Lehrstuhl für Computergestützte Modellierung und Simulation
TUM School of Engineering and Design
Technical University of Munich

Implementation of an IFC file creator and modifier using Visual Programming

Scientific work to obtain the degree

Bachelor of Science (B.Sc.)

at the TUM School of Engineering and Design
of the Technical University of Munich.

Supervised by Prof. Dr.-Ing. André Borrmann
Jonas Schlenger
Lehrstuhl für Computergestützte Modellierung und Simulation

Submitted by Samuils Rulovs [redacted]

Submitted on 28. October 2022

Abstract

Industry Foundation Classes (IFC) are the open international standard in the field of Building Information Modeling (BIM). IFC were created as an interoperable format, which allows the storage, exchange, and sharing of data between participants in the building sector. Another possible application of the IFC file format is a reference or archive of the content describing a model. Therefore Industry Foundation Classes play a significant role in the field of architecture, engineering, and construction (AEC). IFC are a file format that is constantly improved. Every several years a new version of IFC is introduced, providing various extensions. In order to test the features provided by the updated IFC version, a variety of example files should be created. However, software creators need considerable time to support a new IFC version. Hence, other tools should be implemented to create files in such data format.

This work provides a solution for implementing the IFC file modifier and creator. The goal of creating or modifying an IFC file is typically accomplished with the aid of a text editor or visual programming platform, such as Autodesk's Dynamo. In the framework of this thesis, an IFC file creator and modifier is developed using Visual Programming Language (VPL). The main reason for the use of the VPL is a more intuitive approach to the creation of the IFC file compared to the conventional text editors, where all semantics for IFC files are generally updated. Visual Programming Language also reduces the probability of error since the amount of text that should be typed is reduced due to the partial text substitute with VPL elements.

Zusammenfassung

Industry Foundation Classes (IFC) sind der offene internationale Standard im Bereich der Bauwerksdatenmodellierung BIM. IFC wurden als interoperables Format erstellt, das die Speicherung, den Austausch und die gemeinsame Nutzung von Daten zwischen Teilnehmern des Bausektors ermöglicht. Eine weitere mögliche Anwendung des IFC-Datenformats ist die Gelegenheit, den Inhalt von Modellen zu referenzieren und zu archivieren. Daher sind Industry Foundation Classes ein bedeutendes Datenformat im Bereich Architektur, Ingenieur- und Bauwesen. Dennoch befindet sich dieses Dateiformat in der Entwicklung. Alle paar Jahre wird eine neue Version von IFC eingeführt, die neue Erweiterungen bereitstellt. Um die neuen Funktionen der neuen IFC Version zu testen, sollten verschiedene Beispieldateien erstellt werden. Softwareentwickler benötigen jedoch eine beträchtliche Zeit, um eine neue IFC-Version zu unterstützen. Daher sollten andere Tools implementiert werden, um Dateien in neuem Datenformat zu erstellen.
Diese Arbeit bietet eine Lösung für die Implementierung von der Software, die für das Modifizieren oder Erstellen von IFC-Dateien zuständig ist. Das Ziel, eine IFC-Datei zu erstellen oder zu ändern, wird normalerweise mit Hilfe eines Texteditors oder einer visuellen Programmierplattform wie Dynamo von Autodesk erreicht. Im Rahmen dieser Bachelorarbeit wird die Erstellung von der Software diskutiert, die Dateien im IFC Datenformat unter Verwendung von der visuellen Programmiersprache Visual Programming Language (VPL), erstellt und modifiziert. Der Hauptgrund für die Verwendung von VPL ist ein intuitiver Ansatz zur Erstellung von IFC Dateien im Vergleich zu konventionellen Texteditoren, die üblicherweise für die Erstellung von Semantik in den IFC Dateien verwendet werden. Visuelle Programmiersprachen reduzieren auch die Fehlerwahrscheinlichkeit, da die einzutippende Textmenge durch das Ersetzen der Teile mit VPL Elementen verringert wird.

Contents

List of Figures

List of Algorithms

Acronyms

AEC	architecture, engineering, and construction
BIM	Building Information Modeling
bSI	builldingSMART International
CAD	Computer aided design
CMS	The chair of Computational Modeling and Simulation
IFC	Industry Foundation Classes
IFC-SPF	STEP Physical Format
IOS	IfcOpenShell
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
NIST	National Institute of Standards and Technology
OFF	Object File Format
OIP	TUM OpenInfraPlatform
Rhino	Rhinoceros
TUM	Technical University of Munich
VPL	Visual Programming Language

Chapter 1

Introduction

Due to the high technological advancement, industries are shifting towards digitalization. At first architecture, engineering, and construction (AEC) shifted from manual drafting to computer-aided design (CAD). Currently, CAD is being replaced by Building Information Modeling (BIM) since BIM brings with it a variety of new solutions and possibilities which are not encountered in CAD. Building Information Modeling is a process of representing construction projects through virtual models in digital form. Therefore, each participant in the construction project can view and adjust information regarding the project through a virtual model in the BIM viewer without having to rely on physical copies. As a result, the productivity of teams participating in the construction is enhanced, and the chance that some information will go missing is reduced. Thereby the probability that the project will be ready on time grows, and the rate of issues regarding a project decreases.

However, many software creators develop their own BIM solutions. Each developed solution may rely on various data formats that are usually incompatible with one another. Consequently, when various parties involved in a construction project use different software solutions, information may be lost due to the incompatible data formats supported by these applications.

Intending to secure a successful exchange between different parties, a non-profit organization builldingSMART International (bSI) is developing a data exchange format, which should ensure interoperability between different BIM software solutions. This data exchange format is Industry Foundation Classes (IFC).

Industry Foundation Classes are an object-oriented data model responsible for storing and exchanging data describing construction objects. Since the construction industry involves many independent parties that need to rely on one another, the such data format is crucial. Assuring the exchange of information between these parties, IFC secure a grasp of the project status, progress, and possible problems.

Technical University of Munich (TUM) takes part in the development and support of the IFC community. The chair of Computational Modeling and Simulation (CMS) at the Technical University of Munich (TUM) is developing an open-source BIM visualization software named TUM OpenInfraPlatform (OIP). OIP provides tools to read IFC files and visualize the result.

1.1 Motivation

Industry Foundation Classes (IFC) are a file format that is constantly improved, and new versions of this data format provide extensions not supported by previous versions. For example, all IFC versions before IFC4 primarily focused on describing buildings and their components. Figure 1.1 shows the evolution of the IFC and the history of IFC versions up to the IFC4 version. Now with the introduction of the IFC4.3 schema, infrastructure constructions within domains of railways, roads, ports, and waterways are introduced and covered in the IFC (buildingSMART International, 2022a). It is also planned to release the IFC5 version, enhancing parametric abilities and including infrastructure domains(BibLus, 2022). Thus, creating new example files to test added features and finding flaws in these feature implementations is essential for future development.

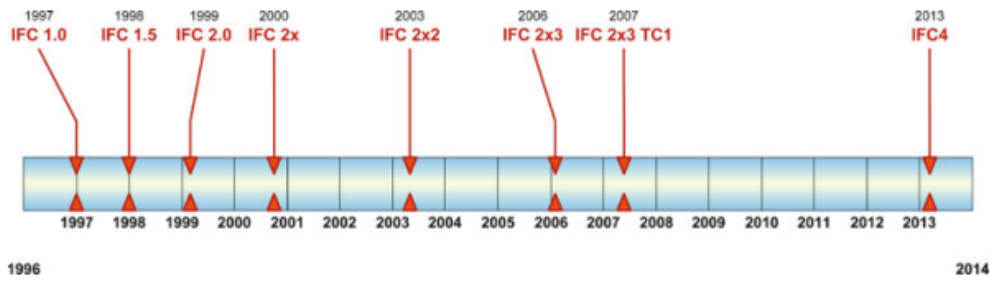


Figure 1.1: History of IFC versions Borrmann et al. (2015)

The new IFC example files are primarily created or modified using a text editor or visual programming platform such as Dynamo from Autodesk. Using text-based editors may result in errors in the file since different typos can occur during the text file creation. At the same time, it is arduous to create functioning IFC files in such a way because of the amount of data that should be included in the file.

Another approach to creating and modifying an IFC file is a Visual Programming Language (VPL). VPL provides graphical components, such as nodes, widgets, buttons, and symbols. Portions of the information can be predefined using graphical components, thus reducing the amount of information users need to write. As a result, the probability of errors is reduced.

Moreover, Visual Programming Language is a more intuitive approach since it resembles the logic of the Industry Foundation Classes. Both VPL nodes and IFC entities contain a various attributes describing them. Nodes in VPL can describe relations between other nodes, similarly to IFC entities. Therefore, using Visual Programming Language for creating IFC files can be more appropriate than using text-based editors.

Different software creators have already worked on this issue. For example, Autodesk has developed its VPL application, Dynamo. The main objective of the Dynamo is to customize building information using a graphical programming interface. Thus, it is focused mainly on the geometries of the building parts or other projects.

However, geometry visualization is sometimes not required for modifying IFC files. Instead, it is necessary to modify IFC files by adjusting each entity's semantics. It is, therefore, necessary to develop Visual Programming language software, which will replace simple text editors, where all semantics for IFC entities are currently updated.

1.2 Problem Statement

Today, the semantics of IFC files is mainly modified and created using text editors. Creating an IFC file in the text editor requires writing a substantial amount of text, which may lead to typos. In addition, some of the IFC entities defined in the created file may not be connected to other IFC entities or contain incorrect attribute values since the text editor does not describe the attributes of IFC entity types. For this reason, a more intuitive and user-friendly approach should be implemented to simplify the creation and modification of IFC files.

1.3 Objective

This bachelor thesis aims to create software that will allow the creation and modification of Industry Foundation Classes (IFC) file using Visual Programming Language (VPL). Upon reading the IFC file, such software retrieves every IFC entity from the file and creates nodes containing information about retrieved entities. Newly created nodes form a structure based on potential connections of identified IFC entities. Users can create, alter, and delete the nodes as needed by interacting with them in the Visual Programming Language environment. After the necessary changes to nodes are complete, nodes are transformed back into IFC entities with updated information. Finally, IFC entities are returned to the IFC file.

1.4 Methodical approach

In the framework of this thesis following steps should be accomplished, which should result in the creation of the IFC file creator and modifier:

1. Review other Visual Programming Language software which has already incorporated necessary functions.
2. Decide which programming language and, consequently, which libraries should be used for software development.
3. Create a prototype application, which will have access only to the limited number of IFC entities but will accomplish all objectives expected from the software.

1.5 Thesis Structure

This thesis is structured into six chapters. Chapter 2 elaborates on the theoretical foundations behind Industry Foundation Classes, Visual Programming Language, and Flow-based visual scripting software Ryven. Chapter 2 also discusses similar solutions to the problem presented in this thesis. Chapter 3 addresses the development method of an IFC file creator and modifier. Subsequently, chapter 4 describes the program's implementation using methods addressed in chapter 3. An example-driven description of the use cases is provided in chapter 5. Finally, in chapter 6 all findings are summarized and evaluated, and improvements are suggested.

Chapter 2

Theoretical foundations

2.1 Industry Foundation Classes (IFC)

Industry Foundation Classes (IFC) are the object-oriented data model, which is currently selected as an open international standard in the field of Building Information Modeling (BIM) (ISO, 2018). IFC were created to store, exchange and share data of construction objects. This information can be later shared with other participants during construction regardless of their software applications. In year 2013 Industry Foundation Classes (IFC) became ISO certified (buildingSMART INTERNATIONAL, 2022b).

IFC development started in 1994 by the former International Alliance of interoperability, now known as buildingSMART International (bSI). The multinational software corporation Autodesk formed this organization, thereby beginning the initiative. The initiative's main goal was to create a format independent of commercial software creators. The independent and interoperable data format should help to overcome the data exchange problem in case two sides of the exchange use different commercial software incompatible with one another.

A construction project is a complex system in which every party has a different responsibility. For this reason, each team involved in construction needs a greater understanding of the other team's progress and possible problems the project faces. Therefore, it is essential that information is exchanged between these teams in order to ensure the project's success.

However, different teams involved in construction projects often use different software solutions since there is no universal software solution capable of addressing all specific construction tasks. As a result, the exchange of information between teams relies on the compatibility of the software each party uses.

One of the motivations why it was necessary to develop an independent and open international standard similar to IFC, was presented in the research conducted by the National Institute of Standards and Technology (National Institute of Standards and Technology (NIST)) in the year 2004. Based on this research, over 15,8 Billion US-Dollar were lost in 2002 because of the "inadequate interoperability" of different software solutions in the construction industry. (GALLAHER et al., 2004)

The IFC data schema is based on various technologies, especially the data modeling language EXPRESS. Similarly to the EXPRESS data modeling language, the entity-relationship model is created for IFC, which can consist of various entities and relations.

These entities and relationships are organized in the inheritance hierarchy, allowing child entities to inherit all attributes from the parent entities. Figure 2.1 illustrates the IFC entity inheritance hierarchy.

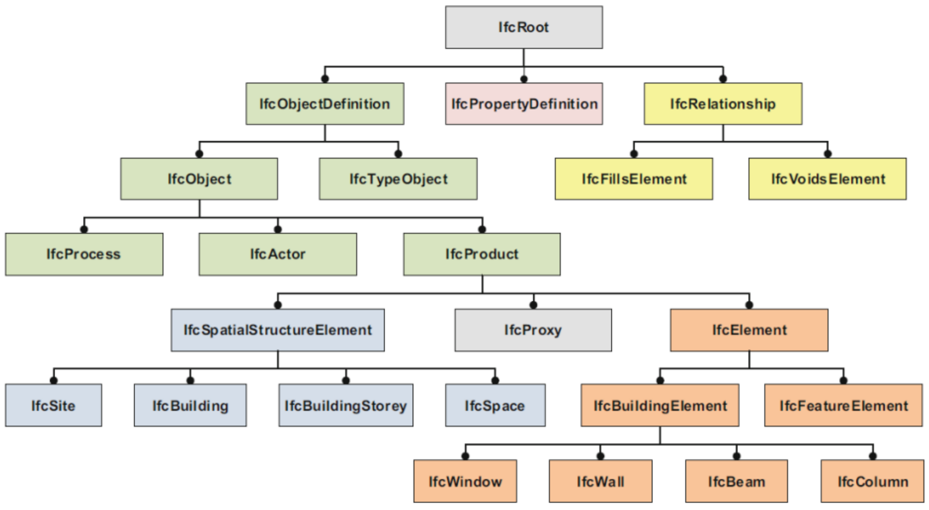


Figure 2.1: Inheritance hierarchy (BORRMANN et al., 2015)

A crucial part of IFC entity-relationship model are relationships. The relationship in IFC describes an interaction between different entities. Such interactions are defined by intermediary objects, which describe all interaction details through their attributes. This form of describing a relationship between several entities is called an objectified relationship. Figure 2.2 displays the relationship classes of the IFC schema.

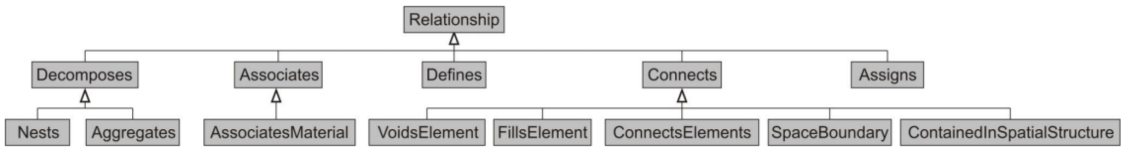


Figure 2.2: Relationship classes (BORRMANN, 2021)

On the other hand, IFC entities describe objects, processes, and further details required in the construction project. Each entity consists of attributes describing the properties of the construction project details. Thus, in the entity-relationship model, every object or process during construction can be defined using entities and linked using relationships.

The IFC schema can be expressed using various data and file formats. The most widespread data format is the ASCII format. The main advantage of the ASCII format is the ability to read and write files using this format in a simple Text editor. The most common text format to express the IFC schema is STEP Physical Format (IFC-SPF). This file format has compact dimensions and a data ending ".ifc" (BIBLUS, 2020). In the framework of this thesis, IFC-SPF will be used as a primary file format for IFC.

2.2 TUM Open Infra Platform

The TUM Open Infra Platform (OIP) is open-source software for visualizing Building Information Modelling models. This software is currently being developed by the chair of Computational Modeling and Simulation (CMS) at the Technical University of Munich (TUM). The core programming of the software is conducted using C++ programming language. (HECHT & JAUD, 2019)

The primary data format which is used in OIP is Industry Foundation Classes (IFC). To support IFC file format, the IFC Geometry Converter was created in the Core of TUM Open Infra Platform. The objective of the IFC Geometry Converter is to convert various geometric descriptions specified in IFC into a triangle mesh. Conversion of geometries into triangle meshes allows the rendering engine to understand and visualize information from IFC file.

Apart from IFC, Open Infra Platform supports other data formats. OFF Geometry Converter supports Object File Format (OFF). OIP also includes the Point Cloud Processing module, which enables the visualization of several point cloud file formats. Figure 2.3 demonstrates the software architecture of the TUM OpenInfraPlatform including IFC Geometry Converter, OFF Geometry Converter, and Point Cloud Processing module.

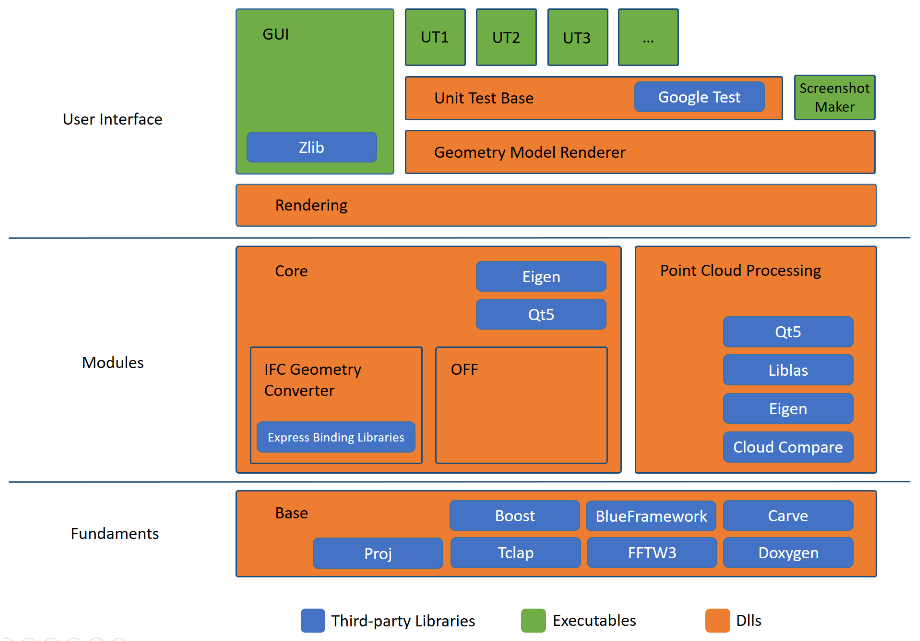


Figure 2.3: TUM OpenInfraPlatform software architecture (THE CHAIR OF COMPUTATIONAL MODELING AND SIMULATION, 2019)

2.3 Visual Programming Language

Visual Programming Language (VPL) is a programming language that uses graphical components such as widgets, buttons, and symbols for software development. A visual programming language allows the illustration of programming code, thus allowing newcomers and people with little technical skills in the field of programming to understand the processes hidden behind the code. VPL allows users to use a more intuitive approach to create the program; for example, it allows creators to use the "drag-and-drop" function and other interface tools (PREIDEL et al., 2017). Figure 2.4 provides an example of the Visual Programming Language in the Dynamo environment.

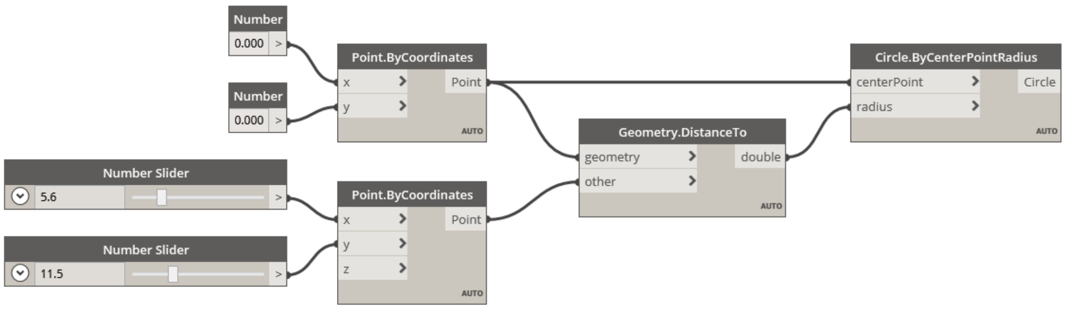
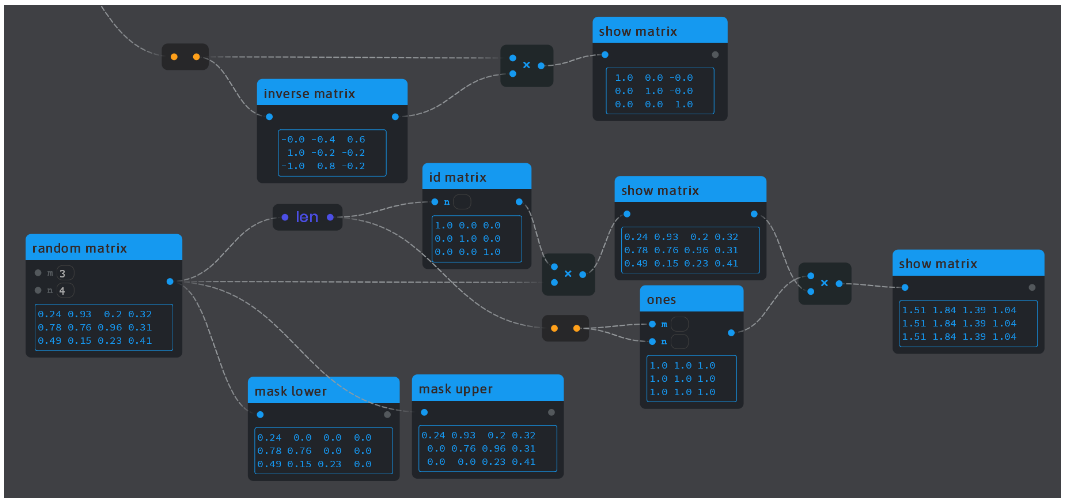


Figure 2.4: Visual Programming script in Dynamo environment (AUTODESK, 2022)

Unlike classical programming languages, ordinarily text-based, visual programming language usually contains different components or nodes with specific properties. Each node is connected to other nodes, thus creating a linked-together system. This allows VPL to be more readable compared to text-based programming languages, such as Java or C++.

2.4 Ryven

Ryven is a flow-based visual scripting environment created by LEON THOMM (2022) student from Swiss Federal Institute of Technology in Zurich, developed using Python programming language. Rather than using lines of code to implement software, Ryven software uses a more graphical or diagrammatic representation. Figure 2.5 shows the diagrammatic representation of the process in the Ryven environment.



```
1.0  0.0 -0.0
0.0  1.0 -0.0
0.0  0.0  1.0
```

```
-0.0 -0.4  0.6
 1.0 -0.2 -0.2
-1.0  0.8 -0.2
```

```
1.0 0.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
```

```
0.24 0.93  0.2 0.32
0.78 0.76 0.96 0.31
0.49 0.15 0.23 0.41
```

```
0.24 0.93  0.2 0.32
0.78 0.76 0.96 0.31
0.49 0.15 0.23 0.41
```

```
1.0 1.0 1.0
1.0 1.0 1.0
1.0 1.0 1.0
```

```
1.51 1.84 1.39 1.04
1.51 1.84 1.39 1.04
1.51 1.84 1.39 1.04
```

```
0.24  0.0  0.0  0.0
0.78 0.76  0.0  0.0
0.49 0.15 0.23  0.0
```

```
0.24 0.93  0.2 0.32
 0.0 0.76 0.96 0.31
 0.0  0.0 0.23 0.41
```

Figure 2.5: Ryven flow based visual scripting environment (LEON THOMM, 2022)

The core part of the visual scripting environment is a node. It is defined as a class in Ryven's node library. Each class has its own attributes describing the node's properties, inputs, and outputs. In addition, during node class definitions, there is an option to define various methods and additional variables, thus describing the functionality of a particular node.

Ryven has some built-in nodes which allow users to create various algorithms using these nodes. However, the Ryven environment allows new node creation and specification, thus allowing a broad scope of new program implementations using the Ryven environment.

2.5 JavaScript Object Notation

JavaScript Object Notation (JSON) is a textual format for data exchange. JSON is based on JavaScript, however, it is an independent data format and can also be used by other programming languages.

JSON was introduced at the beginning of the 21st century as a format for communication between JavaScript clients and back-end servers. It gradually expanded and became the most common human-readable format since it is helpful for data exchange between various systems. Nowadays, JSON is used as the universal standard for data exchange.(TYSON, 2022)

JSON is structured using objects and arrays. An object is a dictionary containing different elements, with every element having key-value pairs assigned to itself. On the other hand, the array is an ordered list of values. However, an array can also contain a list of objects. This means that using these two structures, JSON can model object relations, which can vary in complexity. Algorithm 2.1 demonstrates the structure of the JSON file for the flow-based visual scripting environment Ryven.

Algorithm 2.1: Example of the JSON structure for Ryven environment

```
{
    "scripts": [
        {
            "title": "hello␣world",
            "variables": {},
            "flow": {
                "algorithm␣mode": "data",
                "nodes": [],
                "connections": [],
                "GID": 6,
                "flow␣view": {
                    "drawings": [],
                    "view␣size": [
                        6400.0,
                        4800.0
                    ]
                }
            },
            "GID": 1
        }
    ]
}
```

2.6 Related Work

This section discusses the use of visual programming languages in the construction industry. The most common Visual Programming tools are Dynamo and Grasshopper, based on the paper written by COLLAO et al. (2021) that analyzed the use of visual programming tools in infrastructure projects.

2.6.1 Dynamo

Dynamo is an open-source visual programming environment developed by Autodesk, a leading architecture, engineering, and construction (AEC) software developer. The main objective of the software is to customize building information using a graphical programming interface.

The process of visual programming in the Dynamo environment can be described as an action sequence completed in the established algorithm. Such an algorithm is completed with the assistance of textual fragments of the programming code, nodes, developed in advance. The sequence of such nodes forms a script. fig. 2.6 demonstrates an example of the script created in Dynamo.

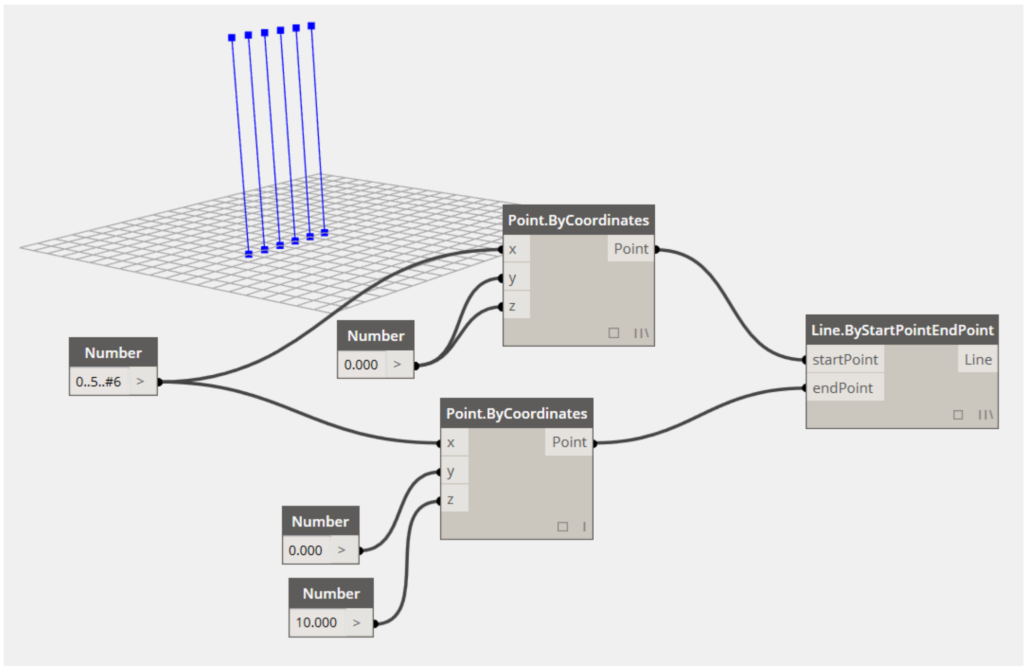


Figure 2.6: An example of the script in Dynamo environment (YANG, 2015)

Dynamo is oriented more toward creating scripts that can manipulate various objects in Revit since it was created as assisting tool for Revit. Thus, it is focused primarily on the geometries of the building parts or other projects.

2.6.2 Grasshopper 3D

Grasshopper is a visual programming language and environment integrated into Rhinoceros (Rhino) 3D modeling software. Both Rhino and Grasshopper are developed by Robert McNeel & Associates (2022)

Grasshopper uses VPL to manipulate different components onto a canvas. Similarly to Dynamo, Grasshopper focuses primarily on geometries. Furthermore, it encompasses numerical, haptic, textual, and audiovisual algorithms for the creation of parametric models and generative artworks (Saili, 2021). fig. 2.7 demonstrates the node diagram in the Grasshopper environment.

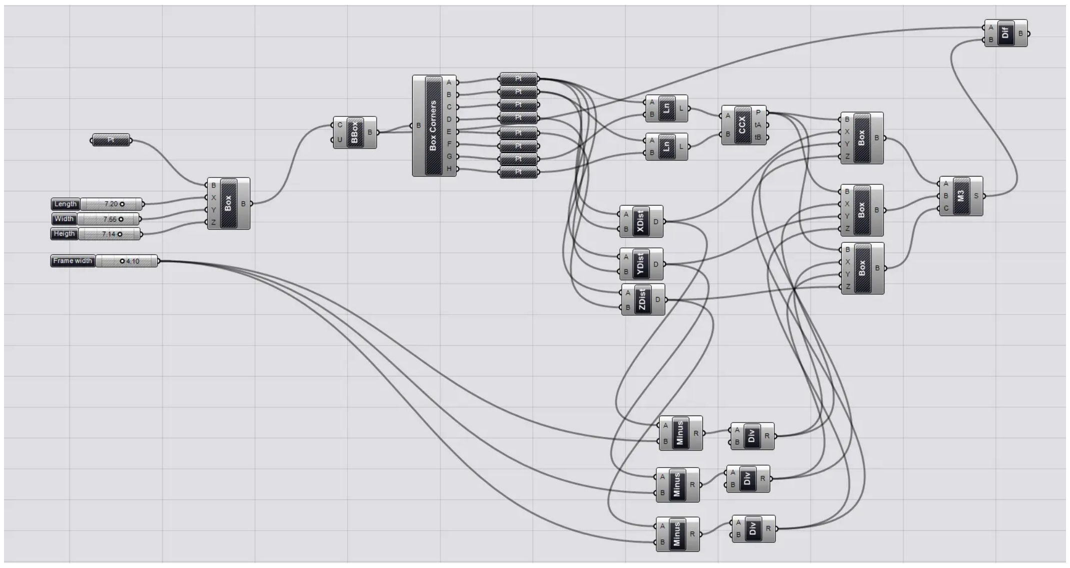


Figure 2.7: The node diagram in the Grasshopper environment (Saili, 2021)

Chapter 3

Method

Chapter 3 describes the decisions underlying the implementation of the IFC file creation and modification software. Section 3.1 discusses how the Ryven flow-based visual scripting environment was selected for use in software development. The approach used to implement the software is described in section 3.2.

3.1 Selection of the libraries for implementation

Creating an IFC file is mainly accomplished using a text editor or visual programming platform. Compared to text-based software, VPL relies on graphical components, such as nodes, widgets, buttons, and symbols, thereby reducing error probability. Moreover, VPL is a more intuitive approach since it resembles the logic of the Industry Foundation Classes.

Therefore, Visual Programming Language was selected to create an IFC file creation and modification software. It was decided to develop the software using a VPL library. Since none of the other VPL libraries found were written in Python or C++ and were open-source, libraries Chigraph and Ryven were selected as prime candidates. Chigraph is a new visual systems programming language created by Russell Greene and Aun-Ali Zaidi (2017) and compiled using LLVM . LLVM is a collection of compiler and toolchain technologies developed by a research group from the University of Illinois (2007). Both LLVM and Chigraph are written in C++. Ryven, on the other hand, is a flow-based visual scripting environment developed using Python by Leon Thomm (2022).

Since the Chigraph programing language was written using C++, it could be a more suitable choice. For example, relying on C++ programming language allows Chigraph to incorporate an early binding generator from TUM Open Infra Platform, which is required to represent each entity in the IFC schema as a corresponding class in the programming language. By applying functionalities of the early binding generator in the IFC file creator and modifier, all entities from the IFC schema would be included in the solution. In addition, based on the screenshots on GitHub, the Chigraph programming language contains all tools necessary for developing the software.

However, various issues with Chigraph software were revealed during the installation process. Many components required by Chigraph were missing or outdated, and usually, some components conflicted with each other. At the same time, the Chigraph build usually fails on the Windows operating system.

The Ryven environment, on the other hand, was installed without any issues. Instead of relying on the external compiler and libraries conflicting with one another, it uses a small number of external libraries, all incorporated into Python. Thus Ryven is a preferable library for developing the IFC file creation and modification software. Although it cannot incorporate an early binding generator from TUM Open Infra Platform, thus including all entities from the IFC schema, it has the option to use a library IfcOpenShell (IOS) for Python that operates with the IFC schemas (IfcOpenShell, 2022).

In the end, it was decided to use Ryven flow-based visual scripting environment due to the absence of issues during the installation and compiling of the software.

3.2 Design

The design of the IFC file creator and modifier implementation using Ryven can be divided into several steps that will lead to the complete cycle of the software use. Figure 3.1 illustrates the workflow of the IFC file creator and modifier.

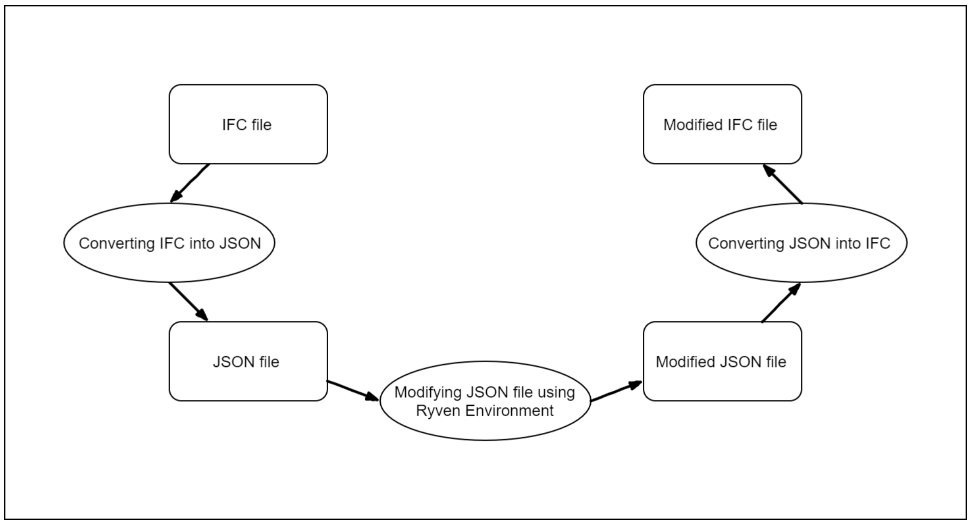


Figure 3.1: The workflow of the IFC file creator and modifier

Initially, an IFC file is opened with the IFC file creator and modifier. IFC files are expressed via STEP Physical Format (IFC-SPF). The Ryven environment, on the other hand, accepts only files with JSON extension. For this reason, IFC files should be converted from STEP format into JSON file format.

The conversion process into JSON starts with retrieving information from the IFC file and its arrangement into dictionaries. It was chosen to save data describing IFC entities into dictionaries since every IFC entity consists of different attributes, thus preventing storing information in lists. Following this, the information in the dictionaries must be encoded in binary format due to Ryven's requirements, which accept only binary information. Once the information is encoded, dictionaries are combined into a nested structure similar to a JSON file, thus preparing information for export. Finally, the resulting structure is saved into the JSON file. However, the conversion process into JSON is skipped if a new IFC file is created from scratch.

After converting an IFC file into JSON, the newly created file is opened in the Ryven environment. As mentioned in section 2.4, the core part of the Ryven is a node. Therefore every IFC entity is presented as a node containing every attribute and the $IFC\ ID$ of the IFC entity it describes. $IFC\ ID$ is a STEP numerical identifier unique for every IFC instance in the file. As the only way to distinguish IFC entities, it is an essential component of the IFC file. As a result, $IFC\ ID$ was included as the first parameter in each node's properties.

Representing IFC entity types with corresponding nodes allows users to interact with information from the IFC file. For example, users may adjust the attribute values of the IFC entity by interacting with the node, which contains information about the entity. Ryven also allows the definition of new IFC entities of the IFC file or the removal of unnecessary ones via adding or removing corresponding nodes.

However, to interact with nodes, each node describing an IFC entity type should be incorporated into the Ryven node library. Otherwise, a file containing undefined node descriptions will not be opened by the Ryven environment. Therefore, the Ryven node library is composed of node classes, each describing the functionality and properties of the node it represents.

Once the IFC file has been modified through interaction with Ryven nodes, the changes are saved to the JSON file. However, receiving changes in the initial format is mandatory for the further use of the created file. Thus, the JSON file should be converted back to the IFC file expressed in STEP Physical Format (IFC-SPF). Similarly to conversion from IFC to JSON file format, the conversion process starts with retrieving information from the file and its arrangement into dictionaries. Following this, the information in the dictionaries is decoded back to the text format since STEP Physical Format is text-based. After decoding, information is written into the new IFC file based on the IFC-SPF requirements, thus completing all operations.

Chapter 4

Implementation

Chapter 4 describes an IFC file creation and modification software implementation using Visual Programming Language. The implementation of the program **fromIFCtoJSON** for converting an IFC file into an JSON file is discussed in section 4.1. A description of the Ryven flow-based visual scripting environment, its functionalities, and the node classes incorporated into this environment is presented in section 4.2. Lastly, section 4.3 indicates how the **fromJSONtoIFC** program performs the conversion from JSON into the IFC file format.

4.1 Converting IFC file into JSON file

For an IFC file to be opened in a Ryven environment, it must be converted into the specific structure of the JSON file format required by Ryven. For this reason, a program should be created to convert IFC files into JSON file format per requirements. This section describes the program's implementation responsible for the conversion process from IFC file format into JSON.

For illustration purposes, all processes described in this section will be accompanied by examples from the IFC example file "tessellated-item.ifc". This example file was selected as a foundation of the IFC file creator and modifier prototype due to the small number of IFC entities it owned and the variety of IFC entities simultaneously.

4.1.1 Retrieving information from IFC file

Upon launching a program, the IFC file is opened using IfcOpenShell (IOS). IfcOpenShell is an open-source software library that operates with the STEP Physical File (IfcOpenShell, 2022). It provides a basis for retrieving, modifying, and creating IFC data models. Within the framework of this program, IOS is used only to retrieve IFC data models from the IFC file.

After opening the IFC file with IfcOpenShell, all entities contained in the IFC file are counted. It allows it to iterate through all IFC entities without relying on string access to the entities. Normally all entity type names must be known and predefined by the user. However, this is avoided by accessing IFC entities using their $IFC\ IDs$ instead of their type names. $IFC\ ID$ is a STEP numerical identifier unique for every IFC instance in the file. A simplified example how IFC entity is accessed using $IFC\ ID$ is illustrated in Algorithm 4.1. The result is visible in Algorithm 4.2.

Algorithm 4.1: Accessing IFC entities using IFC ID

```
import ifcopenshell

ifc = ifcopenshell.open("tessellated-item.ifc")
entity = ifc.by_id(1)
print(entity)
```

All essential information is retrieved from every IFC entity during the iteration procedure. A description of processes assisting the retrieval of information will be given using the IFC entity **IfcProject** as an example.

Iteration starts with the selection of an IFC entity filtered by $IFC\ ID$. **IfcProject** has an $IFC\ ID$ of #1. For this reason, it is selected first. Algorithm 4.2 shows a selected IFC entity.

Algorithm 4.2: IFC entity IfcProject from example-file

```
#1=IfcProject('0xScRe4drECQ4DMSqUjd6d',#2,
'proxy with tessellation', $,$,$,$,(#3),#4)
```

Upon selecting the IFC entity, all information about a particular entity is parsed into the dictionary using IOS function `get_info()`. This function returns a dictionary of the entity instance's properties and values. However `get_info()` function returns property values in the altered form. As a result, the Ryven environment would receive values different from the ones it should receive from IFC file. In order to avoid this, all values in the newly created dictionary are overwritten using attribute values from the IFC entity in the string format, thus preventing alterations to the values later. The difference between a dictionary with altered values and a dictionary after values were overwritten back is shown in Algorithm 4.3.

Algorithm 4.3: The difference in values prior to and after they are overwritten

```
old info
    {'id': 1, 'type': 'IfcProject',
    'GlobalId': '0xScRe4drECQ4DMSqUjd6d',
    'OwnerHistory': #2=IfcOwnerHistory(#6,#7,$,.ADDED.,1320688800,$,$
        ,1320688800),
    'Name': 'proxy with tessellation', 'Description': None,
    'ObjectType': None, 'LongName': None, 'Phase': None,
    'RepresentationContexts': (#3=IfcGeometricRepresentationContext($,'
        Model',3,1.0E-05,#8,$),),
    'UnitsInContext': #4=IfcUnitAssignment((#10,#11))}

new info
    {'id': 1, 'type': 'IfcProject',
    'GlobalId': "'0xScRe4drECQ4DMSqUjd6d'",
    'OwnerHistory': '#2',
    'Name': "'proxy with tessellation'", 'Description': '$',
    'ObjectType': '$', 'LongName': '$', 'Phase': '$',
    'RepresentationContexts': '(#3)',
    'UnitsInContext': '#4'}
```

4.1.2 Arranging retrieved information

With the creation of the dictionary containing all information about IFC entity and its attributes, the dictionary is passed to the initialized object class **IFC_Entity**. Class **IFC_Entity** contains methods responsible for arranging information in a way required by Ryven.

Instead of IFC entities, JSON file required for Ryven environment has an object $"nodes"$, which contains an array of nodes. Each node corresponds to the entity from the IFC file. Every IFC entity's $IFC\ ID$ and all its attributes are stored in the node's array named $"inputs"$. Each $"input"$ contains a description of an $IFC\ ID$ or attribute regarding its type, data type, and widget data. Algorithm 4.4 shows a structure of the $"input"$ of the $IFC\ ID$ in the **IfcProject** node.

Algorithm 4.4: "input" describing the IFC ID in the IfcProject node

```
{
   "type": "data",
   "label": "OwnIFCID",
   "GID": 8,
   "val": "gASVBgAAAAAAAACMAiMxlC4=",
   "dtype": "DType.String",
   "dtype state":
        "gASVPAAAAAAAAAB9lCiMB2RlZmF1bHSUjAEjlIwDdmFslIwCIzGUjANkb2OUj
        ACUjAZib3VuZHOUTowEc2l6ZZSMAW2UdS4=",
   "has widget": true,
   "widget data": "gASVEAAAAAAAAAB9lIwEdGV4dJSMAiMxlHMu"
}
```

Properties $"val"$, $"dtype\ state"$ and $"widget\ data"$ from the example contain details about **IfcProject** entity's $IFC\ ID$ encoded using **Base64** library. **Base64** is a binary-to-text encoding library which translates an **ASCII** string format into a radix-64 representation (Docs, 2022). Algorithm 4.5 illustrates what each property looked before being encoded.

Algorithm 4.5: Properties containing information about IfcProject before encoding

```
{
    "val": '#1'
    "dtype state": {'default': '#', 'val': '#1', 'doc': '',
    'bounds': None, 'size': 'm'}
    "widget data": {'text': '#1'}
}
```

Algorithm 4.5 shows that information regarding $IFC\ ID$ of **IfcProject** was arranged in a specific order for each property type. Therefore, each property required a different method from the **IFC_Entity** class to arrange information in a specific order. Upon arranging information and storing it in certain variables object of the type **IFC_Entity** is saved in the list $nodes$.

4.1.3 Defying connections between nodes

After creating a list $nodes$ describing IFC entities, the next step is to define connections between nodes. As a rule of thumb, a connection between two entity nodes is defined if a relationship exists between two entities. For example, the relationship between two entities in the IFC-SPF file is displayed in the following manner: If one of the IFC entity attributes contains the $IFC\ ID$ of another IFC entity, both entities should be connected.

For example entity **IfcProject** in Algorithm 4.2 contains $IFC\ IDs$ #2, (#3), and #4. This means that the entity **IfcProject** should be connected to the entity **IfcOwnerHistory** (#2), **IfcGeometricRepresentationContext** ((#3)), and **IfcUnitAssignment** (#4).

There may be two variants of connections in the Ryven. Either attribute receives a piece of information from one entity it has a relation to, or it may contain a list of entities connected to this attribute. For example, **IfcProject** can contain only one **IfcOwnerHistory** but can have several entities of a type **IfcGeometricRepresentationContext**. It differentiates based on the presence of brackets. In other words, if an $IFC\ ID$ finds itself in the brackets, it is a part of the entity list.

In order to incorporate both variants into solutions, two classes were implemented in the **fromIFCtoJSON** program: **combine_entities** and **connections**. Class **combine_entities** is responsible for searching all entities which should be included in the list and creating a new node, which will act as an intermediary. Class **connections** defines connections between all nodes, both intermediary and IFC entity nodes and stores them into the list $node\_connections$.

The objectives of nodes defined by class **combine_entities** and connections defined by class **connections** are discussed in the section 4.2.3

4.1.4 Creating JSON file

Under the requirements of the Ryven environment, the JSON file has a nested structure. On the first level of its hierarchy JSON file has three properties: $"generalinfo"$, $"requiredpackages"$, and $"scripts"$. All properties except $"scripts"$ have predefined values since every project uses the same packages to visualize nodes and the same Ryven version.

Property $"scripts"$ contains an array of objects which describe scripts created in spaces of the Ryven environment. In the framework of this thesis, an array of $"scripts"$ has only one object since all IFC file modification processes are completed in one space of the

Ryven environment. This object has several properties, the main one being $"flow"$. In the property $"flow"$ is stored all information regarding $nodes$ and $node\_connections$.

Creating the $flow$ dictionary initiates the process of creating the dictionary, which will later be converted into a JSON file. The dictionary $flow$ receives information by appending lists $nodes$ and $node\_connections$ to the corresponding keys.

Additionally, each node receives an indication of the location in the Ryven environment's space where it should be placed. Positions of the nodes are arranged based on the $IFC\ ID$ of the entities they represent, starting with the first node in the top right corner of the Ryven environment workspace. Other nodes are placed on the plane from right to left and top to bottom, with a constant spacing between nodes. This ensures that each node is appropriately positioned.

Once the dictionary $flow$ has been filled with the necessary information, it is added to the dictionary $script$, part of the list $scripts$. By appending the list to the key $"scripts"$ and adding information for properties $"generalinfo"$ and $"requiredpackages"$, the dictionary for the JSON file is completed. The resulting dictionary is converted into the JSON file using the function `dumps()` from the JSON library.

4.2 Ryven environment

After entity conversion from the IFC file into JSON file format, the JSON file is loaded in the Ryven environment. In Ryven, every node defined in the JSON file is visualized depending on the data JSON file contains about each node in the $"nodes"$ array. The same occurs with the $"connections"$ array: every connection is defined based on the node indexes and indexes of their attributes, which should be connected.

However, to create nodes and connections, Ryven should have all nodes defined as classes, describing each node's properties and functionalities. IFC entity **IfcProject** will be used as an example to demonstrate the structure of Ryven nodes.

4.2.1 IFCNode class

IFCNode class is a parent class for every IFC entity node class. It describes all methods inherited by its subclasses and used to modify the imported JSON file in the Ryven environment. This class describes how information from one entity node can be passed to another node and contains methods that support information transfer. Via these methods, information of output value is defined based on the first input value of the node, which coincides with $IFC\ ID$ of the entity, therefore allowing to pass $IFC\ ID$ to the other node using predefined methods.

For example, in the IFC file, **IfcProject** has an $IFC\ ID$ of #1. Thus by creating an **IfcProject** node and defining its first input as #1, **IfcProject** node will return output value #1. Figure 4.1 shows the **IfcProject** and its output value.

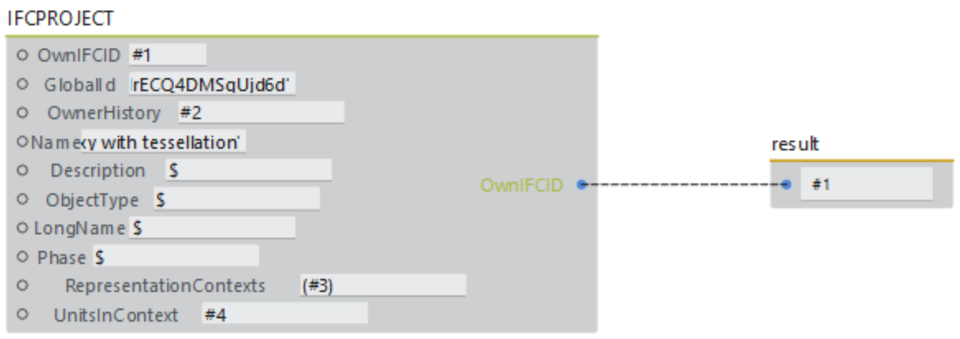

IFCPROJECT
OwnIFCID #1
GlobalId 'rECQ4DMSqUjd6d'
OwnerHistory #2
Name 'y with tessellation'
Description $
ObjectType $
LongName $
Phase $
RepresentationContexts (#3)
UnitsInContext #4

Figure 4.1: Output of textbfIfcProject node

4.2.2 IFC node classes

In order to create a node in the Ryven environment, for every IFC entity type, which should be visualized in Ryven, a IFC node class should be created. IFC node classes are subclasses of the **IFCNode** class. Thus they inherit all functionality from their parent class. The core difference between each node class is the number of inputs each node contains. It varies based on the number of attributes each IFC entity type consists of. Another difference between every node is the labeling of the inputs. Labeling of every input except the first input ($OwnIFCID$) must coincide with an attribute name of the entity type.

For example **IfcProject** consist of 9 attributes: 'GlobalId', 'OwnerHistory', 'Name', 'Description', 'ObjectType', 'LongName', 'Phase', 'RepresentationContexts', 'UnitsInContext'. Algorithm 4.6 demonstrates the input list of the **IfcProject** node class.

Algorithm 4.6: init_inputs list of the IfcProject node class

```
init_inputs = [
        NodeInputBP(dtype=dtypes.String(default="#", size='m'), label='
            OwnIFCID'),
        NodeInputBP(dtype=dtypes.String(default="$", size='l'), label='
            GlobalId'),
        NodeInputBP(dtype=dtypes.String(default="$", size='l'), label='
            OwnerHistory'),
        NodeInputBP(dtype=dtypes.String(default="$", size='l'), label='
            Name'),
        NodeInputBP(dtype=dtypes.String(default="$", size='l'), label='
            Description'),
        NodeInputBP(dtype=dtypes.String(default="$", size='l'), label='
            ObjectType'),
        NodeInputBP(dtype=dtypes.String(default="$", size='l'), label='
            LongName'),
        NodeInputBP(dtype=dtypes.String(default="$", size='l'), label='
            Phase'),
        NodeInputBP(dtype=dtypes.String(default="$", size='l'), label='
            RepresentationContexts'),
        NodeInputBP(dtype=dtypes.String(default="$", size='l'), label='
            UnitsInContext')
    ]
```

IFC Shema has multiple IFC entity types, which had to be included in the software. Therefore an IFC binding generator had to be incorporated into the solution to create nodes for all IFC entity types. However, the primary objective was to create a prototype program that would allow IFC file creation and modification. Thus, nodes were implemented manually and only for the IFC entity types residing in the file "tessellated-item.ifc".

4.2.3 Class combine_entities

As mentioned in subsection section 4.1.3, class **combine_entities** creates a node, which connects several entity nodes and combines $IFC\ ID$ of every connected node into a list. After $IFC\ IDs$ of entity nodes are combined into a list, this class passes it to the other node, which requires a combination of entity nodes.

To better illustrate the objectives **combine_entities** class, the functionality of this class is discussed based on the IFC entity **IfcUnitAssignment**, which is illustrated in Algorithm 4.7.

Algorithm 4.7: IFC entity IfcUnitAssignment from example-file

```
    #4=IfcUnitAssignment((#10,#11))
```

IfcUnitAssignment has only one attribute, $"Units"$. It is a list containing two $IFCIDs$, #10 and #11. This means that the attribute $"Units"$ should be simultaneously connected to the entity nodes **IfcSIUnit** (#10) and **IfcConversionBasedUnit** (#11). However, the Ryven environment does not allow connections to both nodes simultaneously. Thus an intermediary node **combine_entities** should be introduced.

The intermediary node **combine_entities** has several inputs. The number of inputs depends on the number of nodes it should combine. For example, in the case of **IfcUnitAssignment**, node **combine_entities** has two inputs since attribute $"Units"$ should be simultaneously connected to two other entity nodes. Therefore, each node is now connected to its own input from the **combine_entities** node. By connecting to the nodes, **combine_entities** receive an $IFC\ ID$ from every node. Therefore **combine_entities** can now append every $IFC\ ID$ to the list and return the list as an output. Connecting itself to the attribute $"Units"$ of the node **IfcUnitAssignment**, **combine_entities** node passes the output to the attribute. As a result, $"Units"$ receives a list with required $IFC\ IDs$ and is considered connected to both nodes through an intermediary node. Figure 4.2 should provide some insight into the abovementioned processes.

Class **combine_entities** differs from other node classes since it uses actions for its adjustments. Node actions are commands specified in the node class. Upon usage of

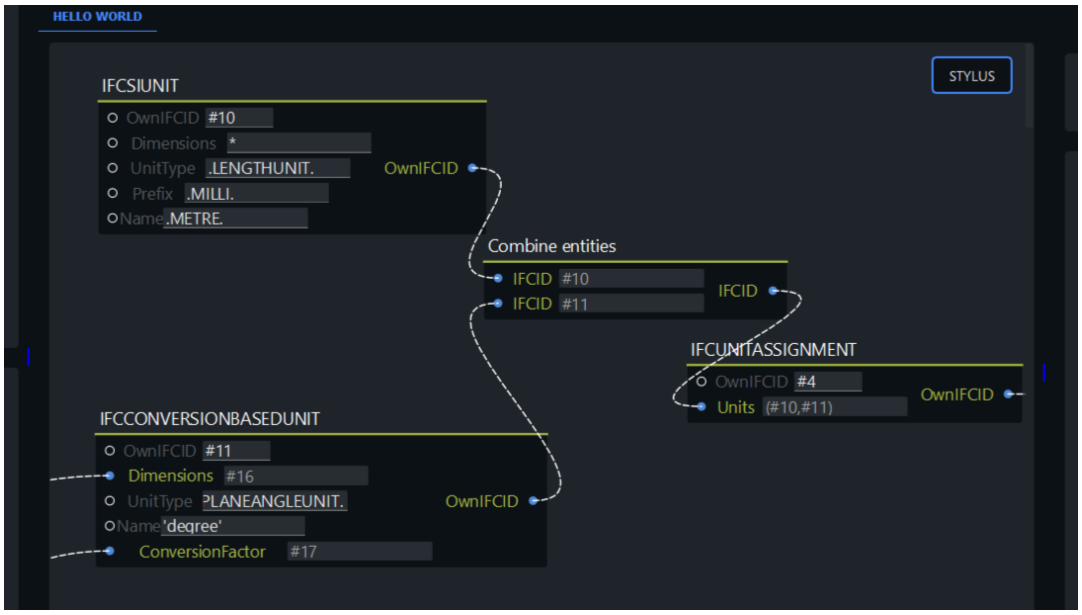


Figure 4.2: IfcUnitAssignment connected to IfcSIUnit and IfcConversionBasedUnit through Intermediary node "combine_entities"

these commands, the structure of the node changes. Node class **combine_entities** have only two actions: "add input" and "remove input i", where i is an index of the input.

Upon initialization,**combine_entities** node class has only one input parameter. By selecting "add input" action `add_operand_input` method is initialized. `add_operand_input` creates a new input parameter and an option to remove it using the method `remove_operand_input`, if it is no longer needed.

4.2.4 Functionalities of the Ryven environment

All modifications are carried out within the Ryven environment. This subsection describes how the IFC file can be modified in the Ryven environment and the functionalities responsible for it.

As soon as the Ryven environment is launched, a user is presented with two options: either open a JSON file, which represents an IFC file that needs to be updated, or start a new project to create a new IFC file from scratch. Upon selecting either of these options, the Ryven environment is opened. In case of opening a JSON file, the Ryven environment creates a node diagram representing an IFC file. Figure 4.3 illustrates the opened JSON file in the Ryven environment.

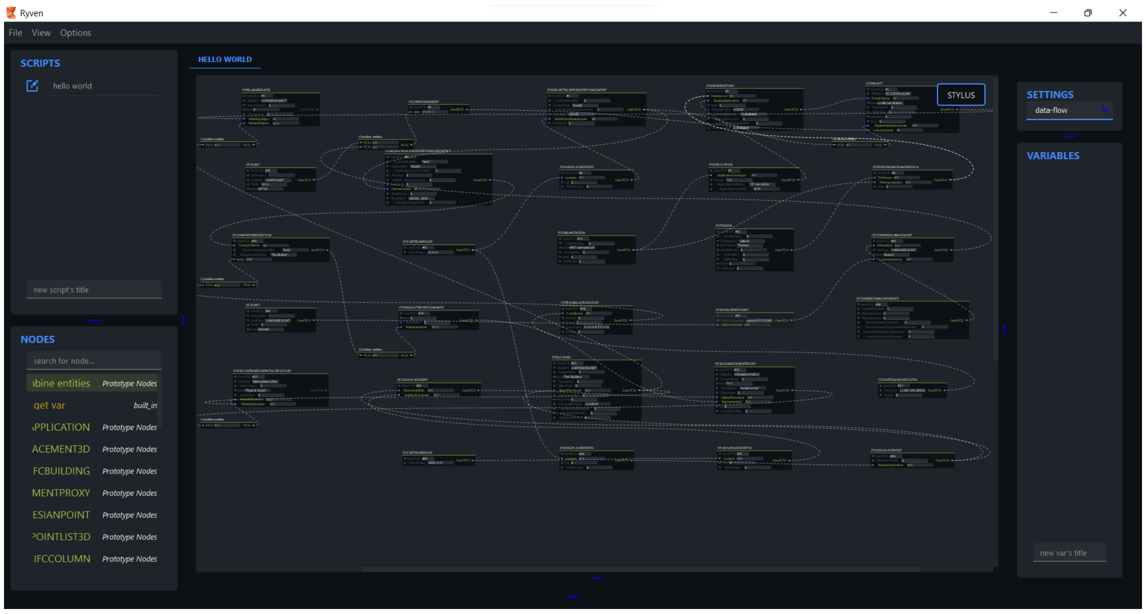


Figure 4.3: A node diagram representing an IFC file in the Ryven environment

Ryven environment has several options for modifying the IFC file. The first option is to create a new node. It is accomplished by "drag-and-dropping" a node from the "NODES" window in Ryven. Figure 4.4 shows a node menu in the Ryven environment.

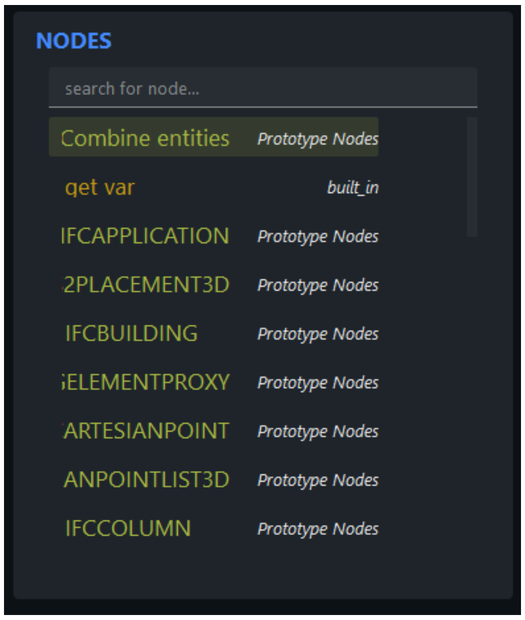

NODES
search for node...
Combine entities Prototype Nodes
get var built_in
IFCAPPLICATION Prototype Nodes
2PLACEMENT3D Prototype Nodes
IFCBUILDING Prototype Nodes
;ELEMENTPROXY Prototype Nodes
ARTESIANPOINT Prototype Nodes
ANPOINTLIST3D Prototype Nodes
IFCCOLUMN Prototype Nodes

Figure 4.4: A node menu in the Ryven environment

A second method of modifying an IFC file is to remove unnecessary nodes by selecting them in the Ryven space and deleting them. It is also possible to modify every node by changing its attributes. As an example, fig. 4.5 shows the node for the IFC entity type **IfcProject** with all its attributes, that can be changed.

In the same way that IFC entities are connected, nodes in the Ryven environment should also be linked. However, there are strict rules in the Ryven environment regarding connections between nodes. For example, a node's output can be connected to several

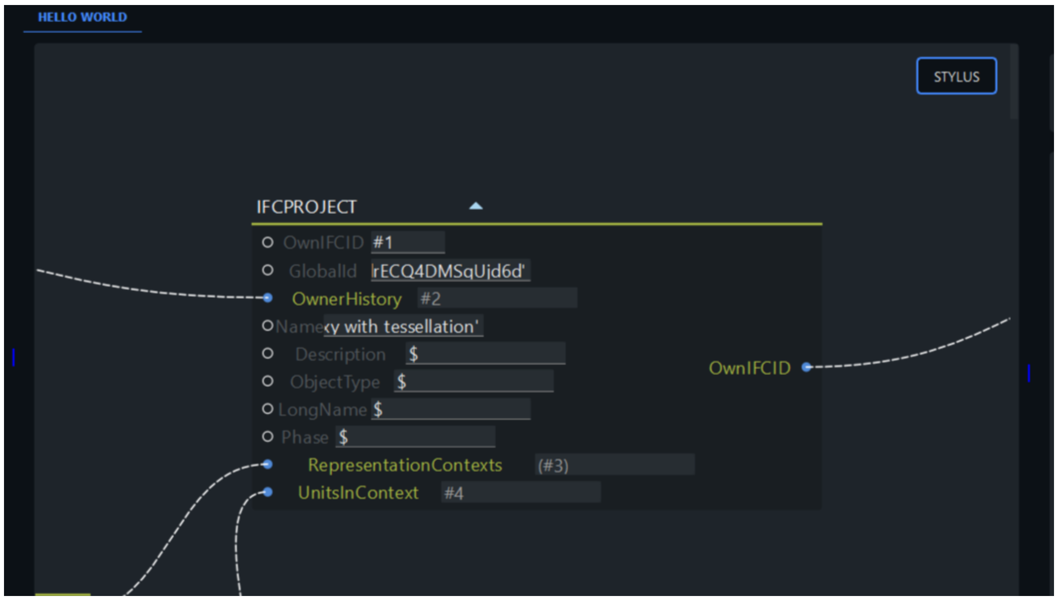


Figure 4.5: IfcProject node in Ryven environment

other nodes' inputs. On the contrary, each node input can receive a value only from one node. When several nodes representing IFC entities should be connected to the other node's input, a **combine_entities** intermediary node should be created. The node **combine_entities** and its functionalities are described in the section 4.2.3.

4.3 Converting JSON file into IFC file

After an IFC file converted into JSON is modified using the Ryven environment, all changes are saved in the new JSON file. However, a created file cannot be opened by most software, and it is mandatory to receive conducted changes in the initial format. Thus JSON file should be translated back into the IFC file format. The following section describes an implementation developed to accomplish this task.

Following the program's launch, the JSON file is opened using the JSON library for python, and all data are stored in the variable as a dictionary. Since only a fraction of the dictionary contains information required for the following procedures, the program searches through the dictionary to find the needed section. As a result, an application receives a list of nodes, where each node is an additional dictionary with information regarding either the IFC entity or a supporting class **combine_entities**.

While iterating through the list of nodes, the program searches for the nodes describing IFC entities. If IFC node is found, data from the node is saved in the object of a type **IFCNode**. **IFCNode** is a class consisting of methods for converting node data into STEP Physical File format. Since the data in the node's dictionary were encoded using the **Base64** library, it is necessary to decode the information back to the **ASCII** format. After the encoding process, data describing the $IFC\ ID$ and attribute values of the IFC entity is stored in a separate dictionary.

Since nodes have been created unsystematically during modification processes in Ryven, IFC entities stored in a newly created dictionary also lack any arrangement. Therefore, to secure the legibility of the created IFC file, entities in the dictionary are sorted based on their $IFC\ ID$, thus ensuring an ascending order of the IFC entities in the file.

After IFC entities are arranged, the program launches a function responsible for writing a new IFC file. Since IFC entities are arranged in ascending order in the newly created file, such an arrangement might differ from the initial file. Nevertheless, IFC file will work as expected since the order in which entities are listed has no impact on the file's content.

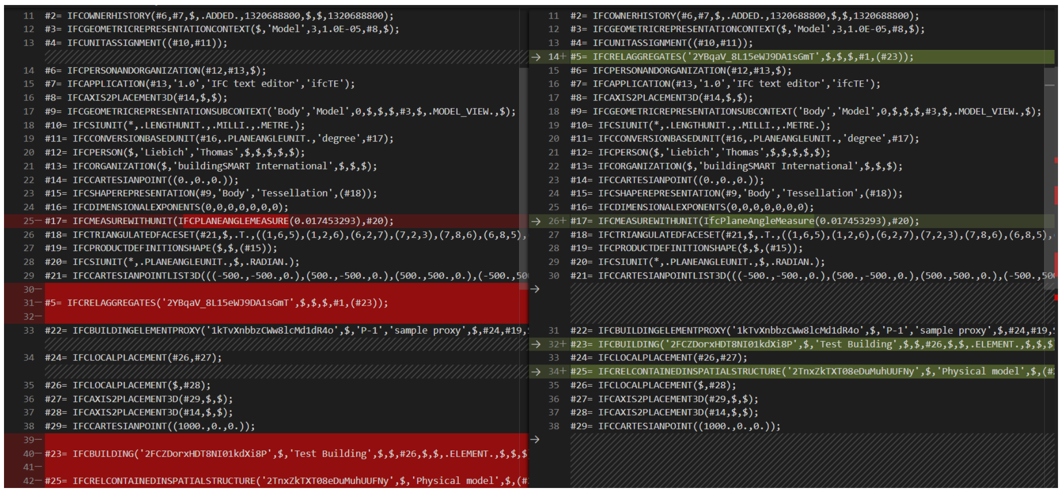
Chapter 5

Use Cases

This chapter describes the practical application of the implemented part of the IFC file creator and modifier prototype. The first example discussed in section 5.1 demonstrates the opening of file $"tessellated - item.ifc"$ since this IFC example file was used as a foundation, and every IFC entity type included in the example file was implemented manually as a IFC node class in Ryven. After that, IFC file creator and modifier limits are tested through examples created from $"tessellated - item.ifc"$ entities. In section 5.4, an example file containing IFC entities not implemented as node classes is loaded to demonstrate the prototype's limitations.

5.1 Rectangle shape represented as tessellated surface

The first example demonstrates changes to the IFC file upon opening it with IFC file creator and modifier prototype and saving it as a new file. During the first test, no alterations were conducted to the content of the IFC file. Therefore, no changes were made to the IFC entity attributes, and no entities were added or removed. However, the structure of the IFC file was changed due to the rearrangement of IFC entities because of the processes within **fromIFCtoJSON** and **fromJSONtoIFC**. The changes to the IFC file are visible in Figure 5.1.

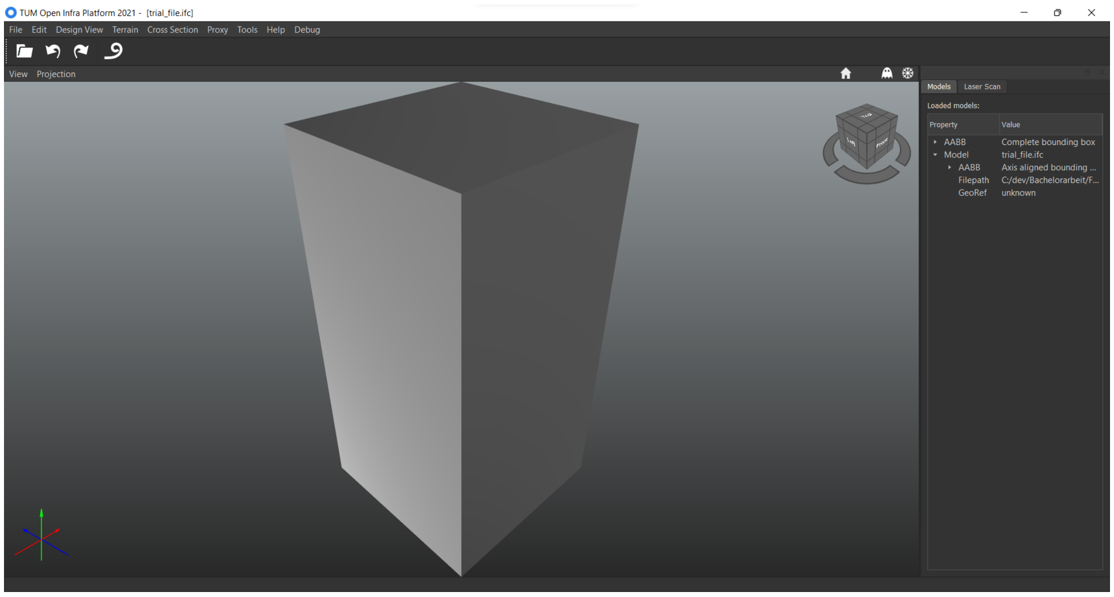


```
#2= IFCOWNERHISTORY(#6,#7,$,.ADDED.,1320688800,$,$,1320688800);
#3= IFCGEOMETRICREPRESENTATIONCONTEXT($,'Model',3,1.0E-05,#8,$);
#4= IFCUNITASSIGNMENT((#10,#11));

#6= IFCPERSONANDORGANIZATION(#12,#13,$);
#7= IFCAPPLICATION(#13,'1.0','IFC text editor','ifcTE');
#8= IFCAXIS2PLACEMENT3D(#14,$,$);
#9= IFCGEOMETRICREPRESENTATIONSUBCONTEXT('Body','Model',0,$,$,$,#3,$,.MODEL_VIEW.,$);
#10= IFCSIUNIT(*,.LENGTHUNIT.,.MILLI.,.METRE.);
#11= IFCCONVERSIONBASEDUNIT(#16,.PLANEANGLEUNIT.,'degree',#17);
#12= IFCPERSON($,'Liebich','Thomas',$,$,$,$,$);
#13= IFCORGANIZATION($,'buildingSMART International',$,$,$);
#14= IFCCARTESIANPOINT((0.,0.,0.));
#15= IFCSHAPEREPRESENTATION(#9,'Body','Tessellation',(#18));
#16= IFCDIMENSIONALEXPONENTS(0,0,0,0,0,0,0);
#17= IFCMEASUREWITHUNIT(IFCPLANEANGLEMEASURE(0.017453293),#20);
#18= IFCTRIANGULATEDFACESET(#21,$,.T.,((1,6,5),(1,2,6),(6,2,7),(7,2,3),(7,8,6),(6,8,5),
#19= IFCPRODUCTDEFINITIONSHAPE($,$,(#15));
#20= IFCSIUNIT(*,.PLANEANGLEUNIT.,$,.RADIAN.);
#21= IFCCARTESIANPOINTLIST3D(((-500.,-500.,0.),(500.,-500.,0.),(500.,500.,0.),(-500.,50

#5= IFCRELAGGREGATES('2YBqaV_8L15eWJ9DA1sGmT',$,$,$,#1,(#23));

#22= IFCBUILDINGELEMENTPROXY('1kTvXnbbzCWw8lcMd1dR4o',$,'P-1','sample proxy',$,#24,#19,

#24= IFCLOCALPLACEMENT(#26,#27);

#26= IFCLOCALPLACEMENT($,#28);
#27= IFCAXIS2PLACEMENT3D(#29,$,$);
#28= IFCAXIS2PLACEMENT3D(#14,$,$);
#29= IFCCARTESIANPOINT((1000.,0.,0.));

#23= IFCBUILDING('2FCZDorxHDT8NI01kdXi8P',$,'Test Building',$,$,#26,$,$,.ELEMENT.,$,$,$

#25= IFCRELCONTAINEDINSPATIALSTRUCTURE('2TnxZkTXT08eDuMuhUUFNy',$,'Physical model',$,(#
```

```
#2= IFCOWNERHISTORY(#6,#7,$,.ADDED.,1320688800,$,$,1320688800);
#3= IFCGEOMETRICREPRESENTATIONCONTEXT($,'Model',3,1.0E-05,#8,$);
#4= IFCUNITASSIGNMENT((#10,#11));
#5= IFCRELAGGREGATES('2YBqaV_8L15eWJ9DA1sGmT',$,$,$,#1,(#23));
#6= IFCPERSONANDORGANIZATION(#12,#13,$);
#7= IFCAPPLICATION(#13,'1.0','IFC text editor','ifcTE');
#8= IFCAXIS2PLACEMENT3D(#14,$,$);
#9= IFCGEOMETRICREPRESENTATIONSUBCONTEXT('Body','Model',0,$,$,$,#3,$,.MODEL_VIEW.,$);
#10= IFCSIUNIT(*,.LENGTHUNIT.,.MILLI.,.METRE.);
#11= IFCCONVERSIONBASEDUNIT(#16,.PLANEANGLEUNIT.,'degree',#17);
#12= IFCPERSON($,'Liebich','Thomas',$,$,$,$,$);
#13= IFCORGANIZATION($,'buildingSMART International',$,$,$);
#14= IFCCARTESIANPOINT((0.,0.,0.));
#15= IFCSHAPEREPRESENTATION(#9,'Body','Tessellation',(#18));
#16= IFCDIMENSIONALEXPONENTS(0,0,0,0,0,0,0);
#17= IFCMEASUREWITHUNIT(IfcPlaneAngleMeasure(0.017453293),#20);
#18= IFCTRIANGULATEDFACESET(#21,$,.T.,((1,6,5),(1,2,6),(6,2,7),(7,2,3),(7,8,6),(6,8,5),
#19= IFCPRODUCTDEFINITIONSHAPE($,$,(#15));
#20= IFCSIUNIT(*,.PLANEANGLEUNIT.,$,.RADIAN.);
#21= IFCCARTESIANPOINTLIST3D(((-500.,-500.,0.),(500.,-500.,0.),(500.,500.,0.),(-500.,50

#22= IFCBUILDINGELEMENTPROXY('1kTvXnbbzCWw8lcMd1dR4o',$,'P-1','sample proxy',$,#24,#19,
#23= IFCBUILDING('2FCZDorxHDT8NI01kdXi8P',$,'Test Building',$,$,#26,$,$,.ELEMENT.,$,$,$
#24= IFCLOCALPLACEMENT(#26,#27);
#25= IFCRELCONTAINEDINSPATIALSTRUCTURE('2TnxZkTXT08eDuMuhUUFNy',$,'Physical model',$,(#
#26= IFCLOCALPLACEMENT($,#28);
#27= IFCAXIS2PLACEMENT3D(#29,$,$);
#28= IFCAXIS2PLACEMENT3D(#14,$,$);
#29= IFCCARTESIANPOINT((1000.,0.,0.));
```

Figure 5.1: The difference in the arrangement of the IFC file

Nevertheless, the IFC file is not impacted by the order in which entities are listed. Thus, the file $"tessellated - item.ifc"$ is open correctly by the **TUM OpenInfraPlatform** as it is visible in Figure 5.2.

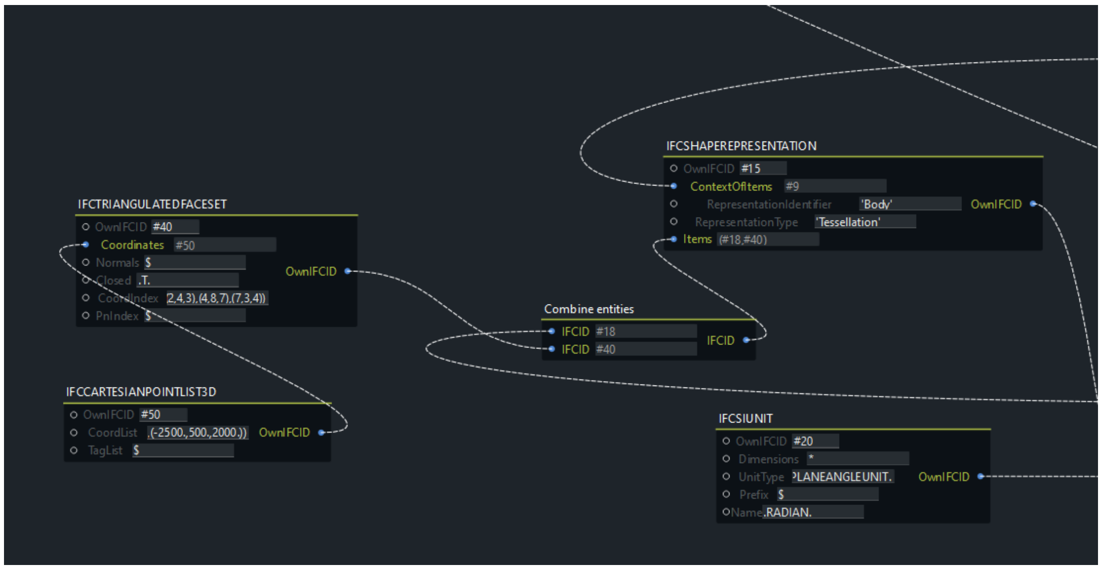


Property	Value
AABB	Complete bounding box
Model	trial_file.ifc
AABB	Axis aligned bounding ...
Filepath	C:/dev/Bachelorarbeit/F...
GeoRef	unknown

Figure 5.2: Tessellated item in TUM OpenInfraPlatform

5.2 Two rectangle shapes represented as tessellated surfaces

The second example shows how new geometries can be added to the initial file using the same entity node. For example, two new nodes are added in the Ryven environment to the file to add a new rectangular shape: IFCTRIANGULATEDFACET and IFCCARTESIAN-POINTLIST3D. IFCTRIANGULATEDFACET node is connected to the **combine_entities** node, thus appending itself to the list, which is passed to the IFCSHAPEREPRESENTA-TION. Figure 5.3 illustrates how the IFCTRIANGULATEDFACET node is passed to the IFCSHAPEREPRESENTATION as a part of the list.


IFCSHAPEREPRESENTATION
OwnIFCID #15
ContextOfItems #9
RepresentationIdentifier 'Body'
RepresentationType 'Tessellation'
Items (#18,#40)
OwnIFCID

IFCTRIANGULATEDFACESET
OwnIFCID #40
Coordinates #50
Normals $
Closed .T.
CoordIndex 2,4,3),(4,8,7),(7,3,4))
PnIndex $
OwnIFCID

Combine entities
IFCID #18
IFCID #40
IFCID

IFCCARTESIANPOINTLIST3D
OwnIFCID #50
CoordList ,(-2500.,500.,2000.))
TagList $
OwnIFCID

IFCSIUNIT
OwnIFCID #20
Dimensions *
UnitType PLANEANGLEUNIT.
Prefix $
Name .RADIAN.
OwnIFCID

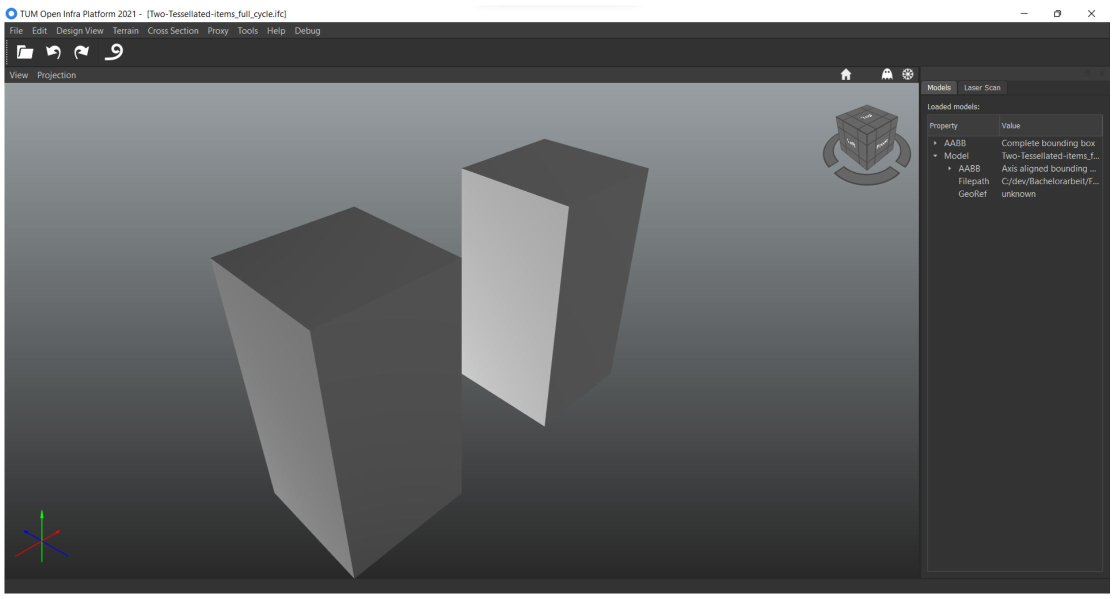
Figure 5.3: Addition of a new tessellated item using Ryven environment

Upon adding IFCTRIANGULATEDFACET and IFCCARTESIANPOINTLIST3D to the IFC file in the Ryven environment, the file is saved in JSON. Following that JSON file is converted to the IFC, and as a result, two new IFC entities are a part of the IFC file. Algorithm 5.1 demonstrates new entities IFCTRIANGULATEDFACET and IFCCARTESIANPOINTLIST3D as a part of the IFC file.

Algorithm 5.1: IFC entities of a new tessellated item

```
#29= IFCCARTESIANPOINT((1000.,0.,0.));
#40= IFCTRIANGULATEDFACESET(#50,$,.T.,((1,6,5),(1,2,6),(6,2,7),(7,2,3)
    ,(7,8,6),(6,8,5),(5,8,1),(1,8,4),(4,2,1),(2,4,3),(4,8,7),(7,3,4)),$
    );
#50= IFCCARTESIANPOINTLIST3D(((-2500.,-500.,0.),(-1500.,-500.,0.)
    ,(-1500.,500.,0.),(-2500.,500.,0.),(-2500.,-500.,2000.)
    ,(-1500.,-500.,2000.),(-1500.,500.,2000.),(-2500.,500.,2000.)),$);
ENDSEC;
```

Figure 5.4 shows a newly created IFC file that has been opened by the **TUM OpenInfraPlatform**.



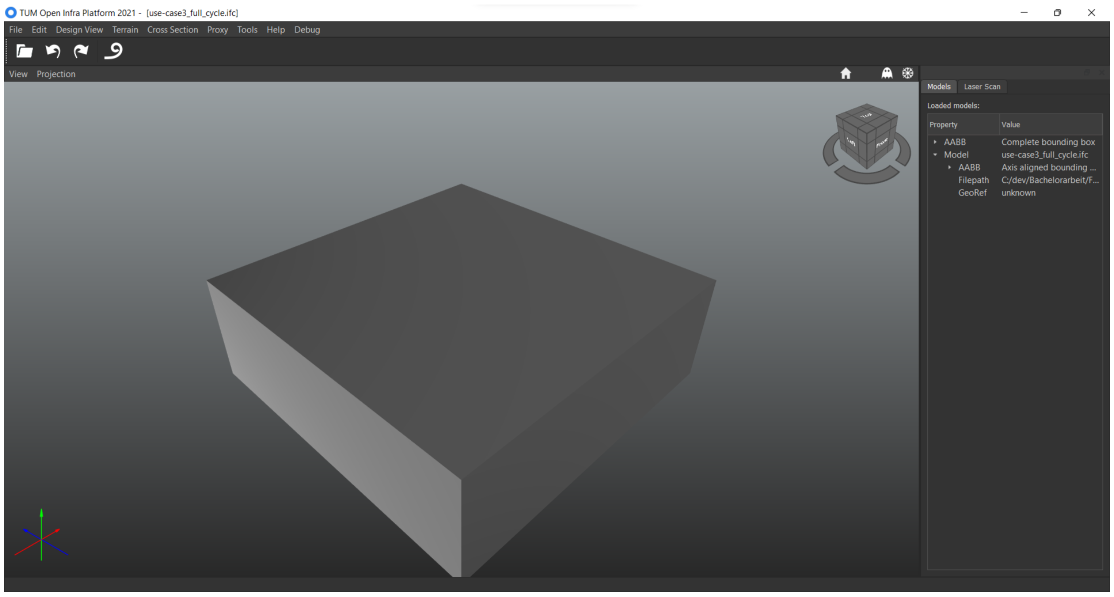
Loaded models:

Property	Value
AABB	Complete bounding box
Model	Two-Tessellated-items_f...
AABB	Axis aligned bounding ...
Filepath	C:/dev/Bachelorarbeit/F...
GeoRef	unknown

Figure 5.4: Two rectangle shapes represented as tessellated surfaces

5.3 New rectangular tessellated surface

The third example demonstrates the creation of a new IFC file using IFC file creator and modifier programs. The new file is created within the Ryven environment by selecting the previously defined nodes for entities from "$tessellated - item.ifc$" and connecting them, thereby developing a structure that describes a new IFC file in the Ryven. It is saved as an JSON file and afterwards converted into the IFC format using **fromJSONtoIFC** program. Figure 5.5 illustrates the results of creating the new IFC file.

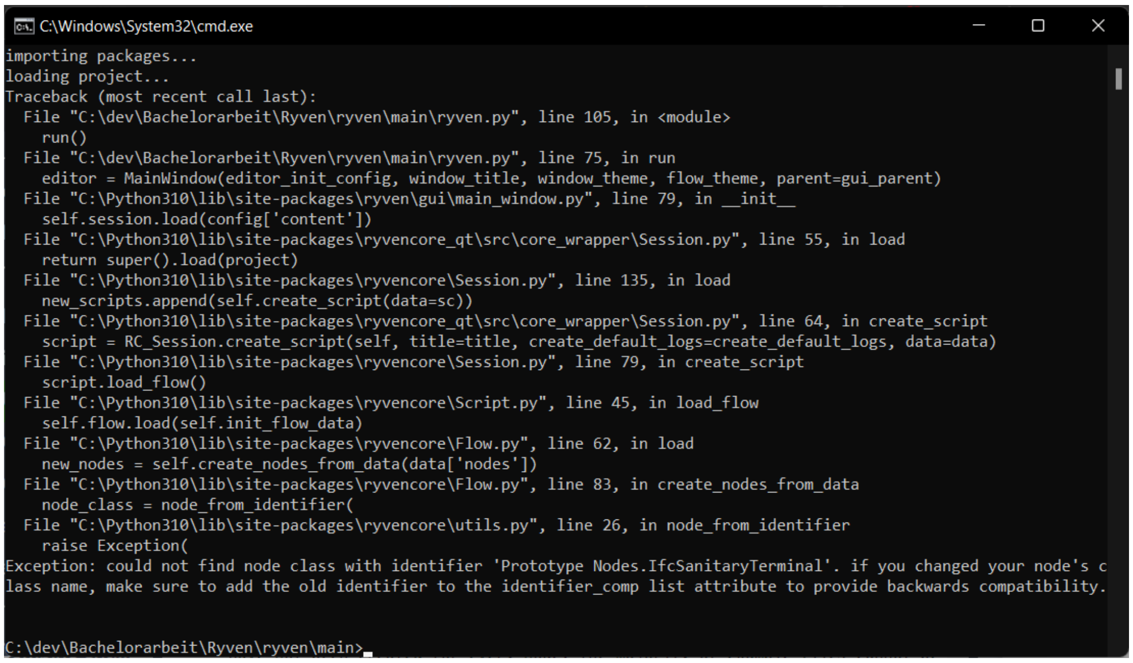


Property	Value
AABB	Complete bounding box
Model	use-case3_full_cycle.ifc
AABB	Axis aligned bounding ...
Filepath	C:/dev/Bachelorarbeit/F...
GeoRef	unknown

Figure 5.5: New rectangular tessellated surface in TUM OpenInfraPlatform

5.4 Unsupported IFC entity types

Forth example demonstrates the limits of the IFC file creator and modifier programs. Node classes describing IFC entity types are created manually and, therefore, have not been created for every node. As a result, most example files cannot be opened. For example, Figure 5.6 illustrates an error message from Ryven stating that nodes for IFC entity type IFCSANITARYTERMINAL cannot be created since no node class exists for this entity type.



```
importing packages...
loading project...
Traceback (most recent call last):
  File "C:\dev\Bachelorarbeit\Ryven\ryven\main\ryven.py", line 105, in <module>
    run()
  File "C:\dev\Bachelorarbeit\Ryven\ryven\main\ryven.py", line 75, in run
    editor = MainWindow(editor_init_config, window_title, window_theme, flow_theme, parent=gui_parent)
  File "C:\Python310\lib\site-packages\ryven\gui\main_window.py", line 79, in __init__
    self.session.load(config['content'])
  File "C:\Python310\lib\site-packages\ryvencore_qt\src\core_wrapper\Session.py", line 55, in load
    return super().load(project)
  File "C:\Python310\lib\site-packages\ryvencore\Session.py", line 135, in load
    new_scripts.append(self.create_script(data=sc))
  File "C:\Python310\lib\site-packages\ryvencore_qt\src\core_wrapper\Session.py", line 64, in create_script
    script = RC_Session.create_script(self, title=title, create_default_logs=create_default_logs, data=data)
  File "C:\Python310\lib\site-packages\ryvencore\Session.py", line 79, in create_script
    script.load_flow()
  File "C:\Python310\lib\site-packages\ryvencore\Script.py", line 45, in load_flow
    self.flow.load(self.init_flow_data)
  File "C:\Python310\lib\site-packages\ryvencore\Flow.py", line 62, in load
    new_nodes = self.create_nodes_from_data(data['nodes'])
  File "C:\Python310\lib\site-packages\ryvencore\Flow.py", line 83, in create_nodes_from_data
    node_class = node_from_identifier(
  File "C:\Python310\lib\site-packages\ryvencore\utils.py", line 26, in node_from_identifier
    raise Exception(
Exception: could not find node class with identifier 'Prototype Nodes.IfcSanitaryTerminal'. if you changed your node's c
lass name, make sure to add the old identifier to the identifier_comp list attribute to provide backwards compatibility.


C:\dev\Bachelorarbeit\Ryven\ryven\main>_
```

Figure 5.6: Unsupported IFC entity type IFCSANITARYTERMINAL

Chapter 6

Conclusion

6.1 Summary

This bachelor thesis aimed to create a more user-friendly approach for creating and editing IFC format files. It is usually accomplished using text editors. However, such a method is less intuitive and can cause typos in the created file, which are not easy to identify afterward. Instead, in the framework of this thesis was suggested to develop software using Visual Programming Language technology, thus implementing a more convenient approach for creating and modifying IFC files.

The Ryven flow-based visual scripting environment was selected as a Visual Programming Language library to avoid creating a software Visual Programming environment from scratch and focus on incorporating tools for IFC file modification. However, Ryven accepts only files with JSON extension. Thus it was necessary to implement a converter that would translate an IFC file format into the format required by the Ryven environment.

After converting an IFC file into JSON and opening it with Ryven, IFC entities can be created and adjusted. It is accomplished via nodes representing any variable or function in the Ryven environment. Therefore, to successfully modify the IFC file, each IFC entity type of this file was defined as a node class, describing how the IFC entity's attributes and $IFC\ ID$ should be incorporated into the node. Ideally, for every existing IFC entity type, a separate node class had to be created. It could be accomplished by incorporating an IFC binding generator and creating the required classes for every IFC entity type. However, since the software was created as a prototype and was required to modify only the IFC example file "tessellated-item.ifc", node classes were implemented manually for IFC entities from this file.

Upon completing the modification of the IFC file via the Ryven environment, all modifications were stored in the newly created JSON file. However, it was necessary to return its initial representation format to the file. Thus, a converter was also implemented to translate a JSON file format back to the IFC format.

As a result, tools were created that translate IFC files into JSON files, modify them using the Ryven environment, and convert results back into IFC files. Thus a method of creating and modifying IFC files using Visual Programming Language was created.

6.2 Evaluation

The developed tools complete all steps required for modifying an IFC file using the Ryven flow-based visual scripting environment. However, the developed solution has certain limitations.

First and foremost, tools in the actual state can only convert IFC files consisting of IFC entities for which node classes in the Ryven node library were defined. Currently, a node class should be created manually for every entity type that is not a part of the example file "tessellated-item.ifc" to launch other IFC files. Otherwise, the tools can only be used with IFC files that contain only entity types presented in the "tessellated-item.ifc" example file.

In addition, the current software tools are separated into three parts, launched independently of one another. For example, an IFC file is opened by the program **fromIFCtoJSON**, which converts it into JSON file format and stores it as a JSON file. Afterward, the Ryven environment is launched separately, and the JSON file received from the program **fromIFCtoJSON** is manually accessed. Each change made using Ryven is saved as a JSON file. Lastly, **fromJSONtoIFC** opens the saved JSON file and, after translating it into STEP Physical Format, saves it as an IFC. Completing this sequence of steps is inconvenient for the user and can also lead to failure due to the possibility of loading a false file.

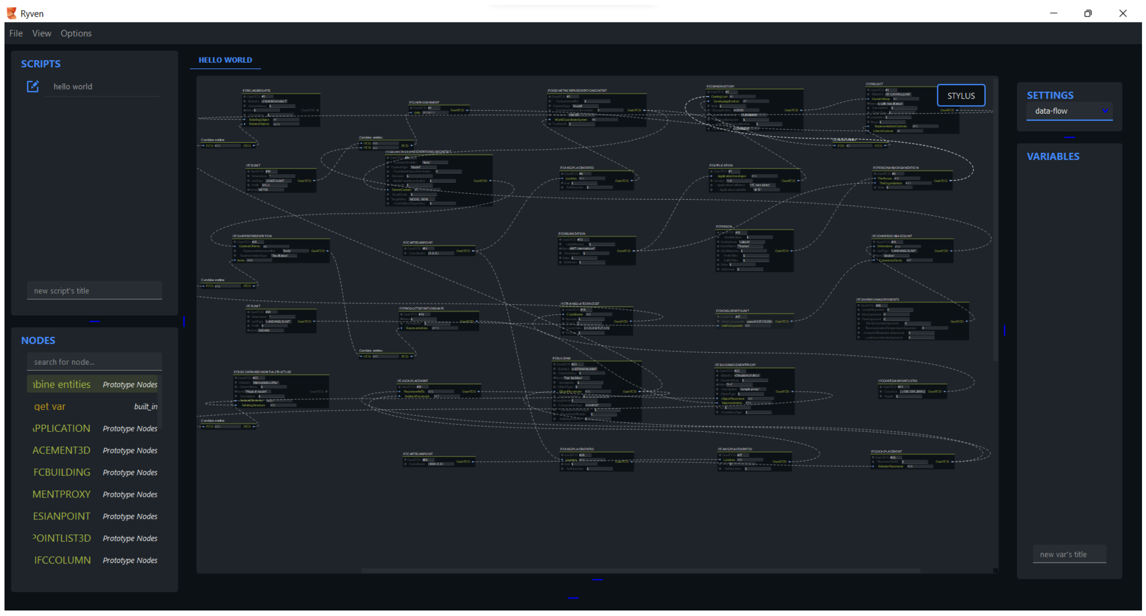

Ryven
File View Options
SCRIPTS
hello world
new script's title
NODES
search for node...
ıbine entities Prototype Nodes
get var built_in
ᴧPPLICATION Prototype Nodes
ACEMENT3D Prototype Nodes
FCBUILDING Prototype Nodes
MENTPROXY Prototype Nodes
ESIANPOINT Prototype Nodes
ᴐOINTLIST3D Prototype Nodes
IFCCOLUMN Prototype Nodes
HELLO WORLD
STYLUS
SETTINGS
data-flow
VARIABLES
new var's title

Figure 6.1: The positioning of the nodes in the Ryven environment

Figure 6.1 demonstrates another limitation of the software. It is related to the quantity of the IFC entities that the Ryven software can open and their placement. Currently, during the creation of the JSON file, the program **fromIFCtoJSON** defines a default position for every node representing an IFC entity. Positions of the nodes are arranged based on the $IFC\ ID$ of the entities they represent, starting with the first node in the top right corner of the Ryven environment workspace. Other nodes are placed on the plane from right to left and top to bottom, with a constant spacing between nodes. Thus, the number of IFC entities that the Ryven environment can open is limited by the dimensions of the Ryven environment workspace and the width of the spacing between nodes. At the same time, such a way of placing nodes in the Ryven environment workspace ignores the connections between nodes, thus decreasing the clearness of this approach.

Furthermore, the user interface of the Ryven environment probably was not developed for the use of nodes with long titles like IFC entities have. Figure 6.2 demonstrates the limitations of the Ryven user interface.

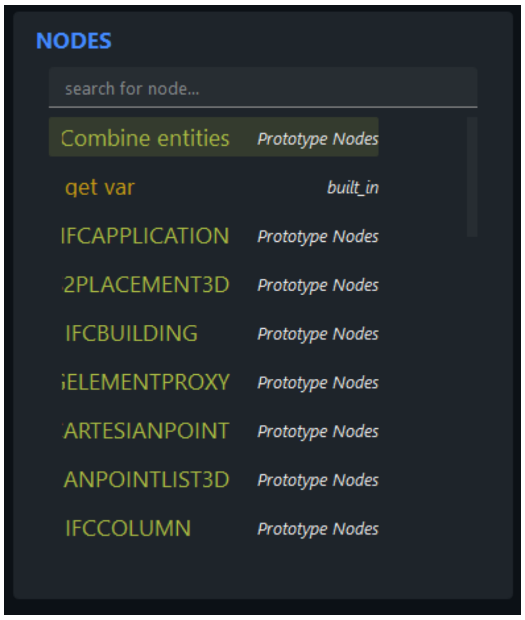

NODES
search for node...
Combine entities Prototype Nodes
get var built_in
IFCAPPLICATION Prototype Nodes
2PLACEMENT3D Prototype Nodes
IFCBUILDING Prototype Nodes
;ELEMENTPROXY Prototype Nodes
ARTESIANPOINT Prototype Nodes
ANPOINTLIST3D Prototype Nodes
IFCCOLUMN Prototype Nodes

Figure 6.2: The limitations of the node menu in the Ryven environment

The node menu has insufficient dimensions to display IFC entity types completely. Therefore, it worsens the user experience while using this program and increases the chance of failure since nodes representing wrong IFC entity types can be selected.

Therefore, despite being able to modify and create IFC files using the Visual Programming Language, this software lacks features VPL was supposed to address. Instead, temporary software for IFC file creation and modification is not intuitive and user-friendly.

6.3 Future development

The prototype software demonstrates promising results in creating and modifying IFC files using the Visual Programming Language. Users can create and adjust IFC files containing IFC entities from the example "tessellated-item.ifc". However, being only a prototype, this solution contains some limitations described in section 6.2.

Therefore there is a variety of improvements that can be completed, thus improving the usability of the software. For example, an IFC binding generator from TUM OpenInfraPlatform should be incorporated into the solution to create and modify IFC files regardless of the IFC entities they contain. One of the possible solutions how to accomplish this is by applying the **pybind11** library. **pybind11** is a header-only library that allows the creation of Python bindings of existing C++ code (JAKOB, 2022).

Another essential improvement is implementing more appropriate placement for IFC entity nodes in the Ryven environment. It can be accomplished by placing a node that is branching out in the middle and arranging its branches around this node. At the same time, a placement hierarchy can be defined by the relationships between entities these nodes represent. For example, the starting node is placed at the top, and all its branches are beneath it.

This can be accomplished only by solving another placement issue described in section 6.2, the Ryven environment workspace limits. The workspace limits should be adjusted based on the number of nodes an IFC file contains and the area nodes occupy.

Therefore the solution provided in this thesis can be enhanced and, as a result, be more suitable for the objectives it was designed for.

Appendix A

Code

The code created for the IFC file creator and modifier prototype can be found on GitHub: https://github.com/SamuilsRulovs/Bachelor_thesis

The Ryven environment can be found on GitHub in the repository from LEON THOMM (2022): https://github.com/leon-thomm/Ryven

Bibliography

AUTODESK. (2022). What is visual programming? [Accessed: 14.10.2022]. https://primer.dynamobim.org/01_Introduction/1-1_what_is_visual_programming.html

BIBLUS. (2020). Ifc format and open bim, all you need to know. https://biblus.accasoftware.com/en/ifc-format-and-open-bim-all-you-need-to-know/

BIBLUS. (2022). What is ifc 5? https://biblus.accasoftware.com/en/what-is-ifc-5/

BORRMANN, A. (2021). Bau- und umweltinformatik ergänzungsmodul vorlesung teil 7: Grundlagen ifc datenmodell.

BORRMANN, A., KÖNIG, M., KOCH, C., & BEETZ, J. (2015). *Building information modeling: Technologische grundlagen und industrielle praxis*.

buildingSMART INTERNATIONAL. (2022a). Ifc release notes [Accessed: 13.10.2022]. https://technical.buildingsmart.org/standards/ifc/ifc-schema-specifications/ifc-release-notes/

buildingSMART INTERNATIONAL. (2022b). Industry foundation classes (ifc) [Accessed: 23.09.2022]. https://www.buildingsmart.org/standards/bsi-standards/industry-foundation-classes

COLLAO, J., LOZANO-GALANT, F., LOZANO-GALANT, J. A., & TURMO, J. (2021). Bim visual programming tools applications in infrastructure projects: A state-of-the-art review. https://doi.org/10.3390/app11188343

DOCS, M. W. (2022). Base64. https://developer.mozilla.org/en-US/docs/Glossary/Base64

GALLAHER, M. P., O'CONNOR, A. C., DETTBARN, JR., J. L., & GILDAY, L. T. (2004). Cost analysis of inadequate interoperability in the u.s. capital facilities industry. https://nvlpubs.nist.gov/nistpubs/gcr/2004/nist.gcr.04-867.pdf

HECHT, H., & JAUD, Š. (2019). Tum openinfraplatform: The open-source bim visualisation software.

IFCOPENSHELL. (2022). Ifcopenshell github repository [Accessed: 23.10.2022]. https://github.com/IfcOpenShell/IfcOpenShell

ISO. (2018). Industry foundation classes (ifc) for data sharing in the construction and facility [Accessed: 23.09.2022]. https://www.iso.org/standard/70303.html

JAKOB, W. (2022). Pybind11 - seamless operability between c++11 and python [Accessed: 26.10.2022]. https://github.com/pybind/pybind11

LEON THOMM. (2022). Ryven - flow-based visual scripting for python [Accessed: 23.10.2022]. https://ryven.org/

PREIDEL, C., DAUM, S., & BORRMANN, A. (2017). Data retrieval from building information models based on visual programming. https://link.springer.com/content/pdf/10.1186/s40327-017-0055-0.pdf.

ROBERT MCNEEL & ASSOCIATES. (2022). Grasshopper—new in rhino 6 [Accessed: 24.10.2022]. https://www.rhino3d.com/6/new/grasshopper

RUSSELL GREENE AND AUN-ALI ZAIDI. (2017). Chigraph github repository [Accessed: 23.10.2022]. https://github.com/chigraph/chigraph

SAILI, S. (2021). Grasshopper 3d: A modeling software redefining the design process. https://parametric-architecture.com/grasshopper-3d-a-modeling-software-redefining-the-design-process/

THE CHAIR OF COMPUTATIONAL MODELING AND SIMULATION. (2019). Tum open infra platform github repository [Accessed: 23.09.2022]. https://github.com/tumcms/Open-Infra-Platform/

TYSON, M. (2022). What is json? the universal data format. https://www.infoworld.com/article/3222851/what-is-json-a-better-format-for-data-exchange.html

UNIVERSITY OF ILLINOIS. (2007). The llvm compiler infrastructure project [Accessed: 23.10.2022]. https://llvm.org/

YANG, S. (2015). Dynamo and computational bim - part 1: Introduction and resources. https://the360view.typepad.com/blog/2015/02/dynamo-and-computational-bim-part-1-introduction-and-resources.html

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.

München 28. 10. 2022 [illegible]

Location, Date, Signature