as a stream of bytes, in other words, how data is serialized (Kleppmann, 2017). Streams are the sequences of bytes that move in and out of the program (Nanyang Technological University, 2013).

Serialization means that a data structure is written into format that can be stored, transmitted, and reconstructed. The storing could happen in a file or in a data buffer. Buffer stores temporarily input data in the memory to be outputted later. Transmitting could be, for example, streaming data over networks, and reconstructing could, for example, occur when data is handled in different environments. (Devopedia, 2020)

The process of reading serialization format and creating identical clones of the original object is called deserialization. Similarly to serialization, where data is stored in stream of bytes, deserialization extracts a data structure from the stream (see Figure 1). After serialization, the data is transferred to be stored in file, database or in memory where they can be streamed to the program to be deserialized. (Hazelcast, n.d.)

Figure 1. Data Serialization and Deserialization process (Hazelcast, n.d.).



Cases, where data serialization is used, are, for example, in

- Persisting state, meaning how long the data persists after closing the application it was created in (MongoDB, n.d.)
- Machine to machine communication
- Representing the configuration

(Zaks, 2016; Devopedia, 2020)

Furthermore, there are five ways to conduct data serialization: **Custom binary representation**, which is used for example with persisting state and machine to machine communication , **built-in binary serialization**, which is included in most object-oriented program languages, such as Java and C#, **text-based representations**, which is popularly used in representing configuration and its most common formats are XML, JSON and YAML, **Embedded SQL or NoSQL databases**, which is usually used only with persisting state, and

the binary cross platform serialization library, which will be the focus of this thesis. One of the serialization libraries, FlatBuffers, will be used in this thesis. Other serialization libraries are, for example, ProtocolBuffers, which is most in common with FlatBuffers. Serialization libraries are fit to be used with all the mentioned data serialization cases. (Zaks, 2016)

2.3.1 Data serialization with FlatBuffers in C++

FlatBuffers is an Open Source, cross platform serialization library that is created by Google, and it is available under the Apache License v2 on GitHub: github.com/google/flatbuffers. FlatBuffers is supported in different operating systems, such as Windows, macOS, Linux and Android and can be used with many programming languages such as C++, Java, C#, Go, Python, Rust, JavaScript, PHP, C, Dart, Lua, and TypeScript. (FlatBuffers, n.d. -a)

Originally FlatBuffers was made to meet the needs of game development and other performance-critical applications. The motivation was to create a serialization library, which considers memory in terms of spreading, accessing, and allocating, and in number of usages. Moreover, when it comes to serialization, developers wanted to reduce the need for parsing since it would mean additional copying, allocating patterns, locality and using temporary data structures – all that causes memory inefficiency. (FlatBuffers, n.d. -b)

Reasons to use FlatBuffers, provided by their website, are:

1. possibility to access serialized data without first parsing or unpacking, which is due to representing data in a flat binary buffer,
2. memory efficiency and speed, where it is only needed to access data in buffer and only part of buffer is required to be in memory,
3. flexibility, meaning that it is compatible forward and backwards, and user can control how to design data structure and what data to use,
4. its small code footprint, referring to the amount of generated code and minimum dependency of one header file,
5. it is strongly typed, meaning that errors happen at compile time,

6. it is convenient to use with generated C++ code and option to parse schemas and text representation, and
7. it uses cross platform code without dependencies, which with C++ means that it works with recent gcc or clang compilers.

(FlatBuffers, n.d. -c)

More details about how the FlatBuffers is used are introduced in chapters 3.2–3.4. In general, FlatBuffers is used with schema files, which define data structure that is used in a program. FlatBuffers schema uses interactive data language, IDL, which has similar syntax to C++. Schema has fields for different variables, and they can be either scalar or compound type (more about data types, see chapter 2.3.3). These fields are optional, and they have default values.

Based on the definition in the schema, a file is generated for the chosen programming language. For example, in the thesis, the C++ programming language will be used and therefore, the schema is compiled into C++ header file. This header file is then included in the actual program along with FlatBuffers library. From the FlatBuffers library, the FlatBufferBuilder class is used for constructing a flat binary buffer and with the functions in generated header file, user can add object to the instantiated buffer. When reading the FlatBuffers, the user obtains a pointer to the root object that can be used to access other fields in the schema. (FlatBuffers, n.d. -c)

2.3.2 Data serialization with binary versus with text

Data serialization can be represented with a binary (binary serialization) or with text (for example, XML and JSON) (Devopedia, 2020). In his article, Beyond JSON — Introduction to FlatBuffers, Maxim Zaks compares JSON, a text-based data serialization format, with FlatBuffers, a binary-based data serialization library. In the Table 3 below, his main findings when using these formats for data serializing persistent data and sending data, are collected. If interested to see more data on FlatBuffers performance compared to other serialization solution, refer to FlatBuffers Benchmark page (FlatBuffers, n.d. -d).

Table 3. Comparing serialization between JSON and FlatBuffers (Zaks, 2016).

Criteria	**JSON**	**FlatBuffers**
Size: the outcome of the process and the smaller the better	With simple messages, JSON is better but can be verbose when handling a lot of repeating data.	With a lot of data that has repeating data structure, FlatBuffers is better. With a small amount of data, it is unnecessarily powerful.
Read and write performance: how fast is the process, meaning converting data in and from memory	Has to be parsed, which means that transient memory is used, and therefore, the efficiency is lesser than with FlatBuffers.	Reads values directly from buffer. Therefore, there is no need for parsing, decoding or creating transient memory.
Human-readability and writability	The best part of JSON.	Binary itself is not readable but with sending data, this is not relevant.
Supporting object-oriented language types: typed data eliminates some bugs and smooths development	Library for OO languages exists but is not straightforward to use.	The schema compiler generates APIs for many languages.
Version support: data might be read between different client versions	Is not a features but there is possibility to write migration code and can be made to be backwards compatible.	With virtual tables, data storing is backwards and forward compatible.

Benefits with text-based data serialization is that it is human-readable, and this type of encoding is the most useful with a small amount of data. Another advantage is that it can also be communicated to other systems regardless of the programming language. However, what text-based encoding gains in readability, it loses in file size. Generally, the binary occupies small physical space in bytes after serialization. Moreover, binary formats are faster and become even faster with compressed data. (Devopedia, 2020)

2.3.3 Data types in C++

Since serialization relates to handling data structures and organizing bytes in stream, the data types, and the number of bytes they use, affect the serialization performance as well as writing and reading of the files (Devopedia, 2020). Therefore, it is relevant to discuss data types more in detail and the size of memory they allocate. In this chapter, the data types for C++ programming language will be introduced.

C++ is strongly and statically typed language. This means that, to compile, every object; variable, function arguments, function return value, must have type that never changes, unless copying value to different type as a result of conversion. In general, there are scalar and non-scalar data types. Scalar type holds a single value that can be arithmetic (integral or floating-point values), a pointer type, pointer-to-member types and std::nullprt_t. Non-scalar types or compound types include array types, function types, class or struct types, union types, enumerations, references, and pointers to non-static class members. (Microsoft, 2022)

C++ has three categories of data types for variable and function; fundamental or build-in, derived and user-defined data types. Fundamental data types are so called primitive data types that can be used directly, and derived data types are derived from them. Usually, fundamental types are scalar types. (GeeksForGeeks, 2023 -a)

Fundamental data types in C++ are **integer**; int, **floating point;** float, double, **Boolean;** bool, and **character**; char. Additionally, std::string, which is compound type that consists of sequence of character (char) sequence, is used in this thesis. In Appendix 1, the memory sizes for these data types are collected in bytes. 1 byte is 8 bits, and out of these bits the first is the most significant and the last the least. The sign of the number, meaning positive (zero, 0) or negative (one, 1), is reserved in the first bit. (Dot Net Tutorials, n.d. -b)

In addition to these types, there are data type modifiers in C++ that modify the length of data types (int, double or char). These modifiers are signed, unsigned, short, and long. (Codecademy, n.d.) In Appendix 1, it is also shown how these modifiers effect the size and range of data types. As it can be seen in the Appendix 1, signed modifier means that both negative and positive values can be used. On the contrary, with unsigned, only zero (0) and positive values are used. Moreover, long modifier increases the size, whereas short modifier decreases the size of data type. (Mahamuni, 2022)

When writing a schema, the struct type is used. Although the schema uses other language than C++, struct has a same function as in C++. Struct is used-defined typed, and once it is defined, it can be used as a fundamental type in a program. The difference to fundamental types is that compiler does not have built-in knowledge of them and learns it during the compilation process. (Microsoft, 2022)

When writing the Flatbuffer project, a vector is used. In C++, vectors are sequence containers, and, similarly to dynamic arrays, can resize automatically. Vector's type is defined between angle brackets, <>, and to access element in the vector, a pointer to the element is used. (GeeksForGeeks, 2023 -b; cplusplus.com, n.d.)

3 Plans and instructions for realization of the thesis goals

As established in the previous chapters, there are many aspects to consider when creating file format for CAD 2D drawing. The theory mainly alludes to the topics regarding publishing file format, which means that the representation is only for viewing and not for editing or another interactivities. However, the issue would remain the same: how to transfer large data sizes efficiently without losing data. Web offers a great way to inspect this because browsers support many graphics viewers and file size affects the performance considerably. Additionally, regarding the direction where the industry is headed, the possibility to view drawings through web or mobile would improve connectivity.

The hypothesis that the data, serialized with FlatBuffers, in the binary file format would be smaller in size, is supported by the theory. Therefore, a binary file format with serialized data with FlatBuffers offer possibilities to improve performance of 2D drawings in Tekla Structure. This thesis attempts to prove that the hypothesis is correct by comparing the file size of formats 1) that is a possible new binary file format constructed with FlatBuffers, 2) that is popular representing vector graphics online, SVG, and 3) that is currently used in Tekla products to transfer data between users, DWG. To do this, a binary file format with FlatBuffers serialized data in bytes needs to be created with several steps. After that, a SVG file format from the same data should be created. Finally, the SVG file is converted into DWG file. The making of binary file format includes steps of making FlatBuffers schema and data serialization. These steps are presented more closely in the following chapter.

3.1 Steps to reach goals of making schema and comparing file formats

As mentioned in the beginning, the output of this thesis is an enabler and will be used as a separate prototype. Since it is separate and works as an individual project, dummy data is used instead of actual model data to create the drawing in a program that uses FlatBuffers for serialization. To validate that serialized data actually can be drawn on the screen, the aim is to visualize this data within this project.

The steps of the process done the scope of this thesis are the following:

1. Create schema file from drawing primitives (see chap. 4.1)
2. Create dummy data and pass it to FlatBuffers (see chap. 4.2)
3. Create buffer with data given to FlatBuffers (see chap. 4.2)
4. Write binary file (see chap. 4.3)
5. Read binary file (see chap. 4.3)
6. Pass binary data to buffer to be used with FlatBuffers (see chap. 4.4)
7. Handle data with FlatBuffers to get values (see chap. 4.4)
8. Pass values and write them to text file (see chap. 4.5)
9. Write SVG file (see chap. 4.5)
10. Compare size between SVG, DWG and new binary file format (see chap. 4.6)

Passing data and creating model (steps 2–3) are the parts of serialization of the data that are done with FlatBuffers schema (step 1). Writing and reading binary file (steps 4–5) is part of validating that the serialized data will pass on to be used in another program that use FlatBuffers. This step also produces a binary file format for the comparison. Passing binary data and handling that given data (steps 6–7) are parts of deserializing of the data to acquire the values to be used in another program. As one goal is to visualize the data and see that it can be used for drawing on screen, the SVG file format is created in steps 8–10. After these steps, if there is a shape inside the defined bounding box and the shape has colour, it can be concluded that the data serialization has been successful. This also produces the SVG file for the comparison.

To check the hypothesis “compression of data to binary file format is smaller and thus more efficient than in other formats, such as SVG and DWG”, the SVG file will be converted with tools found online to DWG file and the file sizes of SVG, DWG and new binary file format will be compared (step 10).

For the process of implementing these steps, a blank project repository in BitBucket, and a text file with the *.fbs* file extension, which enables FlatBuffers to read the file, are created. This will be the schema file. To use FlatBuffers and to execute serialization and deserialization as well as to create binary and SVG file, a project solution is created with Visual Studio 2022.

Next in this chapter, the concept of making FlatBuffers project in general, writing and compiling of the schema, and writing and reading of FlatBuffers is explained. This serves as a basis for chapter 4, where working with a FlatBuffers in practice is presented.

3.2 The structure and syntax of schema

As mentioned in chapte 2.3.1, FlafBuffers schema uses IDL and looks similar to the C family languages. Below, the syntax and types used in FlatBuffers schema are introduce from those part that are relevant to this thesis. For example, there is documentation about unions in the schema, but since they will not be used in this schema, it is not relevant to explain them in detail.

The main way to define object in schema is using tables. The table has a name and fields. Each field in turn has a name, a data type and an optional default value the following way:

```
table Example {
  field_name1:data type (required);
  field_name2:data type (required);
}
```

The default value is zero, 0, for scalar type zero, and for other types null if they are not defined by the user. Renaming fields and tables is possible, but it should be noted that adding a new field is done at the end of a table definition because it would be ignored in compiler. Moreover, fields cannot be deleted, but by marking them “deprecated”, the fields are enforced to not be used anymore. (FlatBuffers, n.d. -e)

Other way to define object is with struct. Contrary to the table, the fields or defaults are not optional. Struct may also contain only scalar types and other structs. Benefit for struct is that it uses less memory than table and is faster to access. (FlatBuffers, n.d. -e)

FlatBuffers schema has built-in scalar types and built-in non-scalar types. The list of these types and aliases, which can be used instead and without affecting code generation, are collected in the Table 4. The types cannot be changed, but same-size data could be casted to another if the current data does not use higher number of bits. For example, in Table 4 uint could be changed into int. Enum is one of the non-scalar types. It defines a sequence of constants that can only have integer types. When enum is declared, the data type is specified with ":" and all the fields in enum would have this type. (FlatBuffers, n.d. -e)

Table 4. Data types and sizes in FlatBuffers Schema (FlatBuffers, n.d. -e).

Size	**Data Type**	**Scalar or non-scalar**
1 byte (8 bits)	byte (int8) ubyte (uint8) bool	Built-in scalar types
2 byte (16 bits)	short (int16) ushort (uint16)	
4 byte (32 bits)	int (int32) uint (uint32) float (float32)	
8 byte (64 bits)	long (int64) ulong (uint64) double (float64)	
Depends on type	Vector of other type, denoted with box brackets; [type]	Built-in non-scalar types
Depends on type	string; only hold UTF-8 or 7-bit ASCII	
Depends on type	References to other tables, structs, enums or unions	

An attribute can be attached to a declaration in a field, a table or a struct. This attribute is optional and would communicate to the compiler, for example, the order priority, which fields are required, and which fields should not be generated anymore. In this thesis, the *required* attribute is used, which is optional and indicates that the field value should always be set. Using this attribute contributes to the forward and backwards compatibility making the schema to be not version specific. If the required field is not initialized in a code, an assert will rise, and the buffer verifier would fail. (FlatBuffers, n.d. -e)

Tables and structs with fields are the basic blocks that build up schema. In addition, schema structure would include namespaces and includes, that could be other schema files. In the end of the schema, the root type is declared. The root type is the root table or root struct of the serialized data. The root dictates where to start the serialization. (FlatBuffers, n.d. -e)

FlatBuffers schema, as many other languages, has its recommended style for naming and formatting. These are:

- UpperCamelCase for table, struct and enum
- snake_case for table and struct field names
- UpperCamelCase for enum values
- UpperCamelCase for namespaces
- Opening brace on the same line as the start of declaration
- Indent is by 2 spaces
- No spaces around ":" with types
- Spaces on both sides with "="

(FlatBuffers, n.d. -e)

3.3 Generating the header file and adding libraries to FlatBuffers project

Compiling of FlatBuffers schema is done with flatc executable through a chosen terminal. Flatc is a Schema Compiler that generates header files from schema. There are instructions how to build flatc executable on Google's FlatBuffers site: Building with CMake. (FlatBuffers, n.d. -f) In this thesis, a prebuilt flatc.exe was used that was built for Tekla Structures and could be found in its private repository.

In command, user can specify the language they want to generate header file for. For example, in this thesis, “c” is used since it stands for C++. Lastly, the name of the schema file, that is generated into header file, is added to the command. (FlatBuffers, n.d. -g) For example, a FlatBuffers schema file called PrimitivesSchema.fbs is used in this thesis. To generate this into a header file for C++ language, a command “flatc -c PrimitivesSchema.fbs” is used.

In order to use FlatBuffers with the new generated header file in program, the generated header file and FlatBuffers C++ library need to be included in the project. The library can be found in FlatBuffers GitHub: github.com/google/flatbuffers/tree/master/include/flatbuffers.

As this thesis uses Visual Studio 2022 project (Console App), additional include library needs to be manually added to the project (Microsoft, 2021). The path to do this is in Visual Studio view and selecting from project the following: Properties -> Configuration Properties -> C/C++ -> General -> Additional Include Directories.

3.4 Writing and reading FlatBuffers

The usage of FlatBuffers, meaning writing and reading FlatBuffers, is explained from the perspective of C++ but on FlatBuffers site, there can be found tutorials for other languages as well. Once the library and the generated header files are included in the project, the same namespace that was specified in the schema is used in the project. FlatBuffers uses data from buffer and for this purpose the writing is started by creating of FlatBufferBuilder instance that can be found in FlatBuffers library. This instance contains the buffer, and the user can give it an initial size, but it is not mandatory since the buffer will grow automatically. (FlatBuffers, n.d. -h)

Before adding data to instance, the instance is called here “exampleInstance” for demonstration, the user needs to create or import the data that would correspond with the data types defined in schema. Values for build-in types can be passed to variable in usual coding manner, but some other types must be passed with the FlatBuffers function. (FlatBuffers,n.d. -g) In these cases, it is usually convenient to give the variable “auto” type, since it can recognize the FlatBuffers type like in the example below:

```
auto exampleString = exampleInstance.CreateString(“Example”);
auto exampleVector = exampleInstance.CreateVector(vector);
short exampleInt16 = 3;
```

Adding data to the buffer is done with the functions generated in header file (FlatBuffers, n.d. -h). For example, if there is a table called Example, function call to create and to add that to the buffer would be along the lines:

```
exampleInstance.CreateExample(exampleInstance, exampleString,
exampleInt16);
```

Once all the tables and structs are serialized with the create function, the final serialization is done for the root table. When this is completed, the buffer is finished by calling the finish method, for example exampleInstance.Finish(root). (FlatBuffers, n.d. -h)

After serializing the data, the buffer can be stored, compressed or transfer. If this is written into a binary file, the user can get the needed pointer and buffer size with flatbuffers methods GetBufferPoint() and GetSize(). (FlatBuffers, n.d. -h)

When reading the FlatBuffers, the program, where the data has been sent, should also include generated header files from schema and FlatBuffers library and use the same namespace that is specified in schema. FlatBuffers is read-only, meaning in C++ that everything is const. Buffer pointer to the data that has been read should be data type of uint8_t. To get the pointer to the root object, the GetExample(bufferPointer) method is used. Now that the user can access to root, they can also access the tables and their fields. (FlatBuffers, n.d. -h)

4 Process of making FlatBuffers schema and writing files for comparison

In this chapter the process and findings of realization of the defined goals will be presented. The following topics are in order of the steps introduced in p. 20: making schema, writing FlatBuffers (serializing), writing and reading binary file format, reading FlatBuffers (deserializing), generating SVG file format and comparing sizes of a new binary file format, SVG, and converted DWG.

4.1 Writing primitives into schema

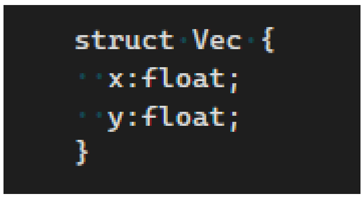
Primitives that are included in schema in the scope of this thesis are point, line, polyline, polygon, circle, and pen. The text primitive is included into final representation, but since some of the topics relating to it, like fonts, are undecided, they are mentioned briefly and recognized to need more development.

In Tekla Structures, drawings are defined with primitive classes where the definitions for member functions and variables are. With these files, it is possible to see how primitives are built and with what data type, and a data structure can be constructed to the schema file.

To further explain this, a small example of how basic line from points is constructed is introduced below.

1. Line has starting and ending point and thus it can be identified that line has two variables.
2. Both variables are points, which tell a coordinate in 2D space along x- and y-axis. Thus, one point would have two variables.
3. Point values, x and y, have a data type of float.
4. Now a struct can be built on bases of x and y data type. This struct is called Vec and it would tell that the struct is a vector containing two fields. (See Figure 2)

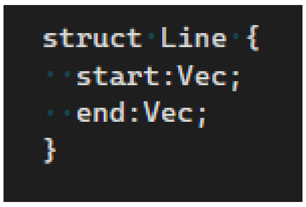
Figure 2. Struct for Point in FlatBuffers Schema.



```
struct Vec {
  x:float;
  y:float;
}
```

5. Having struct for point, a struct for line can be made with the start point and the end point with data type Vec. (See Figure 3)

Figure 3. Struct for Line in FlatBuffers Schema.



```
struct Line {
  start:Vec;
  end:Vec;
}
```

In the Table 5 (pages 28–30), all the primitives that are included in schema, which data types they are constructed of, and how they are written in schema, are presented. There is also a visualization of the shape, where the points of the shape are marked. The screenshots of the actual FlatBuffers schemas are in Appendices 2–3.

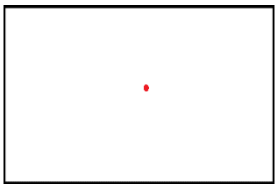
The schema was started with the most basic concept of primitives: **points**. With points, other primitives, such as line that has starting and ending point, can be built, which can be used in turn to build more complex but basic primitives such as polylines and polygons etc. When creating points in Tekla Structures, the program uses the Vector_c type, which uses struct point_t that has three member variables (x, y, z) that are type double. The third variable has a default value of zero and thus works well in 2D scenarios as well.

Although in Tekla Structures' primitives, the values of the points are double, the float type was chosen after discussion with Trimble instructor. It was concluded that float is large enough type to define values for the points. The reasoning behind it was that since these are 2D drawings that are wanted to be formatted for web use and they are not civil engineering drawings including maps with several kilometre accuracy, float should be sufficient type to contain the used values. Moreover, as discussed in chapters 2.3.3 and 3.2, the using of double data type would also double the memory size compared to using float data type.

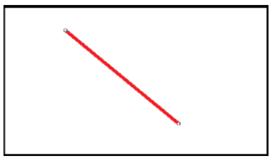
Now that there is a definition for point, a **line** can be defined. In a code, a line segment has a starting and ending point type Vector_c. With **polyline**, there would be more points between the start and end point. Therefore, the points field would be a vector with type of Vec.

Table 5. Representing primitives in FlatBuffers Schema.

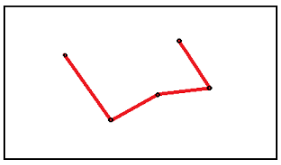
Primitive in Tekla Structures	**Class variables in Tekla Structures**		**FlatBuffers Schema**	**Visualization**
	Type	**Name**		

Point primitive	class Vector	Position	`struct Vec {`
			`  x:float;`
			`  y:float;`
			`}`
				

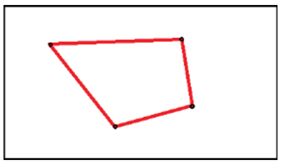
	Note: class Vector inherits from public point and is defined as
	
	struct point
	{
	  double x, y, z;
	}			

Line segment primitive	Vector	StartPoint	`struct Line {`
			`  start:Vec;`
			`  end:Vec;`
			`}`
				

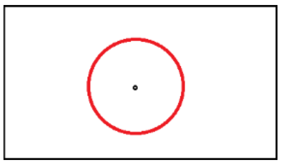
	Vector	EndPoint		

Polyline (not separate primitive class in Tekla Structures)	std::vector <Vector>	Points	`table Polyline {`
			`  points:[Vec];`
			`  style:Pen;`
			`}`
				

Polygon primitive	LoopPrimitive	OuterLoop		

	std::vector <LoopPrimitive>	InnerLoops	`table Polygon {`
			`  position:[Vec]`
			`  (required);`
			`  style:Pen;`
			`}`
				

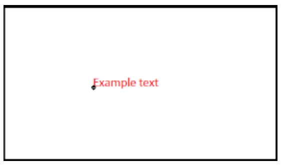
	Note: LoopPrimitive class uses vector with class type of Line segment primitives, Arc segment primitives and Path segment primitives. In thesis, the Line segment primitive is used. In the schema, only the OuterLoop is used.			

Circle primitive	Vector	CenterPoint	`table Circle {`
			`  center:Vec;`
			`  radius:float;`
			`  style:Pen;`
			`}`
				

	double	Radius		

Pen primitive	int	LineColor	`table Pen {`
			` line_color:`
			` LineColorDefined`
			` = Black;`
			` line_type:LineType`
			`= Normal;`
			` line_width:`
			` LineWidth;`
			`}`	

	typedef unsigned short	LineType		
	unsigned char	LineWidth		

Text primitive	std::string	Text	`table TextDisplay {`
			`  text:string;`
			`  font:string;`
			`  position:Vec2f;`
			`  height:float;`
			`  angle:float;`
			`  style:Pen;`
			`}`
				

	std::string	Font		
	Vector	Position		
	double	Height		
	double	Angle		
	double	CharProportion		

Polygon comprises of several points. Therefore, polygon type has vector of points. In schema, the field for points is called positions referring to the vertices of the polygon. Vertices are corner points, which polygon should have at least three. Another way to find the vertices is by checking the index each point would have. In this thesis, position will be enough for drawing polygon with SVG, but in the future development, indexes might be utilized. **Circle** can be draw with two information: circle's centre point and the radius. Centre point is the type Vec, the same that was defined for the point. Radius is a float type.

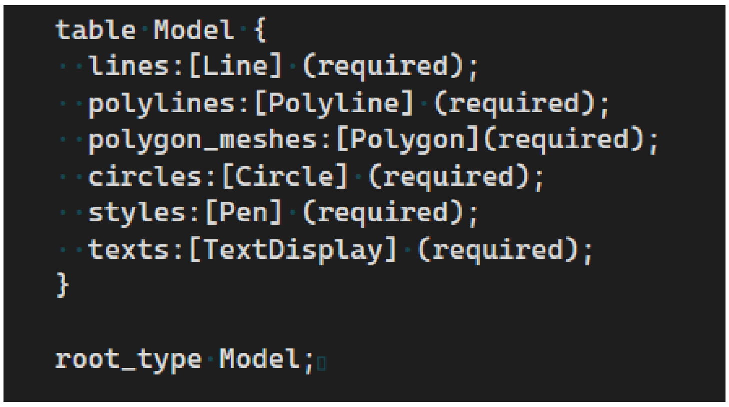
As seen in Table 5, the **pen** data type is included in line, polyline, polygon, and circle and named as a style. This is because each shape could have its own style. Pen is also included as its own table to the schema. The pen primitive contains three variables: LineColor, LineType and LineWidth. The type for the colour is integer (int). LineType has its own typedef that is unsigned short. In schema, unsigned short can be declared with ushort. LineWidth is unsigned char type. Pen has default values and thus does not need to be defined separately for every shape. When looping through the values in program, it would be inefficient to check shapes style with every line. Therefore, it should be noted that, in the program, there could be a function to check if the style has changed and only then update shape's style.

The **text** primitive introduced in Tekla Structures is included in the schema. The fields and their types are a *text,* which is a displayed text in string type, a *font*, which is a name of the font in string, a *position*, which is a Vec type point and tells where the text begins and in Tekla Structures, it is located on the bottom left corner, a *height*, which refers to the font size in pixels (px) in float type, and an *angle*, which is a degree or a radiant value given in float and is used to set the direction of the text in drawing. The topic of text will not be inspected further in this thesis, and it is one topic for further development. The reason is to do with the fonts and need to vectorize them, which is because fonts, that are defined company specifically and some operating system specific fonts, will not be compatible in web, which uses Google fonts. To answer this challenge and to use fonts on web that are not compatible, they need to be made into a vector graphics.

4.2 Writing and serializing dummy data with FlatBuffers

Writing and serializing is done in the project file of the chosen language. In this thesis, the file is TrimBim2D.cpp. Since the data from the Tekla Structure software is not used, the data is created in the same program where **Model** root table is serialized with FlatBuffers. The root table means that serialization of data starts from the table called Model. In order to serialize this, all the objects that are in the root table need to be serialized. Model consists of all the tables and structs that have been specified in schema (see Figure 4).

Figure 4. Model table in FlatBuffers Schema.

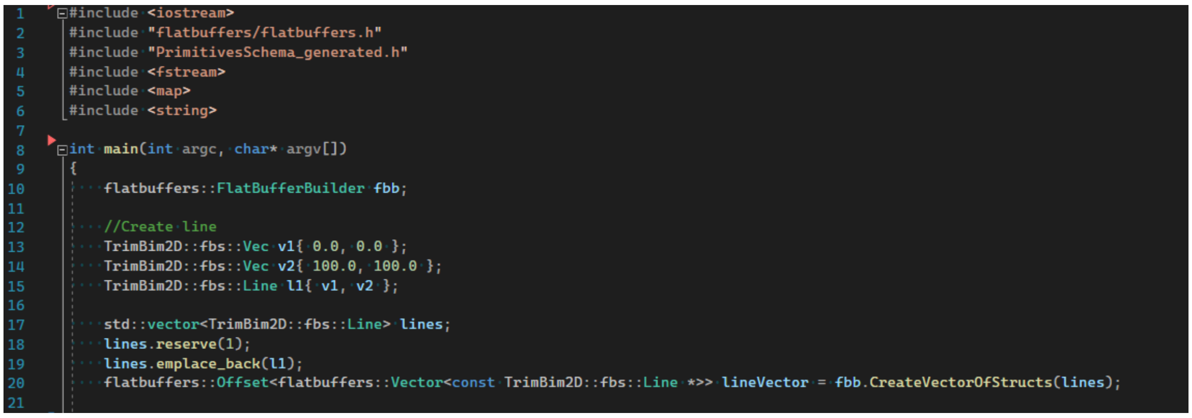


```
table Model {
  lines:[Line] (required);
  polylines:[Polyline] (required);
  polygon_meshes:[Polygon](required);
  circles:[Circle] (required);
  styles:[Pen] (required);
  texts:[TextDisplay] (required);
}

root_type Model;
```

In the scope of this thesis, data and shapes for line, polyline, polygon, circle, style, and text are created. This is done as means to verify that the data is passed to FlatBuffers and, further on, it will be verified whether these shapes can be visualized on the screen with this data.

In Figure 5, the lines 1–10 shows included headers, namespace and creating the instance of FlatBufferBuilder called fbb. To follow up, the line is created with a dummy data. This was done by giving the values to points in float type, meaning that the point has two values presenting x- and y-coordinate (see Figure 5 lines 13–14). Since line is a struct with two values type of Vec, the points defined before can be passed to the line. The line with two points values is then inserted into new vector that could contain more lines.

Figure 5. Start of FlatBuffers project file with headers and creating line.

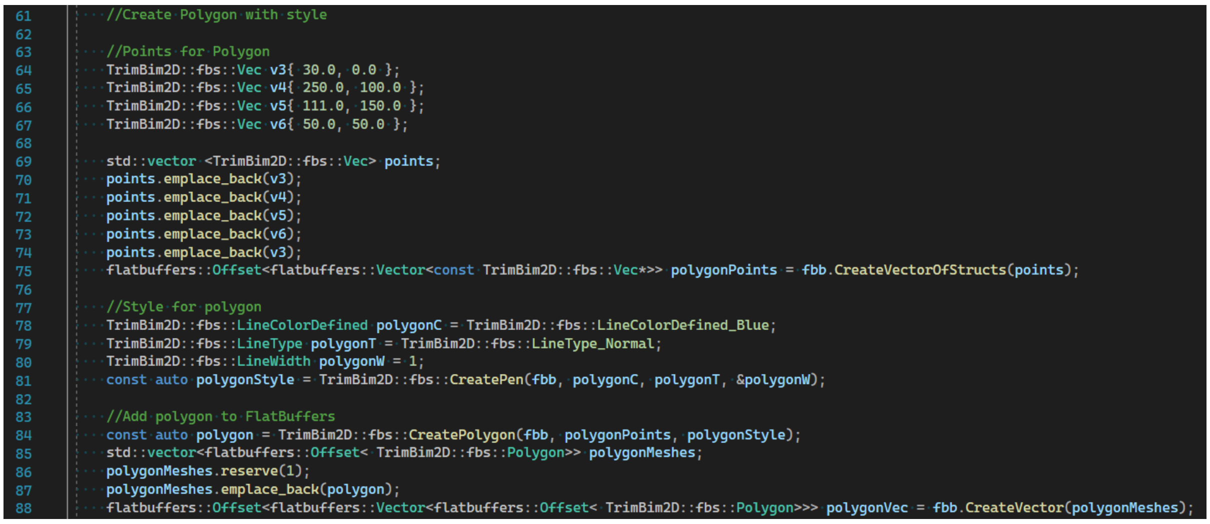


```
1  #include <iostream>
2  #include "flatbuffers/flatbuffers.h"
3  #include "PrimitivesSchema_generated.h"
4  #include <fstream>
5  #include <map>
6  #include <string>
7
8  int main(int argc, char* argv[])
9  {
10     flatbuffers::FlatBufferBuilder fbb;
11
12     //Create line
13     TrimBim2D::fbs::Vec v1{ 0.0, 0.0 };
14     TrimBim2D::fbs::Vec v2{ 100.0, 100.0 };
15     TrimBim2D::fbs::Line l1{ v1, v2 };
16
17     std::vector<TrimBim2D::fbs::Line> lines;
18     lines.reserve(1);
19     lines.emplace_back(l1);
20     flatbuffers::Offset<flatbuffers::Vector<const TrimBim2D::fbs::Line *>> lineVector = fbb.CreateVectorOfStructs(lines);
21
```

With “fbb.CreateVectorOfStructs(lines)”, the line object is collected into a temporary data structure. Objects return values are captured into lineVector, meaning that they are offsets into the serialized data, or, in other words, it is now possible to indicate to the values’ location. Later, when creating the Model with its fields, reference to line’s values can be made with lineVector.

When proceeding into creating a polygon, there is an example of more complex use of flatbuffers vectors and offsets (see Figure 6). When creating the polygon, four points are created with type of Vec. The last point of polygon is the same as the first one because it closes to the starting point. The last point does not have be specified separately here because, for example, in SVG, polygon reference automatically closes the polygon to the first point. After this, the vector with these points is created and they are passed onto flatbuffer::vector further on to the offset, same as with the line. In Figure 6 between lines 78–81, a style for this polygon is also created, and a colour, line type and line width are given. With CreatePen() method, the style is serialize and its offsets are collected into polygonStyle.

Figure 6. Creating Polygon in FlatBuffers project file.



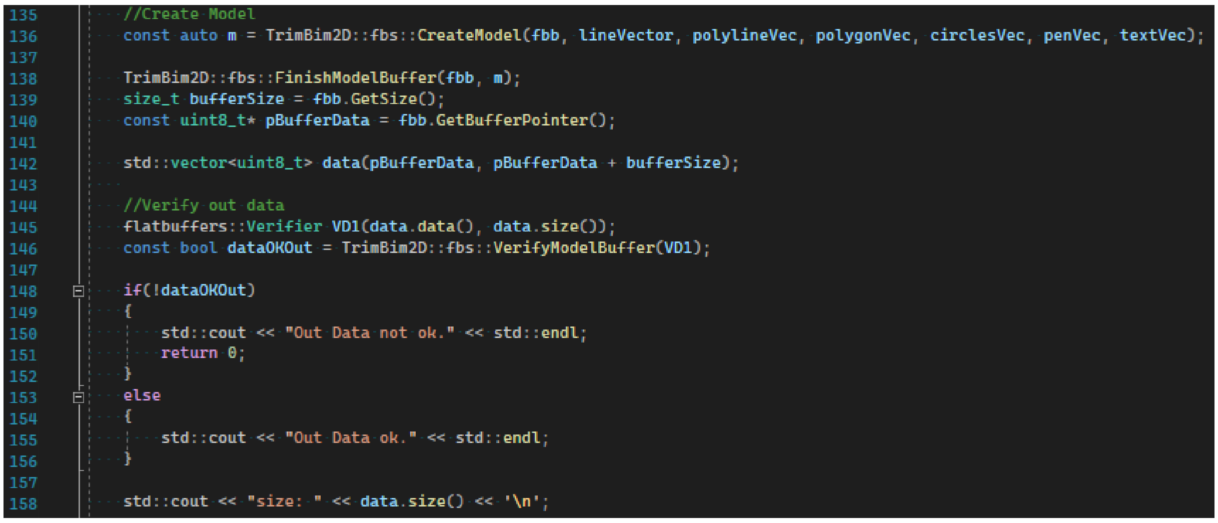
```
61      //Create Polygon with style
62
63      //Points for Polygon
64      TrimBim2D::fbs::Vec v3{ 30.0, 0.0 };
65      TrimBim2D::fbs::Vec v4{ 250.0, 100.0 };
66      TrimBim2D::fbs::Vec v5{ 111.0, 150.0 };
67      TrimBim2D::fbs::Vec v6{ 50.0, 50.0 };
68
69      std::vector <TrimBim2D::fbs::Vec> points;
70      points.emplace_back(v3);
71      points.emplace_back(v4);
72      points.emplace_back(v5);
73      points.emplace_back(v6);
74      points.emplace_back(v3);
75      flatbuffers::Offset<flatbuffers::Vector<const TrimBim2D::fbs::Vec*>> polygonPoints = fbb.CreateVectorOfStructs(points);
76
77      //Style for polygon
78      TrimBim2D::fbs::LineColorDefined polygonC = TrimBim2D::fbs::LineColorDefined_Blue;
79      TrimBim2D::fbs::LineType polygonT = TrimBim2D::fbs::LineType_Normal;
80      TrimBim2D::fbs::LineWidth polygonW = 1;
81      const auto polygonStyle = TrimBim2D::fbs::CreatePen(fbb, polygonC, polygonT, &polygonW);
82
83      //Add polygon to FlatBuffers
84      const auto polygon = TrimBim2D::fbs::CreatePolygon(fbb, polygonPoints, polygonStyle);
85      std::vector<flatbuffers::Offset< TrimBim2D::fbs::Polygon>> polygonMeshes;
86      polygonMeshes.reserve(1);
87      polygonMeshes.emplace_back(polygon);
88      flatbuffers::Offset<flatbuffers::Vector<flatbuffers::Offset< TrimBim2D::fbs::Polygon>>> polygonVec = fbb.CreateVector(polygonMeshes);
```

Between lines 84–88 in Figure 6, polygon with a style is created and added to the buffer instance, fbb. Since vector of points is created into polygonPoints, and its style into polygonStyle, a polygon can be created with method CreatePolygon() on line 84. This is where the serialization happens. Since Model collects a vector of polygons, a so-called nested vector of objects is created, where, in lines 85–85, an additional vector containing polygon offsets is created and, in line 88, collected into temporary data structure.

The same repeats with every primitive that are included in this thesis. To read how circle, pen and text was created in code, see Appendix 4. When comparing sizes of file, the creation of polyline is utilized by looping different number of lines and therefore, the creating of polyline is inspected in more detail in chapter 4.5.

Once every object that are used in Model are serialized, the root table is serialized as shown in Figure 7 on line 136. The root Model includes all the primitives that have been introduced in Figure 4 on page 31. To finalize the buffer, the FinishModelBuffer() method is called on line 138. Additionally, a verification for the data, where it checks that the data in buffer matches FlatBuffers, is added (see lines 145–158 in Figure 7).

Figure 7. Finishing serializing data to FlatBuffers.



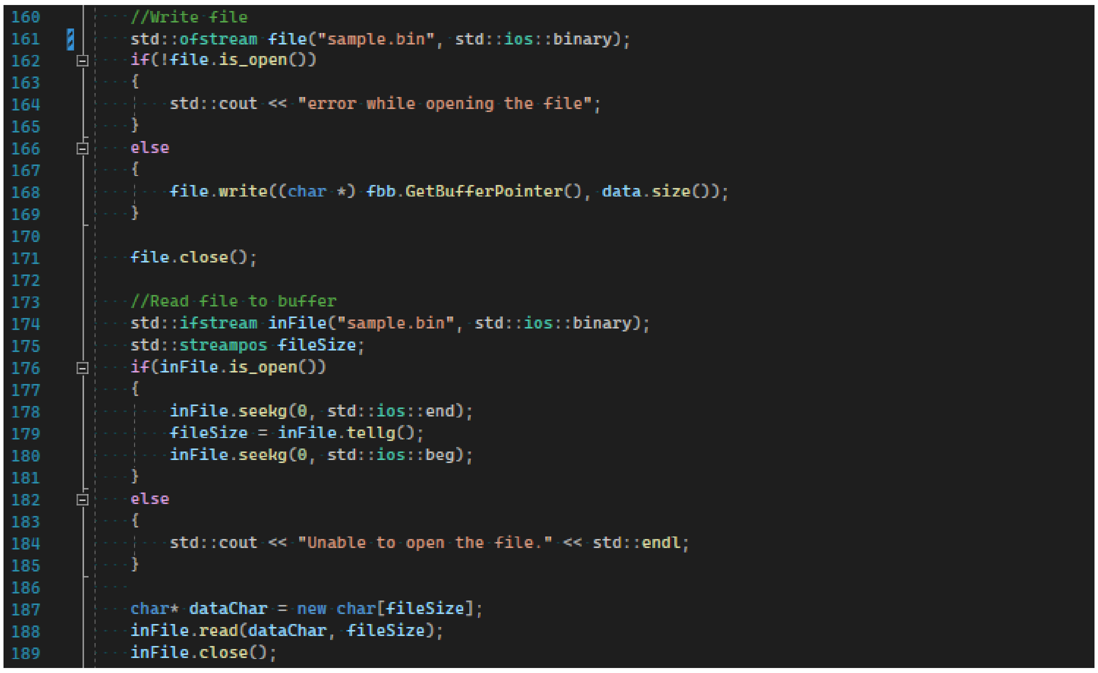
```
135     //Create Model
136     const auto m = TrimBim2D::fbs::CreateModel(fbb, lineVector, polylineVec, polygonVec, circlesVec, penVec, textVec);
137
138     TrimBim2D::fbs::FinishModelBuffer(fbb, m);
139     size_t bufferSize = fbb.GetSize();
140     const uint8_t* pBufferData = fbb.GetBufferPointer();
141
142     std::vector<uint8_t> data(pBufferData, pBufferData + bufferSize);
143
144     //Verify out data
145     flatbuffers::Verifier VD1(data.data(), data.size());
146     const bool dataOKOut = TrimBim2D::fbs::VerifyModelBuffer(VD1);
147
148     if(!dataOKOut)
149     {
150         std::cout << "Out Data not ok." << std::endl;
151         return 0;
152     }
153     else
154     {
155         std::cout << "Out Data ok." << std::endl;
156     }
157
158     std::cout << "size: " << data.size() << '\n';
```

4.3 Writing and reading files in C++

As part of the comparing the sizes of different file formats, a serialized data needs to be written into a binary file. As one goal is to see visually that the data can be used for drawing shapes, the data needs to be also read from this binary file.

To pass the buffer to a file, the buffer pointer and the size of the data are needed to the serialized data (see Figure 7 lines 139–140). When creating output file (see Figure 8 line 161), a name, “sample.bin” is given and the file is defined to be a binary file type, which is important. This is done with stream class ofstream, which can be used to write to the file. In Figure 8 on the line 168, the file is written by passing the FlatBuffers pointer and casted into char pointer (char*) and the size of the data, which is the number of bytes on the stream, that would be written to the file at a time. After writing, the file is closed.

Figure 8. Writing and reading binary file with C++.



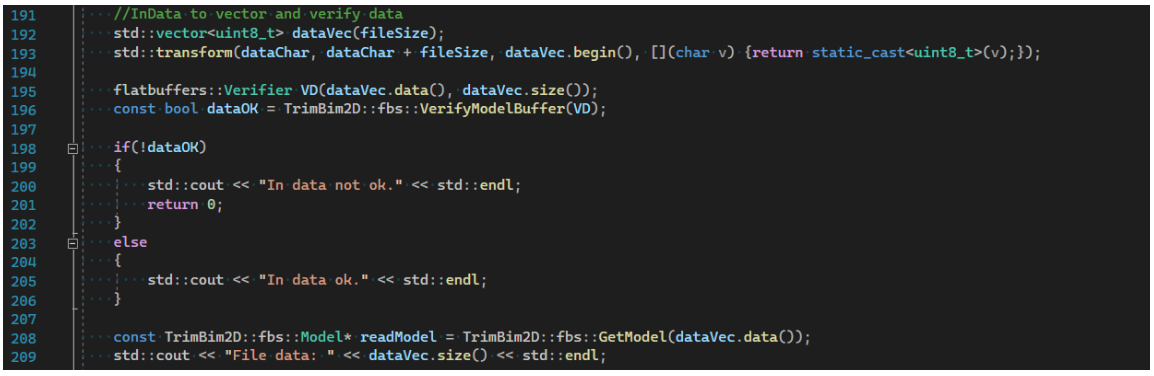
```
160     //Write file
161     std::ofstream file("sample.bin", std::ios::binary);
162     if(!file.is_open())
163     {
164         std::cout << "error while opening the file";
165     }
166     else
167     {
168         file.write((char *) fbb.GetBufferPointer(), data.size());
169     }
170
171     file.close();
172
173     //Read file to buffer
174     std::ifstream inFile("sample.bin", std::ios::binary);
175     std::streampos fileSize;
176     if(inFile.is_open())
177     {
178         inFile.seekg(0, std::ios::end);
179         fileSize = inFile.tellg();
180         inFile.seekg(0, std::ios::beg);
181     }
182     else
183     {
184         std::cout << "Unable to open the file." << std::endl;
185     }
186
187     char* dataChar = new char[fileSize];
188     inFile.read(dataChar, fileSize);
189     inFile.close();
```

When reading the binary file, the ifstream is for inputted data. When opening the file, it should be specified to open as a binary. For reading the file, the size of the data that would be read to the stream is needed. This is done in Figure 8 between lines 178–180. First, the position for the next inputted character is set from the beginning to the end of the stream with seekg(0, std::ios::end). Then, the position of the current character is obtained with tellg() method and it is passed into fileSize variable. This is now the size of the stream in which the data is stored. Finally, the position is set back to the beginning of the stream with seekg(0, std::ios::beg) method. On the lines 187–189 of the Figure 8, the fine is read. First, memory is allocated by the size for the file data. Then, inFile is read to that new memory location. When this is completed, the file is closed.

4.4 Reading and deserializing data with FlatBuffers in program

For reading the binary file, memory needs to be allocated for char data type. However, reading the same data with FlatBuffers requires that the data is in uint8_t type. Although, char and uint8_t should be interchangeable, it was required to transform data to uint8_t type vector (see Figure 9 line 193). To make sure, that data is not corrupted, it was verified with FlatBuffers VerifyModelBuffer() method.

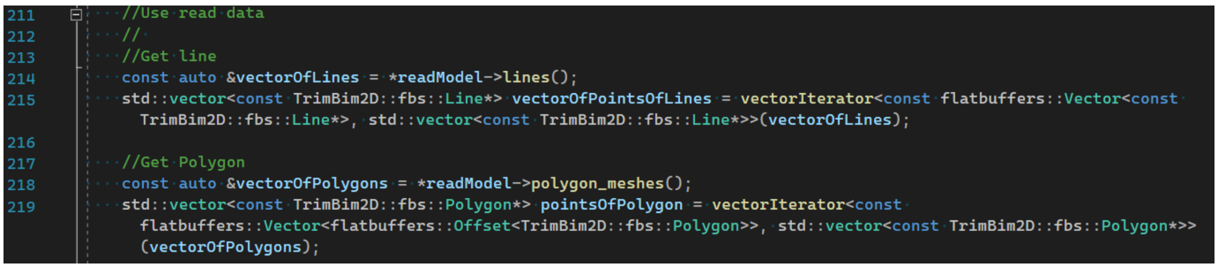
Figure 9. Verifying read data and passing it to buffer.



```
191     //InData to vector and verify data
192     std::vector<uint8_t> dataVec(fileSize);
193     std::transform(dataChar, dataChar + fileSize, dataVec.begin(), [](char v) {return static_cast<uint8_t>(v);});
194
195     flatbuffers::Verifier VD(dataVec.data(), dataVec.size());
196     const bool dataOK = TrimBim2D::fbs::VerifyModelBuffer(VD);
197
198     if(!dataOK)
199     {
200         std::cout << "In data not ok." << std::endl;
201         return 0;
202     }
203     else
204     {
205         std::cout << "In data ok." << std::endl;
206     }
207
208     const TrimBim2D::fbs::Model* readModel = TrimBim2D::fbs::GetModel(dataVec.data());
209     std::cout << "File data: " << dataVec.size() << std::endl;
```

After this, it is possible to deserialize the data with GetModel() method. Since the Model consists of vectors of tables and structs, they need to be looped through to obtain the value that can be used. In the Figure 10, it is demonstrated how the reading of the data for line and polygon is done. For reading other primitives: polyline, pen, circle, and text, refer to Appendix 5.

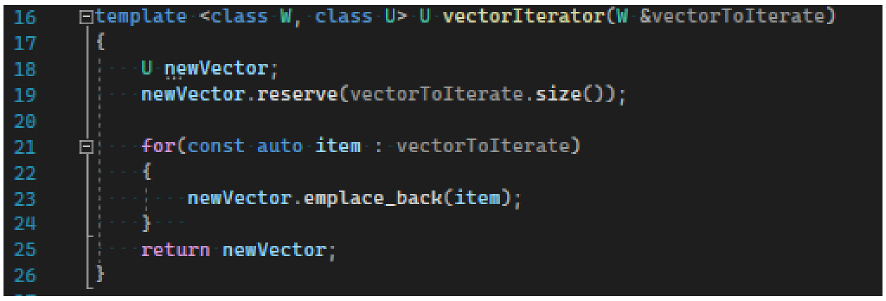
Figure 10. Accessing line and polygon data in FlatBuffers.



```
    //Use read data
    //
    //Get line
    const auto &vectorOfLines = *readModel->lines();
    std::vector<const TrimBim2D::fbs::Line*> vectorOfPointsOfLines = vectorIterator<const flatbuffers::Vector<const
      TrimBim2D::fbs::Line*>, std::vector<const TrimBim2D::fbs::Line*>>(vectorOfLines);

    //Get Polygon
    const auto &vectorOfPolygons = *readModel->polygon_meshes();
    std::vector<const TrimBim2D::fbs::Polygon*> pointsOfPolygon = vectorIterator<const
      flatbuffers::Vector<flatbuffers::Offset<TrimBim2D::fbs::Polygon>>, std::vector<const TrimBim2D::fbs::Polygon*>>
      (vectorOfPolygons);
```

For iterating through the FlatBuffers vectors, a template function, vectorIterator() is created, which enables to repeat the same function for different types of vectors (see Figure 11). When calling the vectorIterator() function (see Figure 10), the type for passed argument and for the return value need to be defined. For example with the line, a vectorOfLine with type const flatbuffers::Vector<const TrimBim2D::fbs::Line*> and return value of type std::vector<const TrimBim2D::fbs::Line*> are passed to the template. In this template function, a new vector with user-defined type is created, the passed vector argument is iterated through and its values are inserted to the new vector, and, lastly, the new vector is returned.

Figure 11. Template function for iterating through FlatBuffer vectors.



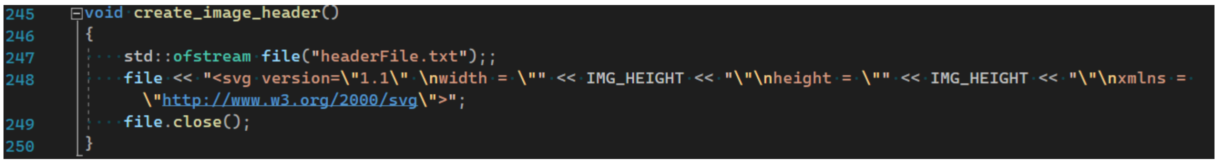
```cpp
template <class W, class U> U vectorIterator(W &vectorToIterate)
{
    U newVector;
    newVector.reserve(vectorToIterate.size());

    for(const auto item : vectorToIterate)
    {
        newVector.emplace_back(item);
    }
    return newVector;
}
```

4.5 Generating SVG file from deserialized data

Once able to access and read data from FlatBuffers, the SVG file can be made for two reasons: 1) to visualize data and validate that the created data in FlatBuffers can be used, 2) to have a web applicable vector graphics file for comparing the file size with binary file.

There is a separate function for writing SVG header (see Figure 12) and for the content of the SVG file (see Figure 13). The header part of the code would be staying the same. Adding and excluding the SVG file content would only happen in create_image_header() function. Both, content, and header, are written into their own text files; headerFile.txt and contentFile.txt.

Figure 12. Function for the beginning for SVG file.

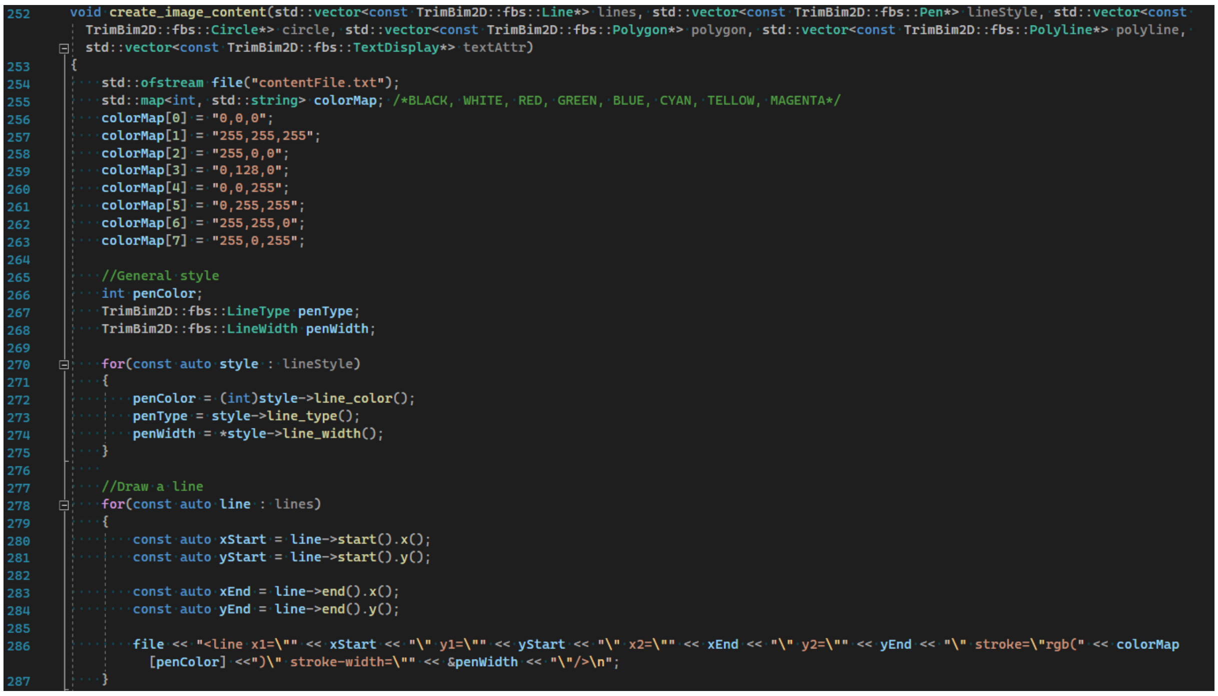


```cpp
void create_image_header()
{
    std::ofstream file("headerFile.txt");;
    file << "<svg version=\"1.1\" \nwidth = \"" << IMG_HEIGHT << "\"\nheight = \"" << IMG_HEIGHT << "\"\nxmlns =
      \"http://www.w3.org/2000/svg\">";
    file.close();
}
```

The SVG file is written between the root <svg></svg>. The macros are used to define the width and height of the canvas. The canvas is where the visualization is done, and if some part of the drawing is outside of it, that part will not be rendered to the screen. In addition, the header includes xmlns, which binds the correct namespace for SVG.

The actual content of the drawing is defined in create_image_content() function. The pointers to the objects in the FlatBuffers are passed to this function where their values are taken for new variables used in SVG file. In Figure 13, there is an example of how a line is written into the SVG file. At the beginning of the function, a map for colour definitions is created. Tekla Structures uses its own definitions for colours, but with SVG, the RGB definition is used. When reading the colour from FlatBuffers, the char value is converted into an int (see line 272 in Figure 13), which works as a key for the map's values. Values in the FlatBuffers are accessed by looping through the vectors (see lines 279–284).

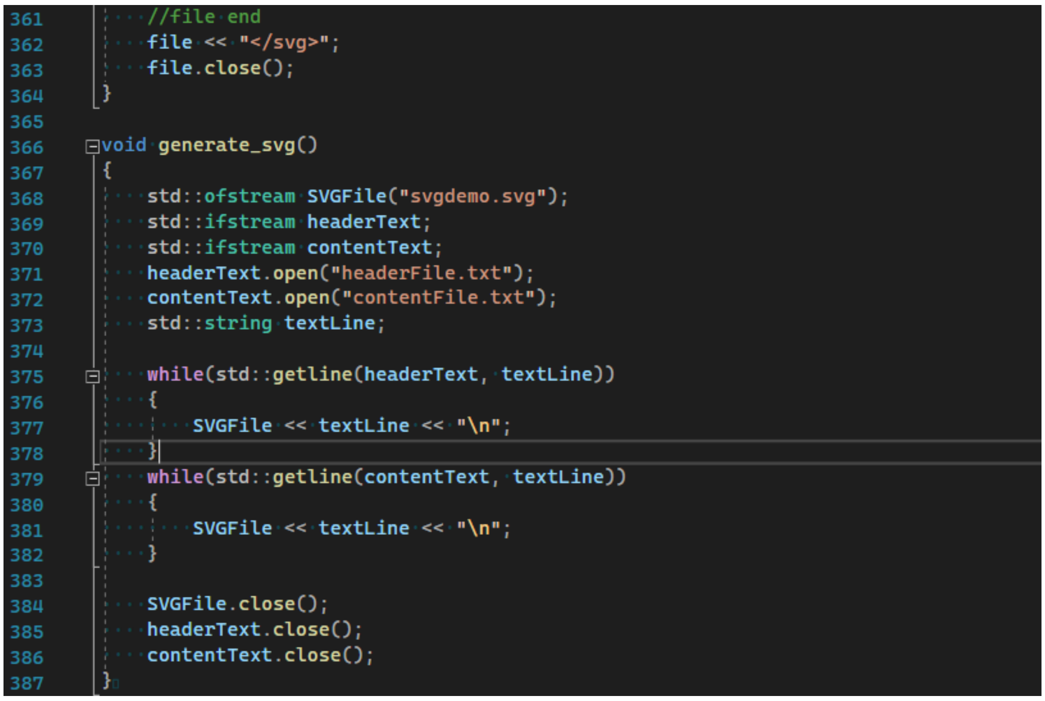
Figure 13. Write line content for SVG file.



```
252     void create_image_content(std::vector<const TrimBim2D::fbs::Line*> lines, std::vector<const TrimBim2D::fbs::Pen*> lineStyle, std::vector<const
          TrimBim2D::fbs::Circle*> circle, std::vector<const TrimBim2D::fbs::Polygon*> polygon, std::vector<const TrimBim2D::fbs::Polyline*> polyline,
          std::vector<const TrimBim2D::fbs::TextDisplay*> textAttr)
253     {
254         std::ofstream file("contentFile.txt");
255         std::map<int, std::string> colorMap; /*BLACK, WHITE, RED, GREEN, BLUE, CYAN, TELLOW, MAGENTA*/
256         colorMap[0] = "0,0,0";
257         colorMap[1] = "255,255,255";
258         colorMap[2] = "255,0,0";
259         colorMap[3] = "0,128,0";
260         colorMap[4] = "0,0,255";
261         colorMap[5] = "0,255,255";
262         colorMap[6] = "255,255,0";
263         colorMap[7] = "255,0,255";
264
265         //General style
266         int penColor;
267         TrimBim2D::fbs::LineType penType;
268         TrimBim2D::fbs::LineWidth penWidth;
269
270         for(const auto style : lineStyle)
271         {
272             penColor = (int)style->line_color();
273             penType = style->line_type();
274             penWidth = *style->line_width();
275         }
276
277         //Draw a line
278         for(const auto line : lines)
279         {
280             const auto xStart = line->start().x();
281             const auto yStart = line->start().y();
282
283             const auto xEnd = line->end().x();
284             const auto yEnd = line->end().y();
285
286             file << "<line x1=\"" << xStart << "\" y1=\"" << yStart << "\" x2=\"" << xEnd << "\" y2=\"" << yEnd << "\" stroke=\"rgb(" << colorMap
                  [penColor] <<")\" stroke-width=\"" << &penWidth << "\"/>\n";
287         }
```

To start the SVG command, the element reference to the line is added (see line 286 in Figure 13), and the start and the end points with x and y values, as well as a style for the line (colour and width) are specified. The SVG file syntax is closed with </svg> and the file, where the text is passed into, is closed with close() (see lines 362–363 in Figure 14) . To read how polygon, polyline, circle, and text are written into SVG file, see Appendix 6.

Figure 14. Function to generate, or write, SVG file.



```
361     //file end
362     file << "</svg>";
363     file.close();
364 }
365 
366 void generate_svg()
367 {
368     std::ofstream SVGFile("svgdemo.svg");
369     std::ifstream headerText;
370     std::ifstream contentText;
371     headerText.open("headerFile.txt");
372     contentText.open("contentFile.txt");
373     std::string textLine;
374 
375     while(std::getline(headerText, textLine))
376     {
377         SVGFile << textLine << "\n";
378     }
379     while(std::getline(contentText, textLine))
380     {
381         SVGFile << textLine << "\n";
382     }
383 
384     SVGFile.close();
385     headerText.close();
386     contentText.close();
387 }
```

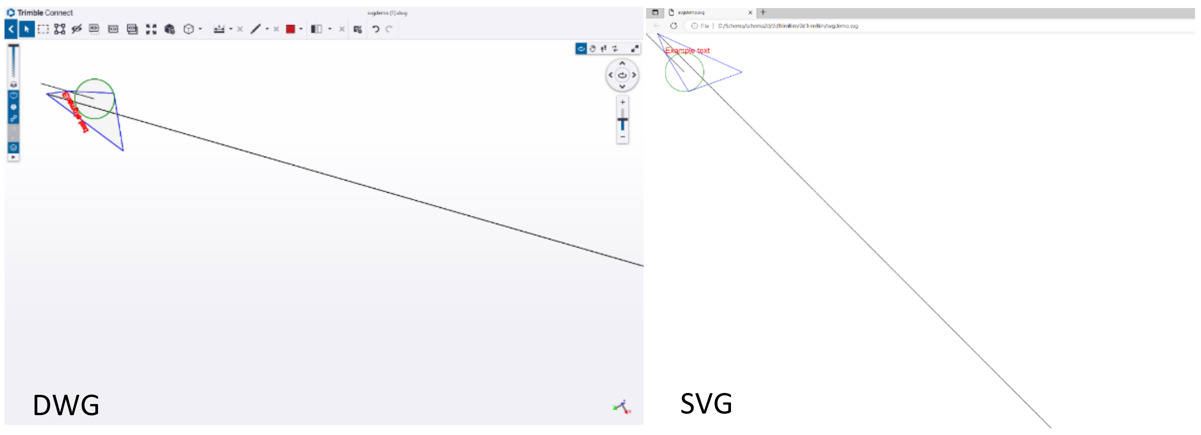
In addition to these functions, there is a final function, where both header and context text file are put together and written into a file with SVG extension (see Figure 14). First, the newly created text files are opened and new SVG file created. Then, the text files are read line by line with std::getline() method (see lines 375–382 in Figure 14) and written into the SVG file. In the end, all the files are closed.

4.6 Comparing file sizes of SVG, DWG and new binary file format

In order to compare different file formats, the binary file with serialized data from FlatBuffers and SVG file with the same data have been created so far. The thesis focus for comparing is between the binary file, and SVG file but the DWG file was decided to be introduces alongside them because DWG is currently used in Trimble Connect for cooperation through cloud. Therefore, it is interesting to see how DWG compares to SVG and to the new binary files. To get DWG file with the same data, the SVG file is converted to DWG file with a converter tool found online at www.online.reaconverter.com.

However, the DWG cannot be the focus of the comparison, because with the converters online, it is possible that some data might be lost in the process and therefore, the file size is only approximate. However, when comparting the visualization of SVG on web and DWG on Trimble Connect, it can be seen that most of the data is preserved (see Figure 15).

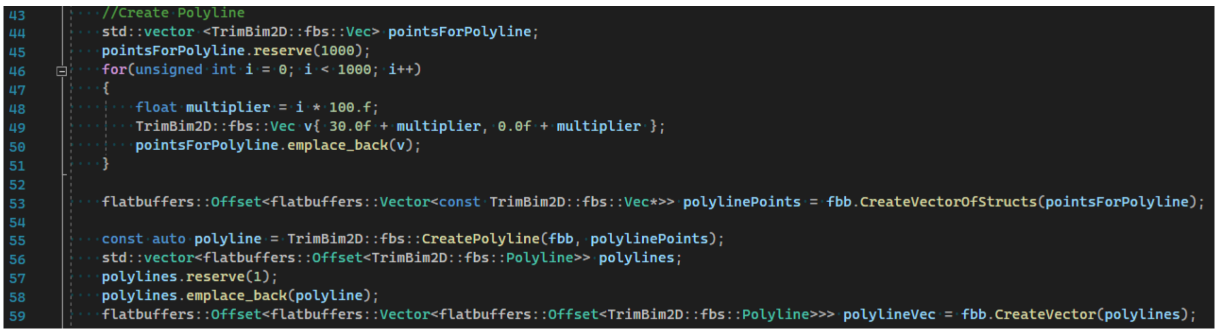
Figure 15. Comparing visualization between DWG and SVG.



Comparing the visualization in SVG and DWG format, an example of how the application specific definition for origin (see chapter 15) might affect visualization can be seen clearly. On SVG, the origin is at the top left corner and in Trimble Connect the origin is at the bottom left corner. This is especially apparent when examining the starting point of the short line. The consequence is that the shapes are interpreted with different coordinates and therefore, shapes appear to be rotated in DWG compared to original SVG image.

To obtain data for comparing different file size, a loop generating as many points as is the loop's range, is created for polyline. For example, in Figure 16 between lines 46–51, the loop range is 1000. Inside the loop, there is a multiplier that assists to obtain new values for the point. After that, the point values are given their value and added to the vector. In the visualization (see Figure 16), this loop can be seen as a long line that in most cases would be partially out of screen. When reading the data and bringing it to SVG, the vector iteration is done with the points, and they are written to the file (see Appendix 5).

Figure 16. Loop for creating points for polyline and writing polyline to FlatBuffers.



```cpp
//Create Polyline
std::vector <TrimBim2D::fbs::Vec> pointsForPolyline;
pointsForPolyline.reserve(1000);
for(unsigned int i = 0; i < 1000; i++)
{
    float multiplier = i * 100.f;
    TrimBim2D::fbs::Vec v{ 30.0f + multiplier, 0.0f + multiplier };
    pointsForPolyline.emplace_back(v);
}

flatbuffers::Offset<flatbuffers::Vector<const TrimBim2D::fbs::Vec*>> polylinePoints = fbb.CreateVectorOfStructs(pointsForPolyline);

const auto polyline = TrimBim2D::fbs::CreatePolyline(fbb, polylinePoints);
std::vector<flatbuffers::Offset<TrimBim2D::fbs::Polyline>> polylines;
polylines.reserve(1);
polylines.emplace_back(polyline);
flatbuffers::Offset<flatbuffers::Vector<flatbuffers::Offset<TrimBim2D::fbs::Polyline>>> polylineVec = fbb.CreateVector(polylines);
```

The comparison was made with 100, 1,000, 10,000 and 100,000 points or loops. In Table 6 on page 42, the results are gathered. The size that is compared, is the data size and not the size on disk. In addition, the size of compressed file done with zip is included.

When comparing to DWG file, it was not possible to reach satisfying results. With the online converter, regardless of the data size, the file size stays the same. There are too many unsettled questions regarding how the converter works and therefore, any conclusions about DWG cannot be made.

The results on Table 6 shows that between binary file and SVG, the binary file is in all of the cases smaller in size. However, the difference decreases with the smaller data size. This supports the theory in chapter 2.3.2, where it was indicated that the benefit of binary files is more apparent when the data is complex and the size of the data large. With smaller data, binary serialization might be excessive since the result with human-readable format is practically the same and as compressed slightly smaller than binary file.

Since Tekla Structure, and CAD programs in general, uses complex data and the data sizes are in abundance, binary format would be recommended since its size could be less than half of the SVG file size.

Table 6. Comparing FlatBuffers binary file, SVG and DWG file sizes.

Number of loops and overall data size	**FlatBuffers binary file size**	**SVG file size**	**DWG file size**
100 loops	1.16 kB (1,192 bytes)	1.56 kB	26.3 kB (27,015 bytes)
	Compressed: 830 bytes	Compressed: 789 bytes	Compressed: 21.1 KB (21,620 bytes)
1,000 loops	8.19 kB (8,392 bytes)	12.1 kB (12,407 bytes)	26.3 kB (26,951 bytes)
	Compressed: 3.92 kB (4,016 bytes)	Compressed: 4.50 kB (4,611 bytes)	Compressed: 21.1 kB (21,620 bytes)
10,000 loops	78.5 kB (80,392 bytes)	135 kB (138,407 bytes)	26.3 kB 26,948 bytes
	Compressed: 37.8 KB (38,750 bytes)	Compressed: 42.6 kB (43,660 bytes)	Compressed: 21.1 kB (21,620 bytes)
100,000 loops	781 kB (800,392 bytes)	2,09 MB (2,198,399 bytes)	Too large to convert online
	Compressed: 355 kB (363,815 bytes)	Compressed: 458 kB (469,026 bytes)	

5 Conclusions

In the scope of this thesis, the FlatBuffers schema, the FlatBuffers project file to create data and serialized them with FlatBuffers, as well as the serialized data into binary file format are made. Moreover, for the comparison section of this thesis, the SVG file from the deserialized data of binary file was created and SVG file converted into DWG file.

The value from this project comes from the schema that can be utilized and used as a basis for high-level schema and data serialization of the 2D drawings in Tekla Structures. Furthermore, by comparing file formats is can be concluded that the hypothesis “the binary file with FlatBuffers is smaller in size than SVG” is correct. However, contrary to the original hypotheses that also included the DWG, the conclusion how this new binary file format compares to DWG was not made due to insufficient conversion with online converter tool.

The schema written in this thesis is low-level and would need fine tuning, adding, and improving, for it to be implemented into Tekla Structures. For example, there is a question about how to handle fonts in the text, as well as the arcs and some other primitives are not yet included in the schema. However, with the visualization of the data with SVG file, it can be confirmed that the current schema is applicable and could be developed further.

The file format, that was produced in as the result, is a publishing format. Other topics for further development could then be adding interactivity and editing possibilities. This would mean in schema that the objects should have, for example, id so that the additional data or changes can be allocated to object specifically. Moreover, in this thesis topics, such as annotation, including dimensioning; measuring horizontal and vertical distance (Gharge, 2021), that are essential for designers and add information to the drawing that cannot be seen in 2D space, are not covered.

All this points to the fact that in order to implement this to CAD 2D drawing, there would be still some topics to consider and work to do. However, as a result of this thesis, there is now an enabler, a base to start further development, and analysis of file size that can contribute into making decision whether continue this development at some point.

Sources

Aalto (n.d.). The Difference between Vector and Raster Graphics. *Guide for Digital Design.* https://digitaldesign.aalto.fi/digital-design-workflows/the-difference-between-vector-and-raster-graphics/

Adobe (n.d.). *DWG vs. DXF*. Comparison. https://www.adobe.com/creativecloud/file-types/image/comparison/dwg-vs-dxf.html

Acaddrafting (26.5.2016). The Magnificence of 2D Technical Drawing. *The engineering design.* https://www.theengineeringdesign.com/the-magnificence-of-2d-technical-drawing/

Akenine-Möller, T., Haines, E. & Hoffman, N. (2008). *Real-Time Rendering*. A K Peters/CRC Press.

Amadine (12.2022). *What Are Scalable Vector Graphics*. https://amadine.com/useful-articles/what-are-scalable-vector-graphics-svg

Andy (8.12.2019). Which Vector File Type Should I Choose?. *Scan2CAD*. https://www.scan2cad.com/blog/tips/vector-file-type-choose/

BlueEnt (8.4.2022). PDF to CAD Conversion: Common Challenges With Their Solutions. *Construction Drawings.* https://www.bluentcad.com/blog/pdf-to-cad-conversion-challenges/

Campbell, J. & Shin, M. (2011). *Essentials of Geographic Information Systems.* From publisher https://www.saylor.org/books/.

Chen, J. & Clarke, K. (2017). *Modeling Standards and File Formats for Indoor Mapping*. [Conference article] 3rd International Conference on Geographical Information Systems Theory, Applications and Management, Santa Barbara, CA, United States.

Codecademy (n.d.). *Data Types*. C++. https://www.codecademy.com/resources/docs/cpp/data-types cplusplus.com, (n.d.). *<vector>.* Containers. https://cplusplus.com/reference/vector/vector/

Czmocha, I. & Pekala, A. (2014). Traditional Design versus BIM Based Design. *Procedia, 91,* 210–215.

Devopedia (2020). *Data Serialization.* https://devopedia.org/data-serialization

Dot Net Tutorials (n.d. -a). *Why Data Types in C++.* https://dotnettutorials.net/lesson/why-data-types-in-cpp/

Dot Net Tutorials (n.d. -b). *Primitive Data Types in C++*. https://dotnettutorials.net/lesson/primitive-data-types-in-cpp/

Eisenberg, D. & Bellamy-Royds, A. (2014). *SVG Essentials, 2nd Edition*. O'Reilly Media, Inc.

English, T.(2020). *Why Engineers Still Create 2D Detailed Drawings*. https://interestingengineering.com/innovation/why-engineers-still-create-2d-detailed-drawings

Eynon, J. (2016). *Construction Manager's BIM Handbook*. John Wiley & Sons, Inc.: Hoboken, NJ, USA.

Fahiem, M. & Farhan, S. (2007, December 29–31). *Representation of Engineering Drawings in SVG and DXF for Information Interchange*. [Conference article] 6th WSEAS International Conference on Circuits, Systems, Slectronics, Sontrol & Signal processing, Cairo, Egypt.

FileFormat (n.d.). *What is a DWG file*. CAD. https://docs.fileformat.com/cad/dwg/

FileInfo.com (21.12.2011). *What is the difference between binary and text files?*. Help Center. https://fileinfo.com/help/binary_vs_text_files

FlatBuffers (n.d. -a). *FlatBuffer Internals*. https://google.github.io/flatbuffers/md__internals.html

FlatBuffers (n.d. -b). *FlatBuffers white paper*. https://google.github.io/flatbuffers/flatbuffers_white_paper.html

FlatBuffers (n.d. -c). *Overview*. FlatBuffers. https://google.github.io/flatbuffers/

FlatBuffers (n.d. -d). *FlatBuffer C++ Benchmarks*. https://google.github.io/flatbuffers/flatbuffers_benchmarks.html

FlatBuffers (n.d. -e). *Writing a schema*. Programmer’s Guide. https://google.github.io/flatbuffers/flatbuffers_guide_writing_schema.html

FlatBuffers (n.d. -f). *Building*. Programmer’s Guide. https://google.github.io/flatbuffers/flatbuffers_guide_building.html

FlatBuffers (n.d. -g). *Using the schema compiler*. Programmer’s Guide. https://google.github.io/flatbuffers/flatbuffers_guide_using_schema_compiler.html

FlatBuffers (n.d. -h). *Tutorial*. Programmer’s Guide. https://google.github.io/flatbuffers/flatbuffers_guide_tutorial.html

Ganovelli, F., Corsini, M., Pattanaik, S. & Di Benedetto, M. (2014). *Introduction to Computer Graphics*. Chapman and Hall/CRC.

GeeksForGeeks (18. 3. 2023 -a). C++ Data Types. *Geeks for geeks.* https://www.geeksforgeeks.org/cpp-data-types/

GeeksForGeeks (20. 3. 2023 -b). Vector in C++ STL. *Geeks for geeks.* https://www.geeksforgeeks.org/vector-in-cpp-stl/

Gharge, P. (29.11.2021). Dimensions in AutoCAD. All You Need to Know. *All3DP*. https://all3dp.com/2/dimension-autocad-command-explained/

Grabowski, R. (2022). *AutoCAD For Dummies*. For Dummies.

Hazelcast (n.d.). *What Is Serialization?*. https://hazelcast.com/glossary/serialization/

Itskawal2000, 123mangooo & jatinsharma_0987 (19.2.2023). Difference Between C++ Text File and Binary File. *Geeks for Geeks.* https://geeksforgeeks.org/difference-between-cpp-text-file-and-binary-file/

Kjell, B. (n.d). *Introduction to Computer Science using Java*. Central Connecticut State University.

Kleppmann, M. (2017). *Designing Data-Intensive Applications. The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly Media.

Logothetis, S., Valari, E., Karachaliou, E. & Stylianidis, E. (2018). Development of an Open Source Spatial DBMS for a FOSS BIM. From publication Shu-Kun Lin (pub.), *Latest Developments in Reality-Based 3D Surveying and Modelling.* (pp. 326–347). MDPI.

Maciej, S., Andrzej, S. & Przemyslaw, B. (2017). *BIM in General Construction.* Lublin University of Technology.

Mahamuni, A. (16.3.2022). *Modifiers in C++*. https://www.scaler.com/topics/cpp/modifiers-in-cpp/

MDN (n.d. -a*). Adding vector graphics to the web*. https://developer.mozilla.org/en-US/docs/Learn/HTML/Multimedia_and_embedding/Adding_vector_graphics_to_the_Web

MDN (n.d. -b). *Positions*. SVG: Scalable Vector Graphics. https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Positions

MDN (n.d. -c). *Canvas API*. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

MDN (n.d. -d). *Introduction*. SVG: Scalable Vector Graphics.
https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Introduction

MDN (n.d. -e). *Adding vector graphics to the web*. Multimedia and embedding.
https://developer.mozilla.org/en-US/docs/Learn/HTML/Multimedia_and_embedding/Adding_vector_graphics_to_the_Web

MDN (n.d. -f). *SVG: Scalable Vector Graphics*. https://developer.mozilla.org/en-US/docs/Web/SVG

Microsoft (11.7.2022). *C++ type system*. C++ language reference. Basic concepts.
https://learn.microsoft.com/en-us/cpp/cpp/cpp-type-system-modern-cpp?view=msvc-170

Microsoft (3.8.2021). *Additional include directories*. https://learn.microsoft.com/en-us/cpp/build/reference/i-additional-include-directories?view=msvc-170

MongoDB (n.d.). *An Introduction to Data Persistence*. https://www.mongodb.com/databases/data-persistence

Nanyang Technological University (2013). *Stream IO and File IO*. C++ Programming Language. https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp10_io.html

Rouse, M.(31.10.2012). Binary File Transfer. *Technopedia.* https://www.techopedia.com/definition/515/binary-file-transfer-bft

Sheldon, R. (06.2022). *Binary file*. Programming. https://www.techtarget.com/whatis/definition/binary-file

Snowbound (2015). Scalable Vector Graphics. The Little-Known Treasure of Document Viewing[Brochure].https://snowbound.com/app/uploads/2022/07/eBook_Scalable_Vector_Graphics.pdf

SVG on the Web (n.d.). Introduction. *SVG on the Web*. https://svgontheweb.com/

Trimble (n.d. -a). *Industries*. Solutions. https://www.trimble.com/en/solutions/industries/overview

Trimble (n.d. -b). *Tekla Structures*. https://www.tekla.com/products/tekla-structures

Trimble (n.d. -c). *Create Model Views*. https://support.tekla.com/doc/tekla-structures/2023/mod_creating_a_view

Trimble (n.d. -d). *Drawing Types*. https://support.tekla.com/doc/tekla-structures/2023/dra_drawing_types

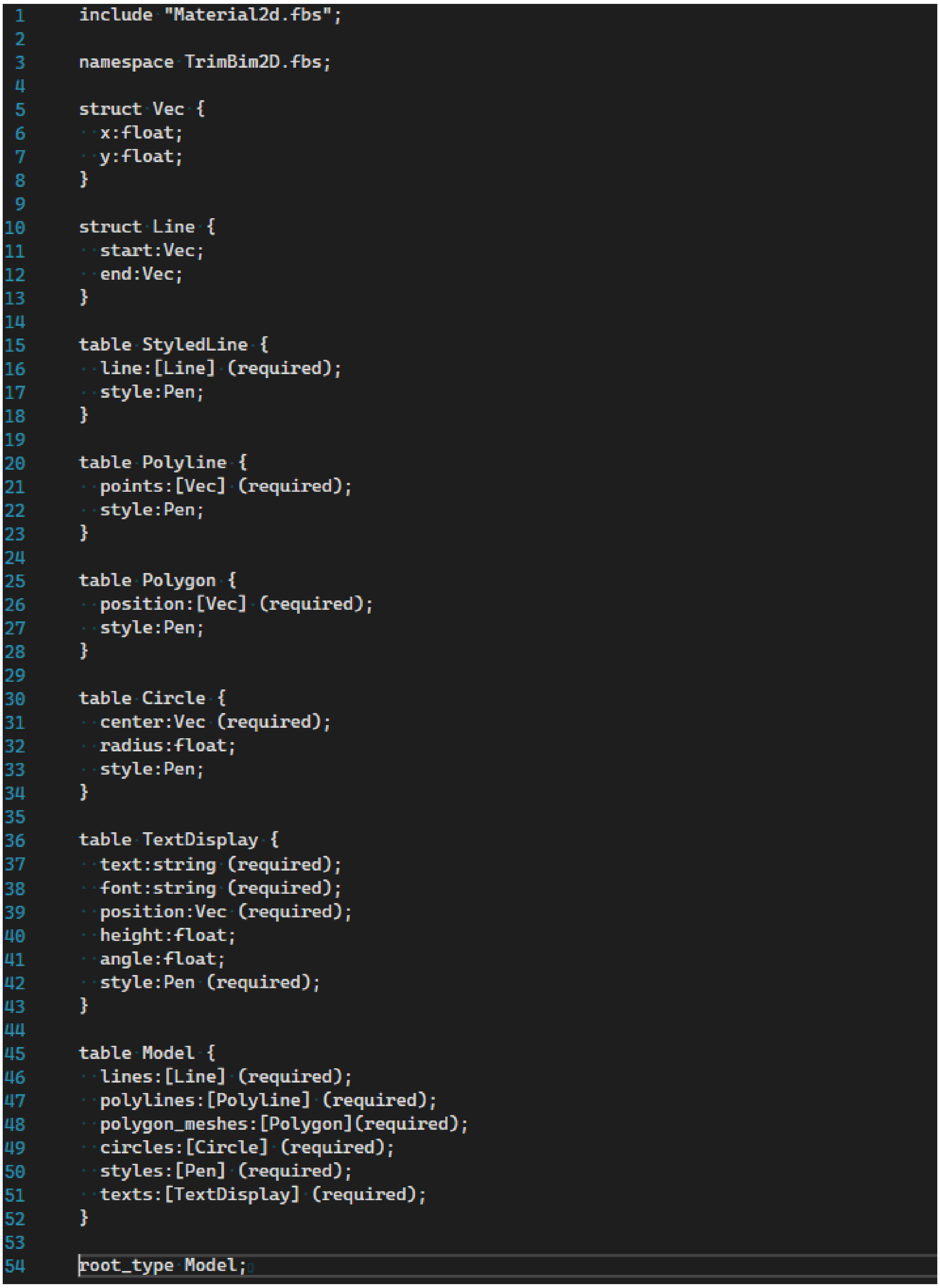
Trimble (2013). *Tekla History*. https://www.slideshare.net/Tekla/tekla-history

Zaks, M. (2.11.2016). Beyond JSON — Introduction to FlatBuffers. *Medium.* https://mzaks.medium.com/beyond-json-introduction-to-flatbuffers-fba1dfd0dcfe

Appendix 1: Table of C++ data types, their sizes in bytes and their range (Windows OS). (GeeksForGeeks, 2023 -a; Codecademy, n.d.)

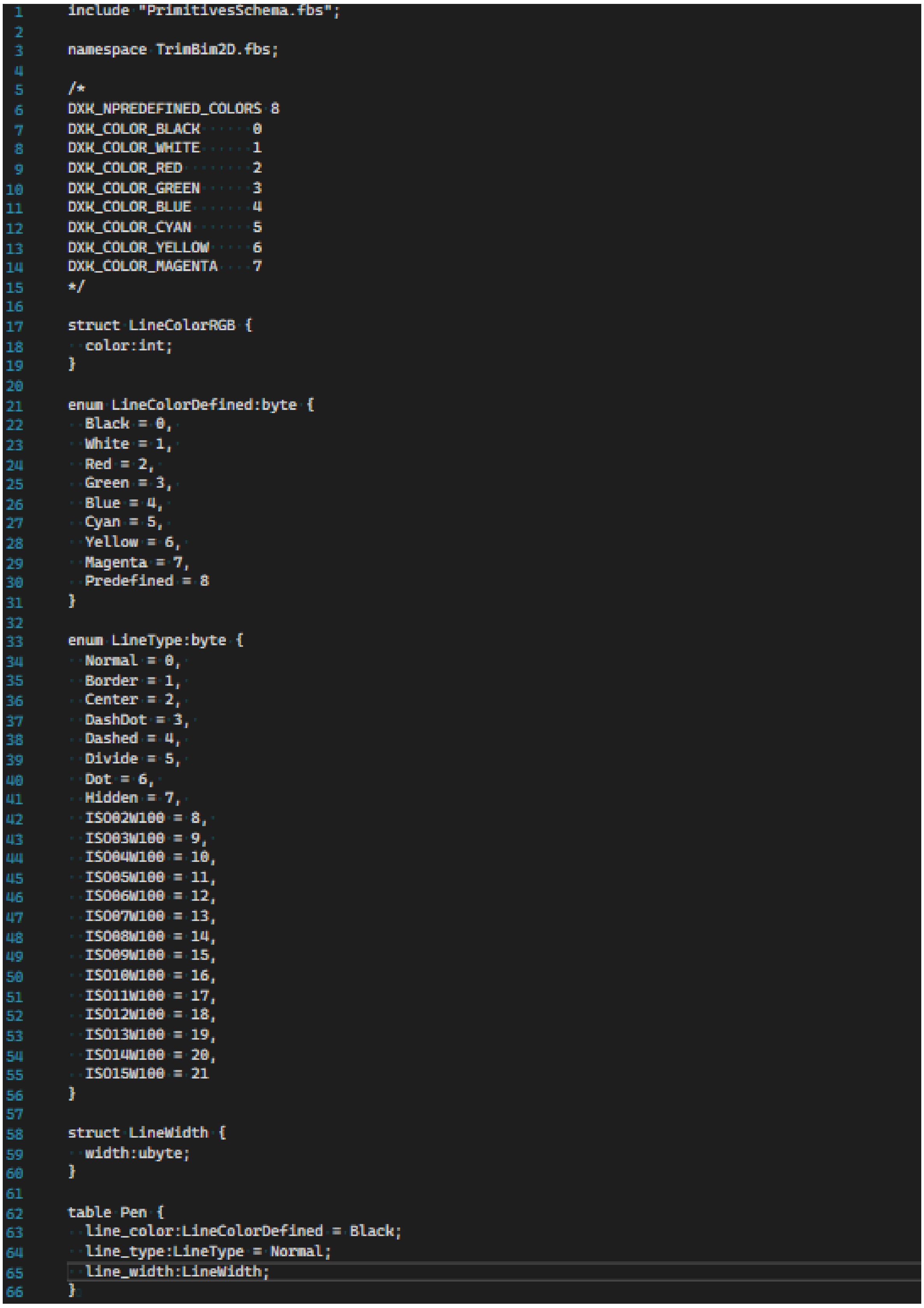
Data type		Memory Size	Range
int	short	2 bytes	-32,768 to 32,767
	unsigned short	2 bytes	0 to 65,535
	unsigned	4 bytes	0 to 4,294,967,295
	(default)	4 bytes	-2147483648 to 2147483647
	long	4 bytes	-2,147,483,648 to 2,147,483,647
	unsigned long	4 bytes	0 to 4,294,967,295
	long long	8 bytes	-(2^63) to (2^63)-1
float		4 bytes	-3.4x10^38 to 3.4x10^38
double	(default)	8 bytes	-2.3E-308 ×10^308 to1.7×10^308
	long	12 bytes	3.4E-4932 to 1.1E+4932
bool		1 byte	True or false
char	(default)	1 byte	-128 to 127
	signed	1 byte	-128 to 127
	unsigned	1 byte	0 to 255
std::string (MSVC compiler)		24 bytes	

Appendix 2: PrimitivesSchema.fbs, the FlatBuffers Schema for point, line, polyline, polygon, circle, text and model



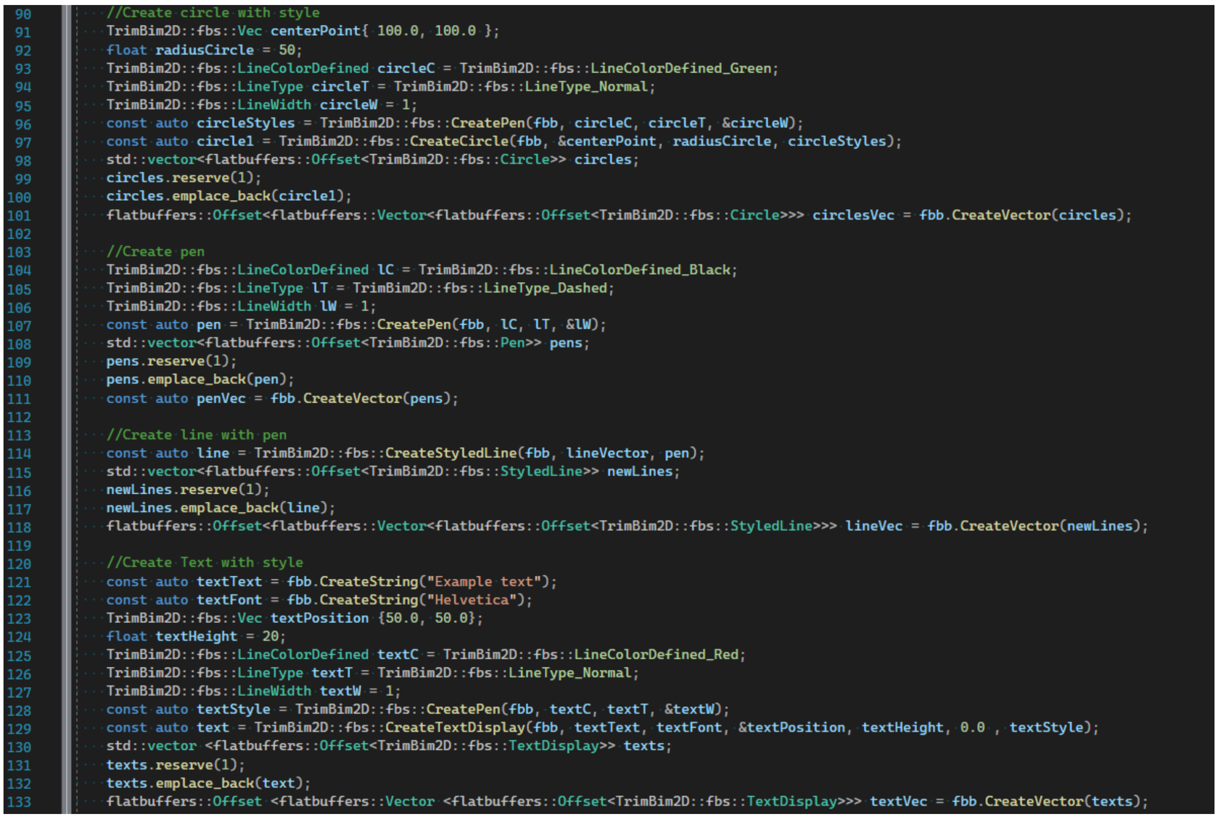
```
include "Material2d.fbs";

namespace TrimBim2D.fbs;

struct Vec {
  x:float;
  y:float;
}

struct Line {
  start:Vec;
  end:Vec;
}

table StyledLine {
  line:[Line] (required);
  style:Pen;
}

table Polyline {
  points:[Vec] (required);
  style:Pen;
}

table Polygon {
  position:[Vec] (required);
  style:Pen;
}

table Circle {
  center:Vec (required);
  radius:float;
  style:Pen;
}

table TextDisplay {
  text:string (required);
  font:string (required);
  position:Vec (required);
  height:float;
  angle:float;
  style:Pen (required);
}

table Model {
  lines:[Line] (required);
  polylines:[Polyline] (required);
  polygon_meshes:[Polygon](required);
  circles:[Circle] (required);
  styles:[Pen] (required);
  texts:[TextDisplay] (required);
}

root_type Model;
```

Appendix 3: Material2d.fbs, the FlatBuffers Schema for pen with colour, line type and line width



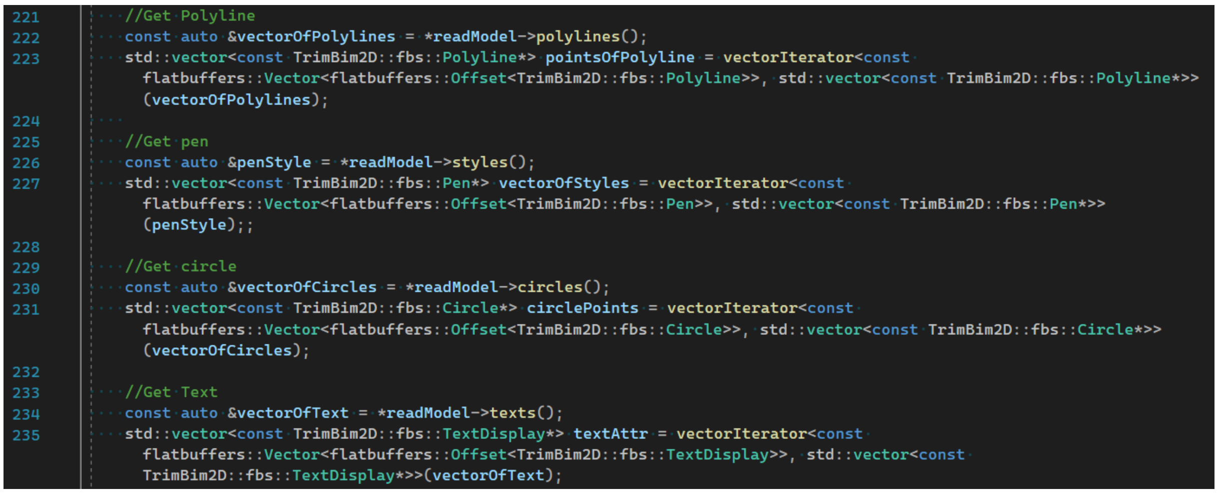
```
include "PrimitivesSchema.fbs";

namespace TrimBim2D.fbs;

/*
DXK_NPREDEFINED_COLORS 8
DXK_COLOR_BLACK      0
DXK_COLOR_WHITE      1
DXK_COLOR_RED        2
DXK_COLOR_GREEN      3
DXK_COLOR_BLUE       4
DXK_COLOR_CYAN       5
DXK_COLOR_YELLOW     6
DXK_COLOR_MAGENTA    7
*/

struct LineColorRGB {
  color:int;
}

enum LineColorDefined:byte {
  Black = 0,
  White = 1,
  Red = 2,
  Green = 3,
  Blue = 4,
  Cyan = 5,
  Yellow = 6,
  Magenta = 7,
  Predefined = 8
}

enum LineType:byte {
  Normal = 0,
  Border = 1,
  Center = 2,
  DashDot = 3,
  Dashed = 4,
  Divide = 5,
  Dot = 6,
  Hidden = 7,
  ISO02W100 = 8,
  ISO03W100 = 9,
  ISO04W100 = 10,
  ISO05W100 = 11,
  ISO06W100 = 12,
  ISO07W100 = 13,
  ISO08W100 = 14,
  ISO09W100 = 15,
  ISO10W100 = 16,
  ISO11W100 = 17,
  ISO12W100 = 18,
  ISO13W100 = 19,
  ISO14W100 = 20,
  ISO15W100 = 21
}

struct LineWidth {
  width:ubyte;
}

table Pen {
  line_color:LineColorDefined = Black;
  line_type:LineType = Normal;
  line_width:LineWidth;
}
```

Appendix 4: Data serializing circle, pen, line with pen and text to FlatBuffers



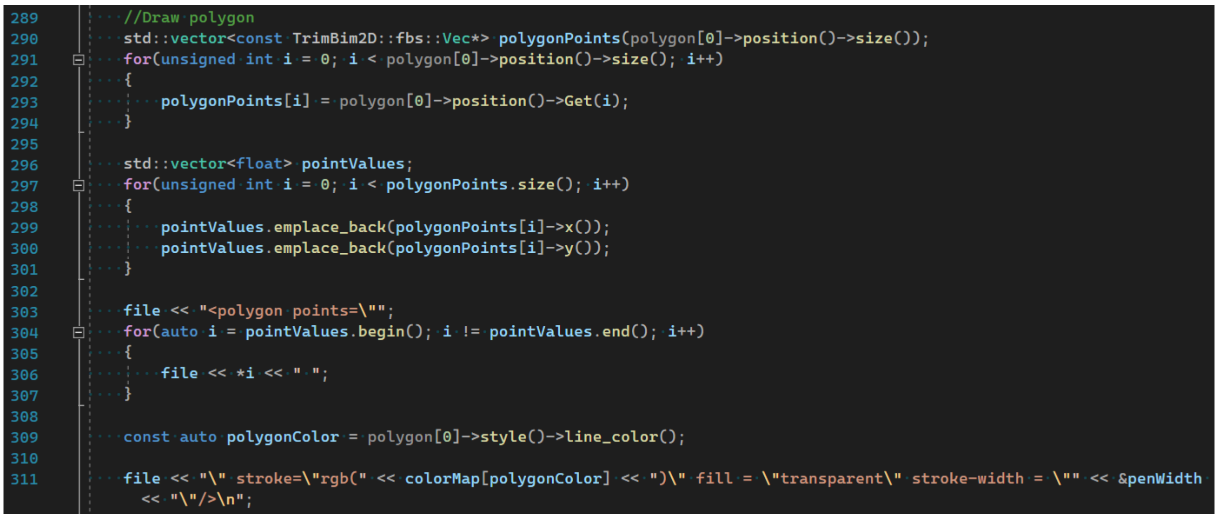
```cpp
//Create circle with style
TrimBim2D::fbs::Vec centerPoint{ 100.0, 100.0 };
float radiusCircle = 50;
TrimBim2D::fbs::LineColorDefined circleC = TrimBim2D::fbs::LineColorDefined_Green;
TrimBim2D::fbs::LineType circleT = TrimBim2D::fbs::LineType_Normal;
TrimBim2D::fbs::LineWidth circleW = 1;
const auto circleStyles = TrimBim2D::fbs::CreatePen(fbb, circleC, circleT, &circleW);
const auto circle1 = TrimBim2D::fbs::CreateCircle(fbb, &centerPoint, radiusCircle, circleStyles);
std::vector<flatbuffers::Offset<TrimBim2D::fbs::Circle>> circles;
circles.reserve(1);
circles.emplace_back(circle1);
flatbuffers::Offset<flatbuffers::Vector<flatbuffers::Offset<TrimBim2D::fbs::Circle>>> circlesVec = fbb.CreateVector(circles);

//Create pen
TrimBim2D::fbs::LineColorDefined lC = TrimBim2D::fbs::LineColorDefined_Black;
TrimBim2D::fbs::LineType lT = TrimBim2D::fbs::LineType_Dashed;
TrimBim2D::fbs::LineWidth lW = 1;
const auto pen = TrimBim2D::fbs::CreatePen(fbb, lC, lT, &lW);
std::vector<flatbuffers::Offset<TrimBim2D::fbs::Pen>> pens;
pens.reserve(1);
pens.emplace_back(pen);
const auto penVec = fbb.CreateVector(pens);

//Create line with pen
const auto line = TrimBim2D::fbs::CreateStyledLine(fbb, lineVector, pen);
std::vector<flatbuffers::Offset<TrimBim2D::fbs::StyledLine>> newLines;
newLines.reserve(1);
newLines.emplace_back(line);
flatbuffers::Offset<flatbuffers::Vector<flatbuffers::Offset<TrimBim2D::fbs::StyledLine>>> lineVec = fbb.CreateVector(newLines);

//Create Text with style
const auto textText = fbb.CreateString("Example text");
const auto textFont = fbb.CreateString("Helvetica");
TrimBim2D::fbs::Vec textPosition {50.0, 50.0};
float textHeight = 20;
TrimBim2D::fbs::LineColorDefined textC = TrimBim2D::fbs::LineColorDefined_Red;
TrimBim2D::fbs::LineType textT = TrimBim2D::fbs::LineType_Normal;
TrimBim2D::fbs::LineWidth textW = 1;
const auto textStyle = TrimBim2D::fbs::CreatePen(fbb, textC, textT, &textW);
const auto text = TrimBim2D::fbs::CreateTextDisplay(fbb, textText, textFont, &textPosition, textHeight, 0.0 , textStyle);
std::vector <flatbuffers::Offset<TrimBim2D::fbs::TextDisplay>> texts;
texts.reserve(1);
texts.emplace_back(text);
flatbuffers::Offset <flatbuffers::Vector <flatbuffers::Offset<TrimBim2D::fbs::TextDisplay>>> textVec = fbb.CreateVector(texts);
```

Appendix 5: Deserializing data for reading pen, circle and text with FlatBuffers

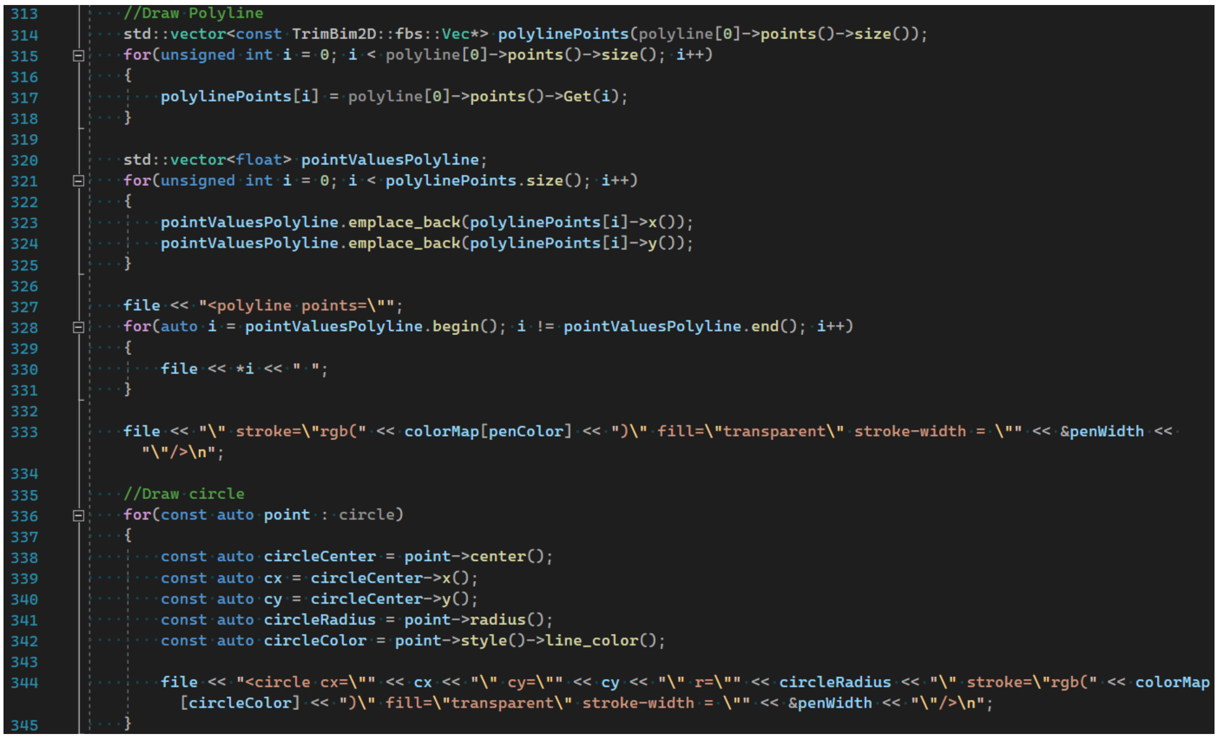


```cpp
    //Get Polyline
    const auto &vectorOfPolylines = *readModel->polylines();
    std::vector<const TrimBim2D::fbs::Polyline*> pointsOfPolyline = vectorIterator<const flatbuffers::Vector<flatbuffers::Offset<TrimBim2D::fbs::Polyline>>, std::vector<const TrimBim2D::fbs::Polyline*>>(vectorOfPolylines);
    
    //Get pen
    const auto &penStyle = *readModel->styles();
    std::vector<const TrimBim2D::fbs::Pen*> vectorOfStyles = vectorIterator<const flatbuffers::Vector<flatbuffers::Offset<TrimBim2D::fbs::Pen>>, std::vector<const TrimBim2D::fbs::Pen*>>(penStyle);;

    //Get circle
    const auto &vectorOfCircles = *readModel->circles();
    std::vector<const TrimBim2D::fbs::Circle*> circlePoints = vectorIterator<const flatbuffers::Vector<flatbuffers::Offset<TrimBim2D::fbs::Circle>>, std::vector<const TrimBim2D::fbs::Circle*>>(vectorOfCircles);

    //Get Text
    const auto &vectorOfText = *readModel->texts();
    std::vector<const TrimBim2D::fbs::TextDisplay*> textAttr = vectorIterator<const flatbuffers::Vector<flatbuffers::Offset<TrimBim2D::fbs::TextDisplay>>, std::vector<const TrimBim2D::fbs::TextDisplay*>>(vectorOfText);
```

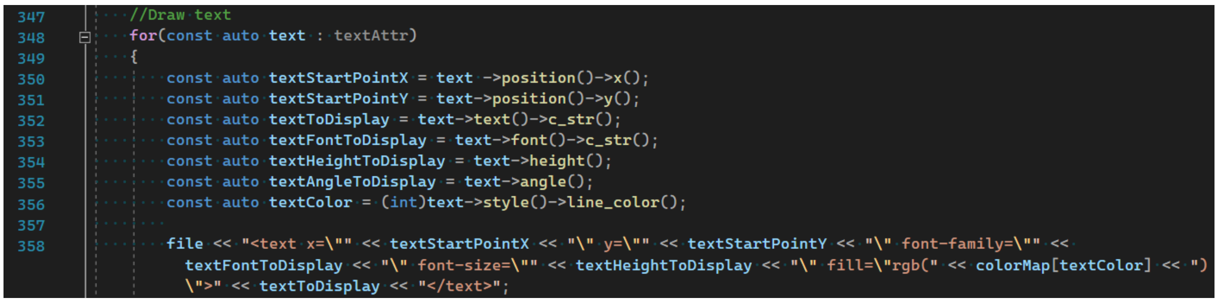
Appendix 6: Write polygon, polyline, circle, and text to SVG file



```
    //Draw polygon
    std::vector<const TrimBim2D::fbs::Vec*> polygonPoints(polygon[0]->position()->size());
    for(unsigned int i = 0; i < polygon[0]->position()->size(); i++)
    {
        polygonPoints[i] = polygon[0]->position()->Get(i);
    }

    std::vector<float> pointValues;
    for(unsigned int i = 0; i < polygonPoints.size(); i++)
    {
        pointValues.emplace_back(polygonPoints[i]->x());
        pointValues.emplace_back(polygonPoints[i]->y());
    }

    file << "<polygon points=\"";
    for(auto i = pointValues.begin(); i != pointValues.end(); i++)
    {
        file << *i << " ";
    }

    const auto polygonColor = polygon[0]->style()->line_color();

    file << "\" stroke=\"rgb(" << colorMap[polygonColor] << ")\" fill = \"transparent\" stroke-width = \"" << &penWidth
      << "\"/>\n";
```



```
    //Draw Polyline
    std::vector<const TrimBim2D::fbs::Vec*> polylinePoints(polyline[0]->points()->size());
    for(unsigned int i = 0; i < polyline[0]->points()->size(); i++)
    {
        polylinePoints[i] = polyline[0]->points()->Get(i);
    }

    std::vector<float> pointValuesPolyline;
    for(unsigned int i = 0; i < polylinePoints.size(); i++)
    {
        pointValuesPolyline.emplace_back(polylinePoints[i]->x());
        pointValuesPolyline.emplace_back(polylinePoints[i]->y());
    }

    file << "<polyline points=\"";
    for(auto i = pointValuesPolyline.begin(); i != pointValuesPolyline.end(); i++)
    {
        file << *i << " ";
    }

    file << "\" stroke=\"rgb(" << colorMap[penColor] << ")\" fill=\"transparent\" stroke-width = \"" << &penWidth <<
      "\"/>\n";

    //Draw circle
    for(const auto point : circle)
    {
        const auto circleCenter = point->center();
        const auto cx = circleCenter->x();
        const auto cy = circleCenter->y();
        const auto circleRadius = point->radius();
        const auto circleColor = point->style()->line_color();

        file << "<circle cx=\"" << cx << "\" cy=\"" << cy << "\" r=\"" << circleRadius << "\" stroke=\"rgb(" << colorMap
          [circleColor] << ")\" fill=\"transparent\" stroke-width = \"" << &penWidth << "\"/>\n";
    }
```



```
    //Draw text
    for(const auto text : textAttr)
    {
        const auto textStartPointX = text ->position()->x();
        const auto textStartPointY = text->position()->y();
        const auto textToDisplay = text->text()->c_str();
        const auto textFontToDisplay = text->font()->c_str();
        const auto textHeightToDisplay = text->height();
        const auto textAngleToDisplay = text->angle();
        const auto textColor = (int)text->style()->line_color();

        file << "<text x=\"" << textStartPointX << "\" y=\"" << textStartPointY << "\" font-family=\"" <<
          textFontToDisplay << "\" font-size=\"" << textHeightToDisplay << "\" fill=\"rgb(" << colorMap[textColor] << ")
          \">" << textToDisplay << "</text>";
```