FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Non-destructive workflow for repeatable creation of virtual prototypes from CAD data
Student:	Gabriela Havranová
Supervisor:	Ing. Jan Buriánek
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Computer Graphics
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Virtual prototyping is currently a popular method used for the inspection and review of 3D CAD engineering models. In the case the model being reviewed is not feasible and needs to be edited, a considerable amount of manual work has to be repeated before the model could be visualized again. The aim of this Bachelor thesis is to design such a workflow that would eliminate the need to repeat the work manually and reduce the production time.

1. Analyse available solutions.
2. Describe the current industry workflow used for the creation of virtual prototypes and propose an improved solution, such that it would require less or no repetition of manual editing of the reviewed model.
3. Implement the software needed for the designed workflow.
4. Test the software and create documentation.
5. Evaluate the results and describe possible extensions.

Electronically approved by Ing. Radek Richtr, Ph.D. on 1 November 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Non-destructive workflow for repeatable creation of virtual prototypes from CAD data

Gabriela Havranová

Department of software engineering
Supervisor: Ing. Jan Buriánek

May 11, 2022

Acknowledgements

First of all, I would like to thank my supervisor Ing. Jan Buriánek for his guidance and insightful suggestions to this thesis. Another huge thank you belongs to my family and friends for their sincere support and encouragement during my whole studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 11, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Havranová, Gabriela. *Non-destructive workflow for repeatable creation of virtual prototypes from CAD data.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Virtuálna prototypizácia je pred vytvorením fyzického modelu jednou z možných alternatív na otestovanie návrhu prototypu. Ak sa počas používateľského testovania ukáže, že je návrh nevhodný, všetky vykonané zmeny na virtuálnom prototype sa zahodia a nová verzia prototypu sa vytvára zo zdrojov odznova. Táto práca predstavuje návrh a softvérovú implementáciu procesu na opakovanú tvorbu virtuálneho prototypu, ktorý opätovne používa zmeny z predchádzajúcich verzií prototypu a aplikuje ich na novú verziu. Implementovaný nástroj automaticky detekuje zodpovedajúce časti verzií prototypov a aktualizuje geometriu zdrojovej verzie modelu, čím šetrí čas potrebný na opakovanú výrobu prototypu.

Kľúčové slová virtuálna prototypizácia, virtuálne CAD modely, proces vizualizácie, virtuálny dizajn, digitálne dvojča

Abstract

Virtual prototyping is one of the feasible alternatives for testing a prototype design before committing to creating the physical model. When the user testing indicates the prototype design was improper, all the post-processing changes made on the virtual prototype are discarded, and the improved prototype version is created from the scratch. This thesis proposes a design and software implementation for a non-destructive workflow for repeatable virtual prototype creation, that reuses changes from previous prototype versions and applies them to a new version. Implemented tool automatically detects corresponding sub-parts of the prototype versions and updates the geometry of the source model version, therefore saving required time for the repeated prototype production.

Keywords virtual prototyping, virtual CAD models, visualizing workflow, virtual design, digital twin

Contents

List of Figures

Introduction

In the process of product development, before committing to making a physical prototype, the method of virtual prototyping is often used to validate the design. For engineering purposes, the mathematical description of 3D models is considered the industry standard for its precision, and it is mostly used in modern CAD software modelers. However, CAD formats, in general, are computationally difficult to manipulate with outside CAD-aimed software, and computer graphic cards are optimized for rendering triangular meshes instead of parametric equations, therefore the model needs to be tessellated before it can be virtually visualized.

Tessellation is a process of approximation of mathematically described model surfaces with triangular mesh. Unless there are no curved surfaces in the model, the tessellated mesh can never be as precise as the original model. The precision of tessellation is defined by the chosen level of detail. The higher the level of detail, the more precise is the mesh approximation. On the other hand, if the chosen level of detail is too high, the number of vertices used for model description can drastically grow, which leads to performance reduction.

The resulting tessellation can considerably vary in visual quality depending on the tool used, the quality of the original CAD model, and other factors. These imperfections of the mesh are typically manually edited in 3D editors, e.g., Blender. Mesh optimizations, texture additions, and other minor changes are being done in this process too, to enhance the visual output of the resulting model. This process would be later referred to as mesh post-processing. After the post-processing, the model is ready to be used as a virtual prototype in any preferred software.

The usual purpose of virtual prototyping is to review the model and edit its possible shortcomings. In the case some adjusting of the model is needed, it needs to be done in the original CAD model, as it is industry standard for manufacturing and construction. After re-adjusting the CAD model, the prototype needs to be reviewed again, firstly tessellated mesh is becoming useless and re-tessellation is necessary. This leads to the main goal of this

bachelor's thesis. To be able to visualize the newly tessellated mesh, the post-processing edits, which were already done once, need to be repeated. The main aim of this thesis is to eliminate the need to reproduce the same edits whenever possible. Further manual editing would be needed on the adjusted parts of the CAD model only, which implies a reduction of the total time needed for prototype creation. Reducing the production time of the prototype creation naturally lowers the total cost of the project which is one of the biggest benefits of this thesis.

The next chapter *Analysis* starts with the requisite theoretical background for the understanding scope of this thesis. Then I will continue with a description of the currently used workflow for virtual prototyping creation. I was not able to find any other thesis or existing software solving this issue, however, there are some tools already resolving similar problems, which I will mention at the end of the chapter. In the second chapter, I will describe my proposal for an improved workflow together with the design of the software needed for this workflow. The third and fourth chapters will be dedicated to the implementation and testing of the software. At the end of the thesis, I am going to sum up the results of this thesis and add the list of a bibliography I have used.

At the end of this introduction, I would like to emphasize that there already are automatic tools allowing to visualize 3D CAD models directly in virtual reality, without any further effort needed. The difference is that this thesis is aiming at more complex sceneries, where we expect the final product to be visually appealing, so the post-processing part of the workflow is a must.

Goals and subtasks

The main goal of this Bachelor thesis is to improve the current virtual prototyping workflow so that the repeated visualizing of the given model will take less time (on average). This includes the design and implementation of software needed for this workflow to work.

Subtasks

Analysis of available solutions on the market
Analyse relevant solutions already available on the market and describe their advantages and drawbacks.

Description of current industry workflow
Explain what exactly virtual prototyping is, and describe the current industry workflow used for the creation of virtual prototypes together with its drawbacks.

Proposal for an improved workflow
Propose an improvement in a workflow so that the repetitive model post-processing will not be needed. Choose the technologies for the implementation of the given software and explain the choice.

Implementation and testing
Implement the software needed and test it on a sample model. Create brief user documentation on how to install and use the tool.

Chapter 1

Analysis

In this chapter, I will explain the requisite theoretical background for understanding the subject of this thesis. I will address the common 3D data structures, how they are implemented and what impact it has on their practical usage. Then, I will describe the current industry workflow used to visualize 3D CAD data. The end of this chapter will be dedicated to a list of available solutions on the market together with my conclusion on their usage.

1.1 3D data structures

To visualize any 3D models, we have to store them first. It may sound like a matter of course, but there are a few different approaches for 3D data representation. I have logically divided available formats into two main groups: *Precise formats* and *Approximated formats*.

1.1.1 Precise - CAD formats

This group of 3D file formats is mainly used in technical fields and industries such as automobile, engineering, building, and architecture [1]. They use mathematical equations for the model representation which implies their main advantage, accuracy. I will refer to this group of precise formats as CAD formats because of their broad usage in computer-aided design software. This group includes:

- **Constructive solid geometry (CSG)**
  This type of representation uses boolean operators and transformations to create complex models from a set of simple primitives. An object is stored as a tree with operators at the internal nodes and simple primitives at the leaves. [2]
- **Boundary representation (BREP)**
  Object in BREP is described by its surface boundaries: vertices, edges,

and faces. [2] These elements define the boundary between interior and exterior points. Boundary representation of the model can be either an approximation or precise description, which depends on whether chosen format supports precise curved surfaces (bicubic surfaces, NURBS surfaces...) or the curved surfaces of the model have to be approximated with polygon mesh.

1.1.2 Approximated formats

In this group, I consider all 3D formats which do not support precise curved surface representation. These surfaces have to be described in another way, e.g. they can be approximated with polygon mesh. The 3D object itself can be either stored as a set of vertices and faces which create a continuous mesh describing the whole model or as a BREP. Approximation of curved surfaces with polygon mesh naturally creates deviations from the original model. With advanced texturing, this is usually not a problem for artistic purposes.

1.2 From mathematical model to graphics card

Modern graphic cards are designed and optimized for rendering a polygonal mesh, rather than B-Rep models. Visualizing solid models currently relies on the tessellation of the models before passing them onto the graphics card (GPU). [3] Direct rendering on the GPU, without a tessellation pre-processing, is currently not widely implemented. Despite its pixel-level quality, smaller memory requirements, and less pre-processing, one of the main problems with existing solutions for rendering solid models directly, is the appearance of crack or gap artifacts between faces due to the approximations of the trimming curves. This prevents the usage of direct GPU rendering in CAD systems. [4]

1.2.1 Tessellation

Tessellation, in computer graphics generally, refers to a process that divides a polygon primitive into smaller structures suitable for rendering. [5] The tessellation of Computer-Aided Design models aims to generate a discrete mesh that approximates the model with simple discrete elements. For a surface mesh are often used triangles or quads. [6] [7] In this thesis we will focus especially on triangular surface mesh generation because of its simplicity and flexibility.

Although numerous meshing methods are available, automated generation of high-quality meshes remains a challenge. [6] Even with modern commercial software (e.g., Ansys and Hypermesh) and open-source packages (e.g., Netgen/NGSolve), generating correct and satisfying meshes is still a time-consuming process that involves an excessive amount of human effort. As the

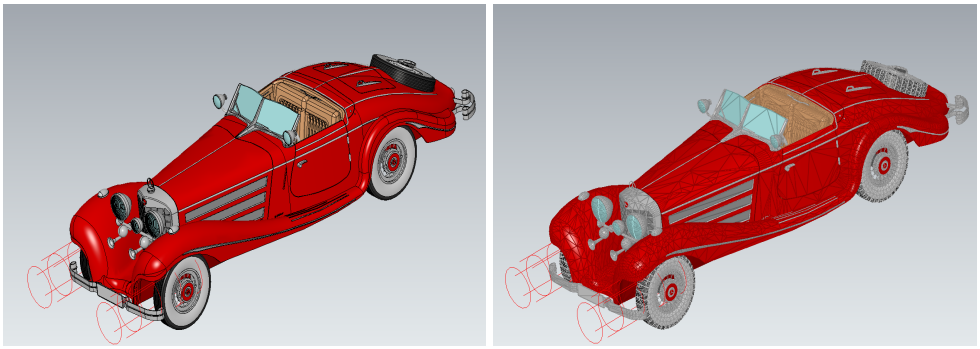

Figure 1.1: CAD model of a car [8] (left) and its triangular tessellation created by CAD Assistant [9] (right)

complexity of the constructed CAD products increases, existing methods cannot always achieve accurate output due to incorrect, degenerate, or ambiguous geometric designs in the CAD system. Even post-processing with mesh repair algorithms experiences difficulties in dealing with these problems. Moreover, some CAD models can also contain many small-scale features within large ones. Preserving such small features in detail creates highly dense mesh locally, which can lead to performance issues. [7]

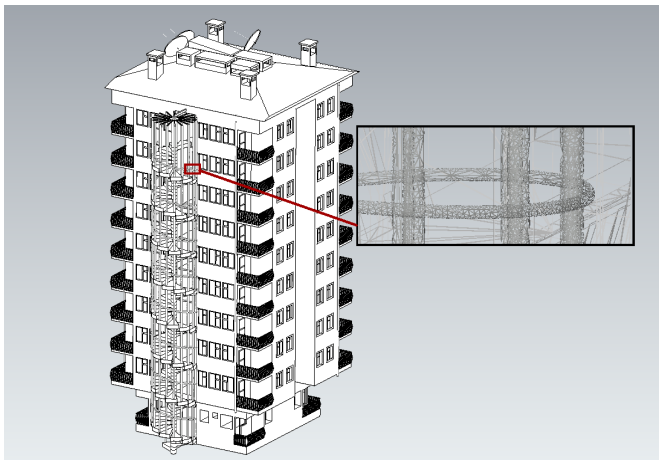

Figure 1.2: *Common issues with mesh generated by tessallating CAD models* - too dense mesh can lead to performance problems in large scenes. Original 3D model from this [10] source, tessellated with CAD Assistant. [9]

1.2.2 Mesh healing

Unfortunately, not all source CAD models are made keeping conventions in mind. Slivers, cross-overs, surfaces with multiple unnecessary patches, super-

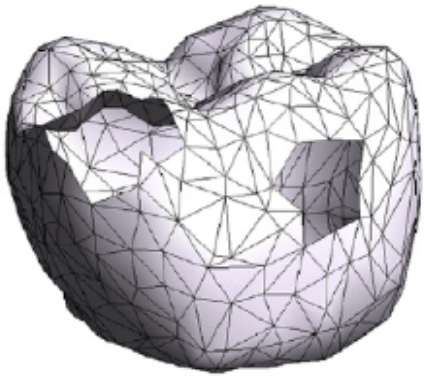

Figure 1.3: *Common issues with mesh generated by tessallating CAD models* - even after shape healing algorithms the mesh can contain gaps. [11]

small model entities, and many other issues that are encountered are often being found in the sourcing CAD model, making the meshing process complicated, and often the results are not as perfect as they are expected to be. [3] Resulting tessellated mesh can be very uneven, overlapping or even gaps and tears occur. [11]

1.2.3 Mesh decimation

Tessellation of source CAD model with lots of details and curves can produce too dense mesh for visualizing. If the response rate of the virtual scene is low, it is feasible to reduce the complexity of the tessellated model by making the mesh more sparse. Instead of manual time-consuming editing, a mesh decimating algorithm can be used to automatically reduce polygon count in the model. The fundamental goal of the decimation algorithm is to reduce the total number of triangles in a triangle mesh while preserving as much accuracy to the original model as possible. [12] [13]

1.2.4 Mesh post-processing

Before the tessellated model is ready to be visualized in virtual reality, it has to be processed first to receive visually appealing results. As described in section tessellation, the process can be fault-prone and result in a defective or unevenly dense mesh. In this part of the process, the graphical engineer fixes and optimizes the mesh to have the least amount of triangles possible (to retain a high frame rate) while maintaining satisfactory visual quality.

Depending on the audience of the virtual review, the model can be added to a more complex scene (e.g. shopping mall can be inserted into the scope of a city district, even if it is not part of the construction). Advanced texturing

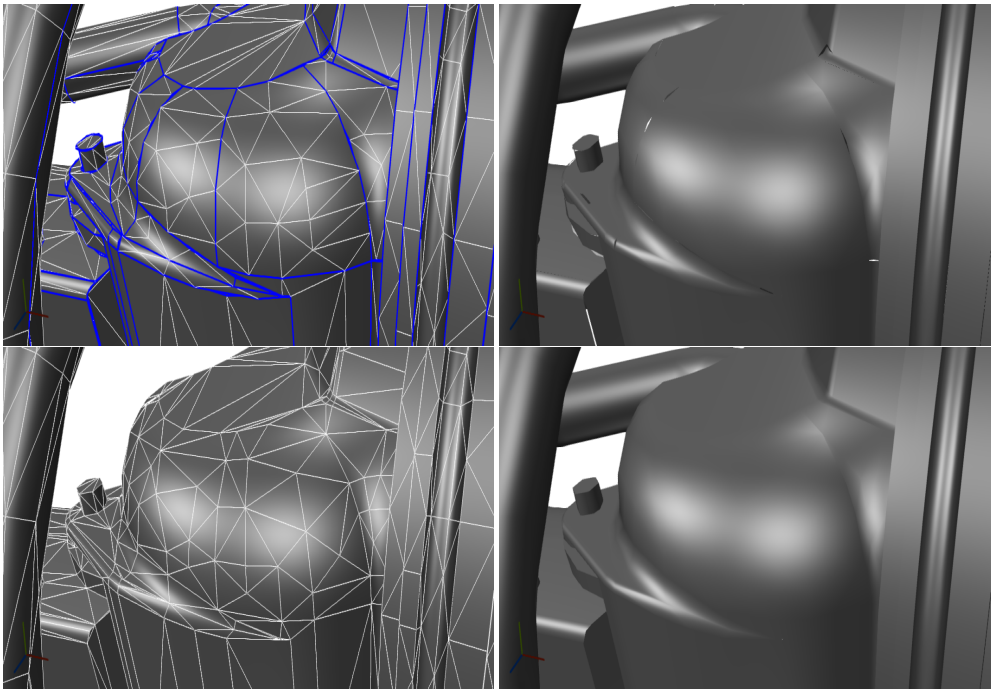

Figure 1.4: Tessellated model with cracked mesh, blue lines represent disconnected edges (upper left). Visible cracks in the shaded version of the same model (upper right). Same tessellated model after applying mesh-healing algorithm (lower left). Shaded repaired model - without visual artifacts (lower right). [3]

and minor details can be added as well. This process can be heavily time-consuming, depending on the exact model and conditions of production.

1.3 Data comparison

While comparing files byte-by-byte can be a working strategy for smaller data sets, comparing large amounts of data and searching for identical sub-parts would be ineffective and time-consuming. In the context of this work, I will describe the strategy to compare tessellated models. Considering the model as a set of geometries composed of vertices in 3D space, we want to find the identical subparts in an effective and fast way. We consider two models (or parts of the model) as identical when all the vertices, normals, and triangles are the same.

1.3.1 Briefly on hash functions

A cryptographic hash function is a mathematical algorithm that maps data of arbitrary size to a bit array of a fixed size - hash value. It is a one-way function, that is, a function for which it is practically impossible to reverse the computation and obtain the input value from the given output. In the ideal case, the only way to find a message that produces a given hash is to attempt a brute-force search of possible inputs to see if they produce a match. Cryptographic hash functions are a basic tool of modern cryptography. A hash function must be deterministic, meaning that the same message always results in the same hash. As a reference for this summary about hash functions was used this [14] article.

1.3.2 Compare-by-hash

Apart from cryptography use, hash functions are helpful in a technique called Compare-by-hash. It is a well-known technique for testing two files for equality. The technique utilizes a cryptographic hash function, such as SHA1 or MD5, to compare the files. [15] Rather than comparing the files byte-by-byte, we compare their hashes instead. If the hashes differ, then the files are certainly different; if the hashes agree, then the files are almost certainly the same (apart from the unlikely event of collision).

1.4 Virtual reality

Virtual reality (VR) simulates the virtual environment for the user to experience a computer-generated world as if it was real, producing a sense of presence, usually by using a VR headset. [16]

Augumented reality (AR) is an experience where designers enhance parts of the physical world with computer-generated input. Designers create inputs—ranging from sound to video, to graphics or GPS overlays in digital content which respond in real-time to changes in the user's environment, typically movement. [17] Digital elements in AR typically have limited interactivity with the real-world environment.

Mixed reality (MR) refers to the blending of the virtual and physical world. It is similar to AR, which simply overlays digital content onto a camera feed of the physical space, but includes an additional understanding of the three-dimensional environment and objects in it. [18] This enables virtual objects to appear to be both in front of and behind physical objects in the space, or appear to interact with them.

1.4.1 Extended Reality (XR)

XR is an emerging umbrella term for all the currently used immersive technologies — augmented reality, virtual reality, and mixed reality and those that will be created as well. All immersive technologies extend the reality we experience by either blending the virtual and real worlds or by creating a fully artificial experience. [19]

In this thesis, I will mostly use the term virtual reality. However, it includes every possible way of prototype virtualization.

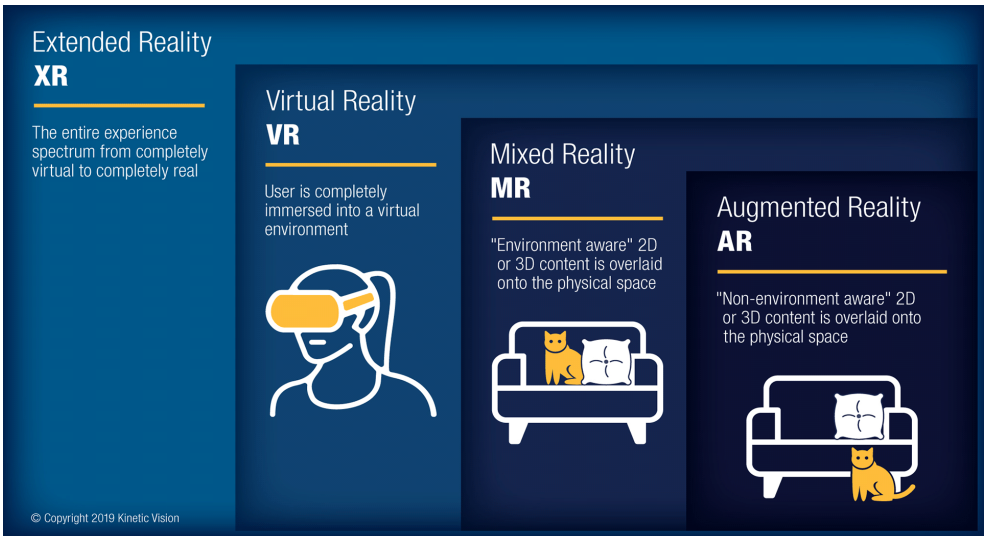

Figure 1.5: Distinction of virtual technologies [18]

1.5 Virtual prototyping (VP)

Considering common principles of human-centered design, the end-user and other stakeholders should be involved in the engineering and design process. [20] Virtual prototyping is an eligible way to get professionals and decision-makers involved by allowing them to interact within the virtual environment rather than simply evaluate the design and product features on a 2D desktop monitor.

VP is the testing and evaluation of specific characteristics of a product or a manufacturing process with the use of a digital model called a virtual prototype in virtual reality. Virtual prototyping aims to detection of faults that can be detected in a compressed time frame before great expenditures are committed. This significantly reduces the number of physical iterations

and thereby the associated manufacturing overheads that lead to faster and cost-effective product development. [21]

In the context of this thesis, I have focused on virtual prototyping and the validation of product design. This includes various analyzes regarding design validation, such as functional testing, form-and-fit testing, or ergonomic testing, which are performed on a model in the virtual environment.

1.5.1 Virtual prototyping workflow

Even though the process of virtual prototype production is not universal, analysis of a few existing workflows [20] [21] [22], all of them have a similar structure as depicted in a Figure 1.6.

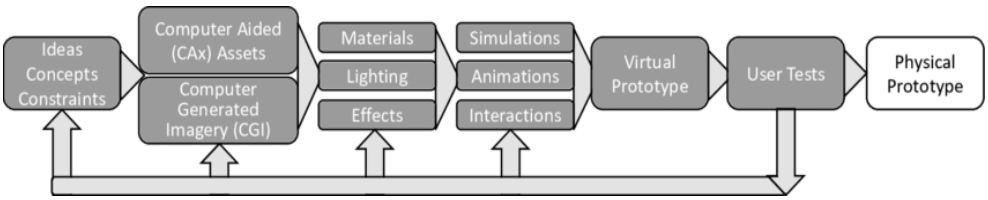

Figure 1.6: Workflow for virtual prototype creation [20]

The process starts with an engineering model prototype (CAD model), where all requirements and ideas are processed by an engineer or architect. Then the model is passed to the graphical designer who adds materials, lighting, and other effects to the model to achieve visually appealing results. Finally, the model is then passed to the virtual environment of choice and user tests are being performed. In the case the design was not feasible, the source model is reprocessed and the workflow starts over again. The interaction of different actors is depicted in a Figure 1.7

1.6 Available solutions

This section is dedicated to a summary of existing solutions available on the market and all software that I found relevant to this thesis. I will use this section in the next chapter *Design of the solution* as a reference.

1.6.1 3D modelling software

3D modelling software is a program used for the creation of a digital representation of any three-dimensional object. Generally, both engineering and artistic-aimed programs are considered 3D modellers. As later shown, the improved workflow will be independent of the source CAD modeller used.

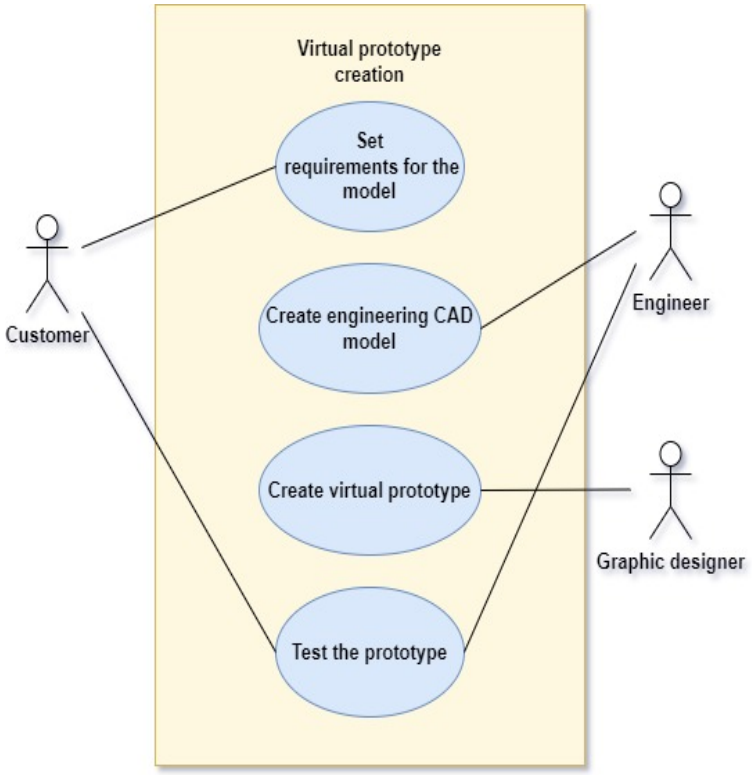

Figure 1.7: Usecase diagram for virtual prototype creation

Therefore I will refrain from listing the CAD modellers and I will focus on the ones oriented toward artistic production.

1.6.1.1 Autodesk Maya

Autodesk Maya is a professional application for creating 3D graphics in digital media. It is one of the most popular programs for creating animated films, 3D film effects, video advertising, the television industry, or creating computer games. [23] Maya integrates 3D modeling, animation, visual effects, and rendering. Creative tools for character creation are included, as well.

Maya is based on an open architecture. Thus, all operations can be scripted using its application programming interface (API) or directly in one of the built-in scripting languages, Maya Embedded Language (MEL) or Python. On the other hand, Maya is not free and the license can be costly for smaller businesses or individuals.

1.6.1.2 Blender

Blender is the free and open-source 3D creation suite. It supports the entirety of the 3D pipeline—modeling, rigging, animation, simulation, rendering, and even video editing and game creation. Blender offers API for Python scripting to customize the application and write specialized tools as well. [24] Blender is a community-driven project under the GNU General Public License, making it free to download and use.

1.6.1.3 3ds Max

3ds Max is another solution from Autodesk, used for architectural and design visualization and animation. It offers interactive environments, fast rendering, and virtual reality (VR) tools. An extensive library of additional plug-in applications from other developers is available, as well. 3ds Max has its own MAXScript language, which is specifically designed to complement 3ds Max and is used for extending the software's functionality. [25]

1.6.2 Available solutions for virtualizing CAD data

There are multiple similar solutions available, however, after closer examination and testing I have found that none of them is actually offering satisfactory results and in every case, at least some additional work in other 3D editor is needed, which again leads to work repetition and unnecessary time expenses. Let's focus on the three following, which I have found the most useful.

1.6.2.1 Unreal Datasmith

"Datasmith is a collection of tools and plugins that bring entire preconstructed scenes and complex assets created in a variety of industry-standard design applications into Unreal Engine" [26]

Unreal's tactic is based on omitting the middleware software and offers the option to load CAD models directly into their visualization software where tessellation and automatic mesh optimization are taking place. Results are seemingly promising. However, Unreal is still not a full-featured 3D geometry editor, rather than visualizing engine. If further post-processing is not needed, Datasmith offers a reasonably good solution. In the other case, it still does not resolve the problem of post-processing repetition. As of spring 2022, Unreal's "Creators license" is free of charge, and you can use it for personal and free projects. If you want to sell your resulting models, you would have to use another licensing plan and pay royalties to Unreal, which may be another downside for some users.

1.6.2.2 Pixyz Studio

"Pixyz Studio is a unique 3D data preparation tool providing the best-in-class Tessellator, enabling the transformation of CAD data from industry-leading solutions (Catia, NX, SolidWorks...) into the lightweight, optimized meshes." [27]

Pixyz Studio is used for the import and tessellation of 3D CAD models and optimization of the generated mesh. The resulting model is almost ready to be used in visualizing software. Almost, because again, it does not solve the problem, when source geometry changes – every change on the resulting model has to be repeated again. Studio claims that you can semi-automate these processes with python scripting, however, the user would have to customize the script for every particular model which leads to additional work and time expenses. Pixyz studio seems to be easy to use, has a simple interface, creates high-quality tessellation, and provides useful semi-automatic mesh decimation tools. A license can be costly (around 1800 € per year, tax excluded).

1.6.2.3 Okino polytrans

Again, mainly tessellation software does not offer the option to further edit the details of the model & advanced texturing. Cheaper than piXYZ studio (around 500 $ per license plus costs of additional packages).

1.6.3 Conclusion of available solutions

After considering all the options above, none of them offers the option to further process the generated mesh from the source CAD model manually. The algorithms used are still not advanced enough to rely on automatic mesh generation exclusively. Furthermore, even in the case, the generated mesh would be perfect, in most cases, the virtual prototype is inserted into a more complex scene to outline the surroundings and this function is not offered by any of the mentioned tools. Thus, I consider creating a new tool, enabling the user to post-process the prototype, as reasonable and supposedly beneficial.

Chapter 2

Design of the solution

In this chapter, I will describe my approach to solving the problem of repetitive model post-processing and propose an improved workflow for repeated virtual prototype creation.

2.1 Repetition of post-processing in detail

As a conclusion of the currently used virtual prototyping workflow, described in Chapter 1, I have created an activity diagram (Figure 2.1) of virtual prototype creation, aimed especially at visualizing CAD data. Red lines which form a loop emphasize the repetitive process of prototype creation, if the user testing or model inspection indicated that the prototype design was not eligible. In the current workflow, whenever the prototype is not feasible, all the changes made by the 3D graphic designer are discarded, the model is edited in the source CAD file and a new version of the virtual prototype is created from the scratch.

2.2 Proposed change in workflow

The main problem of the typical workflow is that even when the changes on the original CAD model are minimal, mesh and scene changes made in the resulting virtual scene are not reused and have to be made all over again. Therefore is natural to try to come up with some way to reapply these changes and save the time needed for repeated virtual prototype production. Figure 2.2 represents the proposed change in the current industry workflow. For the most part, it is identical to Figure 2.1. In the first iteration of the virtual prototype creation, the process remains exactly the same. The change occurs in the second and every next iteration of the virtual prototype creation. In the case that the source model was not feasible and had to be changed, the graphical designer receives the new CAD model version which they tessellate.

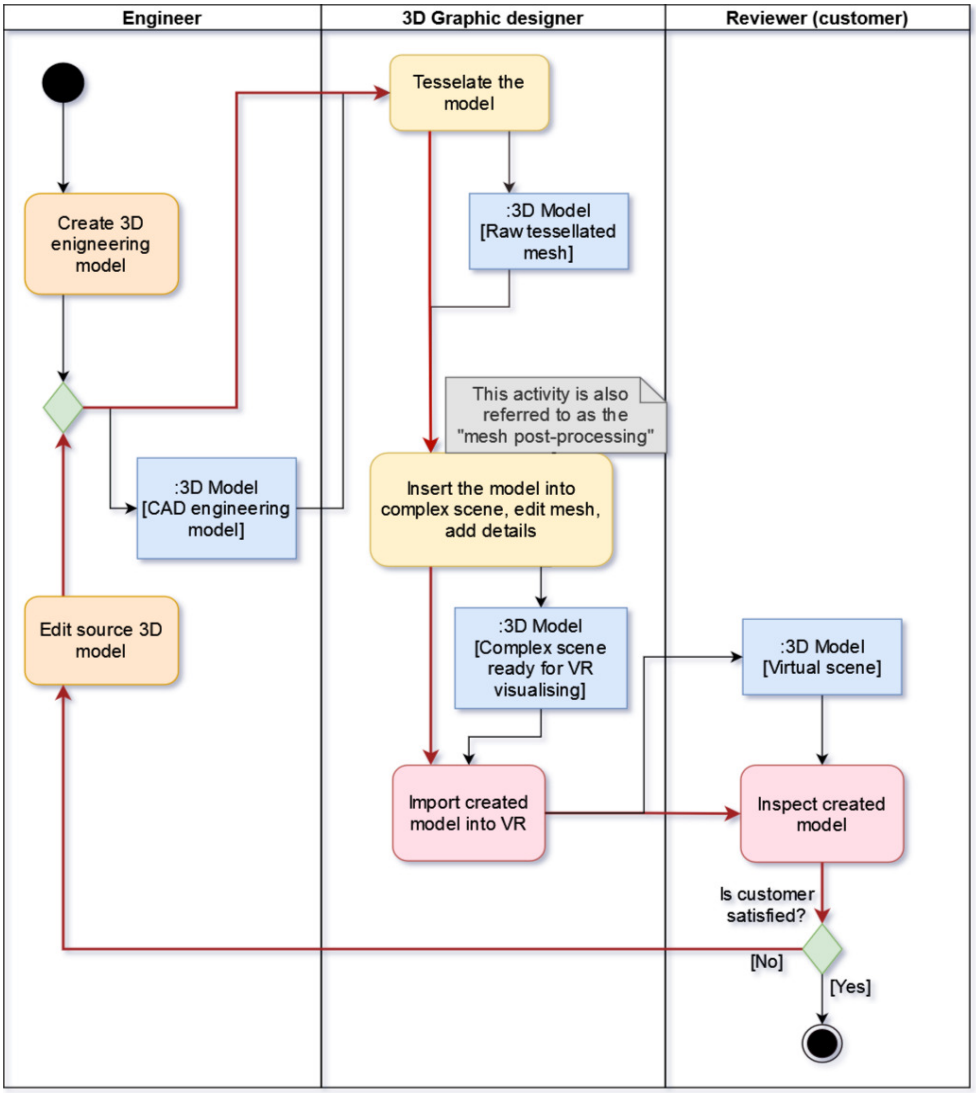

Figure 2.1: Activity diagram for currently used workflow

Now, instead of doing every change on the tessellated model again, the new geometry is only merged into the existing model, while keeping most of the already made changes. This process basically costs no additional time and can save up to hours of repetitive man-work. In the worst-case scenarios, when either the whole model changes, or the tessellated mesh is not divided into sub-parts and the model this method will not save any time, but it will not add any overhead either.

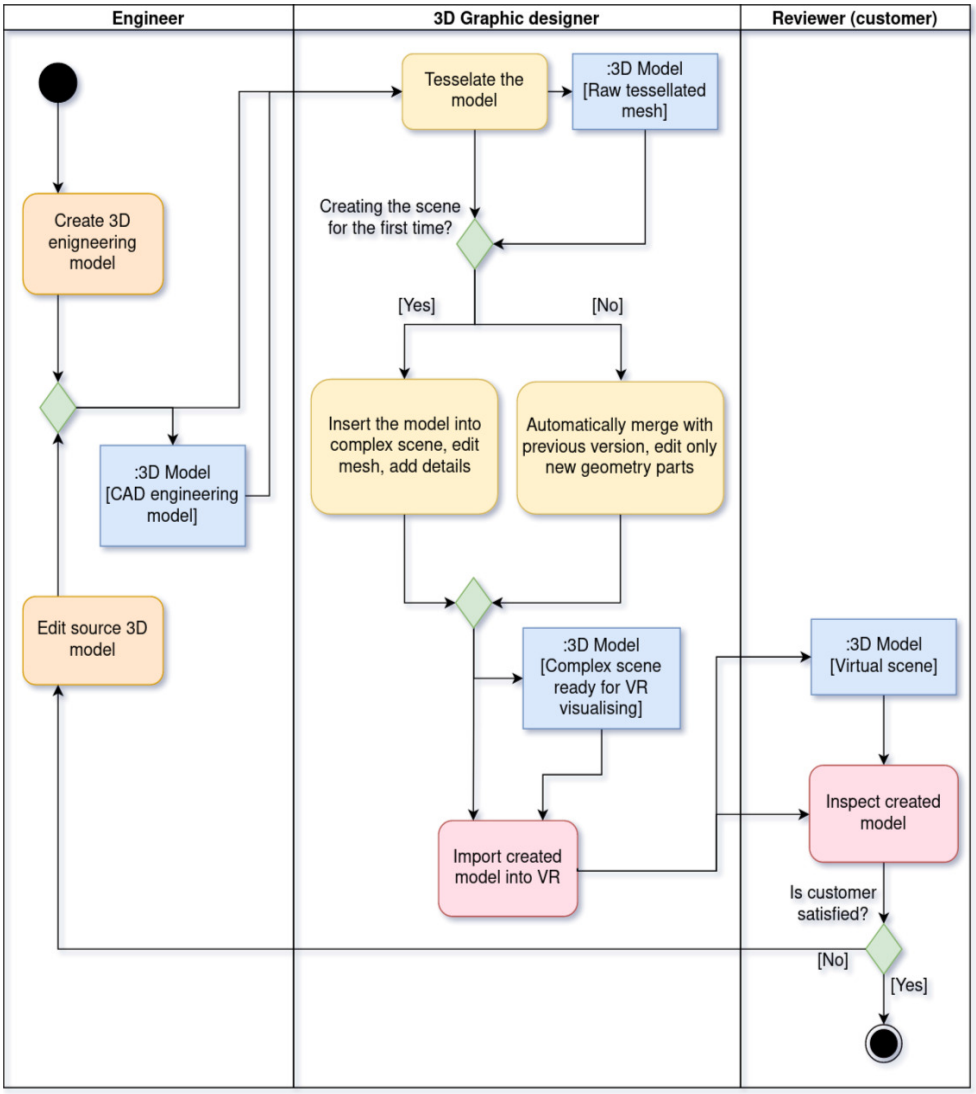

Figure 2.2: Activity diagram for the proposed workflow

2.3 Tool for reapplying changes

My aim is to create such a tool, that would reapply once made model changes. At first, I intended to make use of Blender's Python API (mentioned in the previous chapter). This API enables the user to replicate any changes made in the classic graphical user interface and 3D scene editor. My idea was to save a log file with all changes applied to the first version of the model and then in the second iteration just call all these methods again on the new model version. The problem with this approach was that not every change was logged to the

console so I could not automatically save the exact changes and even in cases when the logging of the commands was successful, even the slightest changes on the model made most of the called methods not applicable anymore. After a few days of playing with this approach, I had to give up this idea.

My second proof-of-concept was more successful. Instead of reapplying the commands, the idea is to load the second tessellated model version in the first version of the resulting virtual scene and move it to the same location in the scene as the first version. Then the script compares respective sub-parts of the models. If the geometry of the given sub-part match exactly, the first version which may contain more data is saved and the new is marked to delete. The remaining sub-parts are tested for similarity. The algorithm tries to calculate, whether the two sub-parts represent the same visual geometry only with one having decimated mesh, or the sub-parts have most of the geometry same and only part of it was changed. If the algorithm determines the former, it means that the user does not wish to update this exact part and the new un-decimated version is marked to delete. In the case of the latter, geometry data of the old sub-part version is replaced by new vertex data, while still keeping the rest of the sub-part attributes e.g. textures, model hierarchy in the scene... If everything fails, all unmatched sub-parts from the previous version of the model are marked to delete and all unmatched sub-parts from the new model version are marked to keep. As the last step before applying the changes and deleting all marked sub-parts, the user can edit these suggestions made by the algorithm and keep sub-parts that are still useful. Moreover, the user can even mark two sub-parts as a "tuple" and those parts will be considered similar and dealt with as explained before. Finally, when everything is ready, the user will apply the changes by clicking on a single button that deletes all unused sub-parts from the scene.

2.4 Comparison of the sub-parts

Theoretically, the order of the vertices in tessellated versions of the same model sub-parts can differ, making it impossible to compare the sub-parts as files. We have to be able to match those tessellations independently on the vertex order. I have inspired by the method compare-by-hash, mentioned in Chapter 1, but instead of using the cryptographic hashing function which would produce different hashes for the same models with different vertex order, I use my own sum formula, which multiplies every vertex's coordinates with large prime numbers and adds them up. At the end of the computation, it adds large multiples of the largest coordinates occurring in the model. This makes hash collisions even less probable. The hash formula ensures that the output is deterministic - the same input geometry will result in the same hash value and even though I am not going to compute the exact probability of the hash collision, by working with large amounts of input vertices the chance

of creating the same hash from different inputs is relatively low. Even in the case of collision, the hashes are only used for fast matching of the same geometry. In the case of a match, also the dimensions of the sub-parts together with other attributes are compared, and only if all of them match, the sub-parts are declared as corresponding. I am aware that making-up own hashing function would be insane in cryptographic and security context, however, for the visual and graphical purposes the function worked reliably, without finding any collision so far.

2.5 Chosen technologies

I have considered multiple options when choosing the suitable technologies for the purpose of this thesis. It was tempting to write a standalone application, with a beautiful, user-friendly GUI which would load, parse and merge the models by itself. However, on the one hand, it would be another piece of software that would a graphical engineer have to use during the virtual prototype creation. On the other, why write complicated and extensive code just to write some small subset of what the current market already offers for free. These are the main reasons why I decided to use Blender and its Python scripting API. I do not have to care about model loading itself, creating GUI from scratch, complicated overhead, 3rd party libraries, and their support. Blender already supports the most common 3D file formats and allows me to directly access internal data structures and manipulate the models with ease. Moreover, changes made to the model can be immediately seen in the 3D scene window, where the accuracy of the model merging algorithm can be checked and adjusted in case of need.

From the other 3D graphic editors I was also considering using Autodesk's Maya. As described in previous Chapter 2 - Analysis, Maya is one of the most popular editors on the current market which also offers extensive Maya API to work with. The main reason, why I have chosen Blender, is simple - Blender is free to download and use, while Maya license can be costly for individuals and small businesses. I would like to support the graphical content creators community and create something which everybody can use. Moreover, Maya supports Python scripting as well. In case of need, only replacing Blender API calls would have to be replaced by corresponding Maya API calls, which are in my opinion structurally similar. Thus, it would not cost much work to make the tool work for Maya too and I consider it as one of the possible extensions of this thesis.

2.6 GUI design

Because the tool should be as automatic as possible, the user interface will be very simple, containing only a few buttons. Blender offers a few different

ways to implement UI for custom tools. I have chosen to create it as a side panel inside the Scene Properties with the name *Model merging tool* because I have found it the most appropriate for this purpose. GUI mockup can be seen in Figure 2.3.

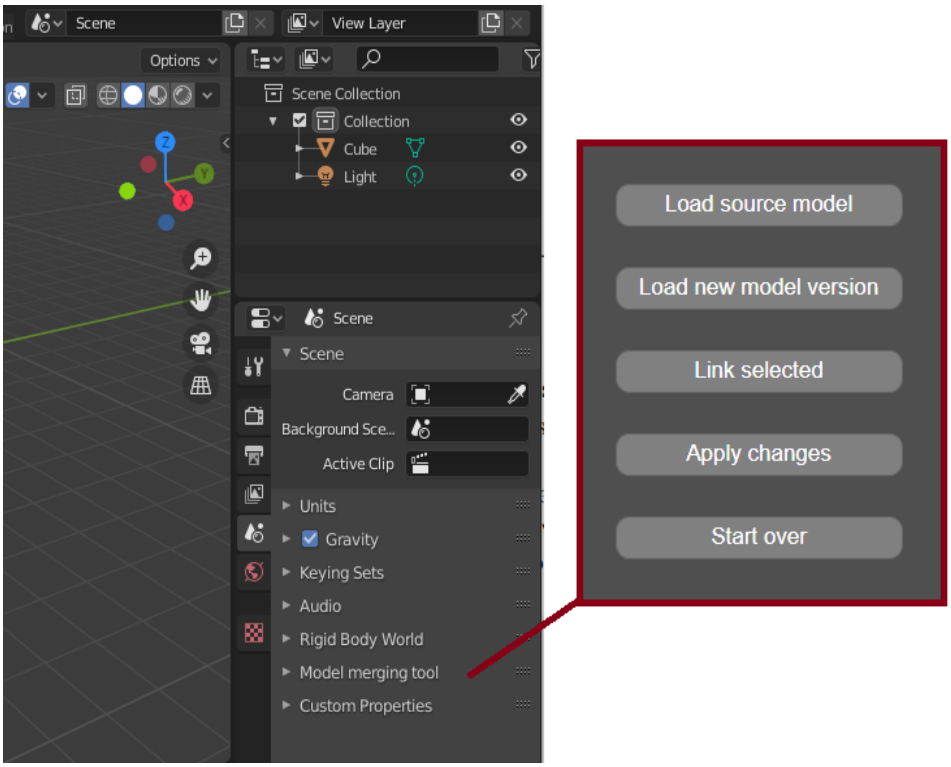

Figure 2.3: GUI buttons mockup

First, the user is expected to load the initial version of the tessellated model, which the first button *Load Source Model* will be programmed to. Clicking on this button will open a basic context window for the user to choose the source model path. The second button from the top, *Load New Model Version*, will be from the user's point of view almost identical, it will show the context window for choosing the model source path as well. The third button *Link Selected* can the user use when the automatic subpart matching algorithm incorrectly decides, that two versions of the same parts are different, or if the user is aware that the geometry changed too much. There will be a constraint that this button can be used only when exactly two sub-parts are selected and are from different model versions. Otherwise, the button will be inactive. The fourth button *Apply changes* is for merging confirmation. The last button *Start over* will delete all metadata stored in the scene. This will not delete anything from the scene, but no model will be marked as a source,

the process has to start all over again from the beginning.

All the buttons will be activated and disabled depending on the context. The user can not load any model if it was just loaded and not confirmed yet. On the other hand, buttons *Link selected* and *Apply changes* can be only used if a new model version was loaded and the merge was not finished yet.

2.7 Tool architecture

Even though Blender supports the source code of the custom tools to be separated into multiple files, I have decided to keep the whole implementation in a single Python file. Splitting the implementation would be appropriate in the case of further functional extension of the tool.

There are two main classes used for model manipulation and storage of the data:

1. **class Model** - this class represents whole loaded model as a single entity
2. **class Subpart** - represents a single piece of geometry in Blender's Scene collection

The remaining classes are implementing Blender operators. Model instance stores its Sub-parts in a member dictionary, where the key is the Sub-part's hash and the value is a list of all sub-parts with the given hash. For the storage of an object's state are used Blender custom properties. Once the scene is initialized, the scene has a defined custom property *state* tag, determining which part of the merging is the tool at the moment. There are three possible states of the tool, *START*, *READY* and *MERGING*. *START* represents the state when no model has been loaded yet. After loading the source model version, the state is switched to *READY*, which indicates that the updated version of the model can be loaded. Last state *MERGING* represents the state during the merging of two model versions when the linking of sub-parts is possible. After confirming the merge process, the tool switches back to state *READY*.

Chapter 3

Implementation

As explained in previous Chapter 2 - *Design of the solution*, I will implement the tool as a Blender script using Python. I will use the latest Blender version, which is currently version 3.1 (spring 2022). This version supports Python of version 3.10.

In this chapter I will explain the parts of the code I have found essential or interesting for the reader. Code snippets will be included too.

3.1 Registering the tool

Blender modules loaded at startup require register() and unregister() functions. These are the only functions that Blender calls from the code, which is otherwise a regular Python module. [28] I have stored all the classes names inside a list so the functions register and unregister can simply iterate through them as seen on 3.1.

```
classes = [LoadModelOperator,
           LoadNewVersionOperator,
           FinalizeMergeOperator,
           LinkSubpartsOperator,
           DeleteMetadataOperator,
           ToolGUIPanel]

def register():
    for cls in classes:
        bpy.utils.register_class(cls)

def unregister():
    for cls in classes:
        bpy.utils.unregister_class(cls)
```

Figure 3.1: Registering the tool

3.2 Setting up GUI

Final user interface inside Blender's side panel is shown on Figure 3.2. As designed, the buttons' activity is toggled on and off, depending of the context. Each of the operators can define class method *poll* [28] which determines whether the operator's *execute* method can be run.

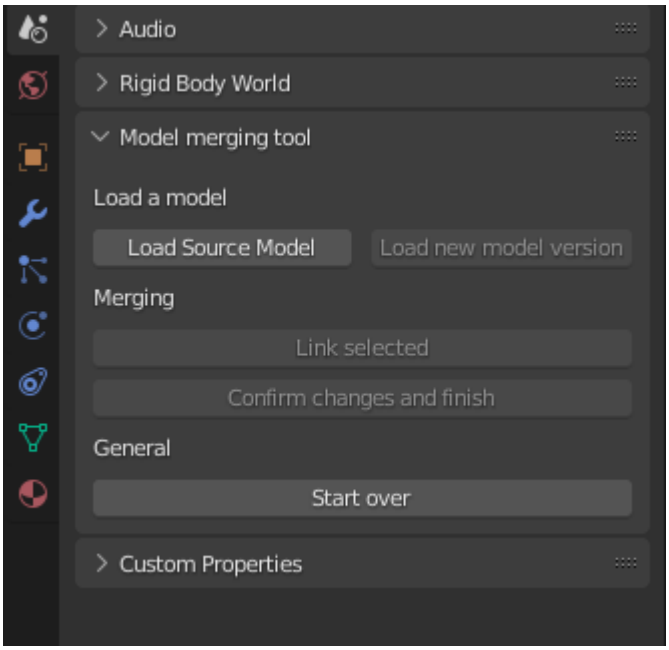

Figure 3.2: Final version of the GUI

Whole panel is defined inside *ToolGUIPanel* class shown on Figure 3.3, which is subclass of an existing Blender type *bpy.types.Panel*. Class members prefixed with *bl_* are used to define the settings and location of the tool. Class method *draw* defines the visual layout of the tool. Blender offers various input methods to use like sliders, drop-down lists, radio buttons and others, however, for the purpose of this tool I used only simple buttons as there are no additional inputs to get from the user. Buttons in UI are divided into three sections: *Load a model, Merging* and *General*.

```
class ToolGUIPanel(bpy.types.Panel):
    """
    Creates a Panel in the scene context of the properties editor
    """
    bl_label = "Model merging tool"
    bl_idname = "SCENE_PT_layout"
    bl_space_type = 'PROPERTIES'
    bl_region_type = 'WINDOW'
    bl_context = "scene"

    def draw(self, context):
        layout = self.layout

        layout.label(text="Load a model")
        row = layout.row()
        row.operator(LoadSourceModelOperator.bl_idname)
        row.operator(LoadNewVersionOperator.bl_idname)

        layout.label(text="Merging")
        row = layout.row()
        row.operator(LinkSubpartsOperator.bl_idname)

        row = layout.row()
        row.operator(FinalizeMergeOperator.bl_idname)

        layout.label(text="General")
        row = layout.row()
        row.operator(DeleteMetadataOperator.bl_idname)
```

Figure 3.3: Setting up the GUI

3.3 Load source model

For source model loading is responsible class *LoadModelOperator* shown on Figure 3.4. By inheriting from *ImportHelper* type we can show a context window for the user to choose their source model. The model path is then stored into *filepath* property which is used to load the model itself. When the loading is done, scene state is then switched to next state *g_READY*. User can no longer load the source model, until the *Start Over* button is clicked and state is equal to *g_START* again.

```
class LoadModelOperator(bpy.types.Operator, ImportHelper):
    '''
    Load .obj model from a file and mark it as a source version.
    '''
    bl_idname = "gh.func_1"
    bl_label = "Load Source Model"  # button label

    # filter .obj models only
    filter_glob: StringProperty(
    default="*.obj",
    options={'HIDDEN'},
    maxlen=255)  # Max internal buffer length

    @classmethod
    def poll(cls, context):
        return scene_state in bpy.context.scene
        and bpy.context.scene[scene_state] == scene_state_start

    def execute(self, context):
        models["source"].load(self.filepath, is_source=True)
        bpy.context.scene[scene_state] = scene_state_ready
        return {'FINISHED'}
```

Figure 3.4: Definition of *LoadModelOperator* class

On the code snippets 3.4 and 3.5 you can see that actual model loading is performed by *Model* class method *load*, with two parameters, *filepath* and *is_source*. This method calls Blender's function to load a model from a file. Then the model's *Subparts* member is initialized. Finally, the model is marked as a source by adding a source tag set to *True* into custom properties of all sub-parts.

```
def load(self, model_path, is_source):
    deselectAll()
    bpy.ops.import_scene.obj(filepath=model_path)
    self.subparts = self.loadSubparts(
                            bpy.context.selected_objects,
                            is_source)
```

Figure 3.5: Load method of Model class

Function deselectAll() is called before the actual model loading to easily distinguish the parts of the model which were loaded most recently and remain selected after loading. Then the selection is passed to the loadSubparts method as a parameter. Even though this method has access to the *bpy.context*, I have passed it as a parameter to highlight the fact that only the

selected parts are the new ones. *Subparts* of the model are stored in a Python dictionary. The computed hash value is used as a key and because of some (even though very small) possibility of hash collision, instead of storing the *Subpart* directly as a value, all Subparts with the same hash are stored inside a list.

3.4 Subpart comparison

Every Subpart object computes its own hash. The computation 3.6 is based on a weighted sum of coordinates of all of its vertices, where every coordinate is multiplied by a different, relatively large, prime number, and then the components are summed up. The sum ensures that the hash remains the same if the internal order of the same vertices was different. After summing up all the vertices, multiplies of Subpart's dimension extremes are added to the hash number, together with a number of vertices. This way is the probability of hash collision even less likely. The coordinates added to the hash sum are rounded to the three decimal precision to neglect possible small vertex location deviations. Lastly, the decimal part of the resulting hash is neglected and only the whole number part is kept. In the case that the hash is too small, the decimal point is moved to the right by 5 places and rounded afterward.

```
def computeHash(self):
    hash_sum = 0
    for vertex in self.object.data.vertices:
        self.updateExtremes(vertex)
        v = vertex.co
        hash_sum += (round(v.x, 3) * 13931
                     + round(v.y, 3) * 74747
                     + round(v.z, 3) * 8887)

    # this should ensure that the hashes will be
    # different for different models in the most cases
    hash_sum += (self.extremes['x_max'] * 13
                 + self.extremes['y_max'] * 71
                 + self.extremes['z_max'] * 73
                 + len(self.object.data.vertices))

    # round to avoid comparing decimals
    hash_sum = (round(hash_sum*10e5)
                if hash_sum < 100
                else round(hash_sum))

    return hash_sum
```

Figure 3.6: ComputeHash method implemented on Subpart class

When both source and updated model versions are loaded, automatic model merging is initiated. By default, all subparts of the new version are marked to keep and source ones to not. The first part of the merging process is finding corresponding subpart hashes in both model versions. If there is a match, it means that the subpart did not change and the original textured version should be kept. The new version is marked to delete and hidden as well, to indicate the fact to the user visually. If any hash from the source version was not found, it is stored into *unmatched_hashes_old* list for later processing. The tags, whether to keep the subpart or not are stored as an object's custom property.

In the second part of the merging process, the algorithm attempts to find similar subparts between the versions inside the *matchCorrespondingSubparts* function 3.7, which takes two lists of Subparts, one from the source version and one new. To avoid comparing each old part with each new (O(m*n) comparisons, where m and n are lengths of lists respectively), both lists are sorted by locations first. Then, the i-th part from the list of old subparts is compared only to the range of $\langle i$ - *difference* : i + *difference*$\rangle$ parts from the other list, at maximum. The *difference* variable is computed by the sum of the difference of the lists' lengths and the addition of the logarithm of the longer list length. The idea behind this formula is that if some parts were deleted, the corresponding parts can have a different index. At the same time, some other parts could be added so the number of the new parts could remain the same, therefore the logarithm part is added as some relative element to the size of the input. In edge cases, the formula is still not bulletproof and can miss some similar parts. However, this is not such an issue because the user can manually link the subparts in a second. On a snippet below is *not* the actual function but the shortened version to outline the idea.

```
def matchCorrespondingSubparts(unmatched_subparts_new,
                               unmatched_subparts_old):
    # ... input lists are sorted here

    # the best strategy is now try to pair corresponding
    # subpart in other array, in case something was deleted
    # we will check *difference* items before and after
    old_len = len(unmatched_subparts_old)
    new_len = len(unmatched_subparts_new)
    difference = math.ceil(abs(old_len - new_len)
                 + math.log(max(new_len, old_len), 2))

    for first in range(0, old_len):
        start = max(0, first - difference)
        end = min(new_len, first + difference)

        for second in range(start, end):
            old_sub = unmatched_subparts_old[first]
            new_sub = unmatched_subparts_new[second]
            sim_fact = old_sub.getSimmillarityFactor(new_sub)
            if (new_sub not in new_subs_matched
            and sim_fact > SIM_TOLERANCE):
                # ... here are the parts linked together
                break
```

Figure 3.7: Snippet of *matchCorrespondingSubparts* function

Subpart method getSimmillarityFactor compares two subparts for similarity and returns a number between 0 and 1, where 1 (with some small deflection) is considered an exact match. This method compares the dimensions, global extremes, and a number of vertices, and each match is represented as a part of the resulting factor. *SIM_TOLERANCE* constant is defined at the beginning of the script and represents a value above which are the parts considered similar and should be linked. After some tweaking, I have set it to a value of 0.75, which seemingly gave the best results.

3.5 Merge the versions

Most of the actual model merging is already done when loading a new model version and comparing the subparts as described in the previous section. All it remains is to actually replace the data if some parts were marked as similar, either automatically or manually with *Link Selected* button. All source subparts which were linked to the new version have their geometry data replaced by a new version, while the material is being preserved. Code snippet 3.8 from the *updateGeometry* method shows how is the data replacement performed.

```
# new part has the name of the original part
# stored as custom property
source_mesh = bpy.context.scene.objects[new_object[MAPPEDTO_TAG]]
material_backup = source_mesh.data.materials[0]
source_mesh.data = new_object.data
source_mesh.data.materials[0] = material_backup
new_object.hide_set(True)
```

Figure 3.8: Updating source mesh geometry with new one

After the user confirmation of the merging process by clicking on *Confirm Changes and Finish* button, all the parts with tag *g_KEEP* set to False are simply deleted, merging is done and in case of need, another version can be loaded again.

3.6 User documentation

As depicted in Figure 3.2, the UI consists of only a few buttons so the usage is relatively straightforward. However, in order to use the tool, the user has to download and install it first.

3.6.1 How to install the tool

1. Download the source Python file and move it into *install_folder/Blender Foundation/Blender 3.1/3.1/scripts/addons*. Change the path according your installation folder and Blender version.

2. Open Blender and go to *Edit ->Preferences ->Add-ons*

3. Add-ons are disabled by default. Find the *Model Merging Tool* in the list and activate it by clicking on the tick box next to it. The tool should be active now. If not, check the Blender's Python console for possible errors.

Another alternative is to open the source code in Blender's *Scripting panel* and run it directly.

3.6.2 Usage

In order to start click the *Start Over* button, which initializes the internal states needed for correct tool execution. The source model should be loaded by clicking on *Load Source Model* button. By using this button instead of built-in import is ensured that only the loaded geometry is tracked and all remaining parts of the scene are ignored by the tool. From now on, all other model versions should be merged with *Load New Version* button. All unmatched

old sub-parts are hidden but the user can link the geometry from new to the original part by clicking on *Link Selected* button. This overwrites the geometry data of the source version with a new one while preserving the materials. Merging is then confirmed by *Confirm Changes and Finish* button. To restart the state of the tool, click on *Start Over* button. This only sets the internal state of the tool to the beginning and does *not* delete any actual object.

Chapter 4

Testing

The script was tested on a model of corporate building from this [29] source. All model versions were edited and exported from AutoCAD [30] into *Wavefront (.obj)* format, which were then imported into Blender using the script.

On a Figure 4.1 is depicted a simple scenery I have modelled, to simulate the use-case when a more complex scene is visualized apart from the actual model.

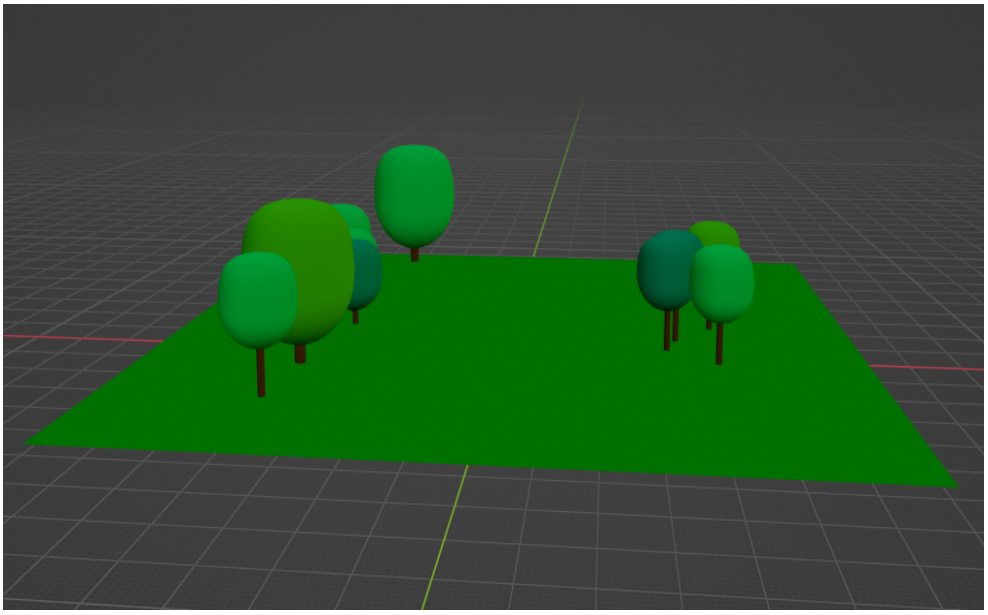

Figure 4.1: A simple background scene created in Blender [24].

The Source tessellated model of the building did not have any assigned materials. In real-world virtual production, a graphic designer would assign materials and textures which would resemble the final building look as much as possible. However, for testing purposes, I have only used a single bright-pink material for the changes and differences to be more visible on the images. This does not affect nor alter the usual usage of the script.

4.1 First iteration

Firstly, using *Load Source Model* button I loaded the initial version of the tessellated model into the scene and moved it a bit further away from the centre of the scene using Blender's translate command. This operation simulates the setting of the model into a more complex scene. For the model post-processing, I have assigned the bright pink material to all subparts of the given model. The initial version of the model is depicted in Figure 4.2.

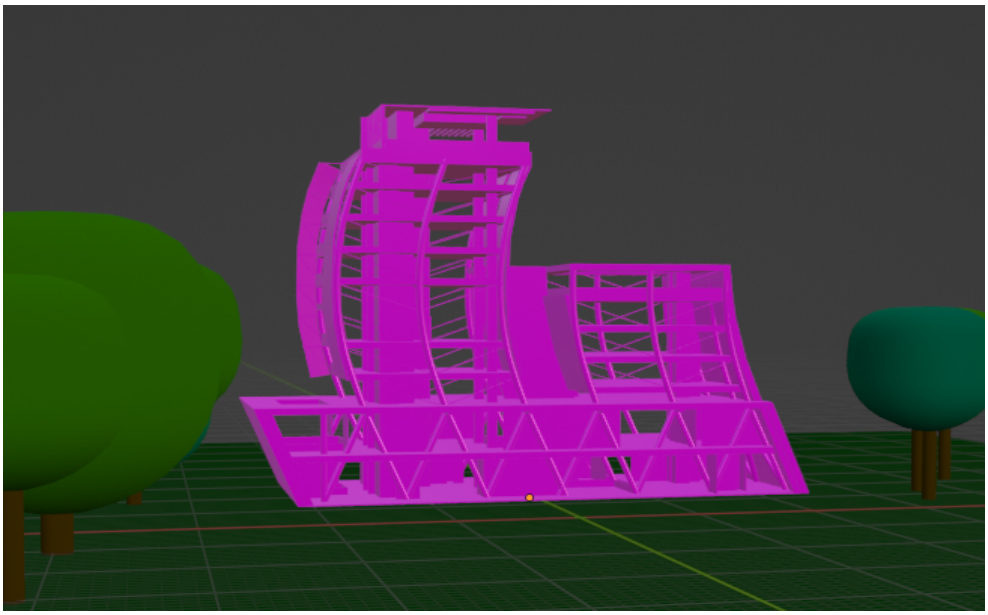

Figure 4.2: Initial version of the model in Blender scene.

4.2 Second iteration

Imagine the model was then visualized in VR using a tool of the choice and the sponsors were not quite satisfied with the design of the building and ordered to make the right part of the building higher. The engineer had to remake the source CAD model, then the model was exported and imported to Blender using the second *Load New Model Version* button. In the Figure 4.3 on the left is depicted the result of the automatic version merge, which, as expected, kept the original textured parts of the building, and the new, edited ones are left without assigned material. Note, that even though the grid on the right side of the building's roof was translated, the algorithm detected the matching shape and automatically assigned the same material as in the previous version. The remaining new parts, which could not be matched with any previous versions, can be linked manually. Changes were not confirmed yet, so all the parts of the previous version are only hidden. For the next test, I have shown the previous version of the roof, selected both old and new versions of the roof, and clicked the *Link Selected* button. The material was correctly transferred to the new roof version and the old version was correctly hidden.

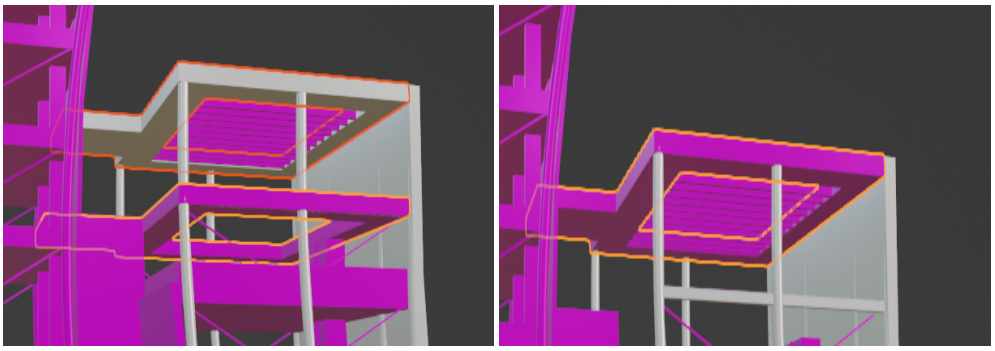

Figure 4.3: Linking of the different versions of the same sub-part. Before linking (left) and after successful linking (right).

For the rest of the model, either the same process of linking or entirely new materials can be applied. After all parts were linked, the merging is finalized by clicking on *Confirm Changes and Finish* button. All hidden unused parts were removed from the scene, as expected.

4.3 Third iteration

The third and final tested iteration was started with the model from the end of the second iteration. I have assigned the bright-pink material to all remaining unlinked parts for better visualization. This test case represents the repetitive process of visualizing and reworking the source models until the sponsors are satisfied with the results. Imagine the sponsors realized, that they actually do not like the appearance of the left part of the building's roof as well. Again, the engineer changed the source CAD model and it was exported to Blender. The textured model from the previous iteration and updated tessellated model without textures can be seen in Figure 4.4.

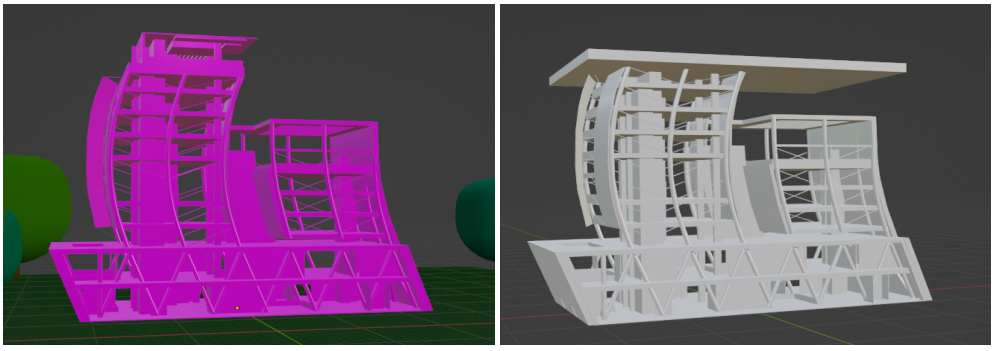

Figure 4.4: Textured model at the end of the second iteration (left). Updated tessellated model without textures (right).

Having the textured model loaded from the previous iteration, I loaded the new version by clicking on *Load New Model Version* button. Again, all the unchanged parts were correctly matched and textured versions were preserved as depicted in Figure 4.5.

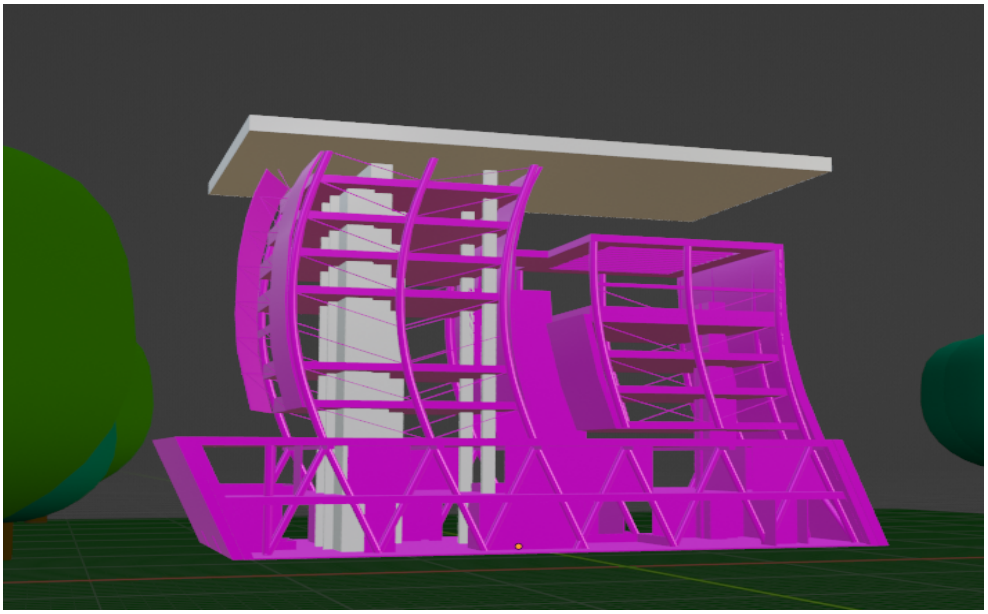

Figure 4.5: Result after automatic merging of the third model version.

4.4 Summary of the testing

To summarize the results of the testing of the tool, all test results were satisfactory and in compliance with the assignment of this thesis. All the model versions were correctly merged and moreover, in the second iteration, the roof grid was automatically matched with its previous version at a different location which can be even more helpful for the user. Tool buttons were inactive or active, depending on the context correctly. Merging of the models could be canceled anytime by using *Start Over* button.

Conclusion

In this thesis I have analysed the process of virtual prototype creation, focusing especially on virtualising CAD data. After understanding currently used processes, I analyzed all available solutions on the market, which I have found relevant in this context. In chapter Analysis, I have mentioned a few tools enabling the user to directly visualise source CAD data. Therefore, in the case that the user does not need to post-process the tessellated CAD model and is willing to pay for the license, they can use the tool directly. In the other case, when the requirements on the visual quality of the output scene are high and mesh post-processing is needed, I have designed a non-destructive workflow for repeatable virtual prototype creation, that enables the user to create new versions of the virtual prototypes in less time, compared to creating a new prototype each time the source CAD model changes. In order for this workflow to work, I have written a Blender plug-in tool, which enables the user to merge the changes made on the new version to the previous one.

At first, it was a challenging task for me to come up with a way, to merge these model versions and my initial idea failed. The second idea was more successful and as described in previous Chapter 4 - Testing, the tool works on the given model as expected, which makes me confident to say that it can help reduce costs related to virtual prototype production.

Possible extensions

One possibility is to improve the heuristics of the subpart comparison. An improved version of the algorithm could be detecting similarities between hierarchies of multiple sub-parts, e.g. when the whole roof with chimney and roof window was replaced by other pieces with a different design. However, this can be matched manually and it does not take too much time to do.

Another functionality extension could be to be able to only highlight the unmatched geometry of two model versions, keeping the possible mesh decimation in mind. This could be helpful when the virtual prototyping workflow

is not abided to, and the new changes are made on the tessellated model first which results in a state when a sourcing CAD model is not representing the exact model which the reviewer approved. Engineers who would want to update the sourcing CAD model could tessellate it and visualize the changes between the models, making it easier to only edit the parts that differ.

Bibliography

1. SAMPAIO, Alcínia Z.; FERREIRA, Miguel M.; ROSÁRIO, Daniel P.; MARTINS, Octávio P. 3D and VR models in Civil Engineering education: Construction, rehabilitation and maintenance. 2010. ISSN 0926-5805. Available from DOI: https://doi.org/10.1016/j.autcon.2010.05.006.
2. FOLEY, James D. et al. *Computer Graphics: Principles and practice*. 2nd edition in C. Addison-Wesley Publishing Company, Inc., 1995. ISBN 0-201-84840-6.
3. UNITY TECHNOLOGIES. *Pixyz Studio 2021.1 Documentation* [online] [visited on 2022-04-13]. Available from: https://www.pixyz-software.com/documentations/html/2021.1/studio/AboutTessellation.html.
4. HANNIEL, Iddo; HALLER, Kirk. Direct Rendering of Solid CAD Models on the GPU. 2011, pp. 25–32. Available from DOI: 10.1109/CAD/Graphics.2011.63.
5. MARK SEGAL, KURT AKELEY. *The OpenGL Graphics System: A Specification (Version 4.0 (Core Profile) - March 11, 2010)* [online] [visited on 2022-04-13]. Available from: https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf.
6. BAKER, Timothy J. Mesh generation: Art or science? *Progress in Aerospace Sciences*. 2005. ISSN 0376-0421. Available from DOI: https://doi.org/10.1016/j.paerosci.2005.02.002.
7. GUO, Jianwei; DING, Fan; JIA, Xiaohong; YAN, Dong-Ming. Automatic and high-quality surface mesh generation for CAD models. *Computer-Aided Design*. 2019, vol. 109, pp. 49–59. ISSN 0010-4485. Available from DOI: https://doi.org/10.1016/j.cad.2018.12.005.
8. JOERG SCHMIT. *Mercedes Benz 500 K* [online] [visited on 2022-04-14]. Available from: https://grabcad.com/library/mb500k-1.

9. OPEN CASCADE. *CAD Assistant* [online] [visited on 2022-05-09]. Available from: https://www.opencascade.com/products/cad-assistant/.

10. YASIN R. *Apartment* [online] [visited on 2022-04-13]. Available from: https://grabcad.com/library/apartment-35.

11. PIRET, Cecile; REMACLE, Jean-François; MARCHANDISE, Emilie. Mesh and CAD Repair Based on Parametrizations with Radial Basis Functions. 2011. ISBN 978-3-642-24733-0. Available from DOI: 10.1007/978-3-642-24734-7-23.

12. SCHROEDER, William J; ZARGE, Jonathan A; LORENSEN, William E. Decimation of triangle meshes. 1992, pp. 65–70.

13. LI, Minglei; NAN, Liangliang. Feature-preserving 3D mesh simplification for urban buildings. 2021. ISSN 0924-2716. Available from DOI: https://doi.org/10.1016/j.isprsjprs.2021.01.006.

14. SOBTI, Rajeev; GEETHA, Ganesan. Cryptographic hash functions: a review. *International Journal of Computer Science Issues (IJCSI)*. 2012, vol. 9, no. 2, p. 461.

15. BLACK J. *Compare-by-Hash: A Reasoned Analysis* [online] [visited on 2022-04-12]. Available from: https://home.cs.colorado.edu/~jrblack/papers/cbh.html.

16. BOWMAN, Doug A.; MCMAHAN, Ryan P. Virtual Reality: How Much Immersion Is Enough? *Computer*. 2007, vol. 40, no. 7, pp. 36–43. Available from DOI: 10.1109/MC.2007.257.

17. INTERACTION DESIGN FOUNDATION. *Augmented Reality* [online] [visited on 2022-04-15]. Available from: https://www.interaction-design.org/literature/topics/augmented-reality.

18. SAEC/KINETIC VISION, INC. *Mixed Reality* [online] [visited on 2022-04-14]. Available from: https://kinetic-vision.com/virtual-interactive-solutions-mixed-reality.

19. QUALCOMM TECHNOLOGIES, INC. *Extended Reality* [online] [visited on 2022-04-14]. Available from: https://www.qualcomm.com/research/extended-reality.

20. ALLMACHER, Christoph; DUDCZIG, M.; KLIMANT, Philipp; PUTZ, M. Virtual Prototyping Technologies Enabling Resource-Efficient and Human-Centered Product Development. 2018, vol. 21. Available from DOI: 10.1016/j.promfg.2018.02.180.

21. CHOI, S.H.; CHAN, A.M.M. A virtual prototyping system for rapid product development. *Computer-Aided Design*. 2004. ISSN 0010-4485. Available from DOI: https://doi.org/10.1016/S0010-4485(03)00110-6.

22. TOVELL, Robert; WILLIAMS, Nina. Genesis: A Pipeline for Virtual Production. 2018. ISBN 9781450358958. Available from DOI: 10.1145/3233085.3233090.

23. AUTODESK INC. *Maya: Create expansive worlds, complex characters, and dazzling effects* [online] [visited on 2022-04-19]. Available from: https://www.autodesk.com/products/maya/overview?term=1-YEAR&tab=subscription.

24. BLENDER FOUNDATION. *The Freedom to Create* [online] [visited on 2022-01-20]. Available from: https://www.blender.org/about/.

25. AUTODESK INC. *AutoCAD* [online] [visited on 2022-05-04]. Available from: https://www.autodesk.com/products/3ds-max/overview?term=1-YEAR&tab=subscription.

26. EPIC GAMES. *Unreal Engine docs* [online] [visited on 2022-02-02]. Available from: https://docs.unrealengine.com/4.27/en-US/WorkingWithContent/Importing/Datasmith/.

27. UNITY TECHNOLOGIES. *pixyz studio* [online] [visited on 2022-01-15]. Available from: https://www.pixyz-software.com/studio/.

28. BLENDER FOUNDATION. *Python API Overview* [online] [visited on 2022-05-02]. Available from: https://docs.blender.org/api/current/info_overview.html.

29. DAVID MARTINEZ DIAZ. *Corporate building* [online] [visited on 2022-05-07]. Available from: https://www.bibliocad.com/en/library/corporate-building_109399/.

30. AUTODESK INC. *AutoCAD* [online] [visited on 2022-04-28]. Available from: https://www.autodesk.com/products/autocad/overview.

Appendix A

Acronyms

CAD Computer aided design

CSG Constructive solid geometry

BREP (B-rep) Boundary representation

NURBS Non-uniform rational basis spline

GPU Graphical processing unit

VR Virtual reality

AR Augumented reality

MR Mixed reality

XR Extended reality

VP Virtual prototyping

API Application programming nterface

GUI Graphical user interface

UI User interface

Appendix B

Contents of enclosed CD

- `src` ..... the directory of source codes
  - `code` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `sample data` .... the directory containing sample models used for testing
- `thesis.pdf` ..... the thesis text in PDF format