

UNIVERSITY OF WEST BOHEMIA
FACULTY OF APPLIED SCIENCES
DEPARTMENT OF GEOMATICS

Time-variable Visualization from Sensor Data Inside Building in a 3D GIS Environment

MASTER THESIS

Bc. Jan Macura

Thesis supervisor
Ing. Karel Jedlička, Ph.D.

Pilsen, 2019

ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA APLIKOVANÝCH VĚD
KATEDRA GEOMATIKY

Časově proměnná vizualizace ze sensorových dat v budově v prostředí 3D GIS

DIPLOMOVÁ PRÁCE

Bc. Jan Macura

Vedoucí práce
Ing. Karel Jedlička, Ph.D.

Plzeň, 2019

ZÁPADOČESKÁ UNIVERZITA V PLZNI

Fakulta aplikovaných věd

Akademický rok: 2018/2019

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Jan MACURA**

Osobní číslo: **A16N0011P**

Studijní program: **N3602 Geomatika**

Studijní obor: **Geomatika**

Název tématu: **Časově proměnná vizualizace ze sensorových dat v budově v prostředí 3D GIS**

Zadávací katedra: **Katedra geomatiky**

Z á s a d y p r o v y p r a c o v á n í :


1. Úvod do 3D geografických informačních systémů (GIS) a Building Information Managementu (BIM) s přihlédnutím k tématu práce.
2. Prozkoumání možností technologie CesiumJS pro dynamickou prezentaci sensorových dat ve 3D prostředí.
3. Implementace řešení na vybrané budově.

Rozsah grafických prací: dle potřeby
Rozsah kvalifikační práce: cca 45 stran
Forma zpracování diplomové práce: tištěná/elektronická
Seznam odborné literatury:

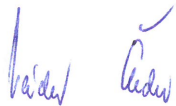
- ABDUL-RAHMAN, Alias a PILOUK, Morakot. Spatial Data Modelling for 3D GIS. Berlin: Springer, 2008. 289 s. ISBN 978-3-540-74166-4.
- JEDLIČKA, Karel. A comprehensive overview of a core of 3D GIS. In: 7th International Conference on Cartography and GIS Proceedings. Sozopol: Bulgarian Cartographic Association, 2018, 464472. ISSN 1314-0604.
- MURSHED, Syed Monjur, AL-HYARI, Ayah Mohammad, WENDEL, Jochen a ANSART, Louise. Design and Implementation of a 4D Web Application for Analytical Visualization of Smart City Applications. ISPRS International Journal of Geo-Information. 2018, 7(7), 276. DOI 10.3390/ijgi7070276.
- PUPO, Luis Henrique Rodríguez. Sensor Data Visualization in Virtual Globes. 2011. 52 s. Diplomová práce. Master of Science in Geospatial Technologies. Vedoucí práce Alain Tamayo FONG.
- TSAI, Fuan, LAI, Jhe-Syuan a LIU, Yu-Ching. An alternative open source Web-based 3D GIS: Cesium engine environment. In: ACRS 2015 - 36th Asian Conference on Remote Sensing: Fostering Resilient Growth in Asia, Proceedings. Manila: 2015.

Vedoucí diplomové práce: Ing. Karel Jedlička, PhD.
Katedra geomatiky

Datum zadání diplomové práce: 15. října 2018
Termín odevzdání diplomové práce: 17. května 2019


Doc. Dr. Ing. Vlasta Radová
děkanka




Doc. Ing. Václav Čada, CSc.
vedoucí katedry

V Plzni dne 15. října 2018

Declaration

I declare that this thesis is my original work of authorship that I have created myself. All resources, sources and literature, which I used in my thesis, are properly cited indicating the full link to the appropriate source.

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

V Plzni dne

.....

Jan Macura

Acknowledgments

On this place, I would like to thank for understanding to all those, who I have been neglecting or ignoring completely during my work on this thesis – namely my partner Kateřina, my family, my friends and colleagues. Big thanks goes to Ing. Karel Jedlička, Ph.D., without his systematic guidance and constructive critique this thesis would never come to life. Last but not least, I thank to Ing. Martin Střelec, Ph.D., for his precious and willing consultations on heat transfer simulation model, and to Ing. Michal Kepka, Ph.D. for his helpful advice with the CesiumJS platform.

Poděkování

Na tomto místě bych chtěl především poděkovat za pochopení všem těm, které jsem během psaní této práce zanedbával nebo zcela ignoroval – zejména mé partnerce Kateřině, mé rodině, mým přátelům a kolegům. Velký dík patří Ing. Karlu Jedličkovi, Ph.D., bez jehož soustavného vedení a konstruktivní kritiky by tato práce nikdy nevznikla. V neposlední řadě děkuji Ing. Martinu Střelcovi, Ph.D. za jeho cenné a ochotné konzultace ohledně simulačního modelu šíření tepla a Ing. Michalovi Kepkovi, Ph.D. za pomoc s platformou CesiumJS.

Abstract

Nowadays, we can find increasing overlap between Geographic Information Systems, Building Information Management and Internet of Things. This thesis focuses on one such interdisciplinary scenario, which involves combination of spatial and sensor data. The spatial data describe the rooms of a building. The sensor data characterise the temperature development in these rooms and are obtained by a heat transfer simulation. Subsequently, a 4D cartographic visualisation of these data was created with the use of a CesiumJS platform. Our overall workflow was divided into three standalone components – a spatial data processing, a simulation of heat transfer and a 4D visualisation. These components have clearly defined interfaces with the use of standardised file formats like O&M or glTF. Hence, the solution is modular. Beside that, all source codes created for this thesis are open and publicly available, thus anyone can adapt and enhance any part of our solution for one's purpose.

Keywords

3D, 4D, BIM, cartographic visualization, CesiumJS, CZML, GIS, heat transfer simulation, Internet of Things, Observations&Measurements, sensors, Smart City, spatial data

Abstrakt

V současnosti můžeme sledovat zvyšující se prolnutí mezi Geografickými informačními systémy, oblastí Building Information Management a Internetem věcí. Tato práce se soustředí na jeden konkrétní případ, který zahrnuje kombinaci prostorových a senzorových dat. Prostorová data popisují místnosti v budově. Sensorová data charakterizují vývoj teploty v těchto místnostech a byla získána pomocí simulace šíření tepla. Následně byla vytvořena kartografická 4D vizualizace těchto dat za použití platformy CesiumJS. Celý technologický postup byl rozdělen do tří samostatných komponent – zpracování prostorových dat, simulace šíření tepla a 4D vizualizace. Tyto komponenty mají jasně definovaná rozhraní pomocí standardizovaných formátů jako O&M nebo glTF. Naše řešení je tudíž modulární. Krom toho, všechny zdrojové kódy vytvořené pro tuto práci jsou otevřené a veřejně dostupné, kdokoli je tedy může adaptovat a rozšířit ke svému účelu.

Klíčová slova

3D, 4D, BIM, CesiumJS, CZML, GIS, Internet věcí, kartografická vizualizace, Observations&Measurements, prostorová data, senzory, simulace šíření tepla, Smart City

Contents

Introduction	13
1 Sensor Data in Building Information Management and Smart City Applications	15
1.1 Relation between BIM and GIS	15
1.2 Relation between GIS and IoT	16
1.3 Previous Works on the Topic	18
2 Contemporary 3D and 4D Geographic Information Systems	20
2.1 3D Spatial data and 3D GIS	20
2.1.1 File Formats Used for 3D Spatial Data	21
2.1.2 3D Virtual Scene in GIS	24
2.1.3 Web 3D GIS	25
2.1.4 Concise Description of CesiumJS	26
2.2 Temporal data in GIS	27
3 Processing of Example Spatial and Sensor Data	29
3.1 Chosen Example	29
3.2 Transformation of Geodata into a Simulation Model	31
3.2.1 Outline of the Geodata Transformation	31
3.2.2 Determining Adjacency of Rooms in the Same Floor	33
3.2.3 Determining Adjacency of Rooms through the Ceilings	41
3.3 Building Simulation Model	42
3.4 Transformation of Geodata into a Visualisation format	46
4 Data Visualisation in Four Dimensions	50
4.1 Schema of the Web 4D GIS Application	50
4.2 Blending 3D Geodata and Sensor Data in a 4D GIS Application	52
4.2.1 Loading 3D Models in the CesiumJS Application	52
4.2.2 Connecting Simulation Outputs into the CesiumJS Application	55
4.3 Possible Use Cases of the Solution	58
5 Discussion	61
Conclusion	63
Bibliography	65
List of Appendices	69

A Source codes	70
B Content of the included DVD	84

List of Figures

1.1	Technologies and challenges in the IoT and SmartCities	18
3.1	A virtual 3D model of the Faculty of Applied Sciences	30
3.2	UML component diagram	31
3.3	Overview of the 3D GIS component in the proposed solution	31
3.4	Three simple rooms in two floors	32
3.5	Floor plan before and after the rasterisation-dilation-vectorisation process	33
3.6	Activities necessary to discover the adjacency of rooms	34
3.7	Comparison of string IDs and their numerical equivalents	37
3.8	Comparison of well-rasterised floor plan and ill-rasterised one	38
3.9	Comparison of well-dilated floor plan and ill-dilated one	38
3.10	Model in QGIS' Graphical Modeler	39
3.11	Intersect algorithm used to find overlapping parts of rooms.	41
3.12	Attribute table of the result of the intersection	42
3.13	The simulation component	43
3.14	Overview of the 3D GIS component in the proposed solution	46
4.1	Designed schema of the 4D GIS web application.	51
4.2	Data visualised in the CesiumJS environment	57
4.3	Time series of temperatures in the building	59
4.4	Detail of a part of the building	60
5.1	Alternative UML component diagram	62

Listings

3.1	JSON representation of adjacency relationship among three rooms	32
3.2	Transformation of string IDs into numerical ones	36
3.3	Example YAML file with heat transfer simulation parameters.	44
3.4	Example of an OM-JSON file for one of the rooms	45
3.5	One of the rooms represented in a glTF format	47
4.1	Enabling the Cesium virtual globe in the HTML document	51
4.2	Function which requests the list of available models	52
4.3	Loading of glTF models and their transformation into CZML	53
4.4	One of the rooms represented in a CZML format	54
4.5	Loading of OM-JSON observations and their transformation into CZML	55
4.6	Temperature observation for one of the rooms	56
4.7	Bounding of the time-dependent function callback to the rooms	57
A.1	horizontalAdjacencyTranslator.py	70
A.2	verticalAdjacencyTranslator.py	74
A.3	room.glTF	76

List of Abbreviations

- 2D – two-dimensional
- 2.5D – two-and-half-dimensional
- 3D – three-dimensional
- 4D – four-dimensional (i. e. spatiotemporal)
- AJAX – Asynchronous JavaScript and XML
- API – Application Programming Interface
- ASCII – American Standard Code for Information Interchange
- BIM – Building Information Management
- CAD – Computer Aided Design
- CAM – Computer Aided Manufacturing
- CityGML – City Geography Markup Language
- COLLADA – Collaborative Design Activity
- CZML – Cesium Language
- DAE – Digital Asset Exchange
- DBMS – Database Management System
- FAV – Faculty of Applied Sciences
- FID – feature identifier
- FOSS – Free and Open-Source Software
- GDAL – Geospatial Data Abstraction Library
- GIS – Geographic Information System
- glTF – GL Transmission Format
- GML – Geography Markup Language
- GPU – Graphics Processing Unit
- GUI – Graphical User Interface
- HTML – HyperText Markup Language
- ID – identifier
- IFC – Industrial Foundation Class

- IoT – Internet of Things
- ISO – International Organization for Standardization
- JSON – JavaScript Object Notation
- KML – Keyhole Markup Language
- MPLv2 – Mozilla Public License Version 2.0
- NASA – National Aeronautics and Space Administration
- O&M – Observations and Measurements
- OM-JSON – Observations and Measurements JSON Implementation
- OGC – Open Geospatial Consortium
- UML – Unified Modeling Language
- URL – Uniform Resource Locator
- WebGL – Web Graphics Library
- WFS – Web Feature Service
- WGS84 – World Geodetic System 1984
- WMS – Web Map Service
- WMTS – Web Map Tile Service
- XML – Extensible Markup Language
- YAML – YAML Ain't Markup Language

Introduction

This thesis addresses the growing connection between Geographic Information Systems (GIS), Smart Cities, Building Information Management (BIM) and Internet of Things (IoT). In nowadays applications, these fields of study complement and influence each other. One such area where the aforementioned disciplines meet is processing of data from sensors, which are located inside a building. It is a trend of current days to make buildings, as well as other constructions, fully controlled and monitored by sensors that are connected to a network. Sensors can collect continual measurements about various types of properties such as temperature, humidity, power consumption, occupancy, or even radiation, oxygen level etc. The measurements can then be send in real time via the network to some datacentre for either storing or further analysis.

The observation data from the sensors are usually displayed in their raw form, but they are rarely visualised with regard to their spatial location. The presented thesis focuses on this neglected possibility, which might lead to better understanding of the data. The thesis also covers a scenario, when the sensor measurements of some property of interest are not directly available, so they are substituted by a simulation model.

The primary goal of this thesis is to find a workflow, which would cover the preprocessing of spatial data, their connection with the measurements and their cartographic visualisation. This visualisation needs to display the data in three dimensions, as the sensors are usually spread out across the building, and it must be able to portray the variability of the observed property through the time. For the cartographic visualisation, the *CesiumJS* platform was selected as a cutting edge technology, hence the discovery of its possibilities for a 4D visualisation is the secondary goal of this thesis. Last but not least, another goal of this thesis is to describe the workflow in a comprehensive methodology.

This thesis is structured as follows: The first chapter gives a brief overview of the disciplines influencing the thesis and describes their overlaps and differences. The second chapter summarises the current state of the art in a domain of spatiotemporal 3D GIS software. The third chapter presents the methodology of preprocessing the

spatial data for their visualisation alongside with their processing for a simulation. The fourth chapter then describes how to use *CesiumJS* platform for the final visualisation. Other possible scenarios, limits of the presented workflow and outlook for potential extensions are discussed in the final, the fifth chapter. Source codes of the shorter programs produced as a part of this thesis are included in the appendices.

Chapter 1

Sensor Data in Building Information Management and Smart City Applications

Number of devices connected to the so called Internet of Things grows rapidly every year.² Such a devices, replete with various sensors, increasingly find their use in Building Information Management (e.g. Chen et al. 2014; X. Liu et al. 2017), as well as in Geographic Information Systems (e.g. Papadopoulos et al. 2018; Kepka et al. 2017), and are essential to many Smart City applications. (e.g. T.-h. Kim, Ramos, and Mohammed 2017; Trilles et al. 2017)

1.1 Relation between BIM and GIS

Each building has a certain lifecycle. This usually involves a stage of projection, stage of construction, stage of operation, possibly some stages of reconstruction and in the end a stage of demolition or disassembly. Each of the stages requires specific data about the building and an interchange of these data between the stages is essential. Traditionally, such interchange would incorporate a lot of paper drawings and written documents. Nowadays, it rather includes CAD drawings and electronic documents but the interoperability of data between individual stages has hardly increased. Professionals involved in each of the stages work with different software and tools and often neither the tools nor the professionals communicate between each other well. This data exchange is especially cumbersome in the transition between the two major phases of the building's lifecycle, as the team of engineers working on projection and construc-

²<https://www.newgenapps.com/blog/iot-statistics-internet-of-things-future-research-data>

tion of the building is usually completely different than the team of facility managers who runs the building after its completion. And this is where Building Information Management comes on stage. (Bonandrini, Cruz, and Nicolle 2005; 29481-1:2016 2016)

Building Information Management, or BIM, is a management of building during all its lifecycle, which is all about its information model. As the Building Information Model is the core part of it, the process is commonly also called Building Information Modelling. BIM addresses the above-mentioned exchange issue by using only one complex but generic model of the building during its whole lifecycle. (Steel and Drogemuller 2009)

Integration of Building Information Management and Geographic Information Systems arose from the natural needs of both industry areas. Although BIM contains detailed information about the building itself, it does not include information about the surroundings. The limitation of this can be illustrated on the spatial planning of construction: optimisation of tower crane's location on a construction site is a classic task that has to be often managed but requires spatial information. Vice versa, GIS is suitable to store, analyse and display the spatial data in a broader context, but it is not designed to work with detailed project data as it is required e.g. by road or railway management authorities. (X. Liu et al. 2017)

Song et al. (2017) in their meta-study summarise, that when converting data between BIM and GIS, there is always a significant loss of data. Even in the case of popular and standardised formats of IFC and CityGML, the conversion is far from ideal.

1.2 Relation between GIS and IoT

An idea of interconnected devices, which communicates between each other through a single network, could be found in the science-fiction literature long ago. As soon as in 1965, Arthur C. Clarke in his apocalyptic short story *Dial F for Frankenstein* expressed worries about the security of such a network, but similar concepts might be found in works by other authors as well. The first realistic concept of a network that could connect smart devices appeared in the 1980s and it started to be called an Internet of Things at the beginning of the new millennium.¹ In the last few years, Internet of Things, or IoT, has become a frequently used term that describes the interconnection of diverse devices via the Internet network.

¹https://en.wikipedia.org/w/index.php?title=Internet_of_things&oldid=897069596

The "thing" connected to the Internet can be essentially any device which has a proper hardware equipment to do so, but overwhelmingly it is a device containing some kind of sensor or a whole range of sensors. Data from these sensors are then sent via the Internet to some server or another device where they are stored and analysed. Smart City in this context is then a broad term describing a municipality which utilises data from sensors in the Internet of Things to enhance the quality of life in the area. Similarly, terms like Smart Home, Smart Grid or Smart Highway are used when talking about these areas enhanced by IoT technologies.

One of such Smart- experiments is SmartCampus¹ by the University of West Bohemia. It currently incorporates data from temperature and humidity sensors, cloud computing platform, data from sensors in parking lots, experimental laboratory connected to the IoT network and other projects currently in the state of preparation.

Smart City applications arise rapidly all over the world, mainly in the largest cities, as a response to their constant growth, which is not going to end any time soon. United Nations estimates that by 2030 more than 60 % of the global population will live in large cities. Smart City applications might play a significant role in better distribution of the cities' resources, thus leading to longer-term sustainability. In this task, many Smart applications have been developed already but many are still waiting to be created and tested, as displayed in figure 1.1. (T.-h. Kim, Ramos, and Mohammed 2017)

A large topic in Smart City related research lies in the field of energy, environment and sustainability that requires visualisation and exploratory analysis of multidimensional (2D, 2.5D, 3D and 4D) data. Such applications require visualisation in granular resolutions and are usually aggregated and illustrated at the building, building surface or building object levels. (Murshed et al. 2018) It is therefore evident that Smart City and IoT fields can be closely related to geomatics and GIScience as well, because of the need for manipulation of spatial data and the need for cartographic representations of the results of an analysis performed on the sensor data.

Since many fields of study and industries and research areas meet when it comes to Smart Cities and IoT, standardisation of interfaces, exchange formats and services is a necessity. Open Geospatial Consortium (OGC) addresses this need with several standards related to the connection of spatial data and IoT:

- Sensor Web Enablement (SWE), a general set of specifications about sensors, sensor data models and sensor web services, (Robin 2011; Echterhoff 2011)
- Sensor Observation Service (SOS), (Na and Priest 2007)

¹<https://www.smartcampus.cz/>

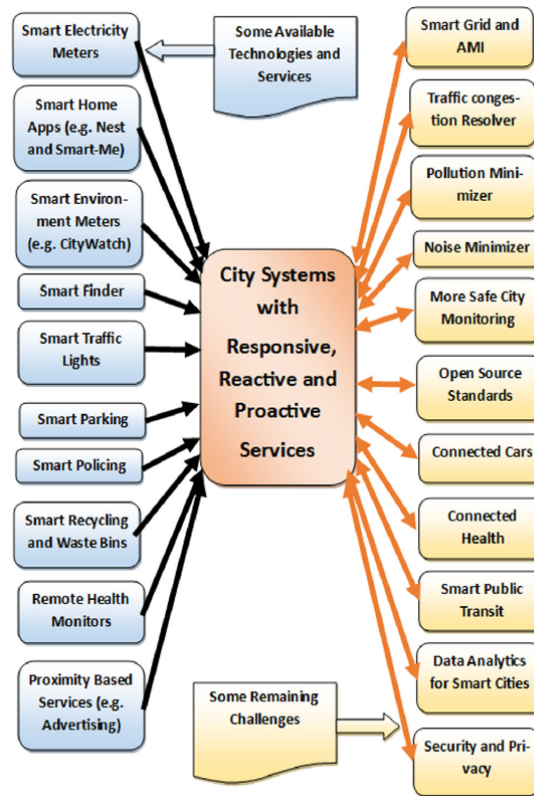


Figure 1.1: Available technologies and services versus remaining challenges in the IoT and SmartCities domain. (source: (T.-h. Kim, Ramos, and Mohammed 2017))

- Sensor Planning Service (SPS), (Simonis and Echterhoff 2011)
- Observations and Measurements (O&M) (Cox 2013) and
- SensorThings API. (Liang, Huang, and Khalafbeigi 2016; Liang and Khalafbeigi 2019)

Most notably of this list, Observations and Measurements standard specifies an exchange format for various types of observation data. This format can be encoded either in XML language or in JSON. (Cox 2013)

1.3 Previous Works on the Topic

Several applications focused to connect the worlds of BIM, IoT, Smart City and GIS has been created already. The most relevant ones to this thesis are described in the following section.

Murshed et al. (2018) created a *4D CANVAS*, a *CesiumJS*-based application for dynamic visualisation of 3D spatial data. Their solution displays time-variable geodata

in CZML format along with large datasets of static spatial data in 3D Tiles and simple dataset of building surfaces in GeoJSON. All the geodata for their application are stored in a relational PostgreSQL database and are converted by a script on the server once they are requested by the application. Based on their experience with various geodata formats in the *CesiumJS* platform, they point out that CZML is the only format able to visualise 4D data, but its file size grows too rapidly if it contains many time samples. The GeoJSON format has similar limits because when served in larger volumes, consumes too much of the client’s memory and causes the application to slow-down rapidly or freeze.

Zhu et al. (2016) focused on air pollution in the city of Karlsruhe. Their application combines two sources of data stored in two databases – a model of the city in 3D CityGML format and data about the air quality from the sensors located on the city tramways. Both data sources are combined in a *CesiumJS*-based visualisation, into which the CityGML data are loaded via Web Feature Service and sensor data via Sensor Observation Service. The sensor data are encoded according to the O&M standard.

De Roo, Bourgeois, and De Maeyer (2017) describes the creation of a prototype of a 4D archaeological GIS, which is based on *CesiumJS* as well. They display one excavated archaeological site in the virtual globes, while their work mainly focuses on the analytical tools available for the data and on the usability testing of the prototype. From the testing on a sample of intended users, they conclude that the time slider present in the application’s GUI was among the most interesting for the users. Its division is however too fine-grained and exact for most archaeological use-cases.

Chapter 2

Contemporary 3D and 4D Geographic Information Systems

First desktop Geographic Information Systems were invented as a simple viewers of 2D data. Later, they incorporated manipulation of layers and analysis tools.² Since then they have evolved significantly, while including more and more tools for various use-cases. This chapter provides an overview of how the current Geographic Information Systems deal with 3D spatial data and how they support the time-variability of the data.

2.1 3D Spatial data and 3D GIS

The need of three-dimensional Geographic Information System, or 3D GIS, arises from many application areas, for example:

- 3D urban mapping,
- archaeology,
- architecture,
- automatic vehicle navigation,
- civil engineering,
- command and control,
- defence and intelligence,

²https://en.wikipedia.org/w/index.php?title=Geographic_information_system&oldid=897367977

- ecological studies,
- environmental monitoring,
- geological analysis,
- landscape planning,
- mining exploration,
- oceanography. (Abdul-Rahman and Pilouk 2008)

To address this need, the support for 3D data was implemented in most of the widely-used GIS software. Among the first notable GIS with at least partial ability to handle a 3D data were *ArcView 3D Analyst* by Esri, *Imagine VirtualGIS* by ER-DAS, *GeoMedia Terrain* by Intergraph and *PAMAP GIS Topographer* by PCI Geomatics. (Abdul-Rahman and Pilouk 2008) Although these software products dates back to the late 1990s or the beginning of the 2000s, data interoperability between different 3D geodata formats is not fully reliable yet. For example as described in Jedlička (2018): "When there is stated that a data format A can be converted to a format B , then such a conversion can mean that only X, Y coordinates are converted, but Z coordinate is partially or completely lost", which is caused by an insufficient support of vertical reference systems and the transformations between them in the GIS software.

To further clarify the terminology, as a three-dimensional, shortly 3D, is marked such an object which can have arbitrary coordinates in three-dimensional space. (Raper 1992) As a two-and-half-dimensional, shortly 2.5D, is marked such an object, whose Z coordinate can be described as a function $Z = f(X, Y)$ of its X and Y coordinates. This limits the 2.5D objects to extruded surfaces and objects alike. (Abdul-Rahman and Pilouk 2008) As a four-dimensional is then marked such an object, whose some property or properties varies over time.(K. Kim, Carlis, and Keefe 2017)

2.1.1 File Formats Used for 3D Spatial Data

There are numerous file formats for storing 2D geodata, which differ in their inner representation of the geographic features and/or in their field of application. The situation with 3D geodata is not different – various file formats exist with various feature representations for various application areas. Storing 3D features is naturally not only in the focus of GIS but also an essential part of other industries like CAD, CAM, BIM, computer graphics, virtual reality or 3D printing as well. The interoperability of 3D file formats between these industries can still be imperfect and challenging.

Two major categories of 3D data representation are surface-based representation and volume-based representation. Surface-based representation describes 3D features as a composition of surfaces, i. e. as a set of two-dimensional manifolds, while volume-based representation describes 3D features as a compound of volumes, i. e. as a set of three-dimensional manifolds. More detailed overview of 3D data representations used in GIS can be found in Abdul-Rahman and Pilouk (2008). Description of the 3D file formats, which are relevant for this thesis follows.

Probably the most popular file format for storing vector spatial data in general is Shapefile. Shapefile was developed by Esri in the beginning of the 1990s and become widespread since then. Although it is usually employed to store regular 2D geodata as a set of either point, line or polygon geometries, Shapefile is also capable of representing a truly 3D features in a form of *Multipatch*. Each Multipatch can consist of triangles, triangle fans, triangle strips and rings and it is therefore considered a surface-based representation. Further details can be found in the ArcGIS 3D Analyst documentation.¹

Among the most common 3D file formats nowadays² is COLLADA, which is also known as DAE format because of its `.dae` extension. COLLADA was published by a non-profit consortium Khronos Group in 2004 as an open standard based on XML language, and it was adopted as an ISO standard in 2013. In contrast to Multipatch, COLLADA is not designed to store just the sole geometry of an object, but can also hold information about its appearance like colour, material, textures and even animations.³ On the other hand, it does not support spatial reference systems and the position of objects is described in Cartesian coordinates. (Barnes and Levy Finch 2008) From this reason, COLLADA files are often accompanied by a simple KML file in the GIS industry. The KML file then stores information about the COLLADA object's location, known as the anchor point, and possibly information about its orientation and scale. KML itself is an OGC standard based also on XML language.

Khronos Group consortium also authored glTF, which is an abbreviation from GL Transmission Format⁴, whose full name refers to the WebGL API, for which it was designed. glTF specification was initially released in 2015 and thus reflects the needs of more modern graphic software when compared to COLLADA. Instead of XML, it

¹<http://desktop.arcgis.com/en/arcmap/latest/extensions/3d-analyst/multipatches.htm>

²<https://all3dp.com/3d-file-format-3d-files-3d-printer-3d-cad-vrml-stl-obj>

³To be fair, Multipatch, just like any other Shapefile, can hold information about colour, textures, etc. as well – in an attribute field. But this information can have arbitrary structure in the attribute table, so the information generally will not be understood by a rendering software. It is the difference, that in COLLADA and glTF, these information has additional semantics.

⁴<https://www.khronos.org/glTF/>

is based on JSON language, which is more suited for the transmission among web applications. Similarly to COLLADA, it was designed to store much more information than just a geometry – glTF can include information about virtual scene, virtual view or camera, textures, materials, animations etc. Geometry of the object is represented in glTF as a set of meshes. Each mesh is then defined by a set of primitives. A primitive can be one of point, line or triangle. Just like COLLADA, glTF supports only one coordinate system and that being the Cartesian. Thus, for usage in GIS, information about the object’s position, orientation and/or scale must be provided separately. Even though some GIS use a local 3D scene with Cartesian coordinates to visualise 3D data, the model coordinates still has to be transformed into the local coordinates, as they likely have different origin, and each model normally has its own coordinate system defined. (Bhatia et al. 2017)

glTF format became a cornerstone for a Cesium Language or CZML. CZML is also a file format, which is based on a JSON encoding. One of the biggest advantages of CZML is that it is completely streamable, i. e. one object does not have to be saved in one file, but it can be partitioned into smaller portions and then send across network in a stream of separate files.¹ An elementary object in each CZML file is a Packet². Each Packet shall contain an ID, which identifies the object it describes and a set of various properties. The geometry of an object can be then described by a simple shape like point, polygon, ellipsoid, etc. or a link to a glTF model can be provided for more complex geometries. Second big difference of CZML, compared to other spatial data formats, is that each its property can be simply made time-dependent. This can be achieved either by providing an `availability` property to the whole Packet, which will restrict its time relevance, or by specifying a time restriction to individual values of some property.³

Analytical Graphics, Inc., which has authored the CZML file format, has invented yet another 3D file format – 3D Tiles. 3D Tiles has been also released as an OGC standard. (Cozzi, Lilley, and Getz 2018) Similarly to CZML, 3D Tiles incorporates the glTF file format to represent geometry of models. But unlike CZML, it is designed to split the whole file into tiles, which can be then rendered separately. This has a great advantage for large 3D datasets – the tiles that are not currently in the visible view of the client does not have to be rendered, which significantly saves computation power. Trade-off for this saving during the rendering is quite complicated and time-consuming creation of the tiles. Another current shortage of 3D Tiles is that their support for time-

¹<https://github.com/AnalyticalGraphicsInc/czml-writer/wiki/CZML-Guide>

²<https://github.com/AnalyticalGraphicsInc/czml-writer/wiki/Packet>

³<https://github.com/AnalyticalGraphicsInc/czml-writer/wiki/InterpolatableProperty>

dependent data is not yet standardised and representation of time in data is considered experimental.¹

2.1.2 3D Virtual Scene in GIS

Same as 2D geodata file is normally not just presented in GIS as it is, but is often combined with other data sources and supplemented with additional information before it is presented to the public, 3D geodata are not just loaded into 3D GIS and displayed as they are, but rather combined with other data to create a cartographic representation of some area of interest.

The model of a particular area of interest in 3D GIS typically consists of:

1. at least one digital terrain model in some of the common representations;
2. 2D geodata which are then draped on such a terrain, just taking over the third coordinate from the terrain model (with all the consequences it has, like necessary triangulation of polygons, adding new vertices to line segments etc.);
3. points, represented by a full 3D symbol from a predefined symbol library (e.g. trees, city furniture etc.) and/or lines and polygons extruded to a specified height to be visualised as planes and volumes respectively;
4. real 3D data structures, usually for models of man-made objects, such as buildings, bridges etc., which are often created in CAD software in a local coordinate system, then converted to a data format suitable for 3D GIS and referenced to other geographic data by an anchor point, direction of rotation and a scale;
5. attribute data, as 3D GIS should not resign on spatial data relationship to the detriment of attribute data typical for GIS in 2D. (Jedlička 2018)

Many contemporary 3D GIS scenes have been created primarily for visualisation purposes and there is a lack of relationship between 3D data and their attributes. Typically a building or even a block of buildings is created as one object, without any further segmentation, so attributes can be then related just to the object as a whole, but not to its particular parts. It results in more generic descriptions of the objects, which also makes any further analysis more generic and thus less detailed. (Jedlička 2018)

¹<https://cesium.com/blog/2015/08/10/introducing-3d-tiles/>

2.1.3 Web 3D GIS

During the last years, the development of GIS in a web environment has grown largely as the web has grown in many other industries as well. Open Geospatial Consortium (OGC) has released many standards concerning geodata on the web, namely the OGC Web Services, from which Web Map Service (WMS), Web Feature Service (WFS) or Web Map Tile Service (WMTS) has become widespread in the GIS industry. The visualisation of spatial data with a third dimension in a web browser was firstly realised by the means of plug-ins and extensions. In 2011, WebGL, a JavaScript API for GPU-driven rendering, was introduced, which opened the doors for native 3D applications running in the web browser. (Wendel et al. 2016)

OGC has later released several documents which focus on a 3D geospatial data as well:

- CityGML, which is an extension to a Geography Markup Language (GML, which itself is based on XML) with an objective to standardise the representation of virtual 3D city models, (Gröger et al. 2012)
- 3D Portrayal Service (3dP), (Hagedorn et al. 2015)
- Indexed 3D Scene Layers (i3s) (Reed and Belayneh 2017) and
- 3D Tiles. (Cozzi, Lilley, and Getz 2018)

In Farkas (2017), the author has examined five open-source web mapping libraries and their applicability in web GIS clients. The reason of focusing on the free and open-source software (FOSS) only, despite there are great proprietary web mapping libraries available, is to gain maximum control over the final application as FOSS grants the freedom of running, studying and adapting, redistributing and releasing improvements to the public. The examined libraries are *Leaflet*, *OpenLayers 2*, *OpenLayers 3*, *CesiumJS* and *NASA Web World Wind*. The first three mentioned ones are common JavaScript mapping libraries only capable of manipulating and visualising 2D data, while the other two libraries are so called virtual globes, which support both 2.5D and real 3D data handling. From these 3D enabled libraries, *CesiumJS* received a better overall score, while being as good as, or better than, *NASA Web World Wind* in every criterion, except of support of known projections. (Farkas 2017)

Among the applications for 3D spatial data visualisations, virtual globes have become very popular. The first widely used virtual globes were *NASA World Wind* and

Google Earth.¹ Current notable virtual globe applications, which are extended by some GIS tools, are *TerraExplorer*² by Skyline Software Systems or *iTowns*³ by IGN and MATIS. Virtual globes usually has only limited capability of user data input, minimal data management capabilities and no or very simple analysis tools, like distance or area measurements. Hence, they are normally not considered a GIS software, but rather a data visualisation software. Consequently, if one wants to use a virtual globe application as a GIS, it has to be extended or connected with other software, e.g. a DBMS for better data management or some analytic geospatial program library which will provide the necessary GIS tools. Naturally, the requirements differ from user to user. A regular end user will probably not require as many features as a member of a scientific community. (Pupo 2012)

2.1.4 Concise Description of CesiumJS

CesiumJS is an open-source JavaScript library for creating a highly customisable virtual 3D globe in a web browser. More precisely, *CesiumJS* is a part of larger ecosystem called simply "Cesium", which is maintained by Analytical Graphics, Inc. Apart from the JavaScript library, it offers a *Cesium ion*, a cloud-based platform for storing, transforming and streaming of 3D spatial data with a "freemium" pricing strategy. The *CesiumJS* library is built on top of WebGL, which transfers the rendering load on the GPU, thus reducing the computing power on CPU. It also allows the application to run in any modern WebGL-enabled browser, without the need to install additional plug-ins. (Tsai, Lai, and Y.-C. Liu 2015; Murshed et al. 2018)

The *CesiumJS* library is distributed with many ready-to-run examples and a server application for Node.js, which behaves as a proxy server between client application and a geodata provider in the web and which is generally used to serve tiles of the terrain in the virtual globe. The examples shared with the library covers most of the CesiumJS capabilities and can be conveniently utilised in an own *CesiumJS*-based application. The most simple *CesiumJS*-based "HelloWorld" application creates the virtual globe in the browser window, which can be freely rotated and zoomed. The virtual globe is by default enhanced by a terrain dataset from *Cesium ion* and the terrain is covered with aerial imagery tiles. These properties of the globe can be changed and customised though. Many spatial data sources can then be added to the virtual scene like custom terrain dataset, image tiles via WMTS, from ArcGIS Server or by other means, 3D

¹https://en.wikipedia.org/w/index.php?title=Virtual_globe&oldid=881531310

²http://skylinesoft.com/SkylineGlobe/corporate/Products/te_web.aspx

³<http://www.itowns-project.org/>

vector geometries from KML, GeoJSON, TopoJSON or CZML, complete 3D models in glTF and large 3D datasets in 3D Tiles. (Tsai, Lai, and Y.-C. Liu 2015)

Beside displaying spatial data, *CesiumJS* is also designed for a management of temporal aspect of the data, for which it provides support for both programmatic management and user interaction. Murshed et al. (2018) explicitly states that "Among the available WebGL-based visualisation frameworks and libraries, Cesium is most suitable to create virtual globes with time dynamic 3D visualisation of geospatial data as it allows native discrete temporal data support."

2.2 Temporal data in GIS

The idea of spatio-temporal GIS dates back at least to the late 1980s when Langran published his dissertation *Time in geographic information systems* (Langran 1989). Since then, many papers has been published on this topic, as evidenced in a *TimeBibliography*¹ project, an on-line bibliography which presents a detailed timeline of texts published with the focus on temporal GIS, spatio-temporal modelling and related topics. (Siabato et al. 2014)

Peuquet (2005) describes three main approaches to represent temporal data in GIS: *a)* location-based representation, *b)* entity-based representation and *c)* time-based representation. Location-based representation is based on a series of time snapshots of a certain location. This is the most simple time representation suitable for any GIS software – each layer of data is related to a certain moment in time (a snapshot) and each layer contains the data about features relevant to that moment. Location-based representation can be used with either vector or raster data and works principally the same for 3D data as well. Although it is simple to maintain and easy to incorporate in "regular" GIS, this representation has its limits, e. g. rapid increase of data volume throughout the time or inevitable redundancy.

Entity-based representation provides a bit more fine-grained level of the changes in an area of interest. In this representation, a time identifier is bound to each feature of a layer, rather than a layer as a whole. The data in this case reflects only the changes in the observed phenomenon, either spatial or non-spatial. This approach has the advantage of no data redundancy but has its drawbacks as well, e. g. it is not trivial to decide how the change of geometry (split or merge of features) affect the non-spatial attributes. The time-based representation is then based on a timeline of events, where

¹<http://spaceandtime.wsiabato.info/>

each event mean some change in the data, no matter if spatial or non-spatial. (Peuquet 2005)

Although the idea of spatio-temporal GIS exists for quite some time, yet in 2013 Goodchild considers possible forms of integration of time in GIS and concludes that "Many difficult and challenging issues will have to be explored" before the integration will become reality. (Goodchild 2013)

Since then, while a uniform way to manage time in GIS is still a "challenging issue", spatio-temporal analysis for Smart City applications attracted attention of both GIS experts and the public. Such a spatio-temporal analysis, like air pollution modelling or crime and epidemic spread analysis, essentially needs the temporal aspects of the data, which is pushing the development forward. (Murshed et al. 2018)

Chapter 3

Processing of Example Spatial and Sensor Data

How previous chapters implies, there are many ways how to represent spatial data and there are many ways, how the spatial data can be used in different industry areas. Here, we focus on the process, which uses spatial data of a building floor plans to retrieve a topological model of the building, which is required to simulate the heat transfer among the rooms. This chapter describes how to obtain the topological model, how to convey it to the simulation model and how to transform the rooms so they can be visualised as time-dependent.

3.1 Chosen Example

Considering available data sources, we have chosen a building of Faculty of Applied Sciences, University of West Bohemia, located in Pilsen, Czech Republic, as our exemplary building. This building, hereinafter referred to as *FAV* (abbreviation from Czech "Fakulta aplikovaných věd"), compounds of two major parts, as displayed in figure 3.1: the eastern wing with four overground floors, in which resides the education part of the faculty, and the western wing with six overground floors, in which the offices of the faculty's research institute NTIS² takes place. Internal disposition of this building corresponds to the dispositions of common office buildings, as it has several floors with various ceiling heights, connected via stairways and elevator shafts.

A floor plan of each level is a bit different, meaning that rooms in individual floors does not overlap perfectly, which adds another complexity to the problem.

The workflow from the input data through a simulation process to a visual representa-

²New Technologies for the Information Society

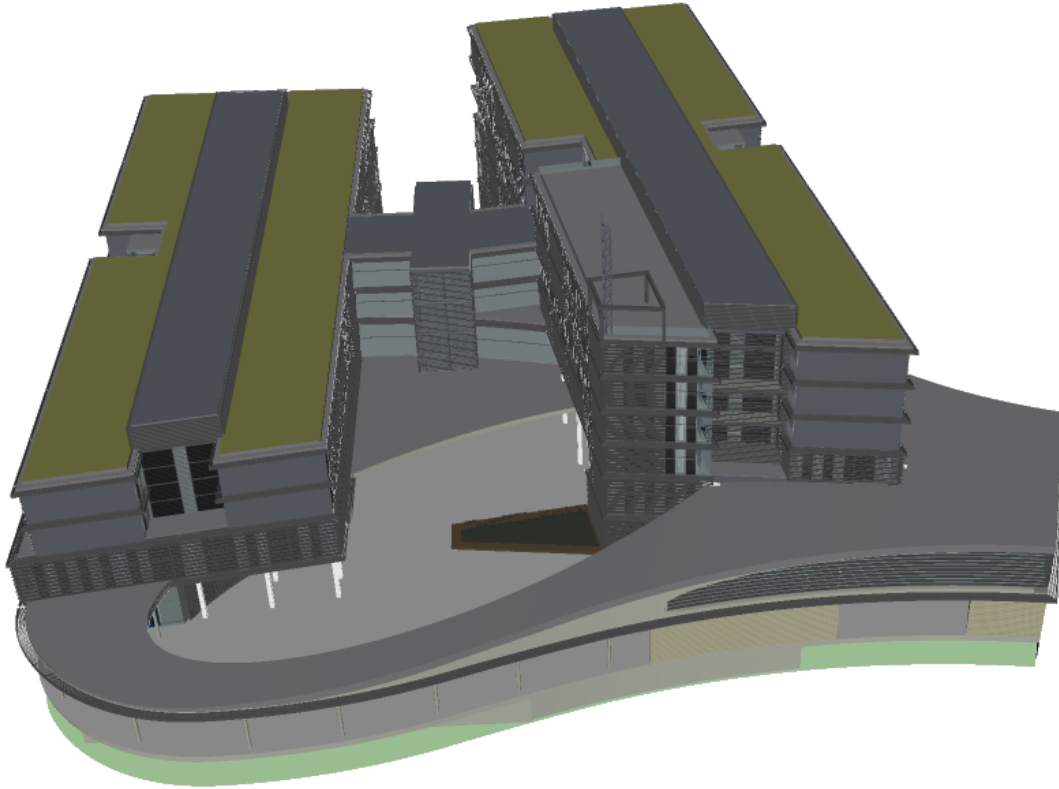


Figure 3.1: A virtual 3D model of the Faculty of Applied Sciences, the chosen exemplar building. (Source: ATELIER SOUKUP OPL ŠVEHLA s. r. o.)

tion of the results was designed as a modular solution with three major components, which can be seen in figure 3.2. The first component is a desktop 3D GIS software, which is used to read the input geodata and process them to retrieve desired information – in our case, data about adjacency of rooms in the example building of FAV. The second component is a heat transfer simulation program written in Java. It utilises the information retrieved from geodata and by means of simulation imitates an output of real-world temperature sensors. The third and last component is a web-based 4D GIS software, which serves to process the sensor data and visualise them. These three components communicate between each other via interchange formats – an application specific JSON file is used to forward data from 3D GIS to Heat Transfer Simulator and a JSON file compliant with the O&M standard is used for conveying data from Heat Transfer Simulator to 4D GIS. Beside the "main path", there is also a direct exchange of geodata between 3D GIS and 4D GIS via the glTF file format. So the data about rooms can be split and merged later as it is proposed, each room must be uniquely identified by an ID, which must be preserved through the whole process.

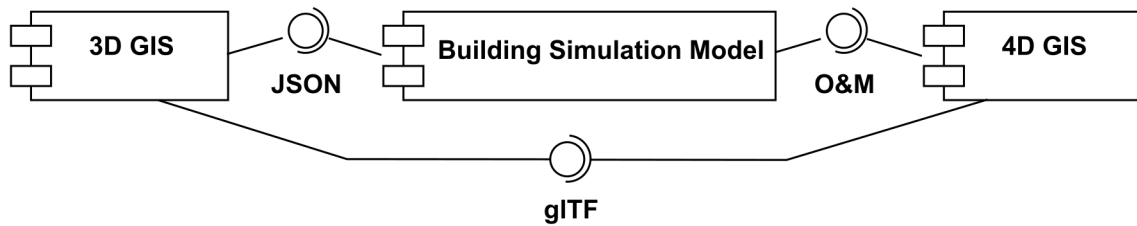


Figure 3.2: UML component diagram showing three main components of the exemplar solution: a 3D GIS, a Building Simulation Model and a 4D GIS.

3.2 Transformation of Geodata into a Simulation Model

3.2.1 Outline of the Geodata Transformation

The first component in our solution is a 3D GIS. Here, spatial analysis and data transformations necessary for the subsequent components takes place. In the figure 3.3, the tasks performed in 3D GIS are displayed as subcomponents. For a heat transfer simulation it is necessary to determine the adjacency of individual zones, i.e. find the shared walls or floors/ceilings between each pair of rooms. This task was divided into two complex steps – *a*) finding an adjacency among the rooms in the same floor (horizontally) and *b*) finding an adjacency among the rooms through the floors and ceilings (vertically) – which are described in the following sections and highlighted in the figure 3.3.

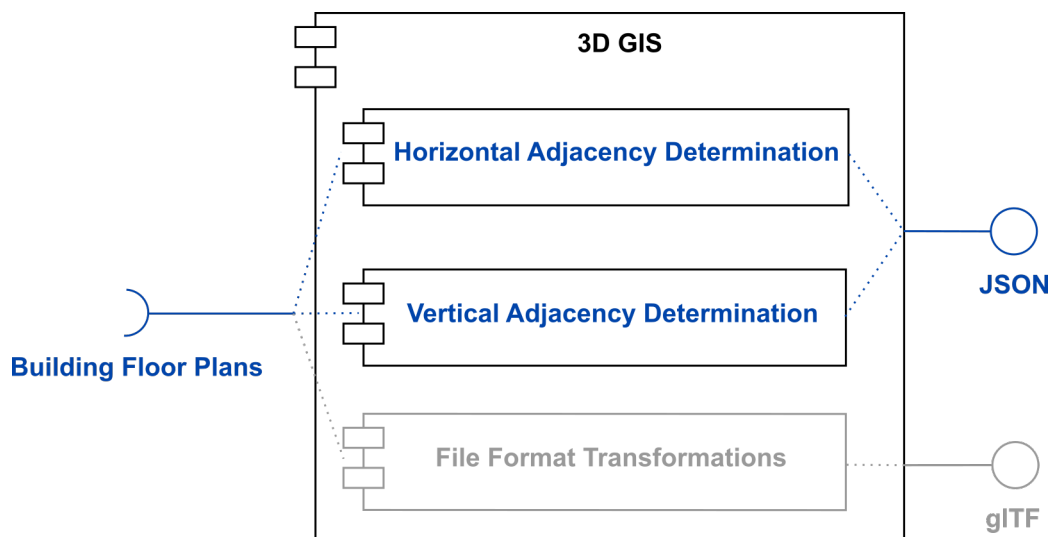
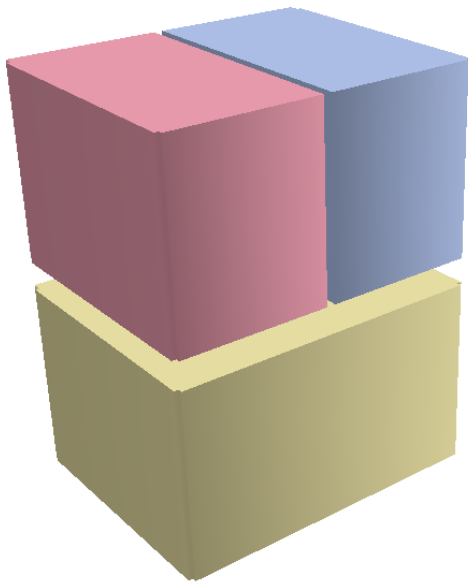


Figure 3.3: Overview of the 3D GIS component in the proposed solution. Subcomponents responsible for the production of a JSON file with adjacency information are highlighted.

During the research, no suitable file format was found which would be designed

to store the topological relations of an object, but not its actual geometry. Hence, a JSON-based file format was designed for this thesis, which omits the geometry, which is redundant in this case, but stores the necessary relations among the rooms. Its structure is explained on a simple example of three rooms, which adjoin each other, as in figure 3.4a. The topological relations among these three rooms are represented in the JSON format as displayed in the figure 3.4b. For clarity, the rooms are identified just by their colours in the figure 3.4a.



(a) The three rooms rendered in 3D view.

```

{"rooms": [
  {"id": "red",
   "volume": 39.41,
   "height": 2.23,
   "walls": [1, 2, 3, 4, 5, 6]},
  {"id": "blue",
   "volume": 39.86,
   "height": 2.23,
   "walls": [4, 7, 8, 9, 10, 11]},
  {"id": "yellow",
   "volume": 68.96,
   "height": 2.23,
   "walls": [5, 10, 12, 13, 14, 15,
            16]}
],
"ambient": {
  "constant": true,
  "walls": [1, 2, 3, 6, 7, 8, 9, 1
            1, 12, 13, 14, 15, 16]
},
"walls": [
  {"id": 1,
   "area": 8.218,
   "leftID": "red",
   "rightID": "-1"},
  ...
  {"id": 4,
   "area": 10.102,
   "leftID": "red",
   "rightID": "blue"},
  {"id": 5,
   "area": 9.135,
   "leftID": "red",
   "rightID": "yellow"},
  ...
]]

```

(b) JSON representation of adjacency relationship among the three rooms. The actual JSON file is shortened for brevity.

Figure 3.4: Three simple rooms in two floors, which share some of their walls.

It is explicitly stated in the JSON file that the "red" room and the "blue" room

share a wall with ID "4", while the "red" room and "yellow" room share a wall with ID "5". Actually, the information is saved twice in the file – firstly, it is kept in the list of walls for each rooms, from which the adjacent pairs of rooms can be inferred. Secondly, it is explicitly stated in the list of walls, that a particular wall connects two particular rooms. This duplicity is designed in the file for several reasons: *a)* for the integrity check and *b)* to simply append additional attributes to both rooms and walls. Although the properties of a wall instance are named "leftID" and "rightID", their assignment is arbitrary for our use case and can be swapped without any effect on the result. These keywords were chosen simply for better readability of the file, then e.g. "ID_1" and "ID_2" or similar naming. Also note that the format makes no difference between adjacency through a shared ceiling/floor and adjacency through a shared wall. That is because the rotation of the building makes no difference in its inner disposition. Actually, the statement from previous sentence is valid for all isometric transformations, as they does not affect the shape of an object. Hence, all adjoining surfaces are simply called *walls*, although they might be ceilings (or floors, depending on the perspective).

As we aim to create a visualisation using free and open-source software, it seems logical to follow this approach during the processing stage as well. From this reason, QGIS software was primarily used during the following process. However, not every step was possible in FOSS, hence a proprietary GIS software was rarely used.

3.2.2 Determining Adjacency of Rooms in the Same Floor

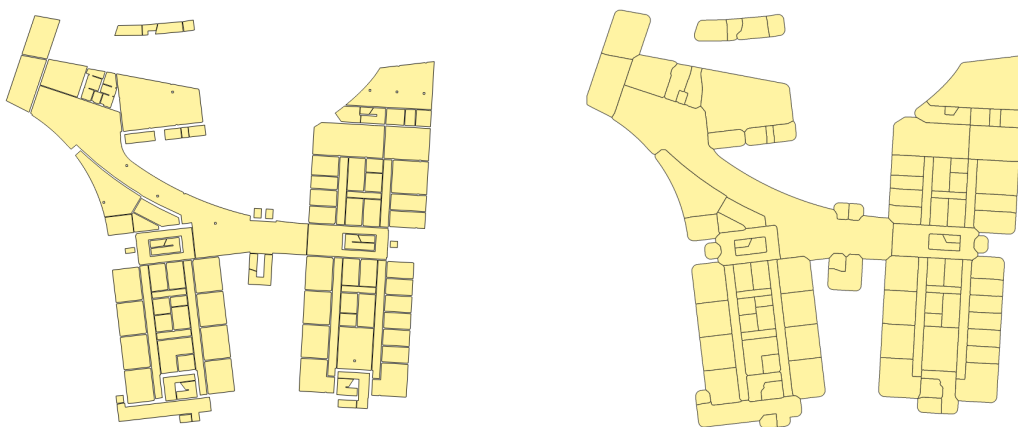


Figure 3.5: Comparison of input floor plan with separated rooms (on the left) and the result of rasterisation-dilation-vectorisation process.

Provided that a building has a digital floor plan available in some CAD or GIS format, it will very likely be a very detailed drawing. In such a drawing, rooms of the building are typically separated by space which corresponds to the thickness of walls between them. As this describes the reality well, it is not well suited for determining the mutual adjacency of individual rooms. To find this adjacency, the space between the rooms must be fully filled and the adjacent rooms must share only an edge, as it is displayed in figure 3.5. Flow of activities which has been done to accomplish this task is displayed at the UML activity diagram in figure 3.6.

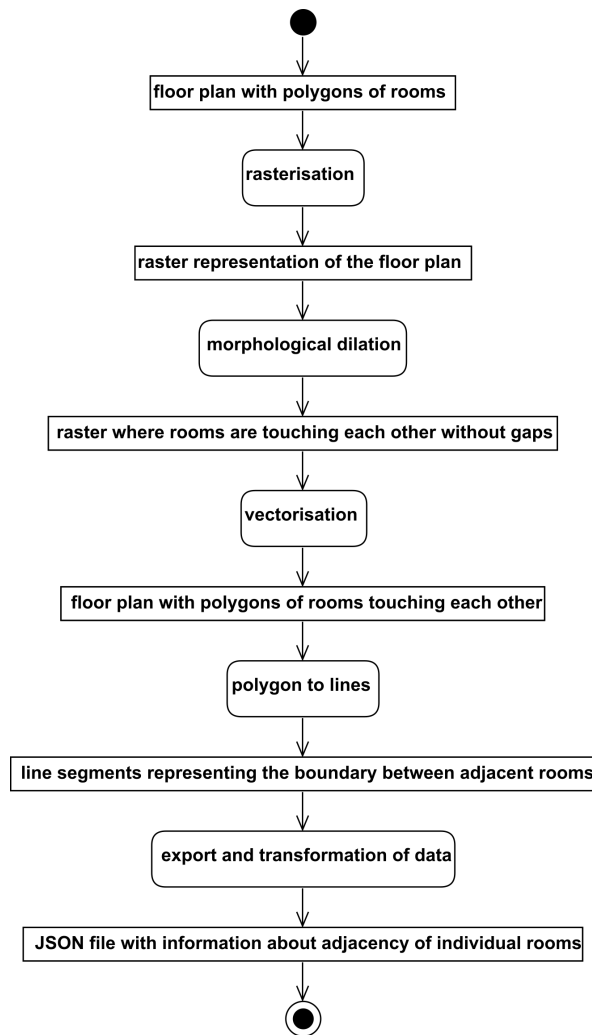


Figure 3.6: UML activity diagram of activities necessary to discover the adjacency of rooms in one floor.

First of all, there must be a unique ID for each room in the floor plan, which must be preserved throughout the processing to allow simple join with the sensor data later. As the first geoprocessing step is a transformation of vector (polygon) layer into a raster, the IDs must be numerical values. There are multiple ways how to transform string-based IDs into numerical ones:

- Manually curated table where for each string ID a numerical one is assigned,
- automated renumbering of all rooms by a range of integers (and storing the mapping between them),
- transformation of strings into integers by a set of reversible rules, e. g. by taking advance of some encoding and decoding or by using some inner logic of naming the rooms.

Each of these options has its pitfalls. We have chosen the last mentioned approach, which disadvantage is, that it can generate unwieldy long integers – if we chose to encode each character into its Unicode decimal code, then for a string of length N it creates an integer of length $3N$. Storing very long values in raster cells is then cumbersome or even impossible. Hence, this method is only suitable for relatively short IDs with a maximum of three or four characters. To minimize this issue in our case, we have combined an inner system of numbering the rooms and converting remaining letters of the original string into their ASCII decimal code. In the FAV building, each room has an ID in a fixed form $UXNNNa$, where U is static (every room ID begins with "U"), X can be one of (C, N, S) , depending on its position inside a building, NNN is a three digits long number (where the first digit corresponds to the number of floor), and a is an optional letter in the end. The IDs in FAV building were transformed by following rules:

- $UC \rightarrow 1$,
- $UN \rightarrow 2$,
- $US \rightarrow 3$,

then three digits long number remains unchanged, and if there are some letters left after it, only they are encoded using an ASCII decimal code. In fact, few outliers were found in the fifth floor, which begins with UNW string, so these three letters were replaced by number 4. At a first glance, such an operation might look too complicated, but it has the advantage of absolute reversibility, so no additional table with ID mappings needs to be stored in this step.

In QGIS, a Field Calculator can be used to simply create new field with numerical IDs, and a Python function to transform the strings to numbers by above-mentioned rules. The code of this function is at listing 3.2 and the result of this operation can be found in the figure 3.7.

Listing 3.2: Python function for transforming string IDs into numerical IDs.

```
1 from qgis.core import *
2 from qgis.gui import *
3
4 @qgsfunction(args='auto', group='Custom')
5 def renameString(value, feature, parent):
6     UX = value[:2]
7     val = None
8     if UX == 'UC':
9         val = 1
10    elif UX == 'UN':
11        try:
12            int(value[2])
13            val = 2
14        except ValueError: # 'UNW'
15            val = 4
16    elif UX == 'US':
17        val = 3
18    else:
19        val = 9
20    if len(value) == 6:
21        if val == 4:
22            ret = "".join( (str(val),value[3:6]) )
23        else:
24            ret = "".join( (str(val),value[2:5],str(ord(value
25                [5]))) )
26    elif len(value) > 6:
27        ret = "".join( (str(val),value[2:5],str(ord(value[5]))
28            ,str(value[6:])) )
29    else:
30        ret = "".join( (str(val),value[2:]) )
31    return ret
```

Following this, a rasterisation of the polygon layer can be done. We have used a method *rasterize* from the GDAL library¹. Depending on the length of IDs, it might be necessary to use some longer datatype for cells, like `UInt32`, as in our case.

A resolution of the raster is another aspect to consider. It depends largely on the

¹<https://www.gdal.org/>

NAME	pxID
US029A	302965
UN128	2128
US114	3114
US117	3117
US118	3118
UN136	2136
UN133	2133
UN131	2131
UN133	2133
UC142U	114285
UC142U	114285

Figure 3.7: Screenshot of an attribute table in QGIS with original string IDs in "NAME" column and numerical IDs in "pxID" column.

geometry of the individual rooms in the building. The raster in this step must preserve all the gaps between rooms, i. e. the pixel should be smaller than the thinnest wall in the building. On the other hand, smaller pixels means larger volume of data and therefore longer processing times. After some elaboration, a resolution of 0.05 m/px appeared to be suitable in our case. Figure 3.8 compares the rasterised floor plans with too low and with sufficient resolution.

When the floor of the building is represented in a raster form, methods of mathematical morphology can be used. To fill the gaps between rooms, a morphological dilation is well suited. (Dougherty 1992) In QGIS, there is a function from GRASS GIS available, named *Region growing* or `r.grow` in the console, which implements the morphological dilation. In other GIS software, the same function might be called *Euclidean allocation*. In the algorithm, there must be explicitly specified a distance in pixels of which the algorithm extends the current rooms. Now it is important to consider the geometry of our building again: if we choose the distance too small, the gaps between rooms won't fill and we won't discover their adjacency. But if we choose the distance too large and the building has little niches or actual outside space between some rooms, we would fill them and incorrectly infer that the rooms are adjacent (have a common wall), while they are not. We ended up with a distance of 25 px, but this value depends vastly on the resolution of raster chosen in the previous step. Figure 3.9

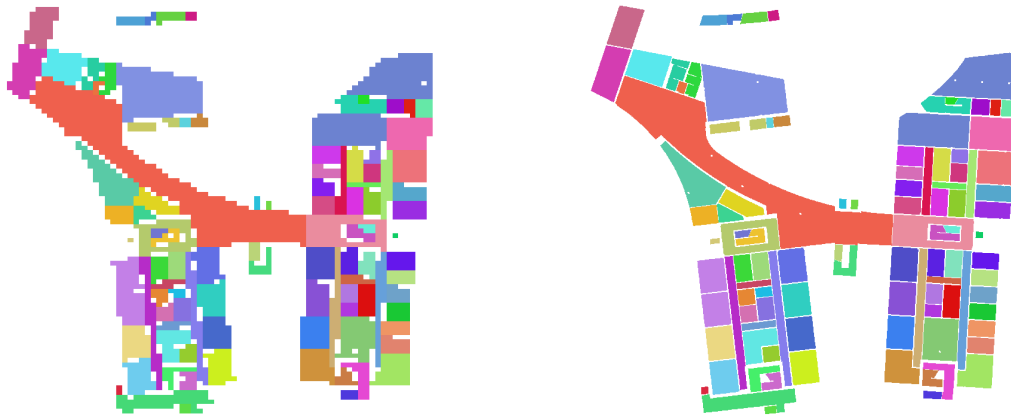


Figure 3.8: Rasterised floor plan with too low resolution (1.2 m/px, on the left) and a resolution which preserves the shape of walls clearly (0.05 m/px). The rooms are randomly coloured for better readability.

shows the results of both well chosen dilation and of a dilation of too many pixels.

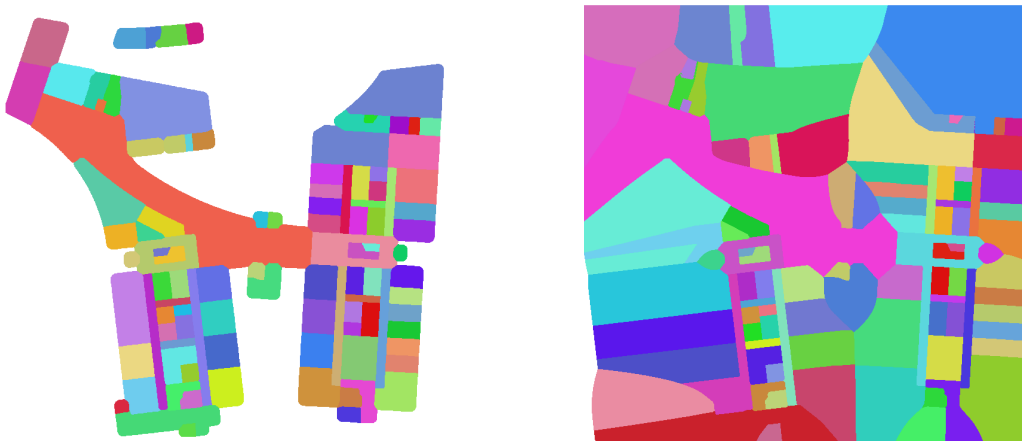


Figure 3.9: Different results of morphological dilation. Dilation of too many pixels can produce unexpected and undesired results (on the right). The rooms are randomly coloured for better readability.

When the rooms touches each other in the raster, it is possible to convert the rooms back to polygons. Function *polygonize* from the GDAL library is suited for this step. The only aspect, which must be kept in mind, is that this function will assign new feature IDs (FIDs) to the polygons, and the numerical IDs from the raster layer will be stored in a field of the attribute table. Figure 3.5 displays the floor plan which was used as an input layer and the newly created floor plan with adjoining rooms.

As the processing of layers is quite complex and tedious and it needs to be performed for each floor of the building separately, we have created a model in the QGIS's

Graphical Modeler¹, which performs the three last mentioned crucial parts at once, while allowing to set only the parameters valid for our objective. It still needs to be run for each floor separately, but speeds up the work-flow noticeably. The graphical model is depicted in figure 3.10 and corresponds to the first three operations in the UML diagram in figure 3.6.

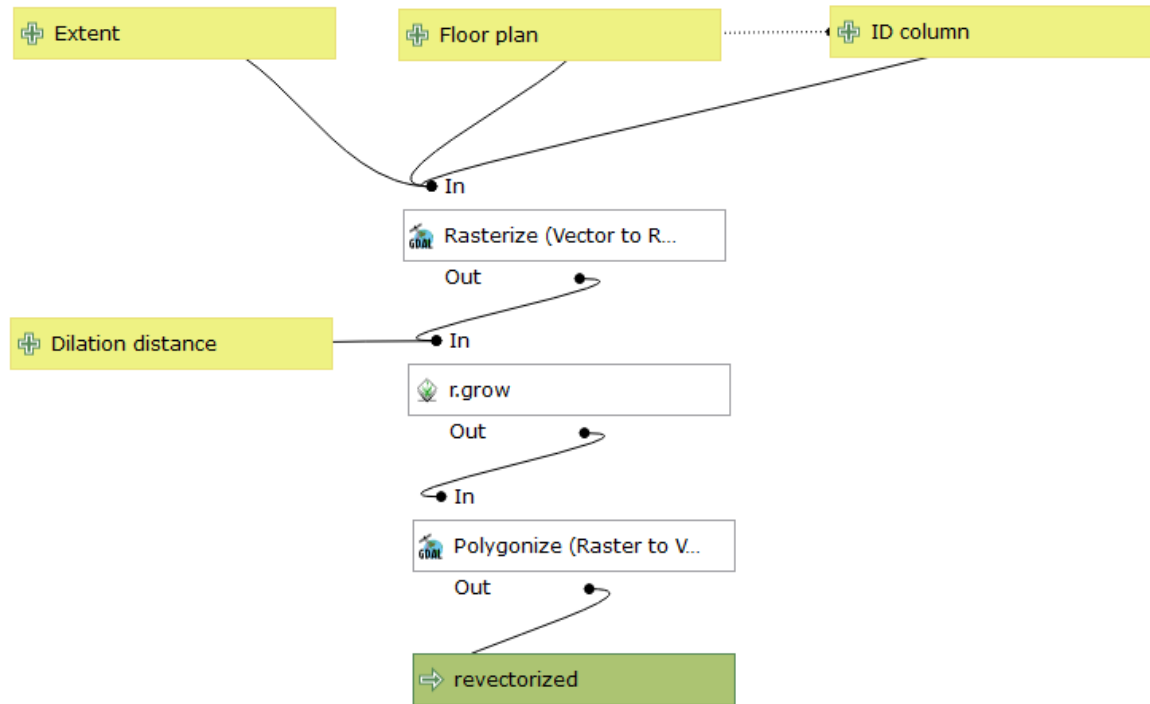


Figure 3.10: Model in QGIS’ Graphical Modeler for simply rasterise-dilate-vectorise any polygon layer.

From the adjoining polygons, we can finally extract the lines representing the boundary between individual rooms. Each boundary describes, which rooms it separates. In QGIS, there is an algorithm *Polygons to Lines*, but it only creates closed lines around each polygon, so they overlap each other in the case of adjacent polygons. Beside that, in GRASS GIS, there is a function called *v.type*, which can transform boundary of polygons into lines. It does what is needed in our case, i. e. it creates line segments, each beginning and ending where corners of three rooms meet, but it holds no information about which polygons of the input layer the particular segment separates. For these reasons, we had to use ArcGIS’s function *Polygon To Line*², which,

¹https://docs.qgis.org/3.4/en/docs/user_manual/processing/modeler.html

²<http://pro.arcgis.com/en/pro-app/tool-reference/data-management/polygon-to-line.htm>

beside creating line segments as the `v.type` does, also creates two additional fields in the attribute table "Left_ID" and "Right_ID".

As a last step of this process, saving the acquired information about adjacency into a file is necessary, so that the heat transfer simulator can read it and use it. This involves exporting the attribute tables of *a*) newly acquired boundary lines (hereinafter referred to as *adjacency file*), *b*) revectorised polygons of adjoining rooms (hereinafter referred to as *mapping file*) and *c*) original polygons of rooms (hereinafter referred to as *properties file*). The *adjacency file* is important because it contains the desired adjacency of rooms, the *mapping file* is necessary because it contains the mapping between newly generated FIDs and the original ones. The *properties file* should contain additional information about the rooms, which might be useful in the process of simulation, like their areas, heights etc.

Finally, a bit of programming was necessary to create an input file acceptable by the heat transfer simulator. A short Python program `horizontalAdjacencyTranslator` does the following: reverses the IDs of rooms back to the original string IDs and then writes the data about the room adjacency into a JSON file. This JSON file can be later fed into the heat transfer simulator, but is not limited to it, as it contains only information about a spatial relationships within the building, it can be generally used to any other type of simulation or modelling.

`horizontalAdjacencyTranslator` can be controlled using a command line and it operates with six parameters:

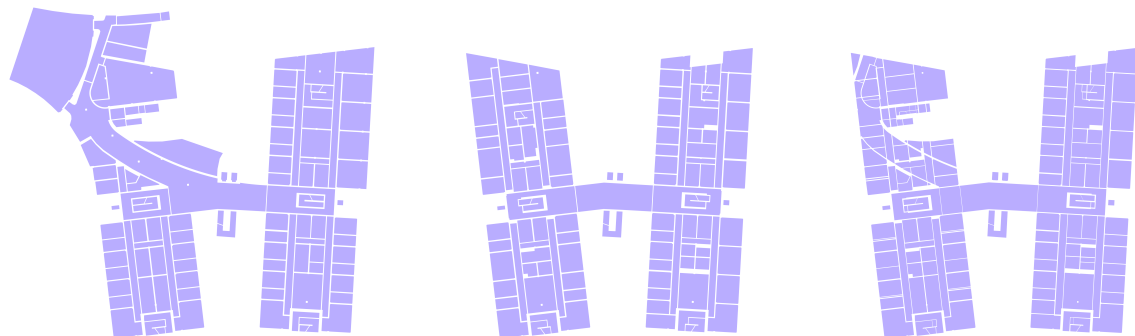
- `-p` or `--properties`,
- `-m` or `--mappings`
- `-f` or `--filename`,
- `-o` or `--outputfilename`,
- `-r` or `--rewrite` and
- `-h` or `--help`.

Parameter `-p` specifies a path to the *properties file*, parameter `-m` path to the *mapping file* and `-f` path to the *adjacency file*. The `-o` parameter can be used to provide a desired name of the newly generated output JSON file, or, if this file already exists, the new adjacency relationships will be appended to it. If the output file exists, but from some reason the user wants to throw it away and start from scratch, the optional

parameter `-r` will cause the file to be overwritten. If parameter `-h` is provided, the program prints its usage details as it is common for command line applications. Complete code of the `horizontalAdjacencyTranslator` can be found in appendix A.1 and on the GitHub website.¹

3.2.3 Determining Adjacency of Rooms through the Ceilings

Compared to the adjacency of rooms in the same floor, as described in the previous section, determining the adjacency of rooms between levels can be determined in GIS quite simply. As long as the floor plans are stored in polygon layers with the same coordinate system, one can simply use the algorithm of intersection to find areas which are adjacent through the ceiling and floor respectively. In both QGIS and ArcGIS, as well as in many other GIS software, the *Intersection* function is natively available as a part of its geoprocessing functions. The input layers of the algorithm and its result are depicted in the figure 3.11.



(a) Floor plan of a second floor of the FAV building. (b) Floor plan of a third floor of the FAV building. (c) Intersection between these two floors.

Figure 3.11: Intersect algorithm used to find overlapping parts of rooms.

That said, the wanted relationship can only be obtained by manipulating vector layers and thus we do not need to transform the IDs as we had to in the previous case, where we needed to manipulate with raster layers as well. However, when performing the intersection between two neighbouring levels, it is important to keep the ID columns from both the input layers. This way, the overlapping part of two rooms will be identified by both the ID of the bottom room and by the ID of the upper room. A short example of the attribute table of the output can be seen in figure 3.12.

When the intersection is computed, it is possible to calculate the area of each overlapping part, which can later be used as a parameter in the heat transfer simulator.

¹<https://github.com/jmacura/shape2lumped>

	NAME	NAME_2	area
1	UC236	UC335	42,726
2	UC237	UC336	46,901
3	UC238	UC337	19,663
4	UC238	UC338	17,033
5	UC239	UC339	20,412
6	UC240	UC340	37,137

Figure 3.12: First few rows of the attribute table of the result of the intersection.

Once the polygon layer of the overlapping parts is ready, we can export it into a table or text file without geometry which can be processed by a program and merged with the JSON file created in the previous step. A Python script `verticalAdjacencyTranslator` has been created to accomplish this. The script can be controlled from the command line and it recognizes four parameters:

- `-f` or `--filename`,
- `-o` or `--outputfilename`,
- `-r` or `--rewrite` and
- `-h` or `--help`.

Parameter `-f` is the name of the table or text file exported from GIS, while parameter `-o` is the name of the JSON file to be created, or, if it already exists, to append the new data into it. The parameter `-r` is optional and can be used to overwrite existing output file if desired. If parameter `-h` is provided, the program prints its usage details as it is common for command line applications. Complete code of the `verticalAdjacencyTranslator` can be found in appendix A.2 and on the GitHub website.¹

3.3 Building Simulation Model

Heat transfer simulation in our case serves to simulate an output of real temperature sensors in each room of the example building. The simulation program is designed to compute how the temperature in the building will change through the time, based

¹<https://github.com/jmacura/shape2lumped>

on the interchange of heat among the individual rooms and the outside temperature, called "ambient". It works with an abstract model of the building, which describes the adjacency among individual rooms and with the ambient, and the thermal conductivity between these elements.

This abstract model is a state-space model of the building, which is also often called a *lumped model*. This model perceives the rooms as *thermal zones*, which have certain heat capacitance C . For two thermal zones with heat capacitances C_1 and C_2 , their mutual heat transfer rate is the given by

$$C_1 \frac{dT_1}{dt} = K_{12}(T_2 - T_1),$$

where t denotes the time, T_1 and T_2 are the temperatures in zone 1 and zone 2 respectively and K_{12} denotes the heat transmission coefficient between the two zones. (Oldewurtel et al. 2010)

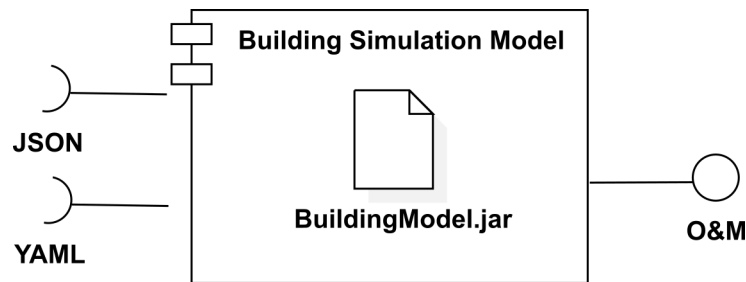


Figure 3.13: The simulation component comprises solely the simulation model written in Java. Beside the JSON file with adjacency information, it can also accept a YAML file with simulation parameters.

The *BuildingModel* program was initially created by Martin Střelec at the Department of Cybernetics¹, Faculty of Applied Sciences, University of West Bohemia. In cooperation with the original author, it was extended so it accepts input data in a form of JSON and YAML files and serves O&M compatible JSON file (shortly OM-JSON) as an output. How it fits to our component model can be seen in the figure 3.13. The source codes of the simulation model are open and currently publicly available for any use at the GitHub website.²

The program itself can be easily run as a command line application. It accepts four parameters:

- `-m` or `--modelFile`,
- `-o` or `--outputfolder`,

¹<http://www.kky.zcu.cz/en>

²<https://github.com/jmacura/BuildingModel>

- `-p` or `--paramsFile` and
- `-h` or `--help`.

Parameter `-m` specifies the input JSON file with information about adjacency among the rooms in the whole building. It is exactly that file, which was obtained by the process described in section 3.2. Parameter `-o` then specifies in which output folder will be saved the result of simulation. The program creates a separate file for each room, so this might be quite a lot of files for buildings with many rooms. With optional parameter `-p`, a YAML¹ file containing desired parameters of the simulation can be set. If parameter `-h` is provided, the program prints its usage details as it is common for command line applications.

The simple YAML file currently allows user to set the following six parameters: a date and time, when the simulation should start; duration, which the simulation should cover; sampling period, in which the simulation steps are computed; initial temperature in the rooms of the building and constant temperature outside of the building. Units of temperatures are degrees Celsius and temporal units must be compliant to ISO 8601 format, where a date is expected in a precision of days, time and duration are expected in hours, minutes and seconds and the sampling period is expected only in minutes and seconds. Example of such a YAML file can be seen at listing 3.3. If no parametric file is provided, the simulation will be computed with default temperature values 10 °C for rooms, 20 °C for outside, it will start simulating since the time it was launched and will simulate next 2 hours with a sampling period of 15 minutes.

Listing 3.3: Example YAML file with heat transfer simulation parameters.

```
simulationStartDate: 2019-04-28
simulationStartTime: "8:10:00"
simulationDuration: "02:30:00"
samplingPeriod: "10:00"
initialTemperature: 12.0
outsideTemperature: 25.0
```

Once the simulation finishes, the program creates a separate file with the temperature observations for each room of the model. This file is a JSON file following Observations&Measurements standard by OGC, namely its *DiscreteTimeSeriesObservation* type. It contains the result of the heat transfer simulation in a form of points

¹YAML Ain't Markup Language (YAML™) Version 1.2 : <https://yaml.org/spec/1.2/spec.html>

in time, correlating with the sampling period of the simulation, where for each point in time a temperature value is attached. Considering the rooms depicted in the figure 3.4a, the OM-JSON file with the simulation result for the "red" room is shown in the listing 3.4. Content of the file for other rooms will only differ in the ID of the room and the temperature values. Structure of the OM-JSON file is standardised and thus independent on the inner implementation of the Simulation Model.

Listing 3.4: Example of an OM-JSON file for one of the rooms. Some less relevant metadata and rest of the time instants are omitted for brevity and better readability of the code.

```
{
  "resultTime": "2019-05-16T00:19:56.513Z",
  "result": {
    "metadata": {},
    "defaultPointMetadata": {
      "uom": {
        "href": "http://qudt.org/vocab/unit#DegreeCelsius"
      },
      ...
    },
    "points": [
      {
        "time": {
          "instant": "2019-06-18T11:00:00Z"
        },
        "value": 22.0
      }, {
        "time": {
          "instant": "2019-06-18T11:02:00Z"
        },
        "value": 21.765143063285144
      },
      ...
    ]
  },
  "observedProperty": {
    "href": "http://environment.data.gov.au/def/property/air_temperature"
  },
  "phenomenonTime": {
    "end": "2019-06-18T15:00:00Z",
    "begin": "2019-06-18T11:00:00Z"
  },
}
```

```
"featureOfInterest": "red",  
"id": "red_temperature",  
...  
}
```

3.4 Transformation of Geodata into a Visualisation format

For the geodata to be easy to visualise in 3D in the web environment, it is favourable to transform them into a format which was designed for such purposes. There are many options among spatial data formats for this task, but as we intent to use *CesiumJS* platform for the visualisation, the glTF file format, which was described in section 2.1.1, is the most suitable one. This section covers the transformation process from a 2D polygon layer to a glTF file, which is the last task of our solution in the 3D GIS, as displayed in figure 3.14.

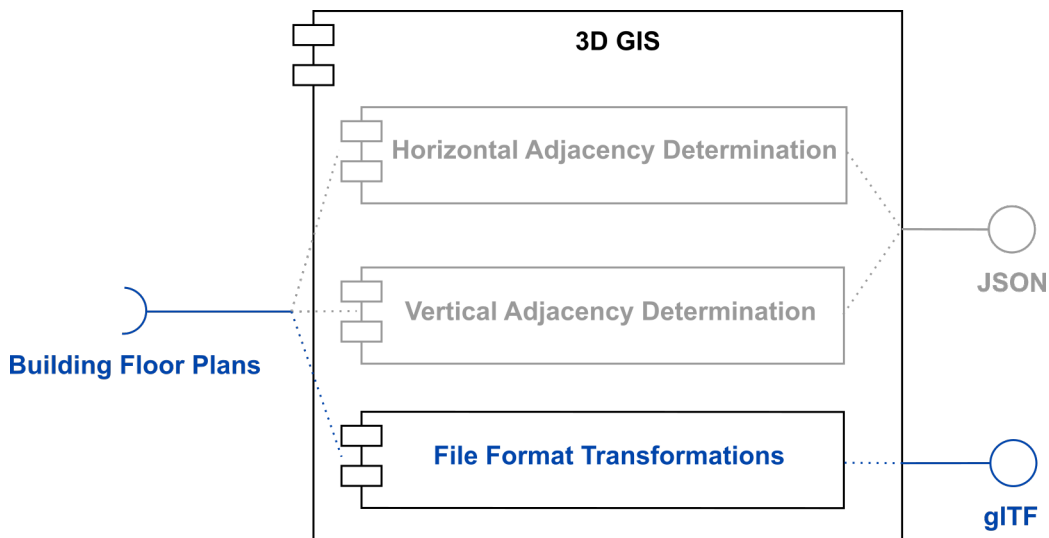


Figure 3.14: Overview of the 3D GIS component in the proposed solution. Subcomponent responsible for the production of a glTF file is highlighted.

So far, we have only worked with floor plans of the building, which are ordinary 2D geodata. These must be first converted into a 3D geodata so they can be visualised properly. In fact, visualising just 2D geometry is technically also possible, but in the case of visualising building rooms, the result would not be meaningful. Thus we need to extrude the flat polygons of individual rooms in each floor into their proper height, so they become solid figures in 3D space. Some attempts were made with storing

2.5D polygons in the GeoPackage format, but the extrusion in GRASS GIS was not successful. For this reason, we have used several tools in ArcGIS to transform ordinary 2D polygonal Shapefile into Shapefile containing a Multipatch geometry. For a brief description of Multipatch see section 2.1.1. The conversion work-flow was following: *a)* Open the Shapefile containing a floor plan in *ArcScene*, *b)* extrude the floor plan layer into desired height (the height must be stored in the attribute table) and *c)* use *Layer 3D to Feature Class* tool to export the extruded polygons into a Multipatch Shapefile. These steps lead to a proper 3D geodata that can be converted into the glTF format afterwards.

However, the transformation from Shapefile to glTF is not as simple as it may sound like. Many roads were probed, but only few led to a successful end. Among the tools tested to directly convert Multipatch Shapefile into glTF were e.g. *FME Desktop*¹ by Safe Software or *PiXYZ Studio*² by PiXYZ Software, which both are advanced commercial tools, but both produced glTF files which were unable to display in *CesiumJS*. Other attempts involved multi-step conversion through some third data format, from which the COLLADA format appeared to be the most promising. Hence, we used *Multipatch to Collada* tool in AcGIS to convert the Multipatch Shapefiles into COLLADA files, but neither the conversion from COLLADA to glTF was without obstacles. E.g. a Node.js based application *collada2gltf-web-service*,³ which was initially used for the conversion, produced invalid glTF files, based on the online glTF Validator⁴ by Khronos Group. Finally, a *COLLADA2GLTF*⁵ command line tool written by Khronos Group in C++ language has successfully produced valid glTF files, which were possible to display in *CesiumJS*.

The structure of glTF file, as described in section 2.1.1, is presented in a concise and significantly shortened version in the listing 3.5. The complete code for the glTF model of one room, namely the "red" room from the figure 3.4a, can be found in appendix A.3.

Listing 3.5: One of the rooms represented in a glTF format.

```
{
  "scenes": [
    {"nodes": [0]}
  ],
  "scene": 0,
```

¹<https://www.safe.com/fme/fme-desktop/>

²<https://www.pixyz-software.com/studio/>

³<https://github.com/AnalyticalGraphicsInc/collada2gltf-web-service>

⁴<http://github.khronos.org/glTF-Validator/>

⁵<https://github.com/KhronosGroup/COLLADA2GLTF/>

```

"nodes": [
  {
    "children": [1],
    "matrix": [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 0.0, 0.0, 1
      .0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0],
    "name": "Z_UP"
  },
  {
    "mesh": 0,
    "children": [3, 2],
    "name": "red.dae"
  },
  {"mesh": 1},
  {"mesh": 2}
],
"meshes": [
  {
    "primitives": [
      {
        "attributes": {
          "NORMAL": 1,
          "POSITION": 2
        },
        "indices": 0,
        "mode": 4,
        "material": 0
      }
    ],
    "name": "red.dae3"
  },
  ...
],
"accessors": [
  ...
],
"materials": [
  ...
],
"bufferViews": [
  ...
],
"buffers": [
  ...
]

```


Chapter 4

Data Visualisation in Four Dimensions

In chapter 3 we have described, how to prepare data for the desired spatio-temporal visualisation in three dimensions. In this stage, we have a set of glTF files along with KML files, which together stores all necessary information about geometry and position of the objects, i.e. rooms, we are about to visualise, and a set of O&M files, which contains the temperature samples in periodic time instants. In this chapter we describe how to blend data from all these files together and present them visually in an exemplar 4D application.

4.1 Schema of the Web 4D GIS Application

Before we start displaying the data, we must set up the application as a whole. As previously stated, our exemplar visualisation is based on *CesiumJS* platform, which itself relies on Node.js environment. Node.js is a cross-platform open-source JavaScript interpreter and runtime-environment, which is generally used to run applications on a server back-end, but it can be quite easily installed on a regular personal computer and used to run a local server. *CesiumJS* requires Node.js so it can dynamically serve image tiles on its virtual globe, for which it needs a local server to run.² That said, *CesiumJS* runs JavaScript both at the server side and the client side of the application.

Our application was designed similarly to *CesiumJS*. The major part of it runs in the client's browser, but as it needs to read a lot of files from the file system, it also needs small application server running on its own. It would be possible to design the application as a pure browser based solution, but that option would require to adjust the code of the application with every change in the file system. On the other hand, the API based solution we have proposed and implemented allows us to easily add or

²<https://stackoverflow.com/questions/31428956/run-cesiumjs-with-no-server-requirements>

remove objects in the visualisation. In other words, it allows to display more or less sensors as they are added or removed from the file system. Figure 4.1 displays a simple schema of the designed *CesiumJS*-based application.

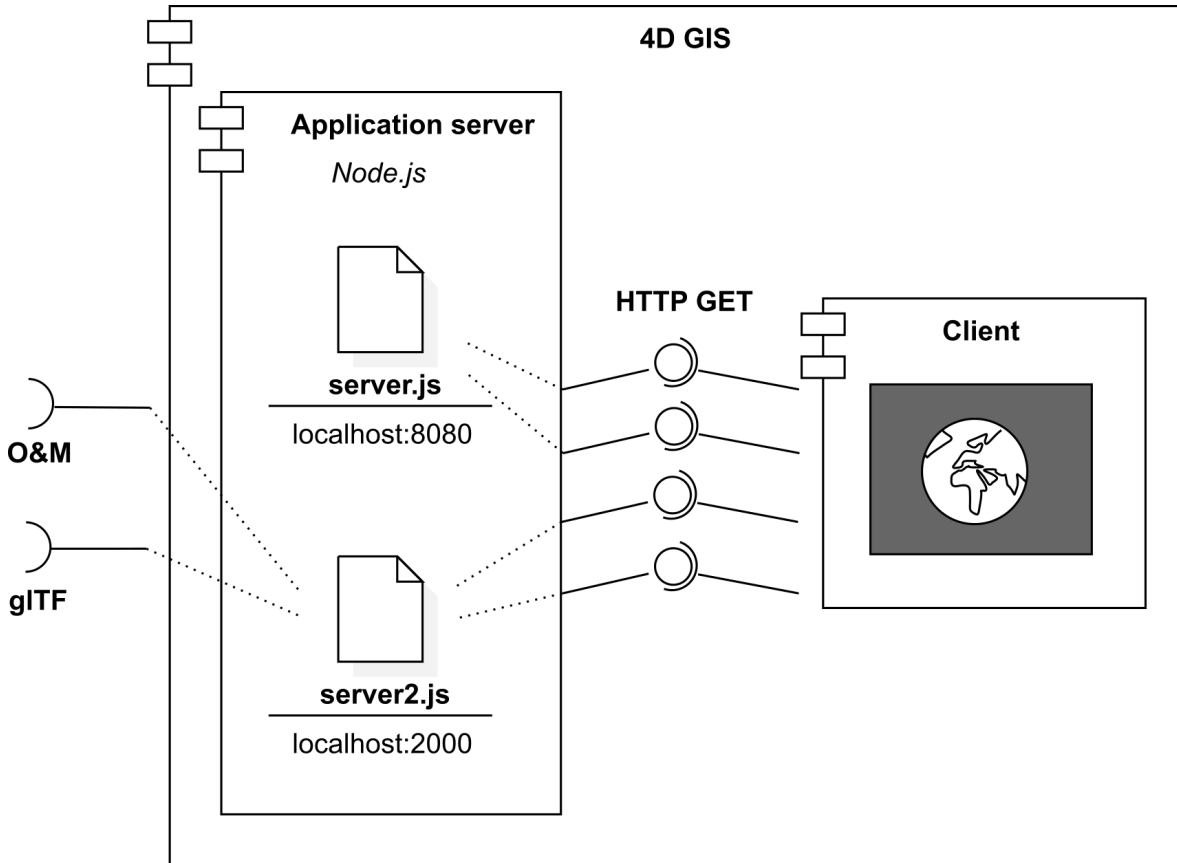


Figure 4.1: Designed schema of the 4D GIS web application.

The initial set up of any *CesiumJS*-based application is as simple as displayed in listing 4.1. If the *CesiumJS* libraries are attached to the HTML file and the server on Node.js is running, then a virtual globe will appear in the browser.

Listing 4.1: A code snippet which enables Cesium virtual globe in the HTML document

```

1 | <div id="cesiumContainer"></div>
2 |   <script>
3 |     var viewer = new Cesium.Viewer('cesiumContainer');
4 |   </script>
5 | </div>

```

4.2 Blending 3D Geodata and Sensor Data in a 4D GIS Application

4.2.1 Loading 3D Models in the CesiumJS Application

As the application is designed to be as independent on the underlying data as possible, it is accompanied by a simple server API running on Node.js. This API accessible under `http://localhost:2000/v1.0/` URL and understands basically these four methods:

- `http://localhost:2000/v1.0/` which returns the URIs of the remaining methods,
- `http://localhost:2000/v1.0/Things` to get a list of all models of rooms currently available on the server,
- `http://localhost:2000/v1.0/Observations` to get a list of all observations currently available on the server (but in our case, this list correlates with the previous one) and
- `http://localhost:2000/v1.0/Observations(temperature_ID)`, where ID is one of the IDs retrieved by the previously mentioned method.

The methods were inspired deeply by the SensorThings API standard by OGC, but for our use-case implementing all the methods required by the standard would be a bit like gilding the lily. Hence, only this tiny subset of the standard was implemented by now, but it allows for simple extension in the future for more robust applications, without the need to rework the API all over.

Once the application receives the list of glTF room models available, it starts to load them one by one and converting them into a CZML format. This approach was chosen as a more generic way than a pre-conversion by another tool. Function `getZoneList()`, responsible for loading the list of rooms, is a regular AJAX request and the function returns a Promise object which resolves once the list is retrieved. The usage of Promise API introduced in ECMAScript 2015 is followed in the whole application, which leads to a non-blocking and asynchronous code. Listing 4.2 shows how the asynchronous functions are chained, when they need to be executed sequentially.

Listing 4.2: Function which requests the list of available models

```
1 | let zones = getZoneList();  
2 | let zonesL = [];
```

```

3 | zones.then(zoneList => {
4 |     zonesL = zoneList.result;
5 |     return loadZones(zonesL);
6 | })

```

Function `loadZones()`, which is in shortened version displayed in listing 4.3 is quite complex but important enough, so we will walk through the interesting parts of its code. Lines 1–5 contains a declaration of new and empty CZML document. Each CZML document must begin with this preamble. This document is later loaded into the Cesium on line 7, after a declaration of new source of data for Cesium on line 6. Line 8 adds this data source into a Cesium Viewer object, which has the consequence of rendering the data in the virtual globe. Lines 9–21 are responsible for the actual conversion of data and proper Promise returning. Some juggling with Promises was necessary because some parts of the code must be performed for each model in the list while some must be only run once when all the models are loaded. From these final lines, the line 12 is most important as it calls the function `generateCzmlItem()`, which does convert each glTF into CZML.

Listing 4.3: Function `loadZones()`, responsible for loading the glTF models and their transformation into CZML

```

1 | let czml = [{
2 |     "id" : "document",
3 |     "name" : "CZML Model",
4 |     "version" : "1.0"
5 | }];
6 | let dataSource = new Cesium.CzmlDataSource();
7 | dataSource.load(czml);
8 | viewer.dataSources.add(dataSource);
9 | return new Promise((resolve, reject) => {
10 |     resolve(Promise.all(
11 |         zones.map((zone) => {
12 |             return generateCzmlItem(zone).then(czmlItem => {
13 |                 return dataSource.process(czmlItem);
14 |             }).catch(err => console.log(err));
15 |         })
16 |     ).then((res) => {
17 |         return dataSource;
18 |     }).catch(err => console.log(err))

```

```

19 |         );}
20 |     );

```

The `readCzmlItem()` function will be described just briefly, but it contains one important and probably unexpected operation, which needs to be mentioned: a rotation. That is because the orientation of axes in the COLLADA model is different than the one defined in the CZML Specification. Hence, all the models must be rotated by $\frac{\pi}{2}rad$ around the axis pointing up, so they display properly. Beside this, the `readCzmlItem()` function also requests a KML file with the same ID as the room model in glTF and uses the WGS84 coordinates from it to specify an anchor point on the Earth's surface for each model. It is also interesting to note that the built-in `dataSource.process()` function is asynchronous, so the models are being loaded in parallel.

If we use the example of "red" room from the figure 3.4a once again, this room in CZML format will look as depicted in the figure 4.4. The `position` property in the listing specifies the anchor point of the model, while the `orientation` property describes the rotation around this anchor point, as already described. The orientation is defined by a unit quaternion, also called *versor* in mathematics, which uniquely determinates a rotation in 3D. (Hamilton 1844–1850) The geometry itself is only linked via a `glTF` property. Notice the yet empty property `fav_temperature` – it will be used for storing the temperature in the room in given time instances, but only after the observation data are loaded as well.

Listing 4.4: One of the rooms represented in a CZML format.

```

{
  "id": "red",
  "name": "Red room",
  "position": {
    "cartographicDegrees": [
      13.352512130150579,
      49.726485530048365,
      15.87
    ]
  },
  "orientation": {
    "unitQuaternion": [
      -0.040021278002015104,
      0.34192608109629546,
      0.10914691092161553,
      0.9325083400214091
    ]
  },
}

```



```

"properties": {
  "fav_temperature": {}
},
"model": {
  "gltf": "./models/gltf/red.gltf",
  "scale": 1
}
}

```

4.2.2 Connecting Simulation Outputs into the CesiumJS Application

In this phase, the simulation output behaves like any other IoT sensor. As the application works with the O&M file, which is a standardised format, it can consume data from any source of the observations, if it provides the result of measurement in the form of OM-JSON.

In terms of the JavaScript code, the part that is responsible for loading the sensor data into the CZML is a function `readObservations()` chained after the `loadZones()` Promise. The code of `readObservations()` function is on listing 4.5. There are two important functions used inside it – first being the `readOM()` function, which is an AJAX wrapper very similar to the `getZoneList()` mentioned above, while in this case it retrieves a single OM-JSON file for each room.

Listing 4.5: Function `readObservations()`, responsible for loading OM-JSON observations and transforming them into CZML packets.

```

1 | function readObservations(dataSource, zoneList) {
2 |   return new Promise((resolve, reject) => {
3 |     resolve(Promise.all(
4 |       zoneList.map(entity => {
5 |         let id = entity.name;
6 |         return readOM(id).then((data) => {
7 |           setTimeTempVars(data);
8 |           let czml = generateCzmlUpdate(data);
9 |           console.log("obser:", czml);
10 |           return dataSource.process(czml);
11 |         }).catch(err => console.log("inall:", err));
12 |       })

```

```

13 |         ).then((res) => {
14 |             return dataSource;
15 |         }).catch(err => console.log("out all:", err))
16 |     })
17 | }

```

The second important function here is `generateCzmlUpdate()`, which creates a very simple CZML Packet. This Packet only contains the ID of the feature which was observed by that particular sensor, i. e. in our case an ID of the room, and the observed property with its values throughout the observed timespan. Observed property in our case is indeed the temperature in the room and the observed timespan correlates with the duration which was set as a simulation time in *BuildingModel*. If we describe it on the example of the "red" room from the figure 3.4a for the last time, then the new Packet will look as in listing 4.6. Cesium recognises the equal IDs and treats this packet as if it were an integral part of the CZML file presented in the listing 4.4.

Listing 4.6: Temperature observation for one of the rooms. Rest of the measurements is omitted for brevity and better readability.

```

{
  "id": "red",
  "properties": {
    "fav_temperature": {
      "number": [
        "2019-06-18T11:00:00Z",
        22,
        "2019-06-18T11:02:00Z",
        21.883454359470754,
        "2019-06-18T11:04:00Z",
        21.765143063285144,
        "2019-06-18T11:06:00Z",
        21.64537704169493,
        ...
      ]
    }
  }
}

```

Finally, but as already mentioned, not necessarily after all observation data are loaded, rather during the time they are being loaded (thanks to the asynchronous nature of the code), the very important event handler function is bound to the graphical entities representing rooms. Without this event handler, the rooms would remain their implicit colour still and would not visualise the change of temperature during the time.

The `addTimeVariableColor()` function, whose code is portrayed on listing 4.7, can not be bound to the CZML source when it is created, but it must be bound to the entities after they are processed by Cesium.

Listing 4.7: Function `addTimeVariableColor()`, which bounds the `temperatureCallback()` event handler to each room.

```
1 function addTimeVariableColor(entities) {
2   entities.forEach(entity => {
3     let cbP = new Cesium.CallbackProperty(temperatureCallback.
4       bind(entity), false);
5     entity.model.color = cbP;
6   });
7 }
```

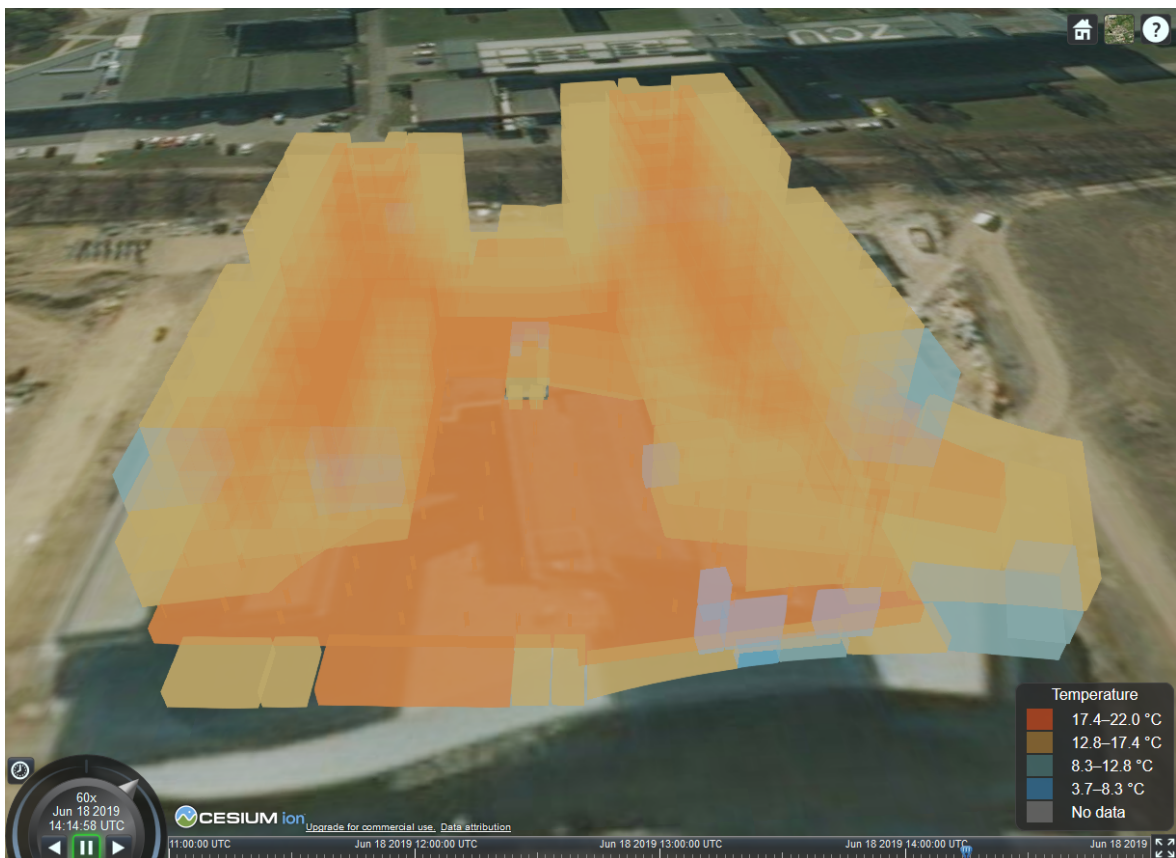


Figure 4.2: Screenshot of the blended data visualised in the CesiumJS environment. Camera is pointing approximately to the same view as in figure 3.1.

All codes of the *CesiumJS*-based application in HTML, CSS and JavaScript are saved on the included DVD as described in appendix B and can be found under the

MPLv2 license on the GitHub website¹, from where anyone can easily fork and edit them for his/her own application. A functional demo of the application described in this section is also available online².

4.3 Possible Use Cases of the Solution

The methodology described in chapters 3 and 4.2 can be used in several use cases. One of them is described here in more detail, because we consider it illustrative.

Simulate and visualise, how the temperature inside the building will change, if:

- the building is heated up to some operational temperature (e. g. 22 °C),
- it freezes in the outside (e. g. -1.5 °C),
- the heating is immediately turned off in the whole building.

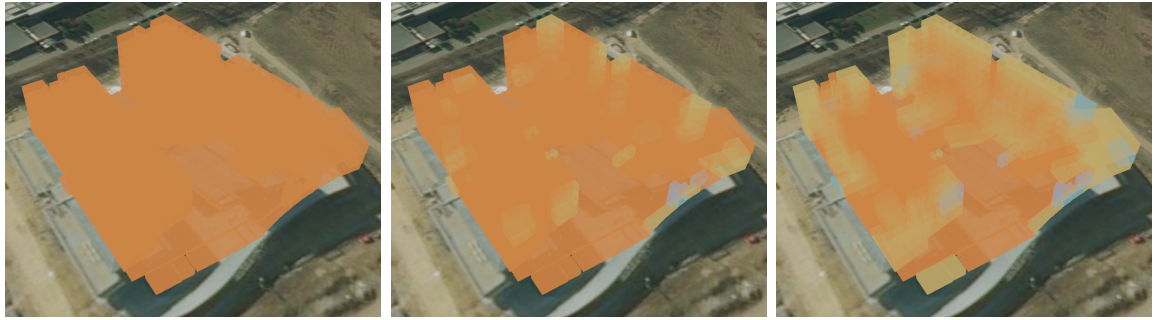
Figure 4.3 displays, how the temperature changes over the building in this scenario. The heating was turned off at 11:00 and the simulation was computed for the following 4 hours. Clear advantage of the visualisation of the temperatures in three dimensions is that the user can investigate the temperature values of individual rooms visually and all in once. The cartographic visualisation allows to zoom in and out, rotate the model and shift the time, so the user can focus on different parts of the building in the same time or on one part in longer time interval. Zoomed-in model with focus on one room is depicted in the figure 4.4.

Similar use cases can be covered with the same workflow quite easily, without the need to change the programmed parts of described components, or with just minimal adjustments. For instance, the vice versa scenario, when the building is cooled down to some temperature, but it is warm in outside and the cooling system goes down, can be examined by just changing the input parameters of the simulation model. There is no need to change anything in the *CesiumJS*-based application, as it will adjust the time frame and colour scale automatically.

Some other interesting scenarios would require only a slight modification of the simulation model. E. g., technically it can not only compute the simulation with a constant ambient temperature, but it may change over time. This would allow either to predict the heat transfer with outside more precisely, based on a weather forecast, or to better reconstruct the temperatures inside the building, based on an outside measurements.

¹<https://github.com/jmacura/observations4d>

²<https://jmacura.ml/observations4d/>



(a) Temperatures at 11:00. (b) Temperatures at 12:00. (c) Temperatures at 13:00.



(d) Temperatures at 14:00. (e) Temperatures at 15:00.

Figure 4.3: The same scene in the CesiumJS-based application, but in different moments of time. The colour of individual rooms is changing based on the temperature value in the given room in the given time.

Although the simulation model is prepared for such cases, this feature is not currently used and would need to be tested.

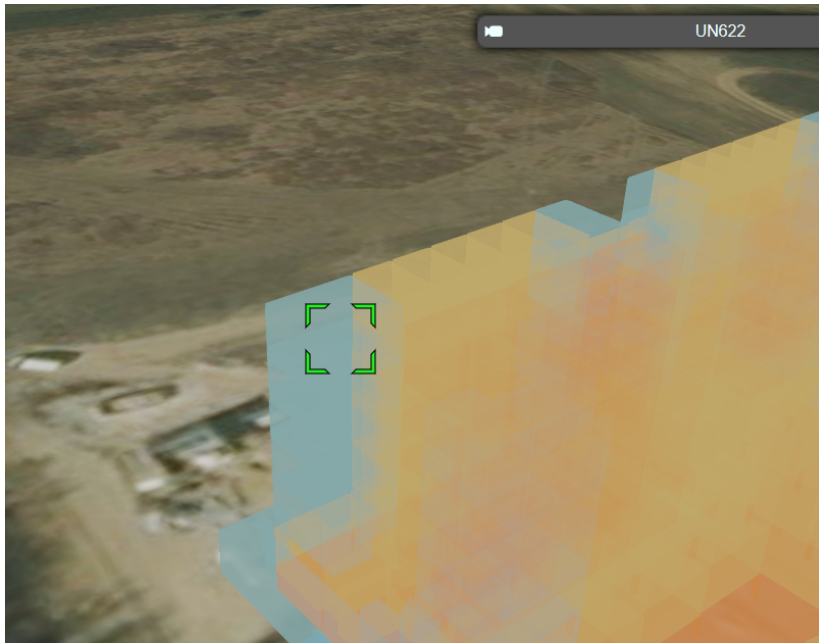


Figure 4.4: User concentrating on a certain part of the building in detail. The ID of the room of interest displays on click.

Chapter 5

Discussion

The solution chosen for the visualisation of sensor data in 4D was designed to demonstrate the technical possibility of it. As a prototype it naturally has its limits. Concerning the simulation model, it was design to also describe a changing ambient temperature (as mentioned in section 4.3), inner heating of the rooms or a heat transfer between the mass of the room itself and objects inside of it, like furniture, machines or humans. These features can be easily added programmatically, but the workflow described in chapter 3 does not cover them. Firstly its because these features were not needed to demonstrate the proof of concept and secondly because of the lack of data about these phenomena.

Another current limitation of the lumped simulation model is its simplicity. As it numerically solves an ordinary differential equation described in chapter 3.3, it has low numerical stability and readily diverges for longer sampling periods. The actual numerical stability depends on many circumstances like number of rooms, their heat capacity and mutual heat conductivity, starting temperature values or duration of the simulation. According to empirical evaluation, a sampling period of one minute seems to be quite secure choice.

Regarding the final visualisation in the *CesiumJS*-based application, it was described on the example of displaying the simulated data, but actually the application does not care where do the observations come from. It can be simulation of heat transfer, simulation of energy consumption, simulation of human movement etc. Or it can be real sensors sending data, either in real time or via some database which would hold the historical measurements. In such a case, no simulation component is needed and the overall UML diagram of components as portrayed in the figure 3.2 would change to a structure displayed in the figure 5.1. Notice that thanks to the clearly defined and standardised exchange formats between the components, O&M and glTF, the final

visualisation would not be affected and it is in fact prepared for such change.

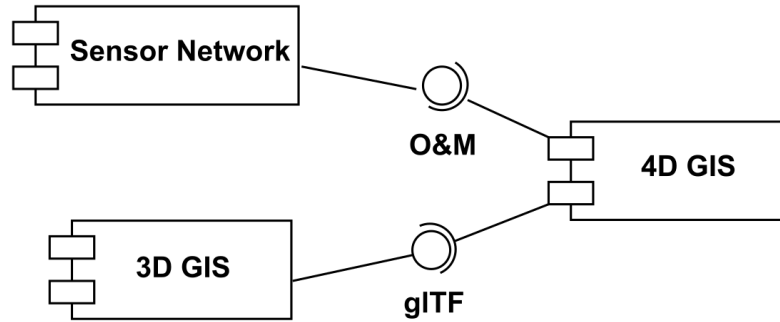


Figure 5.1: UML component diagram showing how the Building Simulation Model component can be exchanged with a Sensor Network component, while not affecting the 4D GIS component at all.

For the application to fully utilise the observations from sensors, it might be interesting to enhance it with a proper back-end including a database, from which the measurements could be read, but also recorded in it by the remote sensors in the same time. As of the *CesiumJS*-based application itself, many aspects can be improved for its higher user-friendliness – e.g. the filtering of rooms in the virtual scene seems to be a useful feature, which could provide better and clearer overview of the data. As such improvements and extensions tend to be specific to a certain use case, its implementations are up to the followers of this thesis.

Another interesting extension would be to experiment with the 3D Tiles file format for rendering the data in *CesiumJS*. Despite the 3D Tiles require more complicated pre-processing, they allow for larger dataset to be visualised in the browser, when compared to CZML.

Although the presented solution is purely a proof of concept, it might find its usage in the current or near-future projects. As the University of West Bohemia currently invests efforts into a Smart City related project *SmartCampus*, which involves connecting sensors through the IoT, along with cloud computing or development of smart devices, the methodology described in this thesis might allow to visualise the outputs of the project in a synoptic and novel manner.

Conclusion

The aim of this thesis was to investigate how to visualise a three-dimensional spatial data in compound with a time-variable observation data describing some phenomenon inside a building. As this problem is relevant in various engineering areas like Geographic Information Systems, Building Information Management, Internet of Things or Smart Cities, a state of the art in these domains, in context of the topic of this thesis, was examined first. During this research, several existing applications with a focus on either time-variable spatial data or interconnection of spatial and sensor data were found. The evolution of spatio-temporal GIS was explored as well, while focusing in deep on *CesiumJS*, a modern JavaScript platform for visualisation of 4D data.

As it was intended to provide the time-variable temperature data by using a heat transfer simulation, a methodology how to infer the necessary simulation parameters from the spatial data was proposed and described. Specifically, how to transform a conventional floor plans of the building into a lumped simulation model. For this purpose, a JSON file describing solely the topological adjacency of individual rooms of the building was designed. Additionally, one geoprocessing model and two Python scripts were created to ease the spatial data processing. Beside that, the spatial data of the building rooms were converted into glTF models which allow visualisation in the targeted web environment.

The JSON file was later successfully embraced in the lumped simulation model for a heat transfer simulation. The simulation model was designed to provide output in a form of the timeserie observations of temperature, compliant with the Observations&Measurements standard.

Finally, a cartographic visualisation combining measurements from the O&M file and geometry from the glTF file was produced. Data from both files were combined into the CZML file format, which is designed to contain a time-dependent spatial data. The application based on the *CesiumJS* platform displays the rooms of the building coloured according to the temperature inside it in given time. Potential user can move around the building in the virtual 3D scene and change the time frame at will. Thanks to the utilisation of the O&M standard, the application is designed so it can

display data from real sensors, instead of the simulation results, without the need for any adjustment. Functionality of the overall solution was demonstrated on the building of Faculty of Applied Sciences, University of West Bohemia.

Source codes of the 4D cartographic application, as well as the scripts used for data processing and the simulation model, were published to a GitHub website, where they are available under an open licence for future use and enhancement. One of possible extensions of the presented work is to enhance the 4D cartographic application with a database, which would allow receiving new remote sensor data and updating the observations in real time. For larger buildings and constructions, it might be worthy to use a 3D Tiles format for visualisation instead of CZML.

Bibliography

- 29481-1:2016, ISO (May 2016). *Building information models – Information delivery manual – Part 1: Methodology and format*. Tech. rep. International Organisation for Standardization.
- Abdul-Rahman, Alias and Morakot Pilouk (2008). *Spatial Data Modelling for 3D GIS*. Heidelberg, ISBN 978-3-540-74166-4. Springer. ISBN: 978-3-540-74166-4.
- Barnes, Mark and Ellen Levy Finch (2008). *COLLADA – Digital Asset Schema Release 1.5.0*. Tech. rep. The Khronos Group Inc., Sony Computer Entertainment Inc.
- Bhatia, Saurabh et al. (2017). *glTF Version 2.0*. Tech. rep. The Khronos Group Inc.
- Bonandrini, Sylvain, Christophe Cruz, and Christophe Nicolle (Jan. 2005). “Building lifecycle management”. In:
- Chen, Jianli et al. (May 2014). “A Case Study of Embedding Real-time Infrastructure Sensor Data to BIM”. In: pp. 269–278. ISBN: 978-0-7844-1351-7. DOI: [10.1061/9780784413517.028](https://doi.org/10.1061/9780784413517.028).
- Cox, Simon (2013). *OGC Abstract Specification: Geographic information – Observations and measurements*. Tech. rep. Open Geospatial Consortium.
- Cozzi, Patrick, Sean Lilley, and Gabby Getz (2018). *3D Tiles Specification 1.0*. Tech. rep. Open Geospatial Consortium.
- De Roo, Berdien, Jean Bourgeois, and Philippe De Maeyer (2017). “Usability Assessment of a Virtual Globe-Based 4D Archaeological GIS”. In: Springer, pp. 323–335. ISBN: 978-3-319-25689-4.
- Dougherty, Edward R. (1992). “An introduction to morphological image processing”. In: SPIE Optical Engineering Press, p. 7. ISBN: 0-8194-0845-X.
- Echterhoff, Johannes (2011). *OpenGIS SWE Service Model Implementation Standard*. Tech. rep. Open Geospatial Consortium.
- Farkas, Gábor (2017). “Applicability of open-source web mapping libraries for building massive Web GIS clients”. In: *JOURNAL OF GEOGRAPHICAL SYSTEMS* 19.3. Heidelberg, E-ISSN 1435-5949, pp. 273–295. ISSN: 1435-5949.
- Goodchild, Michael F. (2013). “Prospects for a Space–Time GIS”. In: *Annals of the Association of American Geographers* 103.5, pp. 1072–1077. DOI: [10.1080/00045608](https://doi.org/10.1080/00045608).

- 2013.792175. eprint: <https://doi.org/10.1080/00045608.2013.792175>. URL: <https://doi.org/10.1080/00045608.2013.792175>.
- Gröger, Gerhard et al. (2012). *OGC City Geography Markup Language (CityGML) Encoding Standard 2.0*. Tech. rep. Open Geospatial Consortium.
- Hagedorn, Benjamin et al. (2015). *OGC 3D Portrayal Service 1.0*. Tech. rep. Open Geospatial Consortium.
- Hamilton, William Rowan (1844–1850). “On Quaternions; or on a new System of Imaginaries in Algebra”. In: *Philosophical Magazine*. URL: <https://www.maths.tcd.ie/pub/HistMath/People/Hamilton/OnQuat/>.
- Jedlička, Karel (2018). “A comprehensive overview of a core of 3D GIS”. In: *7th International Conference on Cartography and GIS Proceedings*. Sozopol. ISSN 1314-0604. Bulgarian Cartographic Association.
- Kepka, Michal et al. (May 2017). “The SensLog Platform – A Solution for Sensors and Citizen Observatories”. In: pp. 372–382. ISBN: 978-3-319-89934-3. DOI: [10.1007/978-3-319-89935-0_31](https://doi.org/10.1007/978-3-319-89935-0_31).
- Kim, Kyungyoon, John V. Carlis, and Daniel F. Keefe (2017). “Comparison techniques utilized in spatial 3D and 4D data visualizations: A survey and future directions”. In: *Computers & Graphics* 67, pp. 138–147. ISSN: 0097-8493. DOI: <https://doi.org/10.1016/j.cag.2017.05.005>. URL: <http://www.sciencedirect.com/science/article/pii/S0097849317300481>.
- Kim, Tai-hoon, Carlos Ramos, and Sabah Mohammed (2017). “Smart City and IoT”. In: *Future Generation Computer Systems* 76, pp. 159–162. ISSN: 0167-739X. DOI: <https://doi.org/10.1016/j.future.2017.03.034>. URL: <http://www.sciencedirect.com/science/article/pii/S0167739X17305253>.
- Langran, G. E. (1989). “Time in geographic information systems”. UMI Order No: GAX90-00269. PhD thesis. University of Washington.
- Liang, Steve, Chih-Yuan Huang, and Tania Khalafbeigi (2016). *OGC SensorThings API: Part 1: Sensing*. Tech. rep. Open Geospatial Consortium.
- Liang, Steve and Tania Khalafbeigi (2019). *OGC SensorThings API Part 2 – Tasking Core*. Tech. rep. Open Geospatial Consortium.
- Liu, Xin et al. (2017). “A State-of-the-Art Review on the Integration of Building Information Modeling (BIM) and Geographic Information System (GIS)”. In: *ISPRS International Journal of Geo-Information* 6.2. ISSN: 2220-9964. DOI: [10.3390/ijgi6020053](https://doi.org/10.3390/ijgi6020053). URL: <http://www.mdpi.com/2220-9964/6/2/53>.
- Murshed, Syed Monjur et al. (2018). “Design and Implementation of a 4D Web Application for Analytical Visualization of Smart City Applications”. In: *ISPRS Interna-*

- tional Journal of Geo-Information* 7.7. ISSN: 2220-9964. DOI: [10.3390/ijgi7070276](https://doi.org/10.3390/ijgi7070276). URL: <http://www.mdpi.com/2220-9964/7/7/276>.
- Na, Arthur and Mark Priest (2007). *Sensor Observation Service*. Tech. rep. Open Geospatial Consortium.
- Oldewurtel, F. et al. (June 2010). “Energy efficient building climate control using Stochastic Model Predictive Control and weather predictions”. In: *Proceedings of the 2010 American Control Conference*, pp. 5100–5105. DOI: [10.1109/ACC.2010.5530680](https://doi.org/10.1109/ACC.2010.5530680).
- Papadopoulos, Antonis V. et al. (Sept. 2018). “Weed mapping in cotton using ground-based sensors and GIS”. In: *Environmental Monitoring and Assessment* 190.10, p. 622. ISSN: 1573-2959. DOI: [10.1007/s10661-018-6991-x](https://doi.org/10.1007/s10661-018-6991-x). URL: <https://doi.org/10.1007/s10661-018-6991-x>.
- Peuquet, Donna (2005). “Time in GIS and geographical databases”. In:
- Pupo, Luis Enrique Rodríguez (2012). “Sensor Data Visualization in Virtual Globes”. diploma thesis. Master of Science in Geospatial Technologies.
- Raper, J.F. (Jan. 1992). “Key 3D Modelling Concepts for Geoscientific Analysis”. In: *Three-dimensional modeling with geoscientific information systems*, pp. 215–232. DOI: [10.1007/978-94-011-2556-7_15](https://doi.org/10.1007/978-94-011-2556-7_15).
- Reed, Carl and Tamrat Belayneh (2017). *OGC Indexed 3d Scene Layer (I3S) and Scene Layer Package Format Specification*. Tech. rep. Open Geospatial Consortium.
- Robin, Alexandre (2011). *OGC SWE Common Data Model Encoding Standard*. Tech. rep. Open Geospatial Consortium.
- Siabato, Willington et al. (2014). “TimeBliography: A Dynamic and Online Bibliography on Temporal GIS”. In: *Transactions in GIS* 18.6, pp. 799–816. DOI: [10.1111/tgis.12080](https://doi.org/10.1111/tgis.12080). eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/tgis.12080>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/tgis.12080>.
- Simonis, Ingo and Johannes Echterhoff (2011). *OGC Sensor Planning Service Implementation Standard*. Tech. rep. Open Geospatial Consortium.
- Song, Yongze et al. (2017). “Trends and Opportunities of BIM-GIS Integration in the Architecture, Engineering and Construction Industry: A Review from a Spatio-Temporal Statistical Perspective”. English. In: *ISPRS International Journal of Geo-Information* 6.12. Copyright - Copyright MDPI AG 2017; Last updated - 2019-03-01, p. 397. URL: <https://search.proquest.com/docview/1988505904?accountid=14965>.

- Steel, Jim and Robin Drogemuller (Feb. 2009). “Model interoperability in building information modelling”. In: *Knowledge Industry Survival Strategy Initiative*. URL: <https://eprints.qut.edu.au/19419/>.
- Trilles, Sergio et al. (2017). “Deployment of an open sensorized platform in a smart city context”. In: *Future Generation Computer Systems* 76, pp. 221–233. ISSN: 0167-739X. DOI: <https://doi.org/10.1016/j.future.2016.11.005>. URL: <http://www.sciencedirect.com/science/article/pii/S0167739X16305519>.
- Tsai, Fuan, Jhe-Syuan Lai, and Yu-Ching Liu (2015). “An alternative open source web-based 3D GIS: Cesium engine environment”. In: *ACRS 2015 - 36th Asian Conference on Remote Sensing: Fostering Resilient Growth in Asia, Proceedings*. Manila.
- Wendel, Jochen et al. (May 2016). “Development of a Web-Browser Based Interface for 3D Data—A Case Study of a Plug-in Free Approach for Visualizing Energy Modelling Results”. In: Springer, pp. 185–205. ISBN: 978-3-319-19601-5. DOI: [10.1007/978-3-319-19602-2_12](https://doi.org/10.1007/978-3-319-19602-2_12).
- Zhu, Wanji et al. (Dec. 2016). “Integration of CityGML and Air Quality Spatio-Temporal Data Series via OGC SOS”. In:

List of Appendices

A	Source codes	70
A.1	horizontalAdjacencyTranslator	70
A.2	verticalAdjacencyTranslator	74
A.3	One room in glTF	76
B	Content of the included DVD	84

Appendix A

Source codes

A.1 horizontalAdjacencyTranslator

Listing A.1: Python function for merging individual outputs of the geodata processing for one floor into a single JSON file.

```
import argparse
2 from csv import reader
import json
4 import sys

6 # Parse command line params
parser = argparse.ArgumentParser(description = 'Creates adjacency
    file feasible for the Heat Transfer Simulation')
8 parser.add_argument('-p', "--properties", action = "store", dest =
    'propertyFileName', required = True, help = 'Name of the CSV
    file with properties (area, height) about each room.')
parser.add_argument('-m', "--mappings", action = "store", dest = '
    mappingFileName', required = True, help = 'Name of the TXT file
    with mappings between IDs used in the adjacency file and th IDs
    which should be used in the output.')
10 parser.add_argument('-f', "--filename", action = "store", dest = '
    adjacencyFileName', required = True, help = 'Name of the (main)
    input TXT file with information about adjacency of individual
    rooms.')
parser.add_argument('-o', "--outputfilename", action = "store",
    dest = 'outputFileName', help = 'Name of the output file which
    will be created. If none is provided, the input file name will
    be used.')
12 parser.add_argument('-r', "--rewrite", action = 'store_true',
```



```

    default = False, dest = 'rw', help = 'Rewrite the output file
    instead of updating it')
args = vars(parser.parse_args())
14 mappingFileName = args['mappingFileName'] if args['mappingFileName',
    ] else "ID_FID_pxID.txt"
propertyFileName = args['propertyFileName'] if args['
    propertyFileName'] else "RoomsAreas.csv"
16 adjacencyFileName = args['adjacencyFileName'] if args['
    adjacencyFileName'] else "RoomsAdjacency.txt"
outputFileName = args['outputFileName'] if args['outputFileName']
    else ""
18 rw = args['rw'] if args['rw'] else False

20 # Search for a room by its ID. Returns room object or None
def getRoomById(id):
22     for room in data['rooms']:
        if room['id'] == id:
24             return room
        return None
26

# FAV-specific function to rename numerical IDs back to their
    normal string form
28 def renameToAlfa(value):
    UX = value[0]
30     val = None
    if int(UX) == 1:
32         val = 'UC'
    elif int(UX) == 2:
34         val = 'UN'
    elif int(UX) == 3:
36         val = 'US'
    elif int(UX) == 4:
38         val = 'UNW'
    else:
40         print("Unknown type of room: {}".format(UX))
    if len(value) == 6:
42         return "".join([val, value[1:4], chr(int(value[4:]))])
    elif len(value) > 6:
44         return "".join([val, value[1:4], chr(int(value[4:6])),
            value[6:]])
    else:
46         return "".join([val, value[1:]])

48 # Recode IDs by using a hash-table

```

```

idTable = {}
50 idTable['-1'] = '-1' # outdoor/ambient
with open(mappingFileName, "r", encoding="utf-8") as fh:
52     csvReader = reader(fh, delimiter=";")
     next(csvReader) #skip heading
54     for row in csvReader:
         idTable[row[0]] = renameToAlfa(row[1])
56
# If there is already a file with adjacencies and the flag is not
  set to rewrite it, read it
58 if len(outputFileName) > 0 and not rw:
     with open(outputFileName, 'r', encoding="utf-8") as fh:
60         data = json.load(fh)
else:
62     data = {}

64 # Adding individual rooms/zones
if not 'rooms' in data:
66     data['rooms'] = []
with open(propertyFileName, 'r', encoding="utf-8") as fh:
68     csvReader = reader(fh)
     next(csvReader)
70     for row in csvReader:
         if not getRoomById(row[0]):
72             data['rooms'].append({
                 'id': row[0],
74                 'volume': round(float(row[1]) * float(row[2]), 2),
                 'height': round(float(row[2]), 2),
76                 'walls': []
             })
78
# Adding outdoor/ambient zone
80 if not 'ambient' in data:
     data['ambient'] = {
82         'constant': True,
         'walls': []
84     }

86 if not 'walls' in data:
     data['walls'] = []
88
# Set auto-counter of walls
90 wallId = max(wall['id'] for wall in data['walls']) if len(data['
    walls']) > 0 else 0

```

```

92 # Adding walls and connecting zones with walls
# Currently, the information is being held dublictly in general "
# walls" array and in "walls" array by each room
94 with open(adjacencyFileName, 'r', encoding="utf-8") as fh:
    csvReader = reader(fh, delimiter=";")
96     next(csvReader) #skip heading
    for row in csvReader:
98         id1 = idTable[row[1]]
        id2 = idTable[row[2]]
100        roomHeight = getRoomById(id1)['height'] if id1 != "-1" else
            getRoomById(id2)['height']
        wallId += 1
102        data['walls'].append({
            'id': wallId,
104            'area': float(row[3].replace(',','.'))*float(
                roomHeight), #l ength*height
            'leftID': id1,
106            'rightID': id2
        })
108        if getRoomById(id1):
            rm1 = getRoomById(id1)
110            rm1['walls'].append(wallId)
        elif id1 == "-1": # because of "-1" ambient room
112            data['ambient']['walls'].append(wallId)
        else:
114            print("Unknown room ID {} cannot be linked with {}".
                format(id1, id2))
        if getRoomById(id2):
116            rm2 = getRoomById(id2)
            rm2['walls'].append(wallId)
118        elif id2 == "-1": # because of "-1" ambient room
            data['ambient']['walls'].append(wallId)
120        else:
            print("Unknown room ID {} cannot be linked with {}".
                format(id2, id1))
122
123 if len(outputFileName) == 0:
124     outputFileName = ".".join( (adjacencyFileName.split('.')[0], "
        json" ) )
    with open(outputFileName, 'w', encoding="utf-8") as out:
126        json.dump(data, out)
    print("File \"{}\" succesfully created".format(outputFileName))

```

A.2 verticalAdjacencyTransaltor

Listing A.2: Python function for merging individual outputs of the geodata processing for one pair of neighbouring floors into a single JSON file.

```
1 import argparse
  from csv import reader
3 import json
  import sys
5
  # Parse command line params
7 parser = argparse.ArgumentParser(description = 'Creates adjacency
  file feasible for the Heat Transfer Simulation')
  parser.add_argument('-f', "--filename", action = "store", dest = '
  adjacencyFileName', required = True, help = 'Name of the (main)
  input CSV file with information about adjacency of individual
  rooms.')
```

```
9 parser.add_argument('-o', "--outputfilename", action = "store",
  dest = 'outputFileName', help = 'Name of the output file which
  will be created. If none is provided, the input file name will
  be used.')
```

```
  parser.add_argument('-r', "--rewrite", action = 'store_true',
  default = False, dest = 'rw', help = 'Rewrite the output file
  instead of updating it')
```

```
11 args = vars(parser.parse_args())
  adjacencyFileName = args['adjacencyFileName'] if args['
  adjacencyFileName'] else "CeilingsAdjacency.csv"
```

```
13 outputFileName = args['outputFileName'] if args['outputFileName']
  else ""
  rw = args['rw'] if args['rw'] else False
15
  # Search for a room by its ID. Returns room object or None
17 def getRoomById(id):
  for room in data['rooms']:
19     if room['id'] == id:
  return room
21 return None
```

```
23 # If there is already a file with adjacencies and the flag is not
  set to rewrite it, read it
  if len(outputFileName) > 0 and not rw:
25     with open(outputFileName, 'r', encoding="utf-8") as fh:
  data = json.load(fh)
```

```

27 else:
    data = {}
29
    # Adding individual rooms/zones
31 if not 'rooms' in data:
    data['rooms'] = []
33
    # Adding outdoor/ambient zone
35 if not 'ambient' in data:
    data['ambient'] = {
37         'constant': True,
         'walls': []
39     }
41
    if not 'walls' in data:
        data['walls'] = []
43
    # Set auto-counter of walls
45 wallId = max(wall['id'] for wall in data['walls']) if len(data['
    walls']) > 0 else 0
47
    # Adding walls and connecting zones with walls
    with open(adjacencyFileName, 'r', encoding="utf-8") as fh:
49         csvReader = reader(fh)
         next(csvReader) #skip heading
51         for row in csvReader:
             id1 = row[0]
53             id2 = row[1]
             if id1 == id2: # room which spans through multiple levels
55                 continue
             wallId += 1
57             data['walls'].append({
                 'id': wallId,
59                 'area': float(row[2]), # area in metres
                 'leftID': id1, # left and right are really arbitrary
                    here
61                 'rightID': id2
             })
63             if getRoomById(id1):
                 rm1 = getRoomById(id1)
65                 rm1['walls'].append(wallId)
             elif id1 == "-1": # because of "-1" ambient room
67                 data['ambient']['walls'].append(wallId)
             else:

```

```

69         print("Unknown room ID {} cannot be linked with {}".
              format(id1, id2))
    if getRoomById(id2):
71         rm2 = getRoomById(id2)
            rm2['walls'].append(wallId)
73     elif id2 == "-1": # because of "-1" ambient room
            data['ambient']['walls'].append(wallId)
75     else:
            print("Unknown room ID {} cannot be linked with {}".
                  format(id2, id1))
77
if len(outputFileName) == 0:
79     outputFileName = ".".join( (adjacencyFileName.split('.')[0], "
        json" ) )
with open(outputFileName, 'w', encoding="utf-8") as out:
81     json.dump(data, out)
print("File \"{}\" successfully created".format(outputFileName))

```

A.3 One room in glTF

Listing A.3: Example of the complete representation of one of the rooms of the building in a glTF format. Only the binary buffers in the end of the files were omitted.

```

{
    "asset": {
        "generator": "COLLADA2GLTF",
        "version": "2.0"
    },
    "scenes": [
        {
            "nodes": [
                0
            ]
        }
    ],
    "scene": 0,
    "nodes": [
        {
            "children": [
                1
            ],
            "matrix": [
                1.0,
                0.0,

```

```

        0.0,
        0.0,
        0.0,
        0.0,
        -1.0,
        0.0,
        0.0,
        1.0,
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        1.0
    ],
    "name": "Z_UP"
},
{
    "mesh": 0,
    "children": [
        3,
        2
    ],
    "name": "red.dae"
},
{
    "mesh": 1
},
{
    "mesh": 2
}
],
"meshes": [
    {
        "primitives": [
            {
                "attributes": {
                    "NORMAL": 1,
                    "POSITION": 2
                },
                "indices": 0,
                "mode": 4,
                "material": 0
            }
        ]
    }
]

```

```

    ],
    "name": "red.dae3"
  },
  {
    "primitives": [
      {
        "attributes": {
          "NORMAL": 4,
          "POSITION": 5
        },
        "indices": 3,
        "mode": 4,
        "material": 1
      }
    ],
    "name": "red.dae5"
  },
  {
    "primitives": [
      {
        "attributes": {
          "NORMAL": 7,
          "POSITION": 8
        },
        "indices": 6,
        "mode": 4,
        "material": 2
      }
    ],
    "name": "red.dae4"
  }
],
"accessors": [
  {
    "bufferView": 0,
    "byteOffset": 0,
    "componentType": 5123,
    "count": 48,
    "max": [
      15
    ],
    "min": [
      0
    ]
  }
],

```



```

    "type": "SCALAR"
  },
  {
    "bufferView": 1,
    "byteOffset": 0,
    "componentType": 5126,
    "count": 16,
    "max": [
      0.0,
      0.0,
      1.0
    ],
    "min": [
      0.0,
      0.0,
      1.0
    ],
    "type": "VEC3"
  },
  {
    "bufferView": 1,
    "byteOffset": 192,
    "componentType": 5126,
    "count": 16,
    "max": [
      3.066779851913452,
      1.9254099130630496,
      4.360000133514404
    ],
    "min": [
      -3.066779851913452,
      -1.9254099130630496,
      0.0
    ],
    "type": "VEC3"
  },
  {
    "bufferView": 0,
    "byteOffset": 96,
    "componentType": 5123,
    "count": 18,
    "max": [
      7
    ],
  },

```

```

    "min": [
        0
    ],
    "type": "SCALAR"
},
{
    "bufferView": 1,
    "byteOffset": 384,
    "componentType": 5126,
    "count": 8,
    "max": [
        0.0,
        0.0,
        1.0
    ],
    "min": [
        0.0,
        0.0,
        1.0
    ],
    "type": "VEC3"
},
{
    "bufferView": 1,
    "byteOffset": 480,
    "componentType": 5126,
    "count": 8,
    "max": [
        3.066779851913452,
        1.9254099130630496,
        4.360000133514404
    ],
    "min": [
        -3.066779851913452,
        -1.9254099130630496,
        4.360000133514404
    ],
    "type": "VEC3"
},
{
    "bufferView": 0,
    "byteOffset": 132,
    "componentType": 5123,
    "count": 18,

```

```

    "max": [
      7
    ],
    "min": [
      0
    ],
    "type": "SCALAR"
  },
  {
    "bufferView": 1,
    "byteOffset": 576,
    "componentType": 5126,
    "count": 8,
    "max": [
      0.0,
      0.0,
      1.0
    ],
    "min": [
      0.0,
      0.0,
      1.0
    ],
    "type": "VEC3"
  },
  {
    "bufferView": 1,
    "byteOffset": 672,
    "componentType": 5126,
    "count": 8,
    "max": [
      3.066779851913452,
      1.9254099130630496,
      0.0
    ],
    "min": [
      -3.066779851913452,
      -1.9254099130630496,
      0.0
    ],
    "type": "VEC3"
  }
],
"materials": [

```

```

    {
      "pbrMetallicRoughness": {
        "baseColorFactor": [
          1.0,
          1.0,
          1.0,
          1.0
        ],
        "metallicFactor": 0.0
      },
      "name": "red.dae3effect"
    },
    {
      "pbrMetallicRoughness": {
        "baseColorFactor": [
          1.0,
          1.0,
          1.0,
          1.0
        ],
        "metallicFactor": 0.0
      },
      "name": "red.dae5effect"
    },
    {
      "pbrMetallicRoughness": {
        "baseColorFactor": [
          1.0,
          1.0,
          1.0,
          1.0
        ],
        "metallicFactor": 0.0
      },
      "name": "red.dae4effect"
    }
  ],
  "bufferViews": [
    {
      "buffer": 0,
      "byteOffset": 768,
      "byteLength": 168,
      "target": 34963
    },
  ],

```

```
{
  "buffer": 0,
  "byteOffset": 0,
  "byteLength": 768,
  "byteStride": 12,
  "target": 34962
},
"buffers": [
  {
    "byteLength": 936,
    "uri": "data:application/octet-stream;base64,(...lots of
      binary data...)"
  }
]
}
```

Appendix B

Content of the included DVD

- dataProcessing – directory with scripts and data relating to chapter 3.2
 - outputs/ – directory with final outputs of the data processing
 - partialResults/ – directory with some important partial results for the processing of the FAV building data
 - shape2lumped/ – directory with scripts used for simplification of the data processing
- latex/ – directory with a source files of the thesis in L^AT_EX format
 - media/ – directory with the figures included in the text
 - bib/ – directory with the bibliography records for the text
 - macura_dp.tex – source L^AT_EX document
 - zadani_jm.pdf – official thesis assignment
- buildingSimulationModel/ – directory with the complete source codes of the *BuildingModel* program
- observation4d/ – directory with the complete source codes for the final 4D visualisation
 - models/ – directory with the rooms of FAV building converted to glTF
 - observations/ – directory with the output of the heat transfer simulation for the FAV building
- spatialData – directory with input spatial data of the FAV building
- macura_dp.pdf – text of the master thesis