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

Exploration of train station's design space via regulation-compliant parametric model

Scientific work to obtain the degree

Bachelor of Science (B.Sc.)

at the Department of Civil, Geo and Environmental Engineering of the Technical University of Munich.

Supervised by Prof. Dr.-Ing. André Borrmann
Jimmy Abualdenien
Sebastian Essser
Chair of Computational Modelling and Simulation

Submitted by Benedict Harder ()

e-Mail:

Submitted on April 30th 2021

Contents

List of Figures

List of Tables

Acronyms

2D	Two-dimensional
3D	Three-dimensional
AEC	Architecture, Engineering and Construction
BIM	Building Information Modelling
CAD	Computer Aided Design
DB	Deutsche Bahn/German Railways
GANS	Generative Adversarial Neural Networks
GCS	Geometric Constraint Solver
GIS	Geographical Information System
IFC	Industry Foundation Classes
JSON	JavaScript Object Notation
LCEA	Life Cycle Energy Analysis
NLP	Natural Language Processors

Abstract

With ever-increasing demands on building performance, the early stage of architectural design becomes an even more significant factor for harnessing a potential building's performance. As decisions done during the early stages have the most impact on the entire project, architects and engineers must consider every possible design choice. Generative Design can help achieve this by automatically generating a large amount of designs, for example with the help of Generative Adversarial Neural Networks. One of the important aspects to consider during the early stages is regulation-compliance. Following guidelines from the beginning eases the design process by eliminating the need for adjustments later on. This thesis focuses on building a script with which regulation-compliant models of the U9-station in Munich can be generated using Autodesk Revit and Dynamo, with regulations originating from official DB (German Railways) guidelines. Furthermore, the possibility and success of translating regulations into code scripts are analyzed statistically and the state of the art of generative design in the early stages is presented.

Chapter 1

Introduction

1.1 Motivation

Since the performance of a building is becoming increasingly significant, energy efficiency to name just one factor, it is progressively crucial to consider regulation during the early phases of the architectural design. Being regulation-compliant during these phases renders the entire process faster, more economical and makes the detailed construction later more convenient to design. The early design phases are amongst the most impactful in regards to later performance of the building (BRAGANÇA, VIEIRA, & ANDRADE, 2014). Making sure that early decisions already follow the necessary regulations and building code means that the architects or engineers can spend more time planning and considering the general layout instead of making changes based on certain regulations not yet fulfilled. A parametric model can aid this process significantly. This type of model follows certain imposed constraints (WASSIM, 2013), which can be as simple as 'these two lines must always be parallel to each other' to 'the euclidean distance between these two objects must never be lower than 2, no matter the circumstances'. As one can see, the second example already somewhat resembles a regulation that can be found in building code. Managing to implement a parametric model that can follow at least some or potentially even all regulations would take away a lot of work of the designers and increase building performance (BRAGANÇA et al., 2014). Additionally, with the help of generative design one could generate as many regulation-compliant models as possible and then choose the most successful one (KRISH, 2011).

One of the aspects to consider during the early phases of architectural design, especially in public buildings, is the ability to evacuate all people inside the building to a safe space (ABUALDENIEN, CLEVER, et al., 2020). With the help of Generative Adversarial Neural Networks one can already simulate pedestrian evacuations simulations based on a given model (ACCU:RATE, 2021). When coupled together with the plethora of regulation-compliant models that can be created with the help of a parametric model, it could be possible to (with a certain degree of surety) find the "best" model relatively quickly, at least based on their performance regarding pedestrian evacuation. This kind of technology could lead to a leaner planning process. On the basis of the U9 station in Munich, this thesis will present a parametric model based on official Deutsche Bahn/German Railways regulations and evaluate the quality of it statistically.

1.2 Structure

The structure of this thesis is as follows:

1. Chapter 2 presents the current state of the art in parametric modelling, generative design, GANS and pedestrian evacuation simulation.
2. Chapter 3 demonstrates the methodology used for implementing real-world regulations into code.
3. Chapter 4 evaluates the success and quality of the parametric model in regards to how much can and could be implemented.
4. Chapter 5 describes the implementation process of the regulations into a *Dynamo* script.
5. Chapter 6 shows different variations of the model on the basis of different input parameters.
6. Chapter 7 discloses limitations with this approach.
7. In Chapter 8 the topic is concluded and ideas and thoughts for future research are proposed.

Chapter 2

State of the Art

Critical to understanding the significance of this subject is understanding the current state of the industry and research regarding the early stages of design, simulation and generative design. By means of literature this chapter will evaluate and present the state of the art accordingly.

2.1 Design exploration in early phases

As mentioned, the early stages in architectural design are the most crucial to building performance. Tasks during these phases include defining and summarizing the core building functions, constraints and requirements and then implementing these into a concept (Bragança et al., 2014). Bragança et al. group these tasks together and call it the "conceptual phase" (Bragança et al., 2014, p. 3), which is also the name used here on out. The reason for the significance of this phase is that these concepts will have a large impact on the general performance of the building (Abualdenien & Borrmann, 2019). As several experts of various domains such as designers, engineers and architects must collaborate intensely (Abualdenien, Pfuhl, & Braun, 2019), they also face the problem of information scarcity. Since only a limited amount of properties of the building are known during this stage, often restricted to footprint and height, a lot of estimations have to be made in order to assess the energy output, material cost, construction time and other essential quantities relevant to the client (Bragança et al., 2014). Therefore, eliminating the need for estimations and instead managing to extract realistic information out of these concepts would be a great benefactor for finding a suitable concept, the basis for a successful performant building.

An aspect of this would be, for example, early-on regulation-compliance. Inspecting for regulation- or code-compliance requires extensive knowledge on the pertinent local and national law (Abualdenien et al., 2019). Including this kind of process during a stage of information scarcity, as mentioned above, is a challenge in itself, in addition to the difficulty of the process. In order to realize these ideas, additional tools to assist experts during the conceptual phase are necessary. In the following, several pieces of software which would facilitate resolving the problems mentioned above will be presented.

2.2 Generative Design in Architecture

2.2.1 Parametric modelling

Developed by SHAH and MÄNTYLÄ, parametric modelling or parametric CAD is a concept of modelling 2D sketches equipped with constraints such as `parallel, coincide, tangential, etc.` (SHAH & MÄNTYLÄ, 1995) and is widely implemented into a variety of softwares such as *Autodesk Inventor, Siemens NX* and more (VILGERTSHOFER & BORRMANN, 2017). The sketches are comprised of geometric objects such as lines, while the parametric constraints define the topology of these objects. The entirety is eventually solved by a Geometric Constraint Solver (FUDOS & HOFFMANN, 1997).

Parametric modelling has a vast range of application, including AEC. SALIMZADEH, VAHDATIKHAKI, and HAMMAD leverage a parametric model developed in *Dynamo* to find suitable layout designs for various photo voltaic modules, accounting for different surface types and building components using BIM (SALIMZADEH et al., 2020). Another application developed by BARAZZETTI and BANFI is combining a parametric BIM-model with geospatial data of GIS to plan infrastructure such as roads, where referencing a global coordinate system is necessary (BARAZZETTI & BANFI, 2017).

2.2.2 Generative Design

Generative Design refers to an intricate process built to help designers accelerate problem-solving and discovering the best designs possible. When considering the assessment of issues of a particular design, FRAZER speaks of the fact that it is “misleading to talk of design as a problem-solving activity” (FRAZER, 2002, p. 253) and instead defines designing as “a problem-finding activity.[...] Indeed the solution is often the very definition of the problem” (FRAZER, 2002, p. 253). Consequentially, FRAZER created “The Generative Evolutionary Paradigm” which describes the process of defining a representation of the problem, creating genetic code on the basis of the representation, extract code scripts to create designs, evaluate the designs and create more code scripts from the successful iterations (FRAZER, 2002, p. 255). This allows the architect to consider a significantly larger amount of different approaches. KRISH breaks down the stages of generative design in a similar fashion, defining them as “1. A design schema 2. A means of creating variation 3. A means of selecting desirable outcomes” (KRISH, 2011, p. 90). KRISH also mentions that this type of design automation is well-suited for creative design problems, especially ones where subjectivity plays a role as in aesthetics. It can, however, also be applied to other less subjective problems such as energy efficiency in buildings (CALDAS, 2008).

2.2.3 GANS - Generative Adversarial Neural Networks

GANS are a type of machine learning network used for generating output data similar to the set of given input data (ZHENG & HUANG, 2018). Adversarial in this case means that two different models are competing against each other. GOODFELLOW et al. describe these as a generative model G and a discriminative model D. While G generates models based on given training data, D tries to identify if a piece of data originates from either the training data or G, with the main objective of G being that D makes a mistake. Meanwhile the main objective of D is to better separate training data and generated data. As G gets better at generating data similar to the input, D gets better at distinguishing it and vice versa (GOODFELLOW et al., 2014). GANS can be used, for example to generate architectural 2D drawings. ZHENG and HUANG use a modified version of GANS called PIXPIX2PIXHD by WANG et al. built to generate 2D pixel-based images. Based on training data of 100 floor plans, each kind of room was given a unique color in order to be distinguishable from one another. Based on this data, the GANS can generate similar floor plans to the training data (ZHENG & HUANG, 2018).

2.2.4 Refinery from Autodesk Revit

Refinery is Autodesk's implementation of generative design for their software-catalog. Available in *Revit 2021*, it allows the architect to explore a larger number of different design options and approaches than usually manageable and thus, can lead to a higher quality product. It is tightly integrated with *Dynamo* for *Revit*. By creating so-called 'studies' used to generate the designs, the user can define the way the generation behaves. An important factor regarding this is the selection of input and output nodes in the *Dynamo* graph. With these selected, one can define whether certain outputs should be maximized or minimized. This can be any number value including, but not limited to, an area, height, or number of rooms. *Refinery* then optimizes the generation process and follows the users guidelines[1].

2.3 Simulation and Life-Cycle Assessment

2.3.1 Types of Simulation

Simulations during the early stages of design can help evaluate the quality and performance of the building. Despite the information scarcity it is still possible to achieve meaningful results. One of the types of simulations is a Life Cycle Energy Analysis. A LCEA comprises of an assessment of the embedded energy which consists of "Construction", "Use", "Disposal" and "Recycling, Reuse and Reutilization" (according to DIN 15978-2012-10) and the energy required during operation which are then merged (ABUALDENIEN, SCHNEIDER-MARIN, et al., 2020). Another type would be a "Structural Preliminary Design", which based on the rough design of the supporting structures, evaluates the structural design during the early stages using a "specialized development system" (ABUALDENIEN, SCHNEIDER-MARIN, et al., 2020, p. 14).

[1] AUTODESK, 2021a.

2.3.2 Pedestrian Evacuation Simulation

When simulating pedestrians there are two main approaches: the hydraulic and the microscopic approach. The former focuses mainly on pedestrian density and its influence on the flow-rate by utilizing empirical data. This approach considers and uses the similarities between pedestrian flow and the flow of liquids (PLUM & JÄGER, 2011). The latter approach focuses on the pedestrian as an individual and allows the simulation of smaller-scale events, such as congestions in certain parts of the pathway. Divisible into either discrete models or continuous ones, which differ in their approach to time and space granularity, this kind of simulation also uses a large amount of computing power (KNEIDL, 2013).

Among the software using this technology, *crowd:it* is one of the more popular ones. Based on the microscopic approach by KNEIDL with the 'Optimal Steps Model' by SEITZ it can accurately simulate the pedestrians' stepping behavior (SEITZ, 2016). After placing and defining objects such as origins, stairs, destinations, paths, etc., the software creates a 'floodfield' containing the force-values and individual experiences. This lets the software realistically predict the behaviour of the pedestrians (SCHOLL, 2019).

The outbreak of the Covid-19-pandemic in 2020 has demonstrated that the issue of social distancing during a pedestrian evacuation needs to be considered and addressed. Especially in hubs of cities such as train stations and pedestrian walking zones, which are used by thousands of people each day, the risk of infection is significant, and maintaining hygiene regulations as well as general security becomes of upmost importance (ABUALDENIEN, CLEVER, et al., 2020). Capacities of platforms during social distancing, variation of safety distances depending on mask-enforcement and places where the minimum distance required cannot be realistically fulfilled/enforced are the main points where ABUALDENIEN, CLEVER, et al. delve into detail.

Chapter 3

Methodology

3.1 Regulations

3.1.1 Level of Regulations

Regulations can be imposed from various sources, each fulfilling a different role and imposing different regulations for the building construction. These include the Federal Building Code, or "Baugesetzbuch", which describes subjects concerning administrative procedures, property rights, land use and more. Following that, is the building code of the individual states, the so-called "Bauordnung"[1], which goes into greater detail as to how a building is supposed to be constructed to minimize public health and safety risks[2]. This also includes fire safety regulations meant to decrease the danger fires exhibit. On the city or municipal level, statutes and bylaws are added to meet the individual demands of the municipality[3]. Additionally to these government-imposed regulations, companies or institutions themselves can also have guidelines as to what their building must look and perform like. In the case of this thesis, the documents 813.0201 and 813.0202 are especially significant because of their relevance to our use case. As a result, we selected them to be implemented as a par metric model. Those two guidelines define how train station platforms as well as their accesses are to be constructed and built.

3.2 Embedding Regulations in a Parametric Model

In order to implement a given set of regulations into a parametric model they need to undergo a process of translation to be in a computer-readable form such as code before being used for the creation of a 3D-model (see Figure 3.1). This translative process is highly variable depending on the style of the regulation, which can take on forms such as a table, a formula, written out and more. Depending on the form, a corresponding piece of code needs to be found in order to ensure that the translation between regulation and code is consistent.

[1]BAK - Federal Chamber of German Architects, 2021.
[2]Bavarian Ministry for Housing, Construction and Transportation, 2021.
[3]Provincial Capital Munich, 2021.

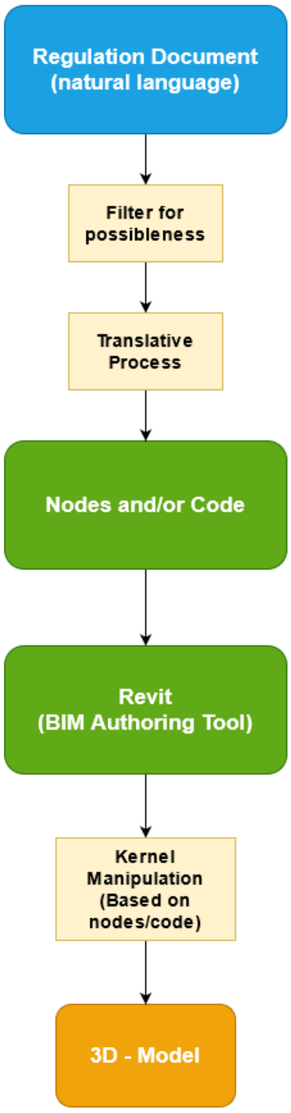

Regulation Document (natural language)
Filter for possibleness
Translative Process
Nodes and/or Code
Revit (BIM Authoring Tool)
Kernel Manipulation (Based on nodes/code)
3D - Model

Figure 3.1: Visualization of the process of translation between regulations and code

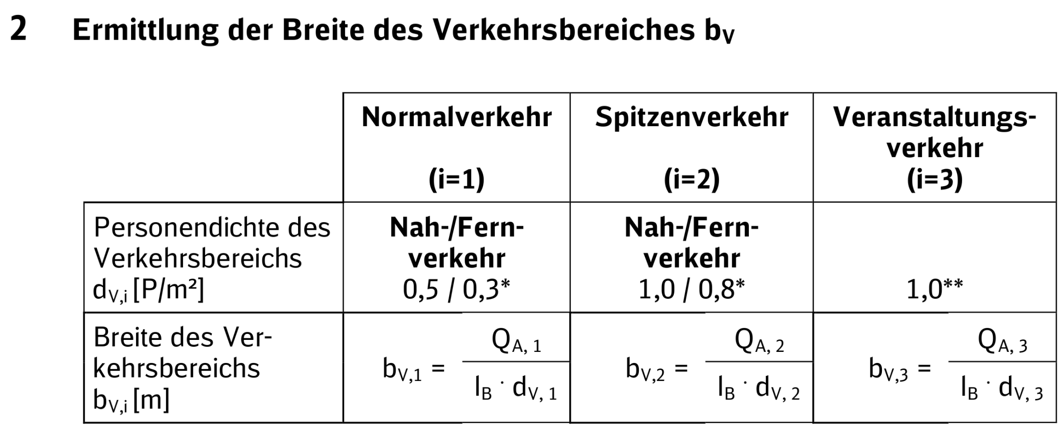
First, an assessment of the types of regulations regarding their forms, must be established:

1. Natural language: in the documents 813.0201 and 813.0202, all regulations contain at least parts that are written in German.
2. Formula: The regulation is in the form of a mathematical formula with parameters and/or variables.
3. Table: Certain parameters are determined dependant on other parameters with the help of tables.

By managing to categorize the rules by their form, we can achieve said consistence by now determining the translative process needed to write code that directly corresponds to the regulation. Of course natural language poses the largest problem here. This kind of regulation can either be handled manually by a person who must read, understand and then translate, or it can be handled by a natural language processor. This software processing can translate natural language in order for computers to understand it. This knowledge can then be used for the computer to write a program (Mihalcea, Liu, & Lieberman, 2006). Using NLPs successfully would be especially convenient for large-scale translation of regulation documents, but considering the scope of this thesis and the number of regulations to translate, a manual approach is better suited given the complexity of such processors. As for the other types (Formula and Table), simpler approaches can be used. Programming languages such as `Python` offer a multitude of ways to achieve this. One prominent example of this will be briefly introduced below.

Table Translation into Code

Tables are quite prevalent in regulation documents and are an excellent way to determine variables whose values depend on other parameters. In Figure 3.2 (813.0201, 2012, p. 60) we can see a excellent example of this kind of table where multiple variables can be identified. Tables such as this one can easily be translated into code with the help of nested `if-statements`. These statements check existing parameters for their values and create a control flow leading to the correct result. The table seen in Figure 3.2 for example could be translated into code as seen in "Table Script":



2 Ermittlung der Breite des Verkehrsbereiches b_V

	Normalverkehr (i=1)	**Spitzenverkehr (i=2)**	**Veranstaltungs-verkehr (i=3)**
Personendichte des Verkehrsbereichs $d_{V,i}$ [P/m²]	**Nah-/Fern-verkehr** 0,5 / 0,3*	**Nah-/Fern-verkehr** 1,0 / 0,8*	1,0**
Breite des Ver-kehrsbereichs $b_{V,i}$ [m]	$b_{V,1} = \frac{Q_{A,1}}{l_B \cdot d_{V,1}}$	$b_{V,2} = \frac{Q_{A,2}}{l_B \cdot d_{V,2}}$	$b_{V,3} = \frac{Q_{A,3}}{l_B \cdot d_{V,3}}$

Figure 3.2: A table to determine platform-relevant variables (813.0201, 2012, A05 p. 2)

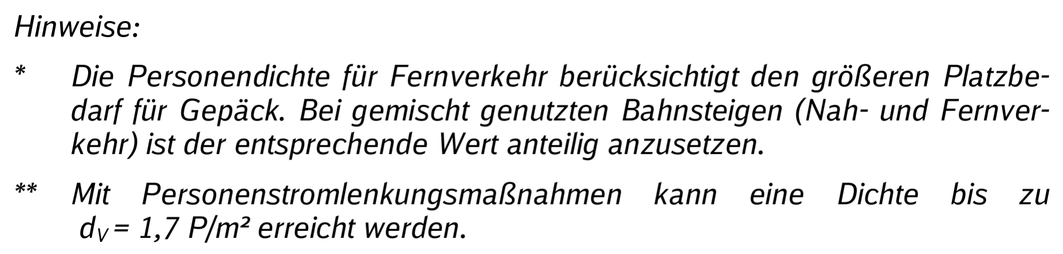
Table Script

```
1  i = 2                 #can be either 1, 2 or 3
2  regional = False      #can be either True or False
3  if i == 1:
4      if regional:
5          d_v1 = 0.5
6      else:
7          d_v1 = 0.3
8      b_v1 = Q_A1/(I_B*d_v1)
9  if i == 2:
10     if regional:
11         d_v2 = 1
12     else:
13         d_v2 = 0.8
14     b_v2 = Q_A2/(I_B*d_v2)
15 if i == 3:
16     d_v3 = 1
17     b_v1 = Q_A3/(I_B*d_v3)
```

If quite lengthy, this script achieves exactly what the table does. With our example values seen in lines 1 and 2, our script would lead to the value of `d` (or `d_v2`) being equal to `0.8` and `b` (or `b_v2`) being equal to `Q_A2/(I_B*d_v2)`. The advantage of this implementation is that it can be scaled indefinitely, or at least up until the system resources have been exhausted.

Issues when Embedding Regulations

As seen above, some forms of regulations are easily translated into code. Problems arise with written regulations where it is more difficult to decipher which parts must be translated. In Figure 3.3 there are two extra regulations to be seen, which correspond directly to the table seen in Figure 3.2 and are located below it.



Hinweise:

* *Die Personendichte für Fernverkehr berücksichtigt den größeren Platzbedarf für Gepäck. Bei gemischt genutzten Bahnsteigen (Nah- und Fernverkehr) ist der entsprechende Wert anteilig anzusetzen.*

** *Mit Personenstromlenkungsmaßnahmen kann eine Dichte bis zu d_V = 1,7 P/m² erreicht werden.*

Figure 3.3: Additions to the table of Figure 3.2 (813.0201, 2012, A05 p. 2)

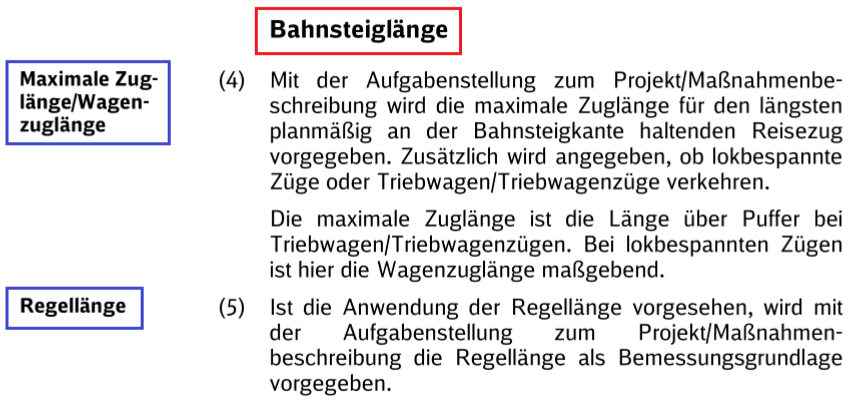
The first asterisk refers to the different densities of passengers depending on it being regional traffic or not. Should the platform be shared between regional and long-distance traffic, the density is to be calculated proportionally. Accordingly, in order to translate this part of the regulation one would have to interpret that one needs to interpolate between the two values in a linear fashion so that one could obtain the appropriate value, then translate that into code. The second asterisk poses a different problem, where in order to implement this one would have to know exactly how passenger flow control measures are defined. Issues during the translative process mainly arise when trying to decipher written out regulations, which sometimes leave room for interpretation or do not go into the level of detail necessary to be fully implemented. Of course, this issue can be addressed by engaging and interacting with domain experts and/or the Deutsche Bahn/German Railways, however this is not within the scope of this thesis.

Chapter 4

Statistical Analysis

4.1 Methodology

The regulations found in 813.0201 and 813.0202 and their respective appendices were assessed on whether they are within the scope of pedestrian evacuation simulation, whether they are implementable and whether they actually have been translated into code. While most regulations are technically capable of being translated into code, not all of them are relevant for our simulation. A good example for a regulation of this type would be the construction detail for the control panel in the elevators. As important as they may be, it barely adds anything of value to simulating pedestrians. The way the assessment was done was by sorting the general categories of regulations, then dividing them up into small-scale detailed regulations. The documents 813.0201 and 813.0202 offer a convenient way of categorizing shown in Figure 4.1:



Bahnsteiglänge

Maximale Zuglänge/Wagenzuglänge

(4) Mit der Aufgabenstellung zum Projekt/Maßnahmenbeschreibung wird die maximale Zuglänge für den längsten planmäßig an der Bahnsteigkante haltenden Reisezug vorgegeben. Zusätzlich wird angegeben, ob lokbespannte Züge oder Triebwagen/Triebwagenzüge verkehren.

Die maximale Zuglänge ist die Länge über Puffer bei Triebwagen/Triebwagenzügen. Bei lokbespannten Zügen ist hier die Wagenzuglänge maßgebend.

Regellänge

(5) Ist die Anwendung der Regellänge vorgesehen, wird mit der Aufgabenstellung zum Projekt/Maßnahmenbeschreibung die Regellänge als Bemessungsgrundlage vorgegeben.

Figure 4.1: Categories in red, detailed regulations in blue (813.0201, 2012, p. 6)

The results will then be compiled into category-specific tables. The first column indicates the regulation, the second one whether it is implementable or not. The third column shows the reason why, as shown below:

1. Family Type: Specific family type missing.
2. Software: Additional software features needed.
3. Document: Further regulation documents necessary.

The next column shows whether or not the regulation is within the scope and the last one shows an estimation of how much of it was implemented into the script. Finally, the results of every category are summarized in order to calculate the total percentage of 'implementable', 'within scope' and 'implemented'.

4.2 Results

Platform Height

Table 4.1: Platform height regulations (813.0201, 2012, 5f)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Standard heights	Yes	-	Yes	100%
Partial heightening	Yes	-	No	-
Total	100%	-	50%	50%

The height of the platform can be chosen from a list of possible standard heights, which is easily translatable into code. The second regulation though refers to changes to a platform after the initial construction of it and would be more complex to implement. One would have to offer an interface for the user in which he or she could input the changes to be made, and then change only a specific part of the model, all of which is out of the scope of this thesis.

Platform Length

Table 4.2: Platform length regulations, (813.0201, 2012, 6ff)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Train length	Yes	-	Yes	100%
Standard-lengths	Yes	-	Yes	100%
Usable length	Yes	-	Yes	100%
Installation length	Yes	-	Yes	100%
Total	100%	-	100%	100%

Platform length is not only within the scope of this thesis, it is also very straightforward to implement. No specific family types or software features are needed, and all formulas and specifications are listed in the available documents.

Platform Width

Table 4.3: Platform width regulations, (813.0201, 2012, 9ff)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Danger zone	Yes	-	Yes	100%
Minimum Width	Yes	-	Yes	75%
Boarding Aids	Yes	-	No	-
Minimum Area	Yes	-	Yes	-
Total	100%	-	75%	43.75%

The width of the platform is also implementable, if not as easily as the previous ones. While the width is calculated differently in the areas with boarding aids, it is not within the scope since boarding aids have not been implemented in the script. The percentage of 75 in line two stems from the four different objects that are placed onto the platform (stairs, escalators, elevators and columns) of which three have been implemented.

Construction

Table 4.4: Platform construction regulations, (813.0201, 2012, 11ff)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Load	Yes	-	No	-
Inspection	No	Document	No	-
Installation Dimen-sions	Yes	-	No	-
Grounding	No	Family Types	No	-
Slabs/Piping	No	Family Types	No	-
Edges	No	Family Types	No	-
Ends	No	Family Types	No	-
Security Area	No	Family Types	No	-
Drainage	No	Document	No	-
Coating	No	Family Types	No	-
Markings	No	Family Types	No	-
Guidance System	No	Family Types	No	-
Total	8.3%	-	0%	0%

Construction details such as the ones in this category are not at all relevant for our simulation and thus, have not been implemented. Additionally, in order to implement them further, family types and/or documentation would be necessary.

Platform Access

Table 4.5: Platform access regulations, (813.0202, 2012, 4ff)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Step-free access	Yes	-	Yes	100%
1000-People-Rule	No	Document	No	-
Barriers	Yes	-	No	-
Total	66.67%	-	33.33%	0%

The elevators and escalators implemented fulfill the requirement of step-free access, and are an important aspect to consider when trying to simulate evacuations. The 1000-people rule though is specified in a different document and could not be implemented.

Obstruction-Free Paths

Table 4.6: Obstruction-free path regulations, (813.0202, 2012, 6ff)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Length	Yes	-	No	-
Width/Height	Yes	-	No	-
Marking	Yes	-	No	-
Parking	No	Document	No	-
Entrances	Yes	-	No	-
Total	80%	-	0%	0%

Even though the model does, in fact, include pathways with no obstruction on them, there is no specific line or line of code that checks whether this is fulfilled. It has been deemed to not be in the scope since unobstructed paths are a given.

Footpath

Table 4.7: Footpath regulations, (813.0202, 2012, 8f)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Width	Yes	-	No	-
Slope	Yes	-	No	-
Drainage	Yes	-	No	-
Coating	No	Family Types	No	-
Guidance System	No	Family Types	No	-
Total	60%	-	0%	0%

For a more detailed explanation, see subsection 4.2.

Stairs

Table 4.8: Stairs regulations, (813.0202, 2012, 9ff)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Step width	Yes	-	Yes	100%
Slope	No	Family Types	No	-
Intermediate landing	No	Family Types	Yes	-
Area in front	Yes	-	Yes	50%
Access-height	Yes	-	Yes	-
Construction of steps	No	Family Types	No	-
Handrails	No	Family Types	No	-
Sweeping Chute	No	Family Types	No	-
Total	37.5%	-	50%	18.75%

The construction detail of stairs could not be dealt with because of missing family types. The 'Area in front' regulation has been deemed around 50% implemented because there is a part of the script that removes columns that are right in front of them, but it does not specifically check whether there are objects in the area right in front of the stairs. When it comes to the access-height, similar to 4.2, the access is unobstructed when it comes to its height, even though there is no part of the script that checks for compliance.

Ramps

Table 4.9: Ramp regulations, (813.0202, 2012, 12ff)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Width	Yes	-	Yes	-
Slope	No	Family Types	No	-
Drainage	Yes	-	No	-
Space in front	Yes	-	Yes	-
Handrails	No	Family Types	No	-
Spur post	No	Family Types	No	-
Coating	No	Family Types	No	-
Guidance system	No	Family Types	No	-
Total	37.5%	-	25%	0%

Ramps, a part of barrier-free access, have not been added at all since all parts of the station are accessible via escalators and elevators, though given the correct families, should pose no problem to implement.

Elevators

Table 4.10: Elevator regulations, (813.0202, 2012, 14ff)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Size	Yes	-	Yes	100%
Construction	No	Family Types	No	-
Control panel	No	Family Types	No	-
Space in front	Yes	-	Yes	-
Access	Yes	-	Yes	-
Awning	No	Family Types	No	-
Trestle	No	Family Types	No	-
Guidance system	No	Family Types	No	-
Total	37.5%	-	37.5%	12.5%

Here again we have a lot of detailed construction for the elevators. Only the parts relevant for the simulations, namely size, have been translated into code.

Escalators

Table 4.11: Escalator regulations, (813.0202, 2012, 16f)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Construction	No	Document	No	-
Combination with stairs	Yes	-	Yes	100%
Slope	No	Family Types	No	-
Width	Yes	-	Yes	-
Storage	No	Family Types	No	-
Speed	No	Software	Yes	-
Security	No	Family Types	No	-
Guidance system	No	Family Types	No	-
Total	25%	-	37.5%	12.5%

Essentially serving the same function in a very similar form, the implementation of escalators is quite similar to that of the stairs. Again, for certain functions a specific family type would be needed. Additionally, the regulation specifying the speed with which the escalators should transport the passengers is certainly within the scope of our project, but *Revit* lacks the necessary software features.

Grade-Separated Intersections

Table 4.12: Grade-Separated Intersection Regulations, (813.0202, 2012, 17ff)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Width	Yes	-	No	-
Height	Yes	-	No	-
Guidance System	No	Family Types	No	-
Standard Design	No	Document	No	-
Total	50%	-	0%	0%

Given that this is a subway station, intersections are redundant.

Grade-Intersections

Table 4.13: Grade-Level Intersections regulations, (813.0202, 2012, 19f)

Regulation	Imple-mentable	Reason	Within Scope	Implemented
Planning	No	Document	No	-
Guidance System	No	Family Types	No	-
Design	No	Document	No	-
Total	0%	-	0%	0%

See 4.2 for more details.

4.2.1 Summary

Now that the different categories have been analyzed, they can be summarized. Table 4.14 shows our total percentages of 'Implementable', 'Within Scope' and 'Implemented' as well the last column, which shows how much of the regulations that were within the scope have been implemented, giving a better idea of how thorough the implementation was.

Table 4.14: Total Percentages

% Implementable	% Within Scope	% Implemented	% of within-scope regulations imple-mented
54%	30.45%	20.08%	68.42%

Of course the percentage of implementable regulations shown in Table 4.14 is not as high as one would hope when considering using this kind of technology in the industry. But we must take into account that most of the reasons why something can not be implemented are either 'Family Types' or 'Document' as seen in Table 4.15. These reasons are among the easiest to solve, given that one would have access to all documents and family types needed.

Table 4.15: Reason Count

Family Types	Document	Software
31	8	2
75.61%	19.51%	4.88%

If we change our data accordingly, eliminating 'Family Types' and 'Document' as a valid reason for not being implementable, we will of course see the percentage of implementable regulations rise, more accurately presenting how feasible it is to translate regulations into code, as seen in Table 4.16:

Table 4.16: Total Percentages with eliminated reasons

% Implementable	% Within Scope	% Implemented	% of within-scope regulations implemented
97.3%	30.45%	16.35%	53.68%

As expected, the percentage increased significantly, 43.3% to be exact. This number obviously gives a vastly superior outlook into the future of this technology. The more regulations are able to be translated into code, the more useful the program becomes.

4.2.2 Validity

As with any statistics, the results as well as the data can be questioned in terms of their validity. Validity is the measurement of how well the data represents reality. Regarding the data, some of the more glaring issues become apparent. Firstly, the method used to assess the implementation of a regulation is likely to not be completely correct, seeing that it was mainly gauged by reading through the regulation and then making an approximation as to which degree that would be possible and in the negative case, estimating what was missing. Only for those regulations that have been implemented to 100% (a list of those can be seen in Table 4.17) can it be said for certain that these are also implementable, since the complete implementation provides proof for that. As for the other regulation data points, though it is highly improbable that they are completely wrong, their validity can be questioned.

Certain categories also pose a problem when assessing whether they have been implemented or not. A good example for this would be the category 'Obstruction-free Paths' (see Table 4.6). Such paths do exist in the models generated by the script, they simply have not been implemented in a 'fool-proof' way. For example, this would mean that a part of the script checks the model every time if the paths are wide enough and whether there are objects obstructing the way. Despite the fact that the model has been created so that there always will be obstruction-free paths, it is difficult to gauge if the regulations have truly been implemented. In these cases, a conservative assessment of 0% implemented has been done. Apart from this kind of evaluation issue, most of the 'Implemented' data points are only educated estimations of how much of a regulation has been implemented.

Table 4.17: Regulations which were implemented 100%

Regulation	**Category**
Standard heights	Platform Height
Train length	Platform Length
Standard lengths	Platform Length
Usable length	Platform Length
Installation length	Platform Length
Danger zone	Platform Width
Step-free access	Platform access
Step width	Stairs
Size	Elevators
Combination with stairs	Escalators

This is because it is sometimes difficult to translate the process into numerical values of completion. An example for this would be the ’Minimum Width’ regulation in the category ’Platform Width’ (see Table 4.3). The 75% number is a result of three of the four different objects placed on the platform being accounted for when calculating the minimum width, though in most cases the number of object types placed would be significantly higher. On the one hand, one could argue that by proving the concept of making the width dependable on the objects given their dimensions is enough to warrant a 100% mark. On the other hand, this does not account for some special cases which could arise when trying to implement the regulation with different kinds of objects. In this case, the 75% seem fair, but may still not accurately represent the true statistic, whatever that may look like.

In summary, this statistical analysis offers an overview of the completion of the implementation process: from evaluating the regulations, to translating them into code. While the validity may be questioned for the reasons mentioned above, the results can still be used to recognize how well this kind of approach to building and planning works.

Chapter 5

Implementation

5.1 Software

5.1.1 Revit

Autodesk Revit is a software designed for BIM and is capable of handling the planning of a project throughout its entire life cycle. By supporting 3D as well as 2D modelling it can automate repetitive tasks such as updating 2D plans based on changes in the 3D model. It also features building components called 'Families' which can be separated into different 'Family Types'. These types then serve as objects which can be placed within the actual model. The families can be custom built by anyone and thus, allows for theoretically any kind of component made by any company to be 3D-modelled and then used in a project[1]. *Revit* also supports a programming interface called the *RevitAPI*. It provides users with an interface to use programming languages such as `C#`, `Visual C++` and `Visual Basic` in order to directly communicate with *Revit* and build plug-in features for it, as well as allowing the automation of repetitive tasks[2]. In our case, *Revit* will be used as the base in order to create, visualize and inspect the model, as well as export the models as an IFC file and was chosen because it supports all of these necessary features in one piece of software.

5.1.2 Dynamo

Additionally to *Revit*, *Dynamo* was used. *Dynamo* is a visual programming software that interacts directly with the *RevitAPI*. Because of its visual aspect, it allows users without much programming skills to interact with the API and write their own script. A *Dynamo* script builds upon its 'nodes', a code-representing box with specific types of inputs and outputs. Connecting these nodes together creates a chain of processes which then get executed and finally visualized in *Revit*[3]. *Dynamo* offers a wide variety of nodes which allows for various kinds of scripts with different purposes such as parameter-dependant grid-based object placement as seen in Figure 5.1.

[1] AUTODESK, 2021c.
[2] AUTODESK, 2021b.
[3] AUTODESK, 2021d.

As for the U9-Trackline script, a selection of nodes together with a quick explanation is compiled in the following list:

1. `Number Slider`: Outputs a `double`. Interval and step width can be specified by the user.
2. `Family Types`: Outputs a family types chosen from a drop-down list. Only types that are loaded into the current document can be selected.
3. `Python Script`: Uses a variable amount of inputs, can process them using the `Python` programming language and then output a single variable, although it can be nested. The python script has access to the *RevitApi*.
4. Various smaller nodes including, but not limited to: `Boolean`, `Code Block`, `Path` and `Number`.

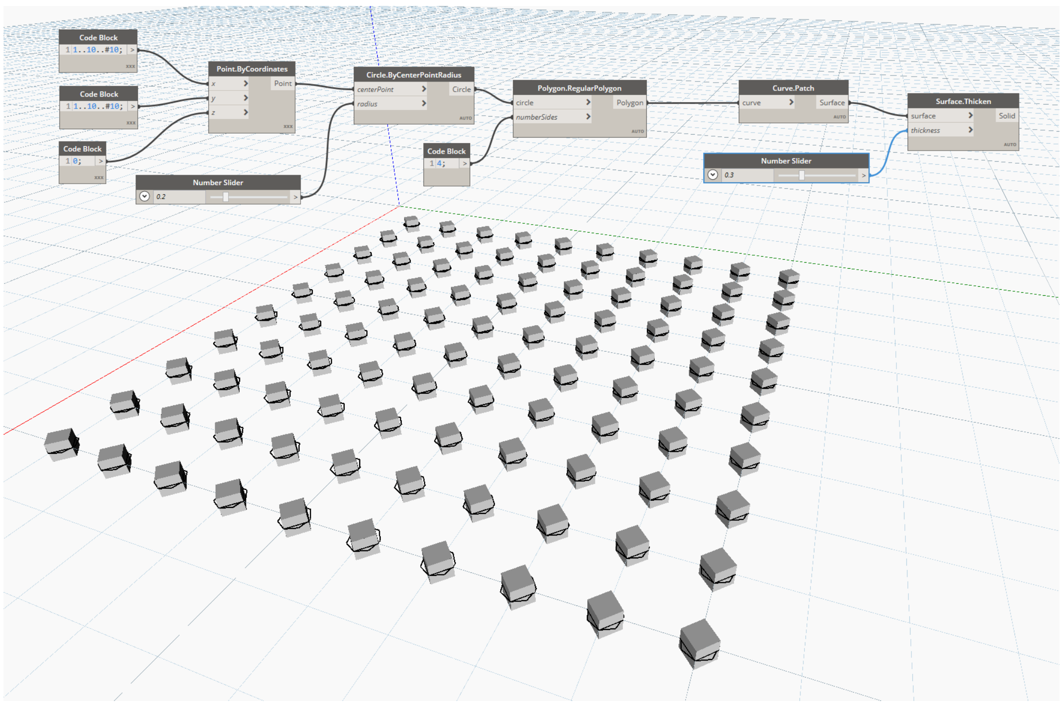


Figure 5.1: An example of a Dynamo script

Since Dynamo allows the user to make changes to the script and view them immediately after the script has completed running, it offers great versatility in debugging and experimenting for this project. This contrasts with a "pure" *RevitAPI* approach, where lengthy compile and load times would hinder the development.

5.2 Code Development

The implementation was achieved using a mixture of *Dynamo* nodes and Python scripts, which also act as *Dynamo* nodes. This makes development easier, since the *Dynamo* script will not be as cluttered with, for example a large amount of "if" nodes. Large chunks of code are written into one python script, which takes inputs, processes them and outputs its results to other nodes. These python-scripts feed into one another and create one fully-functional script, while maintaining a clear structure through their separation. In Figure 5.2 an image of the entire *Dynamo* script with all its nodes is shown.

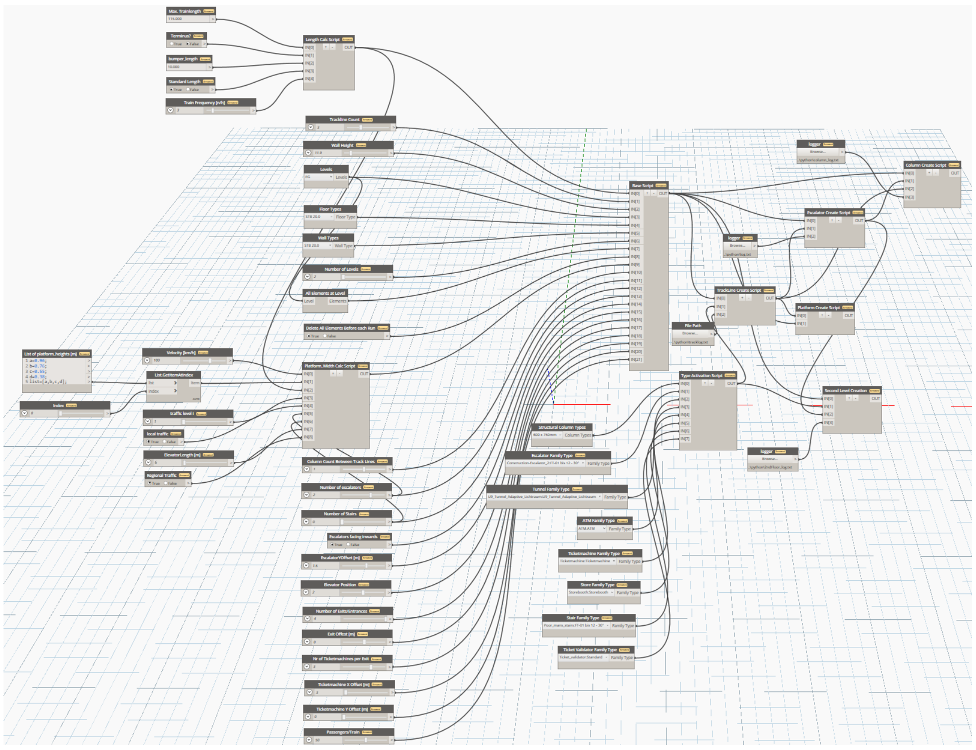


Figure 5.2: The entire U9-Trackline script

When this script is run, a model based on the specified parameters will be created. This model can then be viewed and edited in the *Revit* main window. This script is based on a script made by Jimmy Abualdenien, and has been steadily updated with more features.

5.2.1 Dynamo Nodes

The general arrangement of the script can be structurally and visually divided into three parts (see Figure 5.2): the input nodes to the left, the `Base Script` in the middle and the following python scripts to the right. Each part fulfills its own purpose. Additionally, there are `Family Type` nodes and the `Type Activation Script` to the bottom, where necessary family types are gathered and activated. The input nodes specify, as the name may suggest, the parameters which are either set by the user or are calculated on their own based on other parameters. All of these inputs are lead into the `Base Script` where they are gathered, processed and forwarded to the remaining scripts. There, they are again processed and forwarded.

As explaining the script in its entirety would take too long and contain redundant information, the following subsections will only give a detailed overview of the functionalities of three parts of the script, while only briefly explaining the rest of the script. The three parts are the `Platform Length Calc Script`, the `Platform Width Calc Script` and finally the

`Escalator Create Script`, as these contain the most interesting implementations of the regulations.

Length Script

The Length of the platform is mainly determined by the maximum length of the trains halting at the station. Added to that are values such as the buffer of five meters (813.0201, 2012, p. 7) and bumper-length (when the station is a terminus). Lastly, if selected, the length is raised to the next standard length, according to the regulations (813.0201, 2012, p. 7). So first of all, we add our values to the train length, dependent on whether we have a terminus or not:

Length Script

```
if terminus == False:
        L2 = length + buffer
else:
        L2 = length + buffer + bumper
```

Subsequently, we can set the standard lengths with nested `if statements`:

Length Script

```
if standard_length == True:
        if length >= 60 and length < 90:
                length = 90
        elif length >= 90 and length < 140:
                length = 140
        .
        .
        .
        elif length >=370 and length < 405:
                length = 405
```

The length then gets output to the `Base Script`.

Width Script

In this script, the `platform width` is determined by a number of factors such as the `platform height`, the maximum velocity and traffic level parameters. Based on those, as well as some determination tables, a minimum width is calculated (813.0201, 2012, p. 9ff). Note that the 'installation dimension' is usually based on many more parameters such as cant and radius (813.0201, 2012, A03 p. 6), though since these are out of the scope of this thesis determining the dimension is significantly simpler.

Minimum Width

```
# choose installation dimension based on platform height
if pltfrm_height == 0.960:
        aBa = 1.685
elif pltfrm_height == 0.760:
        aBa = 1.675
elif pltfrm_height == 0.550:
        aBa = 1.670
elif pltfrm_height == 0.380:
        aBa = 1.690

if v <= 160:
        bs = 2.5 - aBa
elif v > 160 & v <= 200:
        bs = 3 - aBa
else:
        bs = 3.7 - aBa
# choose the greater of the two
if 2*bs < 3.3:
        platform_width = 3.3
```

Additionally, the script checks whether the large obstructions (escalators, stairs and elevators) actually fit on the platform under consideration of the minimum distance to the edge of the platform. If not, the value is raised until they do. The function below measures the dimensions of objects to be placed on the platform and widens it should there not be enough space. In this way we ensure that the platform is neither too small nor unnecessarily wide. Besides the distance to the edge, the distance to the danger zone is also taken into account. These values change depending on he dimensions, especially the length, of the object (813.0201, 2012, A04 p.3ff). A larger object, for example, required a greater distance to the edge as well as the danger zone:

Object Check

```
# length, width: dimensions of the object
# dist: the distance of the object to the edge of the platform (if
    placed in the middle)
# bs: width of the danger zone
def checkDistances(length, width, bs, platform_width):
        if length < 1:
                dist = 2*bs + 2*0.9
                if dist < 1.6:
                        dist = 2
                if platform_width < (width + dist):
                        platform_width = (width + dist)
        if length > 1 and length <= 10:
                dist = 2*bs + 2*1.2
                if dist < 2:
                        dist = 2
                if platform_width < (width + dist):
                        platform_width = (width + dist)
        if length > 10:
                dist = 2*bs + 2*1.6
                if dist < 2.4:
                        dist = 2.4
                if platform_width < (width + dist):
                        platform_width = (width + dist)
        return platform_width
```

Base Script

The `Base Script` handles all incoming input parameters and reroutes them to the correct script as well as placement of the basic structures. First, before using numerical values as dimensions we need to convert our values to internal units. The *RevitAPI* uses feet as its standard unit of measurement. In order to convert from meters to feet wen can either divide by `3.2808` or use the included tool:

Unit Conversion

```
revitApiLength = UnitUtils.ConvertToInternalUnits(length,
    DisplayUnitType.DUT_METERS)
```

Along with the unit conversion, the script also handles the deletion of all elements when the *Dynamo* script is executed. This assures when changing parameters before a run, the "old" model gets discarded and only the "new" one is built. Lastly, the script builds basic structures like levels, floors and the surrounding walls. These structures are placed according to the dimensions `width` and `length`, the former depending on the number of track lines the user specified in the `Number Of Track Lines` node, and the latter being determined by the `length` script. Lastly, output variables are compiled into lists, catering to the following scripts needs, and declared as output.

Escalator Create Script

The `Escalator Create Script` handles the placing of escalators, stairs and elevators on the platforms and between the levels. Based on user inputs such as `Number Of Escalators`, `EscalatorYOffset` etc., it evaluates the position's compliance with the regulations and, if needed, overwrites the position for it to be compliant. The script iterates over the number of platforms in the model, as well as the number of escalators and stairs we have. Below, we will take a closer look how this is calculated and achieved.

Firstly, let us view the user-specified parameters:

- `NumberOfEscalators` refers to the number of escalators that will be placed on one side of the platform. Can take on a value from 1 to 3.
- `EscalatorYOffset` refers to the value that the escalators (and stairs) will be offset by in y-direction. A negative value means that is will be moved down, while a positive value means the opposite.
- `NumberOfStairs` is the same as `NumberOfEscalators`, just for stairs. It can take on a either 0 or 1. Instead of increasing the number of stairs, the step width was adjusted in the `Base Script`.
- `Escalators facing inwards` is a `boolean` which determines if the escalators on each side of the platform will face inwards or outwards. In our example, it will be the former.

Additionally, we also have some parameters determined by the general context. These include:

- `frontEnd`, the y-coordinate of an inner platform edge.
- `backEnd`, the y-coordinate of an outer platform edge.
- `minDistEsc`, the minimum distance an escalator or stairs must have to the edge of the platform.
- `revitApiEscalatorWidth`, the width of our escalators.
- `revitApiStairWidth`, the width of our stair steps.
- `escY`, the y-coordinate of the escalator.
- `UpperEscMoved`, a flag set when the upper escalator had to be moved.

In order to find out how to do our calculations, we must first consider how exactly an object is placed at a given point. When placing an escalator on a cartesian grid, the origin point of that specific family type obtains these coordinate values. With our escalator family type the origin point is at the lower left corner, as seen in Figure 5.3. Around this point, we rotate

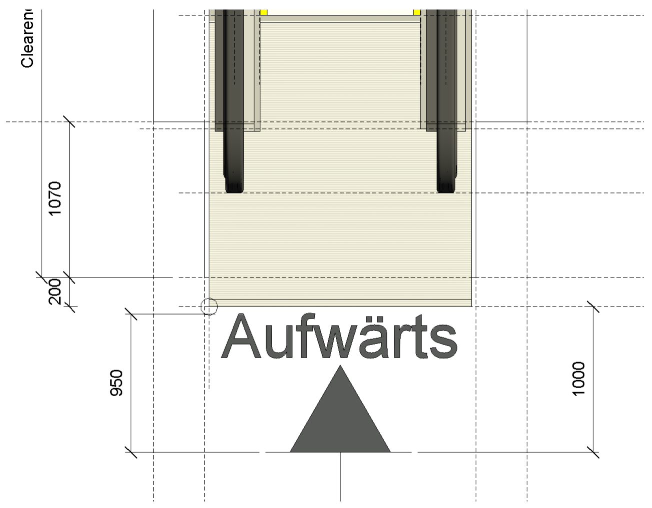


Figure 5.3: The origin is at the lower left corner, as indicated by the circle

our object by 90°counterclockwise, and can then calculate the distance to the inner edge of the platform (`frontEnd`). This can be done by simply subtracting the `frontEnd` from `escY`. This is only done with the first escalator (n == 0), because the distance which the objects need to be moved stays the same for every iteration and can simply be saved for the next escalator or stair. Should the calculated distance be smaller than the minimum distance, we add what is missing. Lastly, we set the `UpperEscMoved` flag to `True`, so that the next iteration (which handles the next escalator) knows that it needs to add the difference:

```
if UpperEscMoved: #this will never trigger during the first iteration
        escY = escY + difference
if n == 0:
    if (escY - frontEnd < minDistEsc):
            difference = (minDistEsc - (escY - frontEnd))
            escY = escY + difference
            UpperEscMoved = True
```

For the outer edge of the platform (`backEnd`) it is not as trivial, since the escalators and stairs get placed from "bottom to top". This means that we need to add the widths of the escalators and stairs to our `escY` value:

```
if backEnd - (escY + (numberOfEscalators*revitApiEscalatorWidth +
   numberOfStairs*revitApiStairWidth)) < minDistEsc:
        difference = (minDistEsc - (backEnd - (escY + (
            numberOfEscalators*revitApiEscalatorWidth + numberOfStairs*
            revitApiStairWidth))))*(-1)
        escY = escY + difference
        UpperEscMoved = True
```

As we now have a y-coordinate, which is compliant with our regulations, we can create our cartesian point and place the escalator. Note that we need to add a `orientationOffset`, which takes on a value of 0 when the escalators face inwards and `revitApiEscalatorWidth` when they are facing outwards. This is because rotation around a corner of the object will offset it by its width when rotated by another

180°counterclockwise. Then, the `Top Offset` is defined in order to reach the height of the next level.

```
escPoint = XYZ(escX, escY + orientationOffset, revitApiplatformHeight +
    levelnr * revitApiWallHeight)
escalator = doc.Create.NewFamilyInstance(escPoint, escalatorType,
   levelList[levelnr],
    Autodesk.Revit.DB.Structure.StructuralType.NonStructural)
escalator.GetParameters("Top␣Offset")[0].Set(revitApiWallHeight)
```

Similar to the `orientationOffset`, one must consider that on the other end of the platform the escalators and stairs are always rotated by an extra 180°counterclockwise, whether they are facing inwards or not. In order to correct the error caused by rotation, the script must again check for the orientation and add the escalator width again:

```
if escOrientation:
        escY = escY + revitApiEscalatorWidth
else:
        escY = escY − revitApiEscalatorWidth
```

After the placement of the escalators has been finalized, we can add the stairs, a wall and an opening in the ceiling in direct dependence to the escalators, as well as iterate over all levels in order to connect them to each other.

Track Line Create Script

The `Track Line Create Script` handles the creation of the track lines by positioning tunnel parts adjacent to each other along the x-axis (dependent on the length of the platform), then iterating across the number of track lines in y-direction as defined by the input. For the middle track lines that are surrounded by platforms from both sides, two get placed in direct succession. This mimics real train stations, where a train can only be accessed by one platform.

Column Create Script

In the `Column Create Script` the columns get placed dependent on how many rows there should be on one platform, as specified by the user. One thing to consider here is to not place columns where the escalators, stairs and elevators are. The `Elevator Position` needs to be checked before placement, as well as the x-axis placement of the escalators. The columns are inserted along the x-axis by iterating over the `tunnelPartCount`, so a list of indices is created, ranging from 1 to `tunnelPartCount`. Then, the appropriate indices get deleted before the `for loop` where the columns get placed.

Platform Create Script

Here, the platform are created as simple floors with the dimensions `length` and `platform width` as well as offset upwards according to the `platform height`.

1+x Level Create Script

This script handles everything that gets placed at the levels above the first one, with the exception of the level-connecting escalators and stairs (since that is more easily done in the `Escalator Create Script`) as well as the floors and wall (handled by the `Base Script`). The user has a few `Number Slider` to adjust the positioning and the count of the exits/entrances, as well as adjust the position of some objects like ticket machines, ATMs and ticket validators. These objects only get placed at the top most level.

Debugging

Dynamo does not allow direct console output, which makes debugging slightly more difficult. Though `print` statements can be added without problem, one does not see their output directly when pressing "run". This can be circumvented by adding a `File Path` pointing to a `.txt` file as well as a few lines of code to the corresponding script:

```
stdout_original = sys.stdout
stdout_file = open(filepath, 'w')
sys.stdout = stdout_file
# insert print statements here
sys.stdout = stdout_original
stdout_file.close()
```

This causes every `print` statement to write to the specified file, making it easier to understand why certain errors happen when running. Apart from this, bugs and errors in the code need to be found and inspected manually by repeatedly running the script and checking the model in the main window. *Dynamo* supports automatic script running, though considering the size of the script and its load time as well as the scale of the resulting model, running the script manually when necessary is a better if less convenient way of checking results. Of course, when merely working on one specific part of the script it is redundant to run the entire code if a developer is checking for escalator compliance nothing about the elevators should have to be executed. As convenient as these sorts of tests would be, the *RevitAPI* wrapper for Python is only available within *Dynamo*, which means that running tests on external python files would not work.

Chapter 6

Variations

In this chapter variations of the model will be presented. All of the variations were generated using the U9-TrackLine script and the images include the user parameters necessary to create it. The images will start with models of the first floor and will gradually show the following floors. In Figure 6.1 we see the general layout of the first floor, complete with platforms, columns, track lines as well as escalators, stairs and elevator shafts.

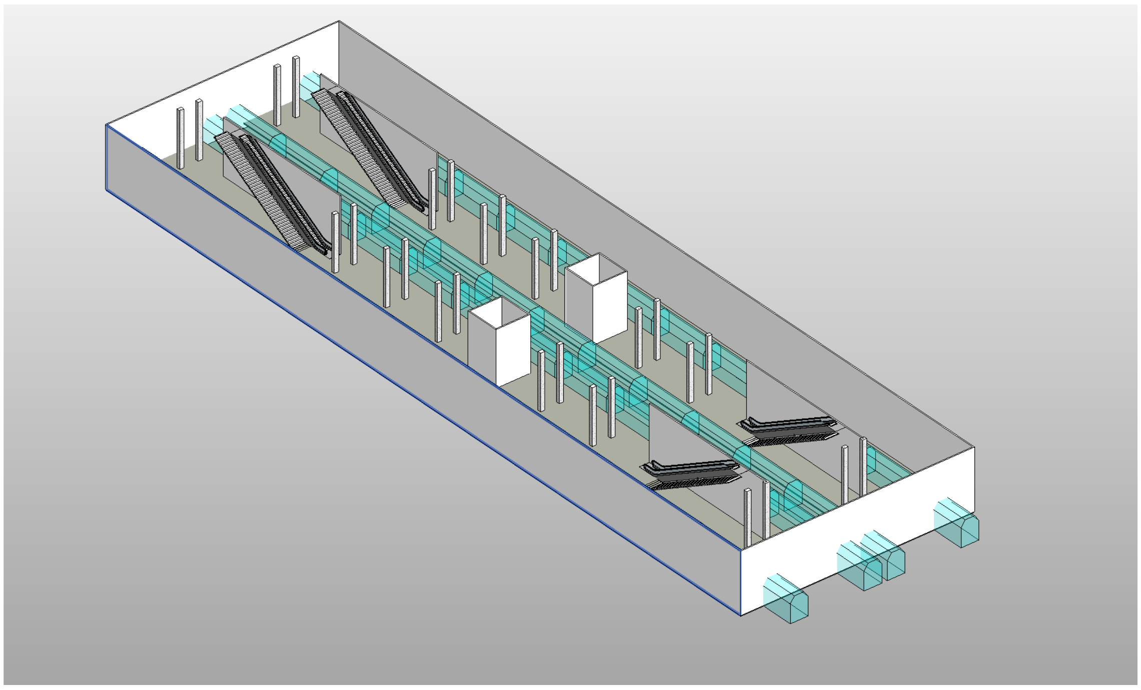


Figure 6.1: General layout of the first floor.

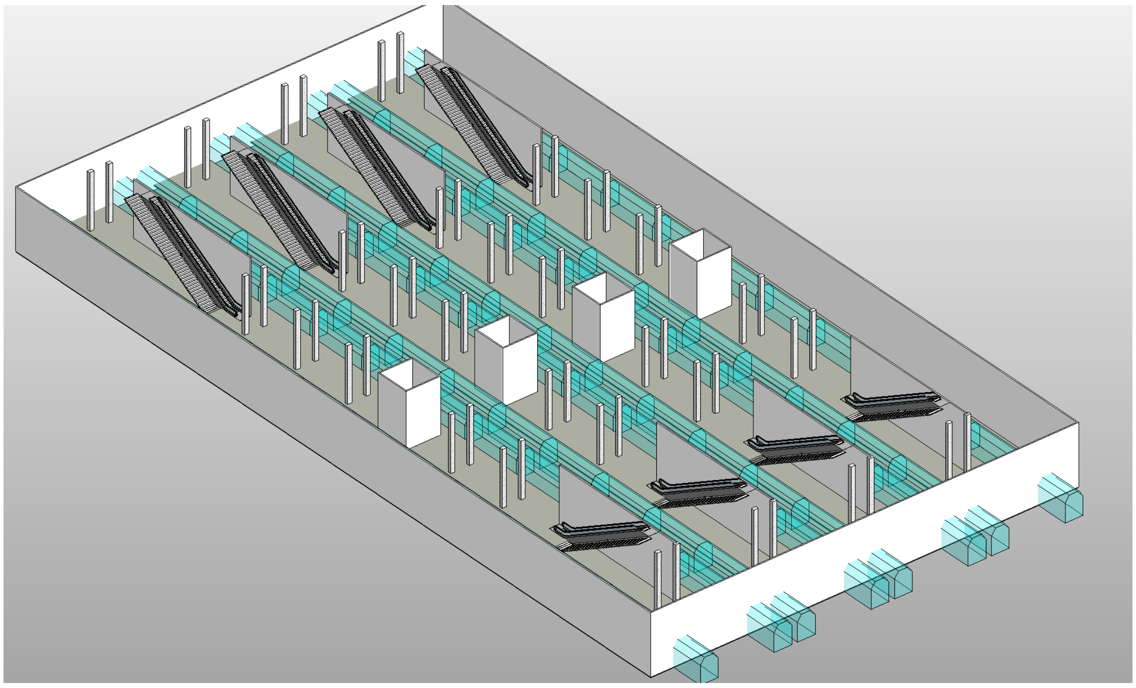


Figure 6.2: `Nr Of Tracklines = 5`, The track line count can be increased and more platforms will be created accordingly. Note that double track lines are counted as one.

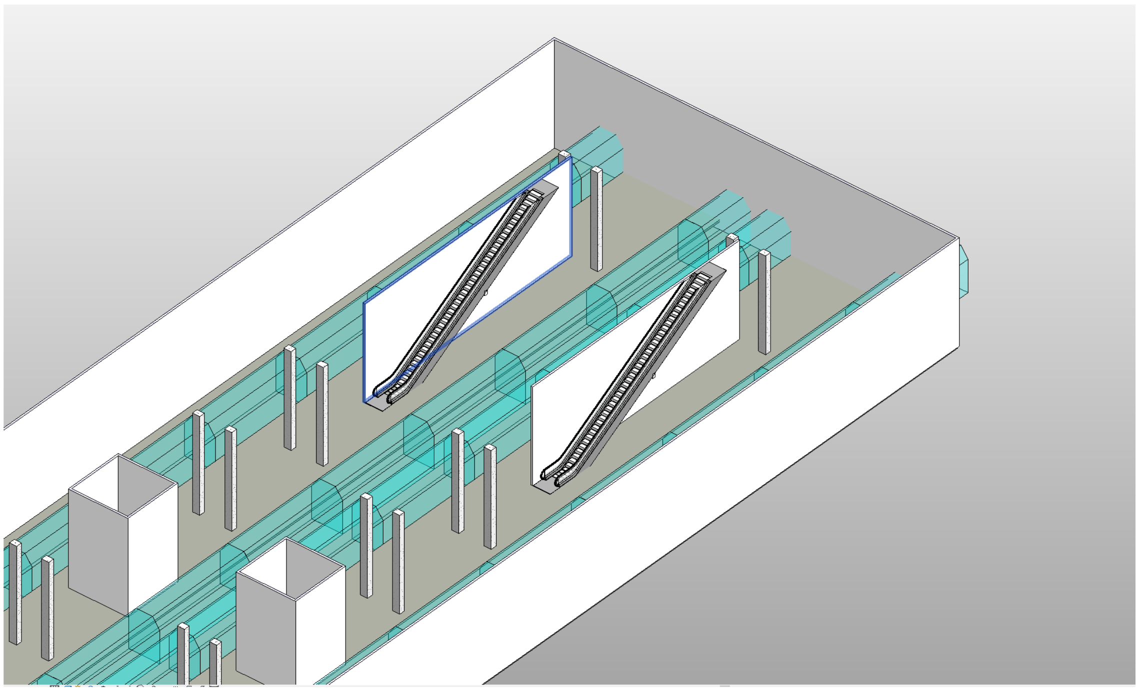


Figure 6.3: `Nr Of Escalators = 1, Nr Of Stairs = 0`, A more detailed view of the single escalators together with the walls.

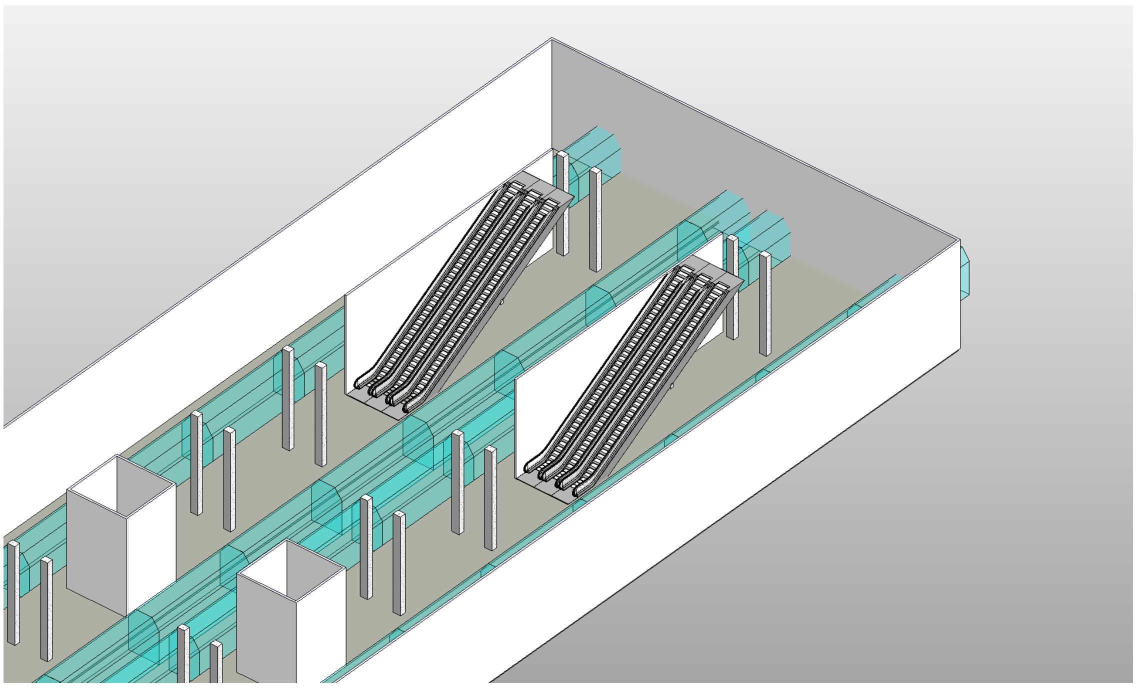


Figure 6.4: `Nr Of Escalators = 3, Nr Of Stairs = 0`, The escalator count can be increased to any value between 1 and 3.

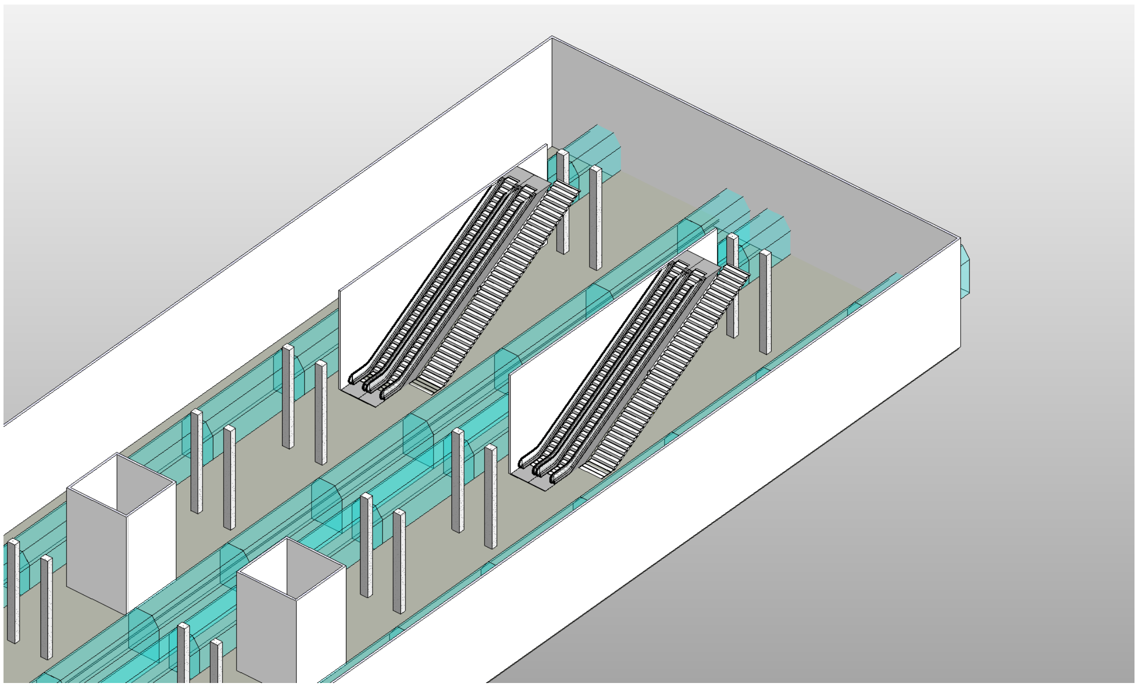


Figure 6.5: `Nr Of Escalators = 2, Nr Of Stairs = 1`, As well as escalators, stairs can also be placed in the same manner.

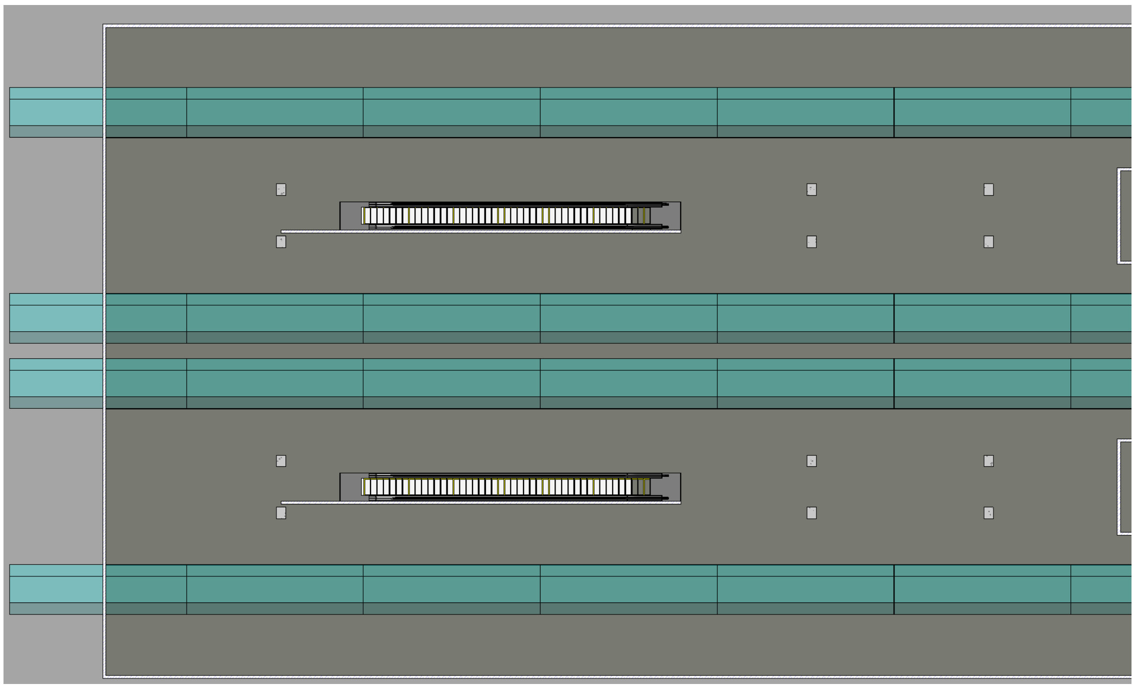


Figure 6.6: `EscYOffset = 0`, A top-down view of a single placed escalator.

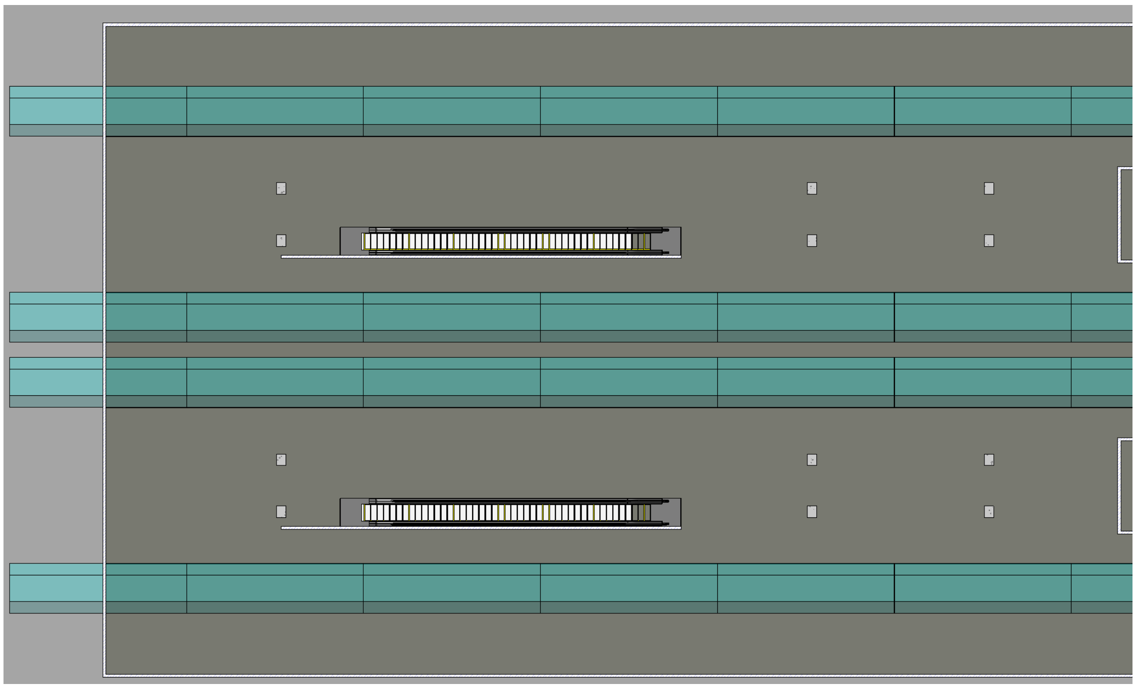


Figure 6.7: `EscYOffset = -10`, The specified offset toward the south of the model should have placed the escalators far from the desired platform, but the logic in the script overwrote the "invalid" parameter.

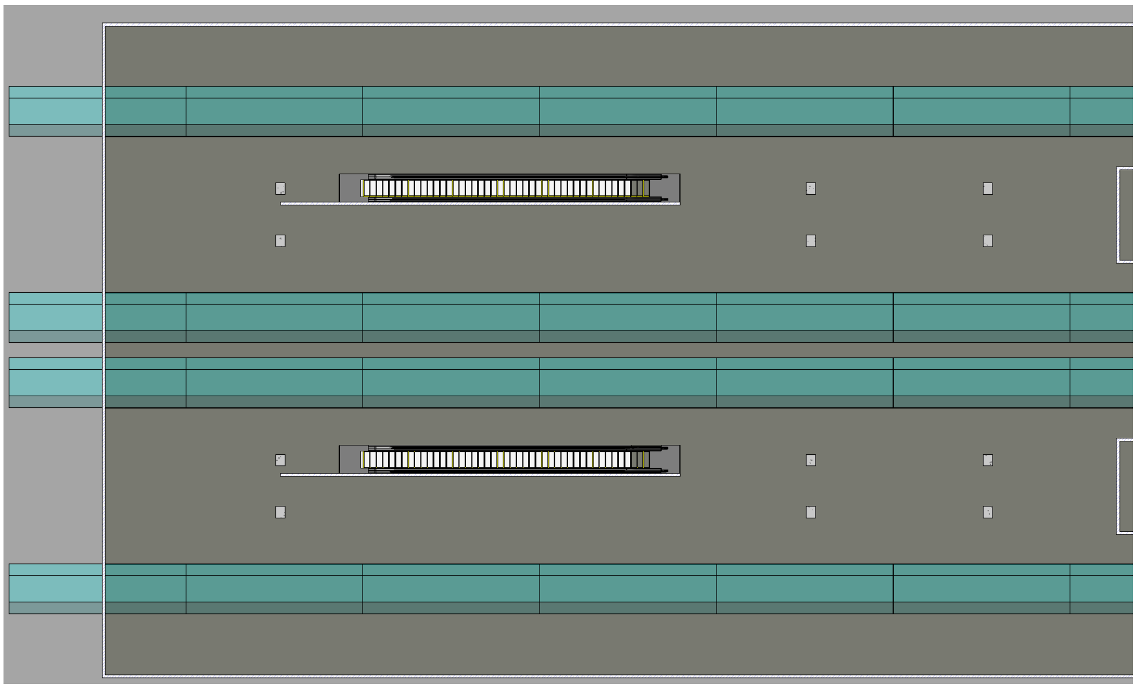


Figure 6.8: `EscYOffset = 10`, Similar to Figure 6.7, just in the opposite direction.

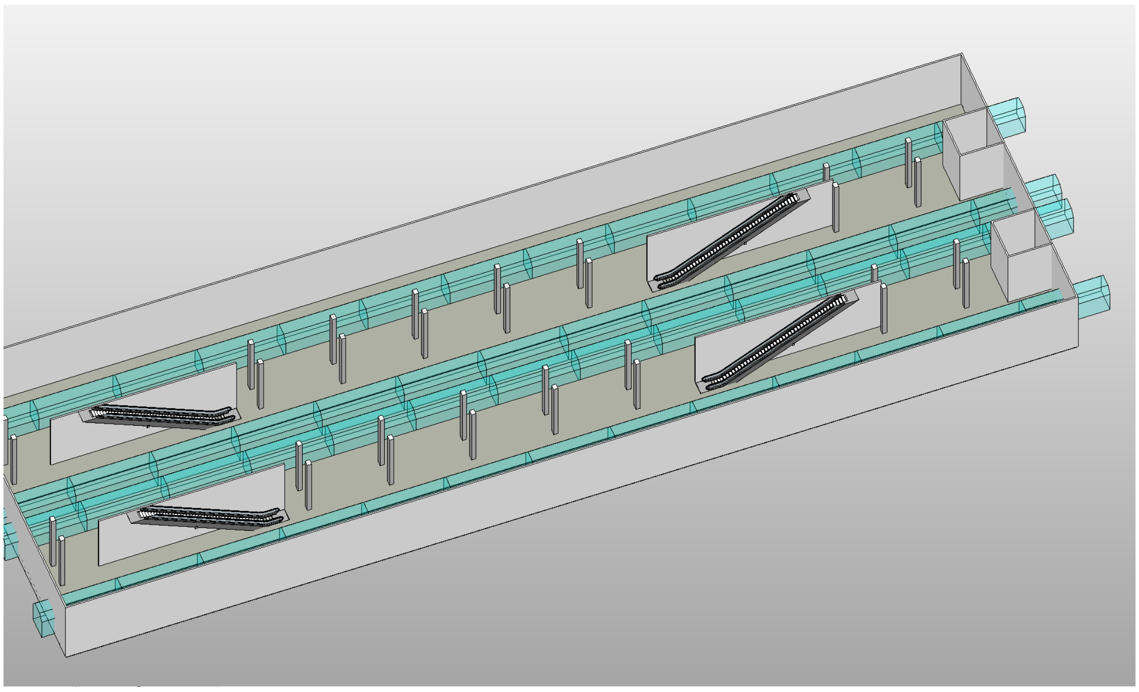


Figure 6.9: `Elevator Position = 1`, The elevator shafts can be placed in the middle or on the ends of the platform.

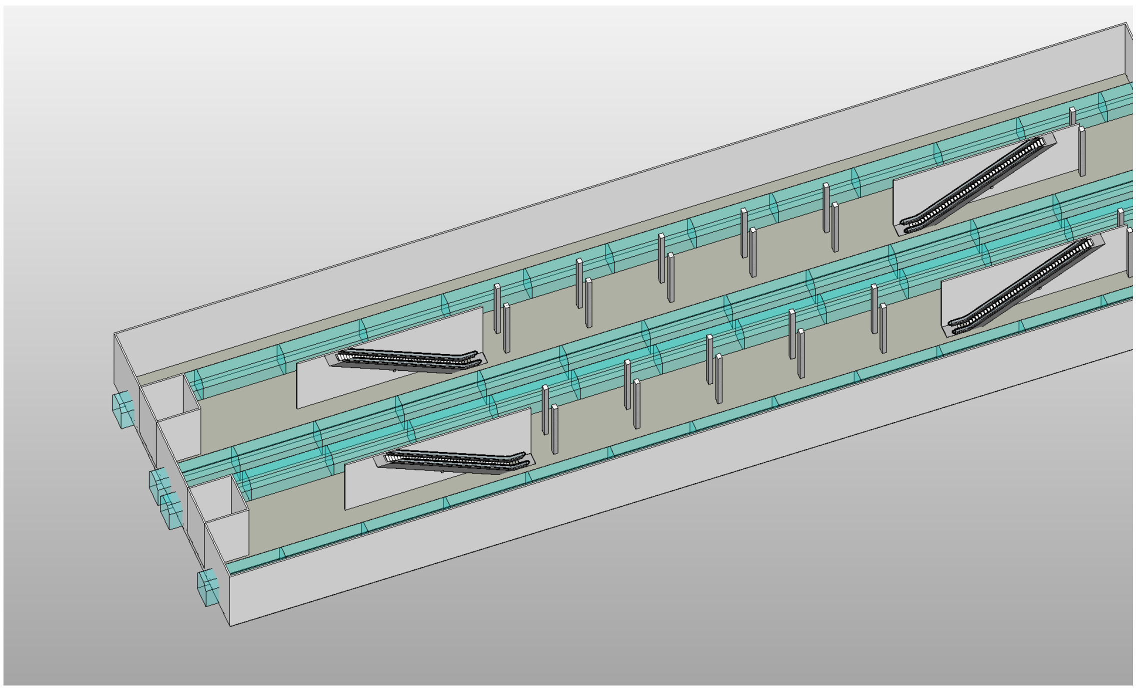


Figure 6.10: `Elevator Position = 3`, Similar to Figure 6.9, with the elevator shafts at the opposite end.

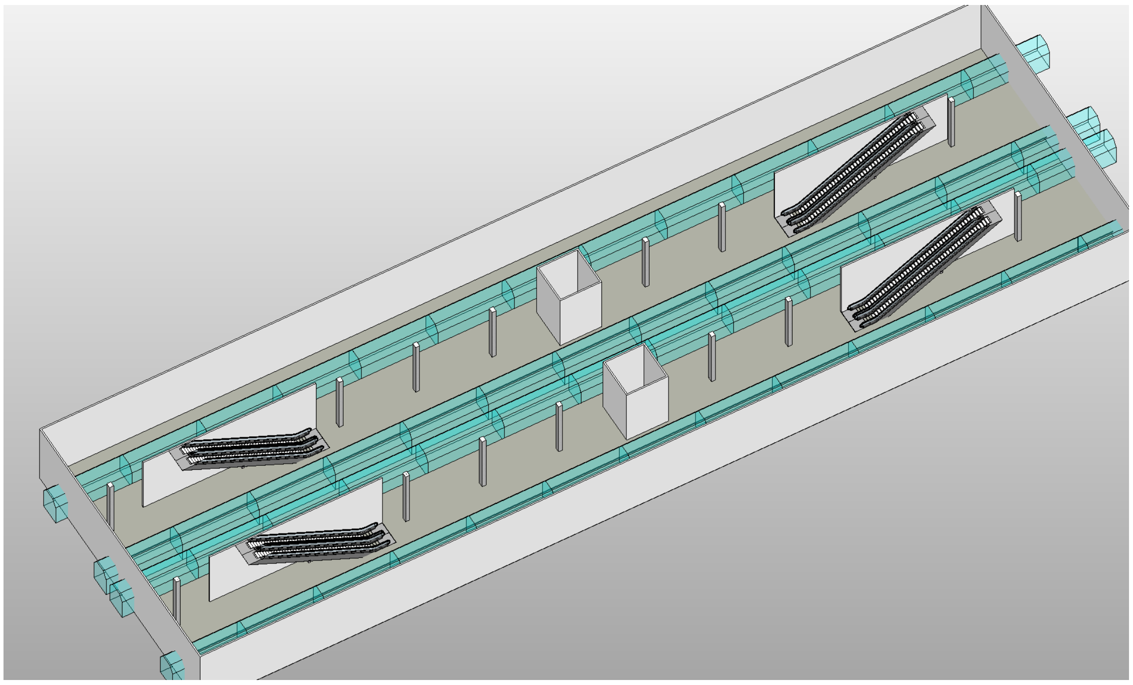


Figure 6.11: `Nr Of Column Rows = 1`, The column count can be adjusted to either 1 or 2 rows on a single platform.

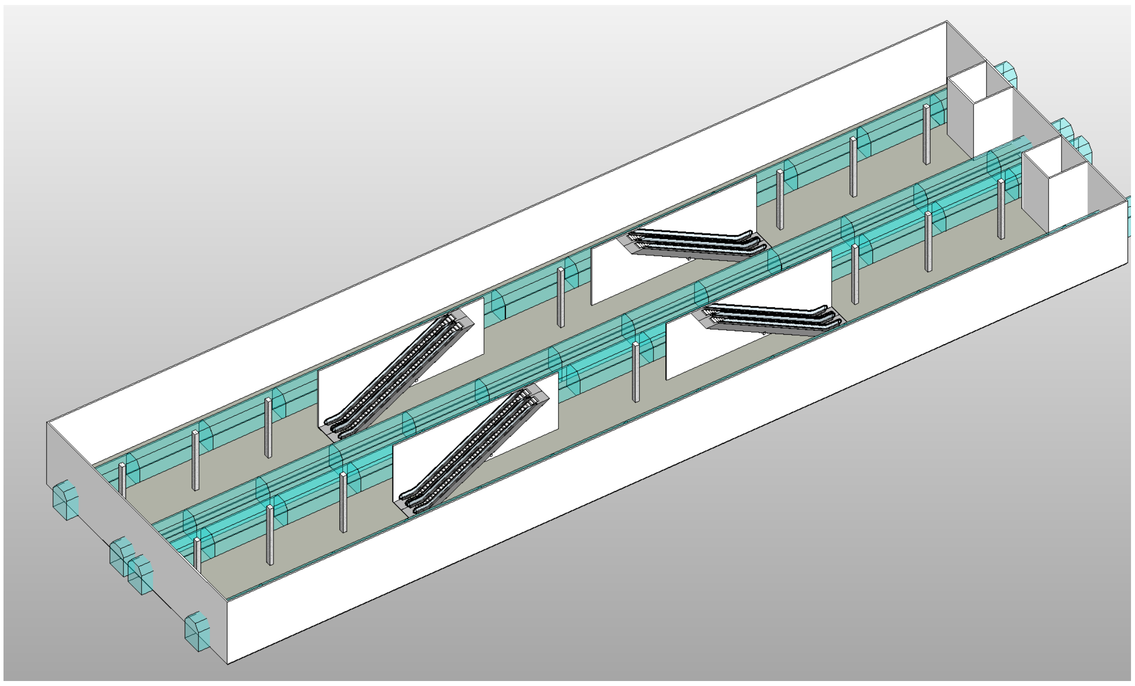


Figure 6.12: `Escalators facing inwards = False`, This changes the orientation of the escalators and stairs, as well as adjusting their position accordingly. The elevator shaft gets moved to the end of the platform should the user have specified a "middle" placement.

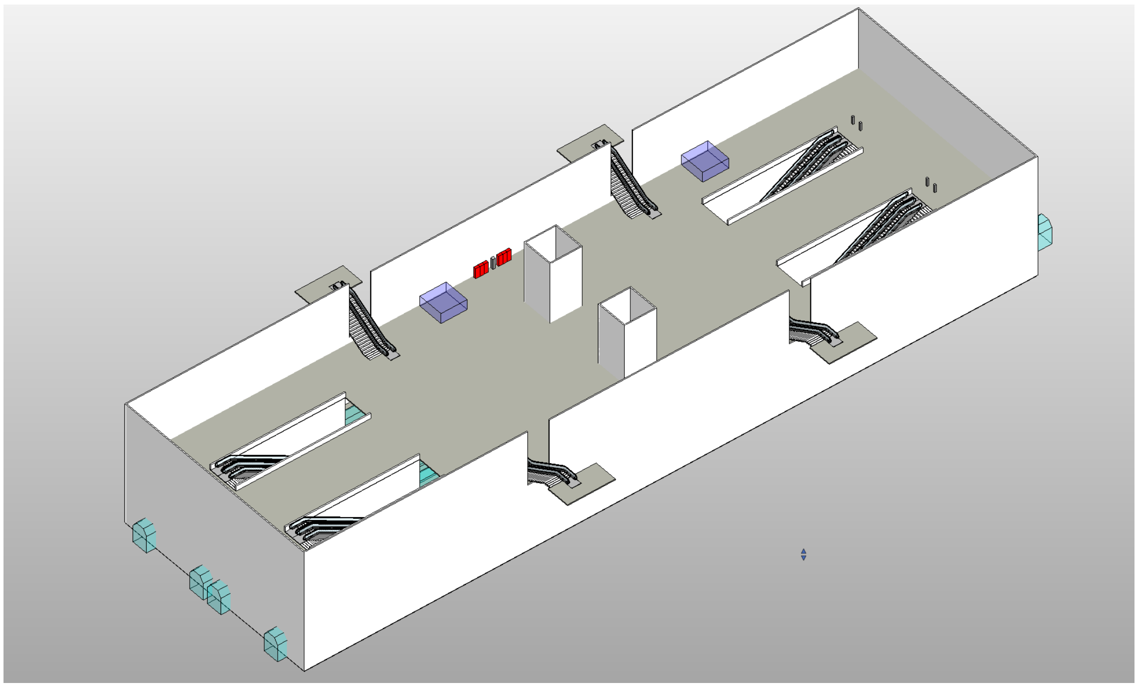


Figure 6.13: `Nr Of Levels = 2`, The general layout of the topmost floor, which in this case is the second.

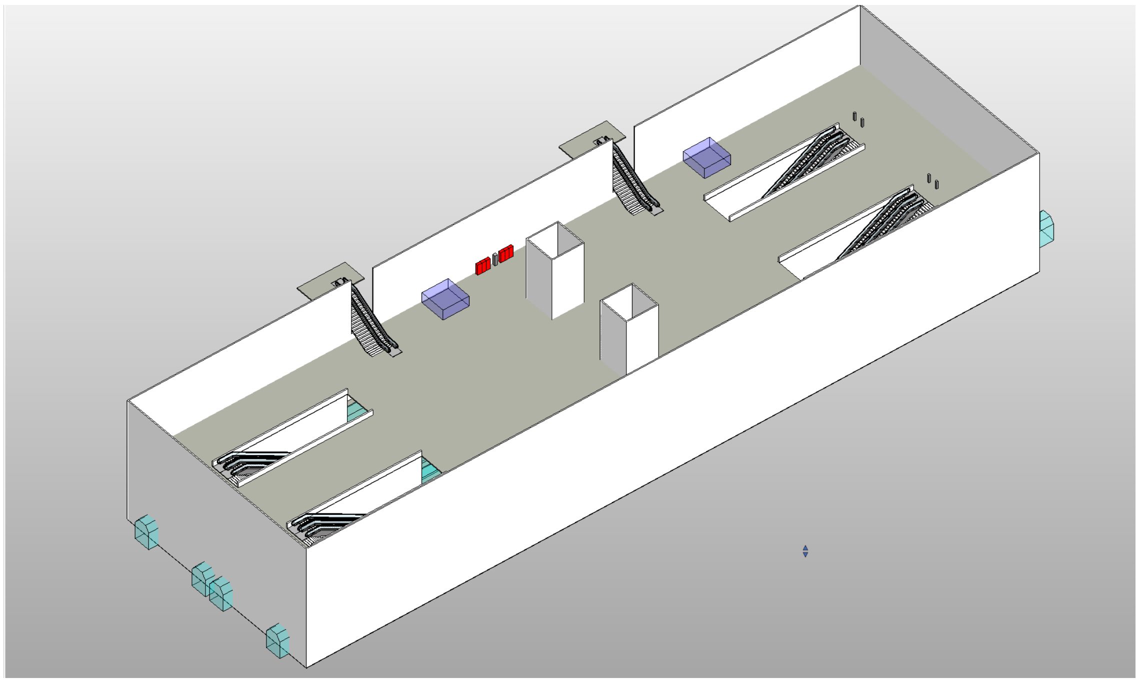


Figure 6.14: `Nr Of Exits = 2`, The number of exits can be changed to any value between 1 and 4. Slabs are added to the end of the exits to declare the destinations for pedestrian simulation.

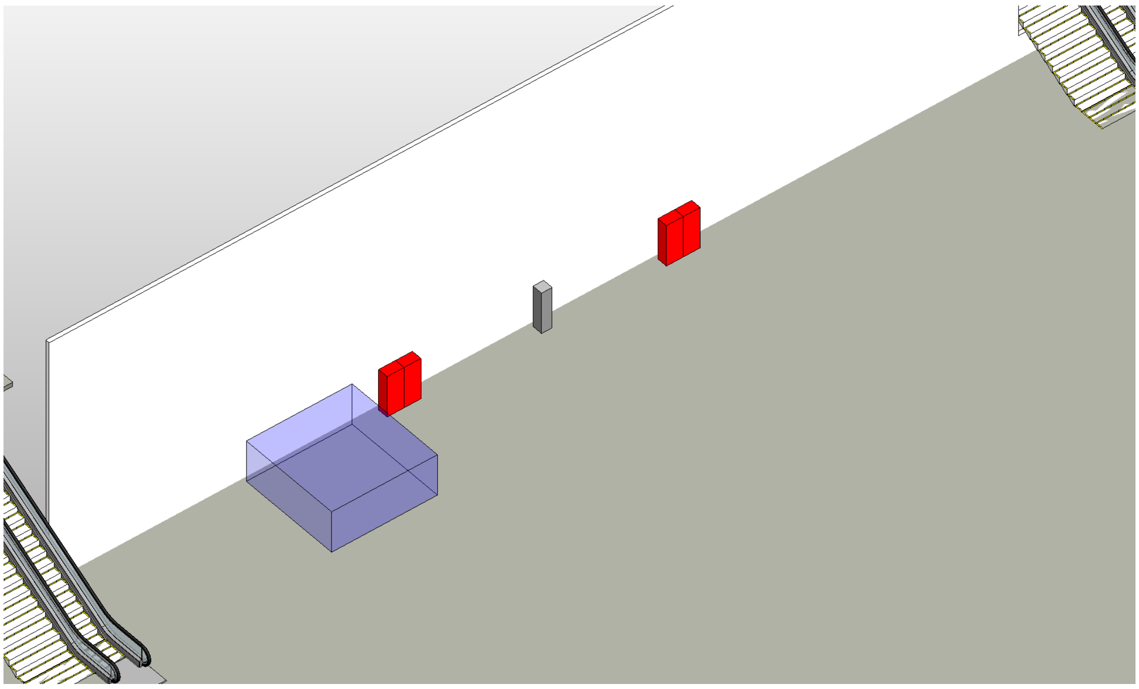


Figure 6.15: `Ticketmachine X Offset = 7, Nr Of Ticketmachines per Exit = 2`, A closer view of the ticket machines (red) which can have their position and count adjusted as well as the ATMs (grey) and storebooths (blue).

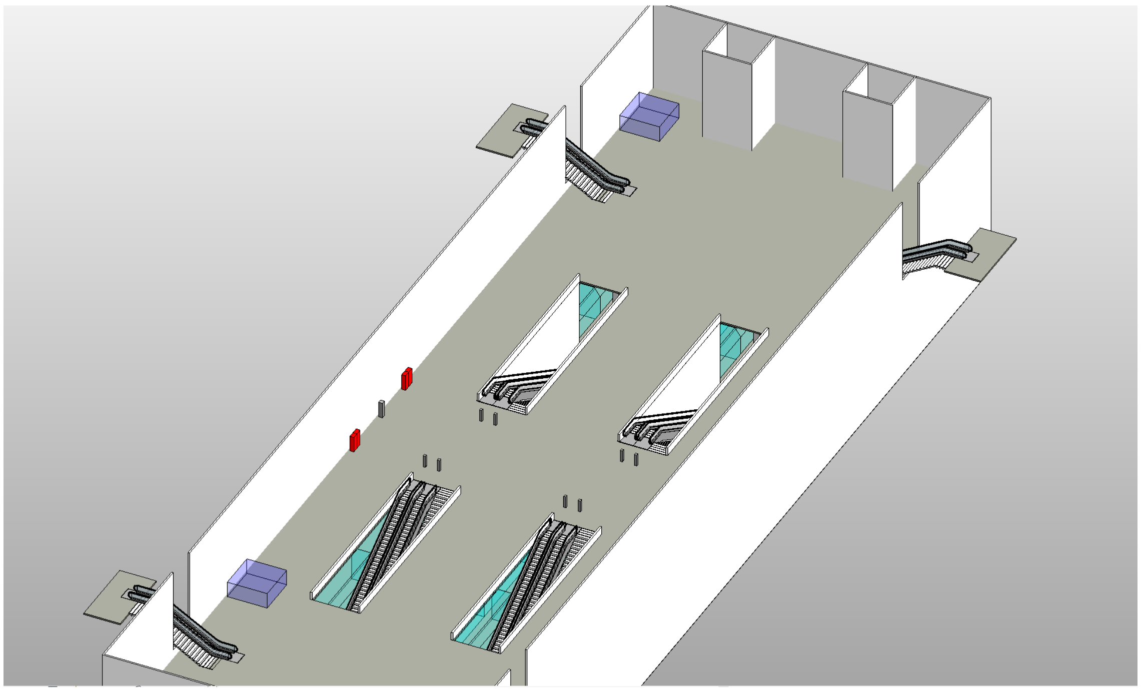


Figure 6.16: `Escalators facing inwards = False`, General layout of the top floor with the escalators and stairs facing outwards.

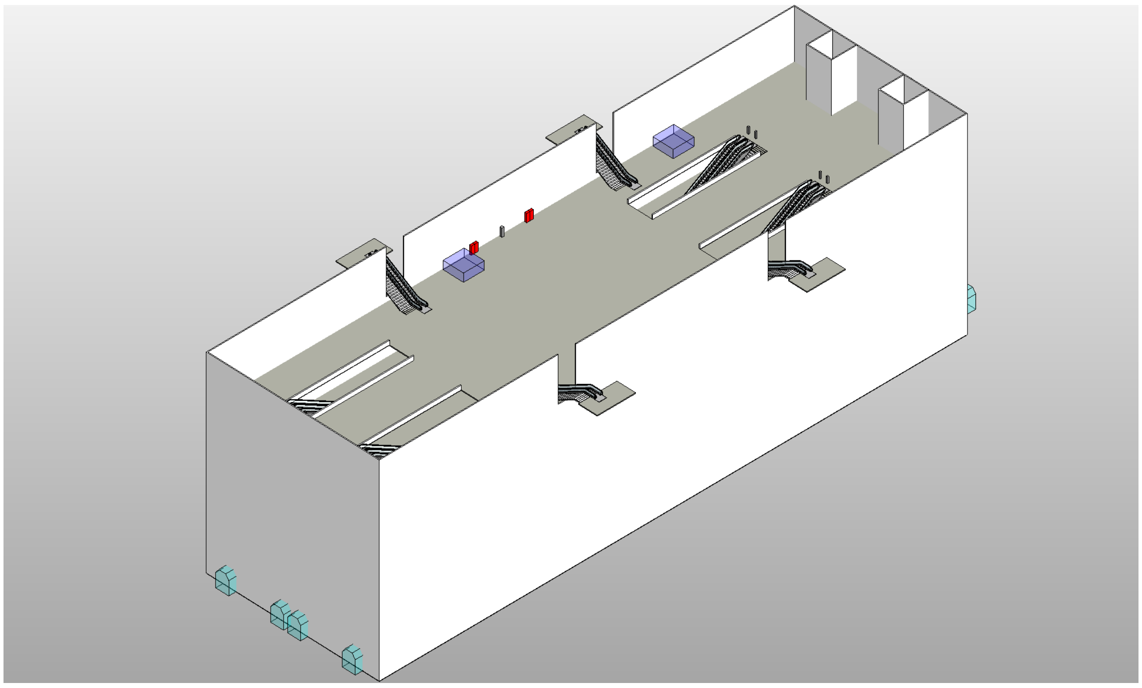


Figure 6.17: `Nr Of Levels = 4`, The level count can be changed to any value larger than 2. The top floor will always contain the exits, ATMs, storebooths and ticket machines.

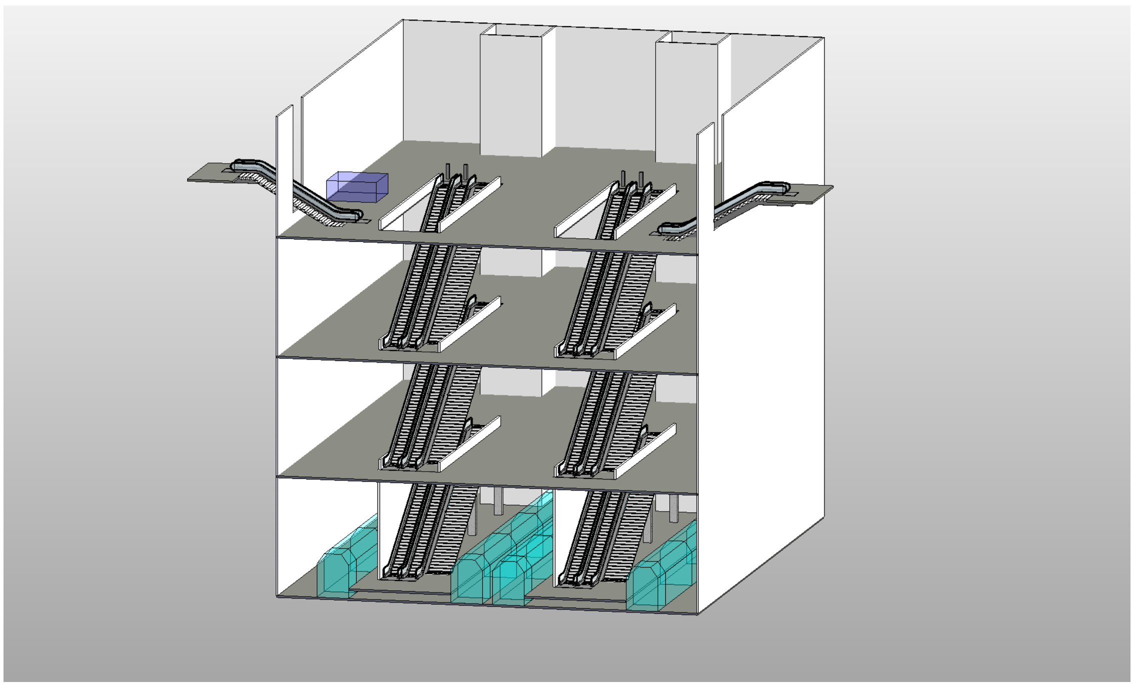


Figure 6.18: `Nr Of Levels = 4`, Regardless of the number of levels, the floors always remain connected.

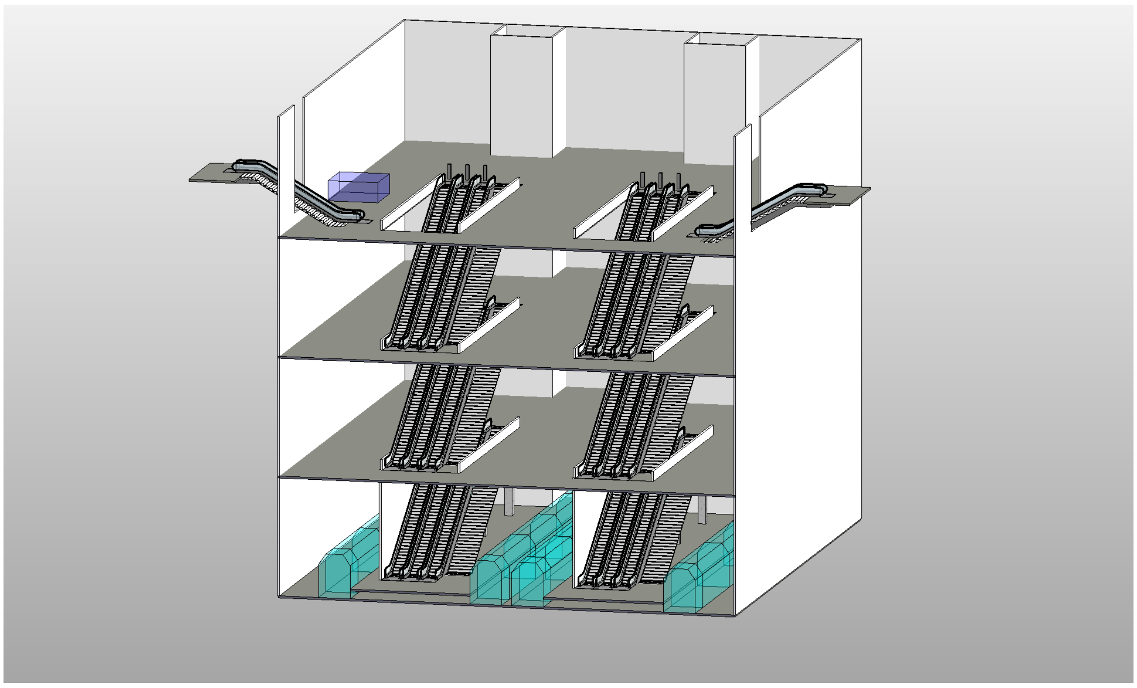


Figure 6.19: `Nr Of Levels = 4, Nr Of Escalators = 3`, The number of escalators is the same for all connecting escalators and stairs.

Chapter 7

Limitations

7.1 Translation

As with any technology, this approach to generative design is far from perfect in all ways. When considering the limiting factors, the first thing that comes to mind is the translative process between natural language regulation document and code. In order for this process to be consistent and of high quality certain conditions must be met:

1. The "translator" must have a deep understanding of the regulations and their applications.
2. Extensive programming skills are required.
3. There must be constant communication with the institution issuing the regulations to eliminate misunderstandings.
4. An independant review as to ensure the quality of the final product must be established.

As mentioned in chapter 3, natural language processing can eliminate the need for a "translator", given that the processor is consistent and of high enough quality itself. Since NLPs were out of the scope of this thesis, it is difficult to gauge the implications they could have on this issue.

7.2 Software

Even though Generative Design is becoming increasingly popular as more research on it is done and even companies like Autodesk incorporating it into their software catalog, the real-world implementation is seemingly still in need of further development. As capable as *Refinery* is in *Revit*, its possibilities of application are limited. The script developed and explained in chapter 5 could not be used to create a study in *Refinery* and automatically generate variations of the model. Even though the script almost exclusively uses `Number Slider` and `Booleans` and input nodes, which can also be declared as `Input` for *Refinery*, no output nodes can be declared since *Refinery* only supports `Watch` nodes filled with a numerical value as `Output`. While this is useful for maximizing and minimizing surface areas, heights and other "simpler" parameters, the sometimes quite complex regulations implemented into the script cannot be used in *Refinery*, or at least not without significantly more effort.

Chapter 8

Conclusion and Future Research

As resource efficiency becomes more important, architects and engineers need to consider every possible approach as an opportunity to increase the quality of their building. What was unachievable in the past, where every possibility had to be thought of, sketched and evaluated, today's technology allows for Computer Aided Design to reach new levels. Generative Design and Generative Adversarial Neural Networks allow for designs to be automatically generated and assessed in quantities so large that they might have been seen as impossible in a not so far past. As discussed, this is especially important for the early stages of design where decisions have the most impact. Additionally, the early stages are a great opportunity to implement the approach because of the level of detail, or even the lack thereof, that early-stage design has.

As for possible future research and work, the problem of translating between natural language and code is one that must be addressed. As useful as natural language processing and even natural language programming are, they create far too much overhead for the issue at hand. Creating regulations, textualising them in a regulation document only to then process it to code is too lengthy and unnecessary of a procedure. Instead, a framework with which regulations can be directly noted in a computer-readable file (for example JSON) that can then be loaded by a script would prove far more useful and consistent. Of course, this would have to be an industry-wide standard for it to be applicable in any way. Furthermore, a fully-fledged design process with Generative Design at its core would most likely need a considerable amount of computing power, especially with Generative Adversarial Neural Networks. Developing a web-based application for it with suitable server infrastructure could serve as a means to generate the designs remotely without having to occupy the local machine. Together with the industry-standard framework discussed above, multiple companies and engineering offices could access the same remote "design generating" service since everyone uses the same standard. Despite the fact that there are likely to be more aspects which need to be researched to a greater extent, Generative Design today is already capable of assisting during the design process. The urgency is to address the immediate limitations of this technology right now because the implications that Generative Design has on the future of the industry are both paramount and crucial.

Appendix A

Files

All necessary files to run the Dynamo script are included in the Sync&Share folder "Bachelorthesis Benedict Harder". Revit 2021 is required.

References

813.0201. (2012). Bahnsteige konstruieren und bemessen. Valid as of 1. May 2012, Internal DB document, Authored by Brantzko, A., Appendices included.

813.0202. (2012). Bahnsteigzugänge konstruieren und bemessen. Valid as of 1. May 2012, Internal DB document, Authored by Brantzko, A., Appendices included.

ABUALDENIEN, J. & BORRMANN, A. (2019). A meta-model approach for formal specification and consistent management of multi-lod building models. *Advanced Engineering Informatics*, *40*(1474-0346), 135–153. doi:10.1016/j.aei.2019.04.003

ABUALDENIEN, J., CLEVER, J., BORRMANN, A., PLATT, A., KNEIDL, A., & SIMON, S. (2020). *Distansim: Implementation of social distancing in pedestrian simulation*. Technische Universität München.

ABUALDENIEN, J., PFUHL, S., & BRAUN, A. (2019). Development of an mvd for checking fire-safety and pedestrian simulation requirements. In *Proc. of the 31th forum bauinformatik*. Berlin, Germany.

ABUALDENIEN, J., SCHNEIDER-MARIN, P., ZAHEDI, A., HARTER, H., EXNER, H., STEINER, D., . . . SCHNELLENBACH-HELD, M. (2020). Consistent management and evaluation of building models in the early design stages. *Journal of Information Technology in Construction (ITcon)*, *25*(13), 212–232. doi:10.36680/j.itcon.2020.013

ACCU:RATE. (2021). Our software crowd:it. https://www.accu-rate.de/en/software-crowd-it-en/. Accessed: 20/04/2021.

AUTODESK. (2021a). Generative design primer. https://www.generativedesign.org/03-hello-gd-for-revit/03-03_running-gd-for-revit. Accessed: 17/04/2021.

AUTODESK. (2021b). Revit developer center. https://www.autodesk.com/developer-network/platform-technologies/revit?us_oa=dotcom-us&us_si=09ebc770-8627-4809-b75a-10da6ba51117&us_st=revit%20api. Accessed: 19/04/2021.

AUTODESK. (2021c). Revit overview. https://www.autodesk.com/products/revit/overview?term=1-YEAR. Accessed: 12/04/2021.

AUTODESK. (2021d). What is dynamo. https://primer.dynamobim.org/en/01_Introduction/1-2_what_is_dynamo.html. Accessed: 19/04/2021.

BAK - FEDERAL CHAMBER OF GERMAN ARCHITECTS. (2021). Regulation of the architectural profession and practice in germany. https://www.bak.de/eng/regulation/. Accessed: 14/04/2021.

BARAZZETTI, L. & BANFI, F. (2017). Bim and gis: When parametric modeling meets geospatial data. In *Isprs workshop on geospatial solutions for structural design, construction and maintenance in training civil engineers and architects, geospace 2017* (Vol. 4, *5W1*, pp. 1–8).

BAVARIAN MINISTRY FOR HOUSING, CONSTRUCTION AND TRANSPORTATION. (2021). Bauordnungsrecht. https://www.stmb.bayern.de/buw/baurechtundtechnik/bauordnungsrecht/index.php. Accessed: 14/04/2021.

BRAGANÇA, L., VIEIRA, S. M., & ANDRADE, J. B. (2014). Early stage design decisions: The way to achieve sustainable buildings at lower costs. *The Scientific World Journal*, *2014*, 365364. doi:10.1155/2014/365364

CALDAS, L. (2008). Generation of energy-efficient architecture solutions applying gene_arch: An evolution-based generative design system. *Advanced Engineering Informatics*, *22*(1), 59–70. Intelligent computing in engineering and architecture. doi:https://doi.org/10.1016/j.aei.2007.08.012

FRAZER, J. (2002). Chapter 9 - creative design and the generative evolutionary paradigm. In P. BENTLEY & D. CORNE (Eds.), *Creative evolutionary systems* (pp. 253–274). The Morgan Kaufmann Series in Artificial Intelligence. San Francisco: Morgan Kaufmann. doi:https://doi.org/10.1016/B978-155860673-9/50047-1

FUDOS, I. & HOFFMANN, C. (1997). A graph-constructive approach to solving systems of geometric constraints. *ACM Trans. Graph. 16*(2), 179–216. doi:10.1145/248210.248223

GOODFELLOW, I., POUGET-ABADIE, J., MIRZA, M., XU, B., WARDE-FARLEY, D., OZAIR, S., ... BENGIO, Y. (2014). Generative adversarial nets. In Z. GHAHRAMANI, M. WELLING, C. CORTES, N. LAWRENCE, & K. Q. WEINBERGER (Eds.), *Advances in neural information processing systems* (Vol. 27). Curran Associates, Inc.

KNEIDL, A. (2013). *Methoden zur abbildung menschlichen navigationsverhaltens bei der modellierung von fußgängerströmen* (Dissertation, Technische Universität München, München).

KRISH, S. (2011). A practical generative design method. *Computer-Aided Design*, *43*(1), 88–100. doi:https://doi.org/10.1016/j.cad.2010.09.009

MIHALCEA, R., LIU, H., & LIEBERMAN, H. (2006). Nlp (natural language processing) for nlp (natural language programming). In A. GELBUKH (Ed.), *Computational linguistics and intelligent text processing* (pp. 319–330). Berlin, Heidelberg: Springer Berlin Heidelberg.

PLUM, A. & JÄGER, G. (2011). Evakuierungssimulationen im rahmen von sicherheitskonzepten - von der konzeption bis zur realisierung. In *Proc. of sicherheit von veranstaltungen*. Köln, Germany.

PROVINCIAL CAPITAL MUNICH. (2021). Satzungen, verordnungen und stellplatzrichtzahlen. https://www.muenchen.de/rathaus/Stadtverwaltung/Referat-fuer-Stadtplanung-und-Bauordnung/Lokalbaukommission/Kundeninfo/Satzungen.html. Accessed: 14/04/2021.

SALIMZADEH, N., VAHDATIKHAKI, F., & HAMMAD, A. (2020). Parametric modeling and surface-specific sensitivity analysis of pv module layout on building skin using bim. *Energy and Buildings*, *216*, 109953. doi:https://doi.org/10.1016/j.enbuild.2020.109953

SCHOLL, J. (2019). *Integration of bim-based pedestrian simulations in the early design stages* (Bachelor's thesis, Technische Universität München).

SEITZ, M. (2016). *Simulating pedestrian dynamics* (Dissertation, Technische Universität München, München).

Shah, J. & Mäntylä, M. (1995). *Parametric and feature-based cad/cam: Concepts, techniques, and applications*. John Wiley & Sons.

Vilgertshofer, S. & Borrmann, A. (2017). Using graph rewriting methods for the semi-automatic generation of parametric infrastructure models. *Advanced Engineering Informatics*, *33*, 502–515. doi:10.1016/j.aei.2017.07.003

Wang, T., Liu, M., Zhu, J., Tao, A., Kautz, J., & Catanzaro, B. (2018). High-resolution image synthesis and semantic manipulation with conditional gans. arXiv: 1711.11585 [cs.CV]

Wassim, J. (2013). *Parametric design for architecture*. Laurence King.

Zheng, H. & Huang, W. (2018). Understanding and visualizing generative adversarial networks in architectural drawings.

Appendix B

Declaration

I hereby affirm that I have independently written the thesis submitted by me and have not used any sources or aids other than those indicated.

Location, Date, Signature