Università della Svizzera italiana

Faculty of Informatics

Bachelor Thesis

December 13, 2019

A robust pipeline to process 3D objects

for virtual reality visualization

Niccolpichini

Abstract

The interest in Virtual Reality (VR) has increased exponentially in the last years, leading to the development of more and more applications and platforms. To prevent motion sickness and nausea, VR comes with strict requirements in terms of performance. HEGIAS, a Swiss company specialized in VR software, is building its VR content on top of A-Frame, a WebVR-based framework. A-Frame relies on the Three.js rendering engine. Three.js is a Javascript library, built on WebGL, a Javascript API for rendering 3D graphics in any compatible browser. This layering yields to a lack of performance and makes it important that 3D scenes are organized in an optimal way, and that the overall geometry is simplified.
The goal of this project is to optimise 3D scene to be visualised in VR by decimating their geometries and organising them in an optimal structure, decreasing loading time and increasing the number of FPS when rendering within A-Frame. This goal is accomplished by automating steps and procedures into a 3D processing pipeline, capable of supporting HEGIAS' web app, which is mainly meant to be used by architects to visualize their projects. This 3D processing pipeline has the responsibility of preparing 3D scenes and objects to be visualized and loaded in HEGIAS' VR environment.

Advisor:
Piotr Didyk
Co-advisor:
Andreas Schmeil

Approved by the advisor on Date:

Contents

1 Introduction

1.1 Motivation

Today, virtual reality (VR) is considered one of the most important research topics in computer graphics [5] , and it has become widely used as a functional tool in many fields. HEGIAS' web app, a VR environment is exploited to allow changes before they actually take place in the real world, possibly saving money and time. As instance, a designer could move the furniture of an apartment in HEGIAS' web app before doing it in the real world, even before actually buying or producing the furniture. VR is, therefore, a really powerful tool.
In the past, VR has typically been restricted to research universities (Figure 1), large corporations, and military for training. Progress in both hardware and software technology has helped to overcome these limitations and will allow VR to be available and useful to a much wider population.

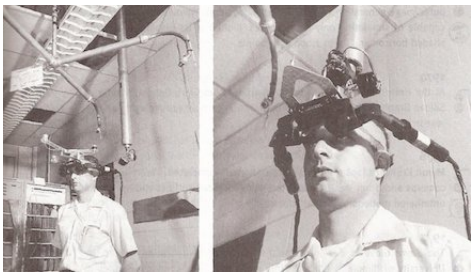


Figure 1. *The Sword of Damocles*, the first virtual reality head-mounted display system, MIT Lincoln Laboratory (1968).

In a VR environment, latency is an important aspect. The feeling of realism in a virtual reality environment highly relies on the latency. Latency is the total time between the movement of the user's head (with a VR headset) and the updated image being displayed on the screen. This includes the times for sensor response, rendering, image transmission and display response. Above 20 ms of latency users feel less immersed and comfortable in the VR environment, as stated by the Oculus developer page[1]. Above 60 ms of latency, users head's motions and motions in a VR environment are out of sync, leading to headache, nausea and vertigo [1]. This kind of illness is known as *Virtual Reality sickness*. It is believed that large latency is one of the primary causes of VR sickness. In an ideal VR environment, the latency would be 0 ms. This is impossible to achieve (with the current technology) but keeping it as small as possible is important. Latency is related to the number of FPS (Frame per Second). A high number of FPS leads to low latency. If the number of FPS is not high enough, the rate of images rendered will fall behind the rate of images in the screen, leading to a high latency and all the problems listed previously.
A virtual reality environment is computationally expensive. Everything is rendered twice, one time for each screen of the headset, leading to a high computational cost. It should now be clear how keeping a high number of FPS in a virtual reality environment is problematic: it is as computationally expensive as important to achieve to assure realism and prevent illness.
Moreover, the loading time of a 3D scene in VR should be as short as possible. Unlike traditional applications where users can do something else during a loading screen like checking the phone, drinking a coffee or whatever keeps them busy while waiting, VR users instead cannot do something else during a loading screen. VR users are trapped in loading screens. For this reason, to make the experience as enjoyable as possible, the loading time should be as short as possible.

1.2 HEGIAS

This project is part of a large application at the moment in developing stage at HEGIAS AG[2], a Swiss company where I'm currently[3] working as a programmer.
HEGIAS' application allows users to view and interact with their 3D models within its web application. A user can navigate through his projects, add new ones or visualise them in VR. Inside the VR environment, the user can import in real time its objects from the library, to move them and to change materials and/or textures.
This application, at the current stage, is mainly meant to be used by architects, and their clients, to plan, visualize

[1] https://developer.oculus.com/design/latest/
[2] https://www.hegias.com/
[3] December13,2019

and communicate better. This is achieved by allowing multiple users to connect simultaneously at the same scene, meaning that each user can share his ideas, move objects such as furniture and make modifications in a fully capable VR environment.

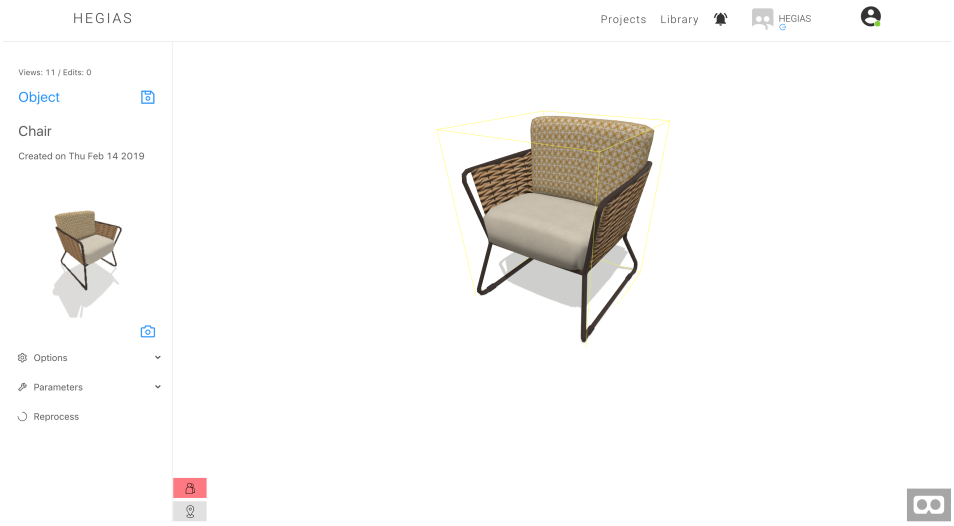


Figure 2. Hegias user interface

1.3 Project goal

The goal of this project is to improve the performance, in terms of FPS, in HEGIAS' web app by simplifying objects' geometry and organizing 3D scenes in an optimal structure. These steps are generally done manually by 3D artist. In this project they are automatised into a 3D processing pipeline with the responsibility of supporting HEGIAS' web app.

Right now HEGIAS' application has no level of optimization. To improve the performance (the number of FPS) within A-Frame, the VR framework used within HEGIAS' web app, some adaptions have been made. The most effective one is to decimate the geometry of objects in a 3D scene. An object with a high number of vertices is computationally more expensive than an object with a low number of vertices. The second improvement is to organize the structure of a 3D scene. Since HEGIAS' web app is meant for architects, the 3D scenes that are uploaded are composed of many identical objects. This means that only one object for each group of identical objects needs to be processed and loaded in the web app. This decreases both the loading time and the time to process a scene. Moreover, if an object present in a 3D scene is already in the database, no processing is required at all.

At the end of the report results on decimating different 3D scenes and objects and their respective decrease in loading time into HEGIAS' web app will be shown.

2 Background

Before proceeding it is worth giving some basic knowledge that I consider essential for this project. The following section focuses on the main concepts that have to be considered as basis for the report. During the report, I will cite this section and each subsection when required for a better understanding.

2.1 Mesh

Mesh and object, are often used as synonyms even if they do not exactly have the same meaning. A mesh is the part of an object that defines the geometry of it (i.e. the vertices and edges). More formally, a mesh is an ordered set, in terms of connectivity, that defines the shape of an object.
An object can have other elements such as armatures, information about animations and/or textures. An object can contain multiple meshes or none at all, as in the case of a *null*, which has no geometry. On the right, there is the image of a human body inside Blender's interface. The human body is composed of different elements such as the head, the chest, the arms, the legs, etc. All these elements are meshes. Moreover, this object has an armature, which is used for animation. This armature does the same work than bones do in the human body, its purpose is to move the meshes, like in a human body the bones move the body. The object in all its complexity is composed by all these elements (armature, meshes etc).

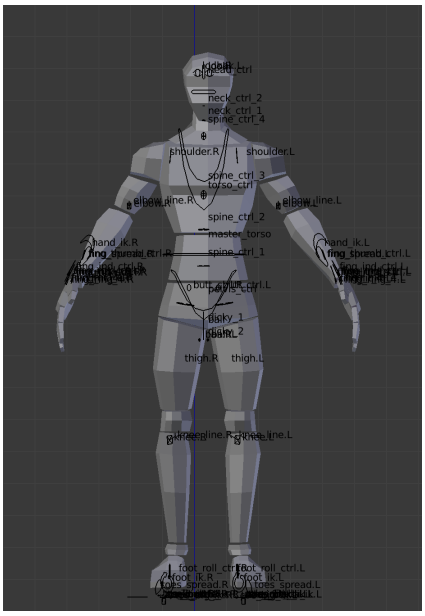


Figure 3. A human body inside Blender's interface

2.2 3D file formats

A 3D file format stores information about 3D objects as plain text or binary data. Their purpose is to encode geometry, materials, animations and all the information about the scene. The position of each object in the scene and lights is encoded in the 3D format. However, not all 3D file formats encode all this information. For example, a *.stl* file format stores only the objects' geometry. In contrast, a *.dae* file format store all the information. There are literally hundreds of 3D file format, almost every CAD software has their own. For example, Cinema4D's 3D file format is *.c4d* and Blender's 3D file format is *.blend*. The presence of so many file formats gives a big problem of compatibility. Suppose two 3D artists want to share with each other their works. The first 3D artist uses Blender, the second one Cinema4D. Therefore, the first one is working on a *.blend* 3D file format, and the second one on a *.c4d* 3D file format. Blender does not support natively a *c4d* 3D file format, and Cinema4D does not support on the same way a *.blend* 3D file format. This compatibility problem can be solved using a neutral 3D file format, such as *.dae*. The two 3D artists can export their works in *.dae* and share them with each other. However, some information is based on the software. For example, Blender Cycles materials will rarely be exported correctly in any 3D file format, into any other unless it is *.blend*. The same applies to Cinema4D and to any CAD software.
The most popular 3D file formats are STL, OBJ, FBX and DAE. Each of them has its strength and weakness.
STL is a neutral 3D file format. It is one of the oldest 3D file formats. It encodes the geometry of an object using a triangle mesh. Nowadays it is primarily used in 3D printing. STL can describe only the geometry of an object, any other information cannot be stored in a *.stl* 3D file format. It cannot store, for example, the colours of an object.
OBJ is another neutral 3D file format, widely used in Computer Graphics. OBJ supports both triangle and polygon faces. It can encode materials in a separate file with the extension *.mtl*. It cannot encode animations and units. Because it cannot encode units, it is not a good choice for architectural models.
FBX is a proprietary 3D file format introduced by Autodesk. FBX supports all the information told before (geometry, materials, animations, etc.). Even if it is a proprietary 3D file format, the way importers and exporters handle this format changes from software to software. The SDK just provides some guidelines.
DAE is a neutral 3D file format developed by the Khronos Group. The intent of the Collada (DAE) 3D file format

was to become a standard among 3D file formats. It is one of the few 3D file formats which supports kinematics and physics.

Even if it is not a popular 3D file format, for this project, some words have to be spent on IFC 3D file format. This 3D file format is meant to be used on architectural models. It is primarily used in Autodesk Revit. This 3D file format supports colours but it does not support textures. Since HEGIAS developed a web app mainly meant for architects, it is not uncommon to see IFC 3D file format being uploaded from a user.

2.3 Non Manifold Meshes

An important property of a 3D mesh is to be a two-dimensional manifold. A definition of surface in the context of computer graphics is "an orientable continuous 2D manifold embedded in $\mathbb{R}^n$". Visually, this can be understood as a solid surface which does not have infinitely thin parts, such that the surface separates "interior" and "exterior" (Figure 4).

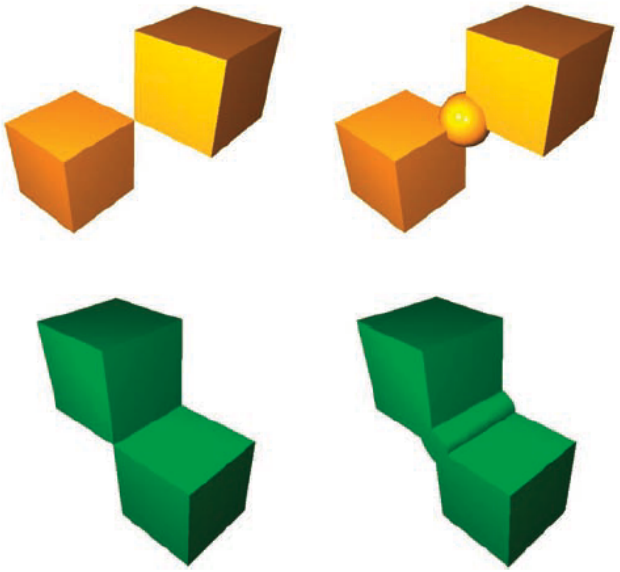


Figure 4. Examples of non-manifold surfaces (on the left) and their respective manifold (on the right). Image taken from [3]

In the first case (top-left) there is a single vertex connecting the two cubes. In the second case (bottom-left) the cubes are connected through an edge. In both cases, there is no separation between interior and exterior of the surface (a vertex and an edge do not have a volume). This is fixed by "filling" the mesh to give it volume in the non-manifold vertex/edge.
Non-manifold configurations (Figure 5) are problematic for most data structures, simulations, Boolean operations and 3D printing. Decimation algorithms can perform poorly on non-manifold meshes, therefore converting a non-manifold mesh to a manifold one before decimating its geometry can prevent errors and bad results.

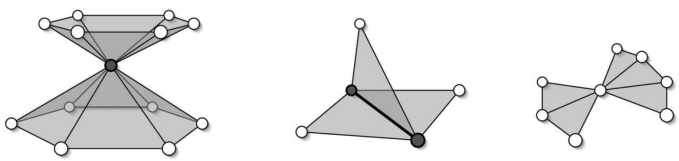


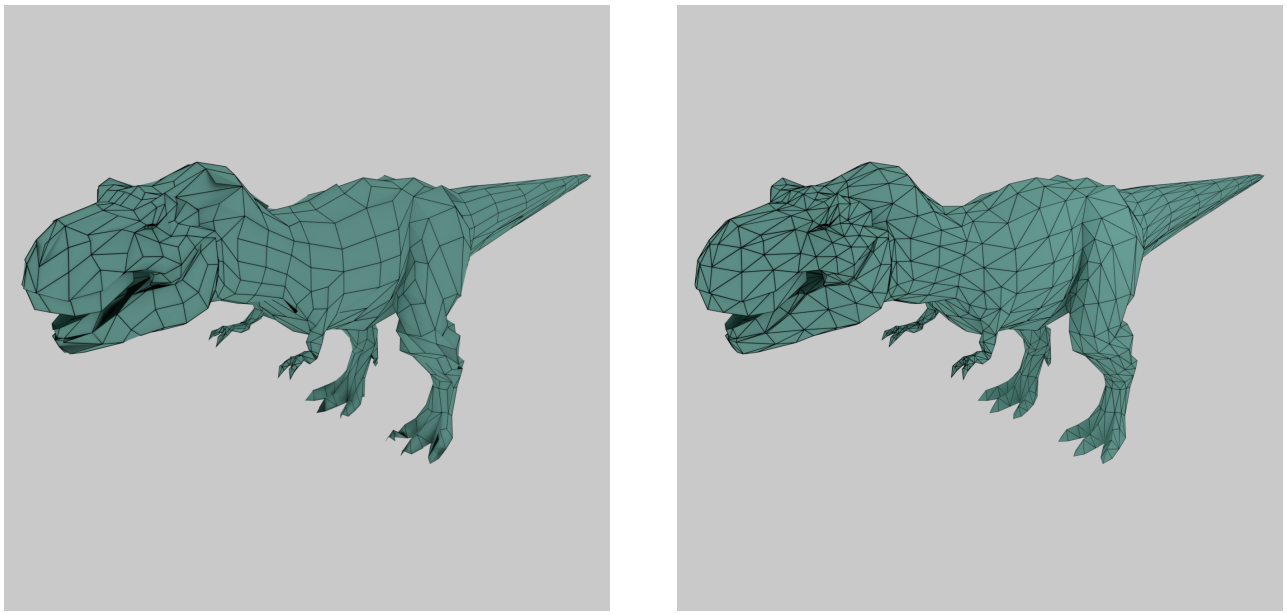
Figure 5. Non-manifold configurations: non-manifold vertex(left and right), non-manifold edge (middle). Image taken from [3]

2.4 Decimation algorithms

The complexity of an object increases along with the number of vertices (or faces). A mesh with a high number of vertices is called *high poly*. In contrast, a mesh with a low number of vertices is called *low poly*. In real-time renderings, such as virtual reality and more specifically for this report, in A-frame, it is important to keep the number of vertices low for performance reasons. An algorithm that reduces the number of vertices of a mesh is called *decimation algorithm*, or *mesh simplification*.
There are many families of algorithms to simplify the geometry of an object. For any 3D application, it is fundamental to preserve the geometry boundaries and to recompute texture coordinates in an optimal way [4]. In order words, the decimated version of the meshes needs to look as much as possible to the original. After studying and comparing different decimation algorithm, I decided to use Quadric Edge Metrics algorithm [6]. This algorithm best fit the needs of HEGIAS, it is fast, does not require the mesh to be non-manifold (which was not a requirement) and, generally, the visual fidelity of the result is high.

2.5 Triangulation

A mesh generally has both triangular, quadrilateral and sometimes Ngon faces (a polygon with arbitrary n vertices). The triangulation of a mesh is a process where every face of it is converted into a triangular face (Figure 6).



(a) Quad domimant mesh

(b) Tris dominant mesh

Figure 6. The same mesh in two different version. On the left it is a quad mesh, on the right a triangle mesh.

Triangulating a mesh before importing it for rendering in A-frame, or any other real-time renderer is important. If a mesh is not triangular, Three.js will triangulate it automatically yielding to increase loading time and potentially wrong results in the topology of the mesh and in face normals' orientations (i.e. flipped normals). Also, triangulating a mesh on the client side (on the HEGIAS' web app) will take much more time than doing it on the server.

2.6 Draw Call

Every real-time engine generates data using the CPU and then sends this data to the GPU so it can be rendered on the screen. This process, is a draw-call, at high-level, and it is repeated for every object and material. To keep things as short as possible, the sub-section focus on what role a draw call plays inside a rendering pipeline and why it is so important to keep a low number of draw calls.
A draw call contains all the information about geometry, textures, shaders, buffers etc. This information is processed by the CPU. It is very expensive for the CPU to make this information available for the GPU [2].

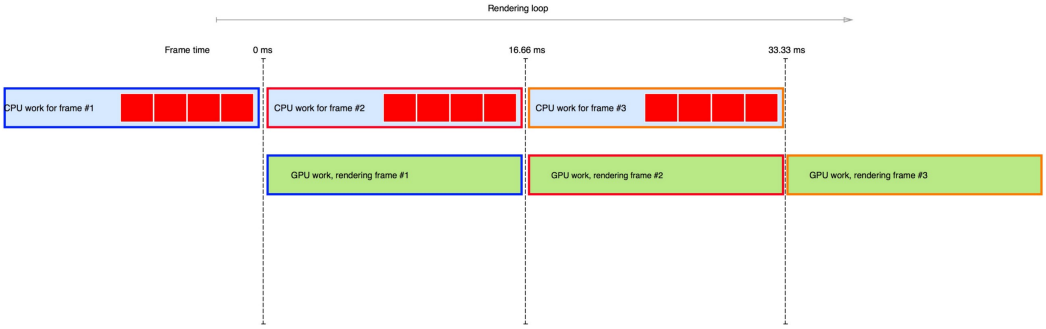


Figure 7. Optimal situation with 4 draw calls (red squares)

Figure 8. Not optimal situation. The GPU is bottlenecked by the CPU and has to wait for it

In the first situation (figure 7) the CPU work *ideally* matches GPU work. When the GPU is rendering the first frame, the CPU is preparing the second frame and so on. However in the second situation (figure 8) the CPU isn't fast enough to prepare the rendering data for the GPU in time, therefore the GPU is bottlenecked by the CPU. The point is that CPU is much slower than GPU and CPU is required for every draw call, implying: the lower the number of draw call is, the better is in terms of performance.

In addition, simplifying the geometry of an object with a decimation algorithm (2.4) reduces the amount of data the GPU has to process. The performance of a rendering engine can be increased by reducing the number of draw calls and the complexity of the geometries.

2.7 Geometry instancing

As explained in 2.6, as the number of draw calls increases, the performance decreases accordingly. Consider the following example. We want to move 10.000 small files 1 KB each from one hard disk to another. This will take a while. Let's now say that we want to move a single file *containing all the small files* of 10 MB (actually 10.000 kB are 9.7 MB). This will take much less time. That is because to move each file some operations, such as preparing the file transfer, allocating memory etc, are required. Similarly rendering many small meshes (which means having many draw calls) is a lot more expensive than rendering one large mesh containing all the small meshes (a single 'large' draw call). This holds as long as the rendering engine is processing meshes with the same materials. For sake of simplicity, I considered only this case, but it is important to keep in mind that additional CPU work, changing shader programs and/or material parameters, has to be done and it is very expensive. Instancing means that you send one mesh for each group of meshes. If a 3D scene has 10 chairs, the CPU sends the geometry of one chair just once instead of many times. To allow these instanced meshes be at different positions, the CPU provides an extra stream of data containing the transformation matrices. Geometry instancing is an excellent technique to decrease the rendering time and therefore increase the number of FPS and since architectural 3D scenes, generally, have a lot of duplicated objects, it suits perfectly with HEGIAS' web app.

3 Tools

The following is a list of frameworks and applications that have been used in this project. Each one of them is introduced by a brief introduction regarding the importance of the project.

3.1 Three.JS

ThreeJS[4] is a cross-browser Javascript library and API used to display 3D computer graphics in a web browser. It uses WebGL[5] an API that allows the rendering of 2D and 3D graphics within browsers. WebGL shares a lot of similarities with OpenGL[6]. This allows Three.js to create 3D content using JavaScript language without relying on any browser-plugin.

3.2 A-Frame

A-Frame[7] is a framework built over ThreeJS currently released under a MIT license. Originally it was born inside Mozilla, lately it became an open source project. Its uses vary but its main purpose is to simplify the making of VR projects. A-frame is naively supposed to be used in HTML elements where every component, such as 3D-objects, images, lights, etc. corresponds to a specific element. This kind of architecture is called Entity-Component-System.
The major advantage of this framework lies on its flexibility which allows easy extendibility. Many libraries have been built over A-Frame such as A-Frame-React[8]

3.3 HTC Vive

HTC Vive[9] is a virtual reality headset developed by HTC and Valve The Vive; it is composed of a head-mounted device, a pair of controllers and one or more tracking systems. Altogether it allows the user to move and interact within 3D scenes. Two major models are currently available. They're quite similar but for the resolution, which is significantly higher on the PRO version. The refresh rate is 90Hz for all models, it is important because frame rate values above 90 FPS have to be maintained in order to avoid nausea and disorientation [1].

3.4 Blender

Blender is an open-source 3D computer graphics software. Its main features include, for this project, 3D modeling, sculpting, UV unwrapping, rigging and texturing. The current stable version is Blender 2.79, this version has been used.
Compared to other software, Blender's main advantage is the great extendibility allowing anyone to write his own scripts and plugins. In fact, Blender allows scripting in Python[10]. A plugin that is worth mentioning is *Blender glTF 2.0 Importer and Exporter*[11] allowing Blender to export to glTF (GL Transmission Format) format. This format extension is *.glb*.
Blender's two render engines: Internal and Cycles. Internal Render, historically, is the first render engine of Blender. It has high compatibility with the main 3D formats. Still, with the advent of Cycles with Blender's version 2.7, it is not much used anymore. Blender Cycles comes with a node system.

[4] https://threejs.org/
[5] https://www.khronos.org/webgl/)
[6] https://www.opengl.org/
[7] https://aframe.io/
[8] https://github.com/supermedium/aframe-react
[9] https://www.vive.com
[10] https://docs.blender.org/api/2.79/
[11] https://github.com/KhronosGroup/glTF-Blender-IO

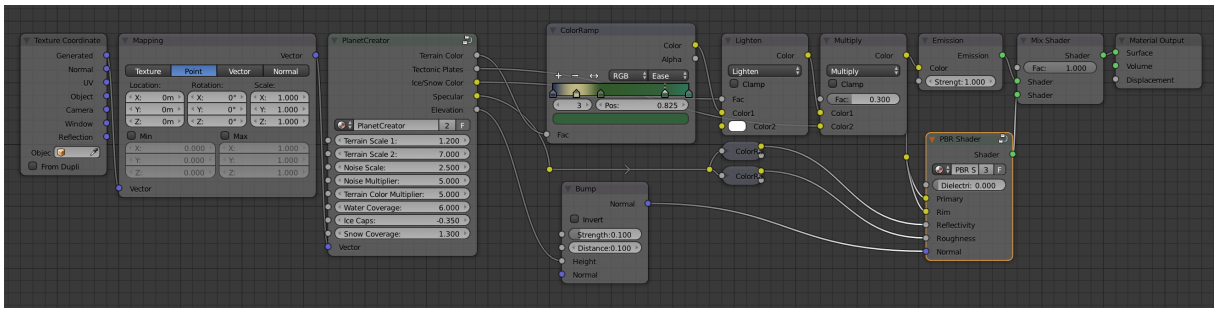


Figure 9. Blender's Cycles node system

Blender glTF 2.0 Importer and Exporter partially supports Cycles nodes. Only a couple of nodes are readable by it. Currently [12] the last Blender stable version is 2.79.

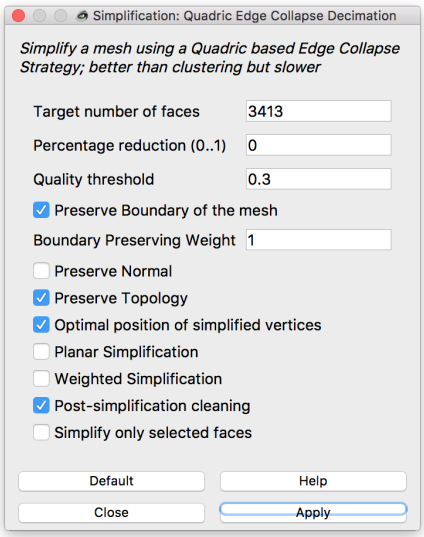
3.5 Cinema4d

Cinema4d is a 3D computer graphics software developed by MAXON. Its main features include 3D modeling, animation, motion graphics and rendering. It allows users to write scripts in 3 languages: C++, Python, and Coffee, now deprecated. For the project, Cinema4D R19 has been used.
An important advantage of c4d over blender is the speed. The most important operations (as joining meshes) are by far faster in Cinema4D than in Blender.

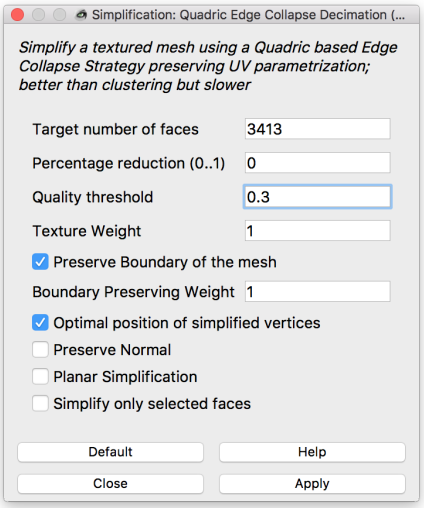
3.6 Meshlab

Meshlab[13] offers a powerful toolset for processing 3D meshes including cleaning filters to remove duplicated and un-referenced vertices, to re-topology non-manifold meshes and to remove overlapping faces, powerful mesh simplification algorithms and tools for curvature analysis. It is free and open source. Originally it was born to process large and unstructured 3D objects arising from 3D scanning, which generally have a lot of vertices and poor quality in terms of topology.

Meshlab, in particular for this project, offers two implementations of Quadric Error Metrics algorithm (2.4) called *Quadric Edge Collapse Decimation* and *Quadric Edge Collapse Decimation (with texture)*. In the image below the Meshlab's user interface of the available for both implementations is shown.



(a) Meshlab's parameters of Quadric Edge Collapse Decimation



(b) Meshlab's parameters of Quadric Edge Collapse Decimation (with texture)

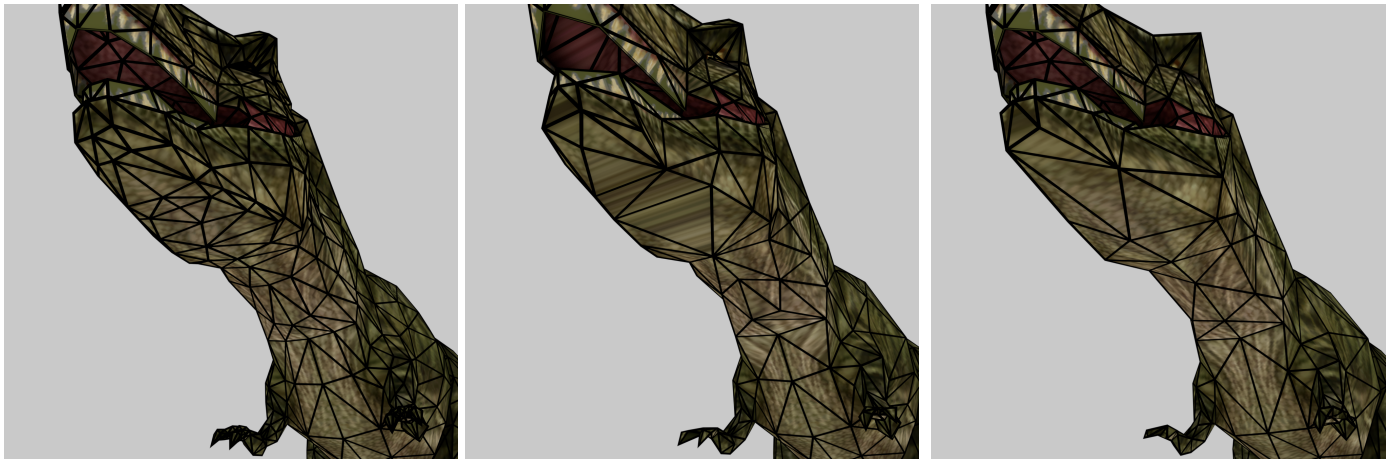
[12] December 13, 2019
[13] `http://www.meshlab.net/`

It is really important to keep the decimated version of the mesh as similar as possible to the original. To achieve this goal, the algorithm has to preserve the boundary of the mesh and the topology. The topology in the computer graphics field refers to the connected components of the mesh structure. Moreover, texture coordinates have to be re-computed in an optimal way to obtain visual fidelity with the original mesh.
Below there is an example applying Cinema4D's decimation algorithm and Meshlab's Quadric Edge Collapse Decimation on a mesh modeled by myself is shown.

(c) 1047 vertices (d) 595 vertices (e) 547

Figure 10. Different results on a mesh applying two versions of decimation algorithms. On the left the original mesh, in the middle the result applying a Cinema4D decimate algorithm and on the right the result applying a Simplification Quadric Edge Collapse algorithm in Meshlab.

At first look, both decimate algorithms seem to perform well. The geometry boundaries are preserved correctly in both algorithms, but texture coordinates are wrong for Cinema4D decimation algorithm. In figure(11) you will see that Cinema4D's decimate algorithm tends to *stretch* the texture, in contrast with Meshlab's decimate algorithm, by using the correct parameters, the texture doesn't get stretched. Meshlab's Simplification Quadric Edge Collapse algorithm outperforms Cinema4D's decimation algorithm in recomputing texture coordinates.



(a) Original mesh (b) Result from c4d's decimate (c) Result from meshlab's decimate

Figure 11. Cinema4D's decimate tends to stretch the texture

Meshlab's and its algorithms play a fundamental role in this project.

3.7 Shadermap

ShaderMap[14] is a textures map generator software developed by Rendering Systems Inc. This software allows the generation of different texture maps from one or more input images. In HEGIAS' 3D processing pipeline this tool is used to generate Normals, Albedo and RMA (Roughness, Metallic and Ambient Occlusion) maps. The role of this application in the 3D processing pipeline is to improve the visual appearance of materials.

3.8 IfcOpenShell

IfcOpenShell[15] is an open source toolkit to convert *.ifc* format to other 3D formats.

[14] https://shadermap.com/home/
[15] http://ifcopenshell.org/ifcconvert.html

4 Overview of the whole HEGIAS' web app architecture

This section aims to give a high-level overview of the HEGIAS' architecture. In the image below, the reader will find a schematization of the interactions between a user, the client (HEGIAS' web app) and the server.

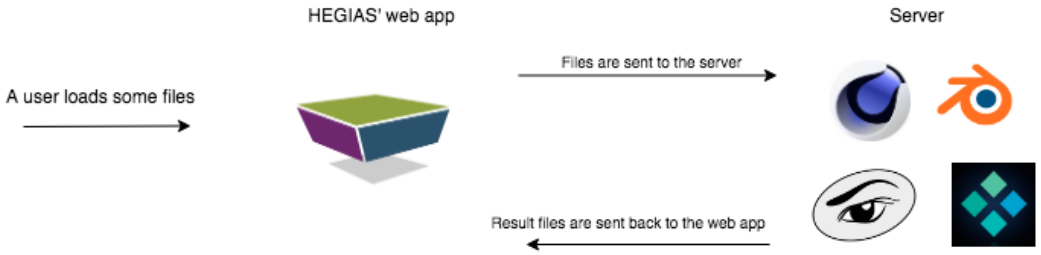


Figure 12. The Hegias' Architecture

A user, within HEGIAS' web app, can choose to upload a 3D scene or an object. The difference is that a 3D scene is composed of many objects, an object is a single entity. Moreover, objects are available on a library an there is the possibility to add in real time (when the user is inside a 3D scene in VR) objects from the library into a 3D scene. Two kinds of uploads are offered, one for a 3D scene and one for an object. After uploading the files into the respective uploader, these files need to be **pre processed** before being ready to be sent to the server. After the pre-processing takes place, files are sent to the server and the **3D processing pipeline can start**. The 3D processing pipeline has some little differences between processing an object and a scene. A single *.glb* 3D file format is generated if a single object has been uploaded within HEGIAS' web app, multiple of them are generated if a 3D scene has been uploaded. In both cases, new texture maps are generated. After the 3D processing pipeline has finished, the resulting files are sent to the HEGIAS' web app to **visualise the results**. A parser reads the resulting files and has the responsibility to correctly load them into A-frame. The parser reads the *.json*, where each node represents an object. With this information, a 3D scene, or an object, is loaded in A-frame.
These steps are schematized below in a more compact and readable way.

1. Pre-processing: file(s) are pre-processed to be ready for the 3D processing pipeline

2. 3D processing pipeline: Optimisation and decimation is performed on an object(s). A *.json* with information about the object(s) is generated.

3. Visualising results: Results from the 3D processing pipeline go through a parser which has the responsibility to upload them into A-Frame

Having a general overview of the architecture, let's now consider the two following example scenarios.

1. A user loads an *.ifc* file of a house.

2. A user loads a *.dae* file of chair with some textures.

In the first scenario, the user uploads a house as a 3D scene in the appropriate uploader within HEGIAS' web app 13). For this example, I took an *.ifc* model from an open repository[16].

[16] `http://openifcmodel.cs.auckland.ac.nz/`

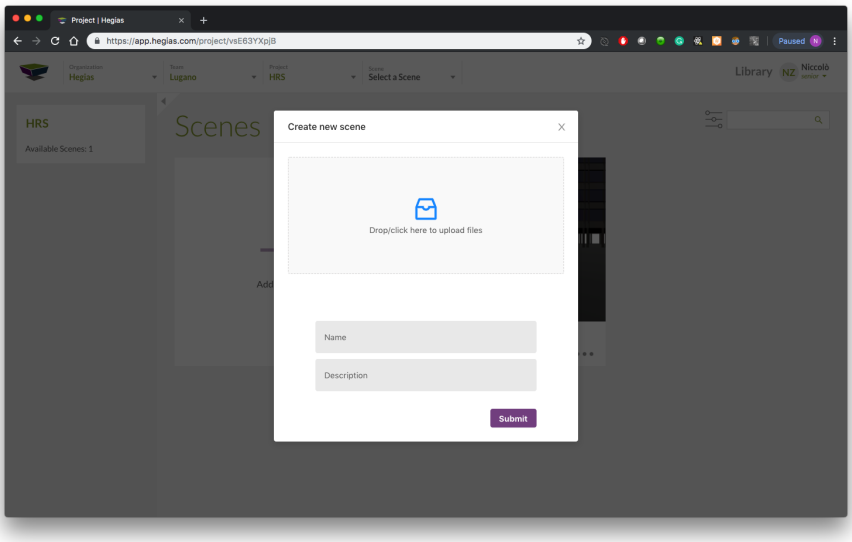
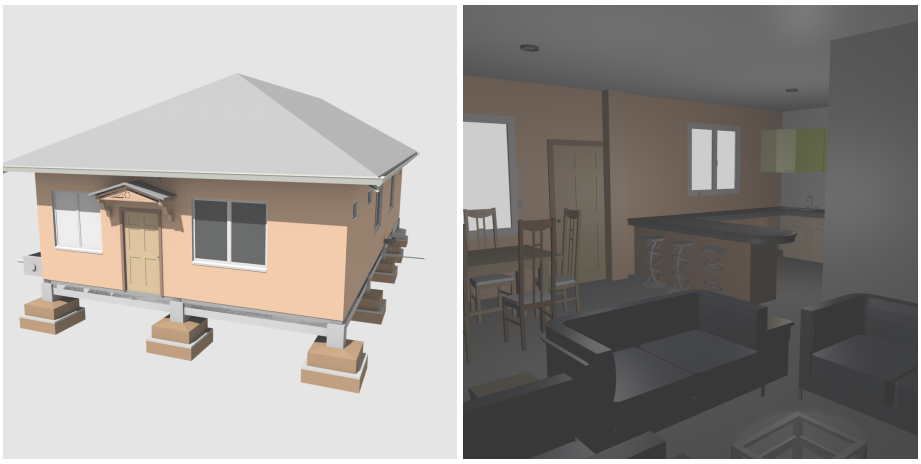


Figure 13. The upload procedure for a 3D scene in HEGIAS' web app

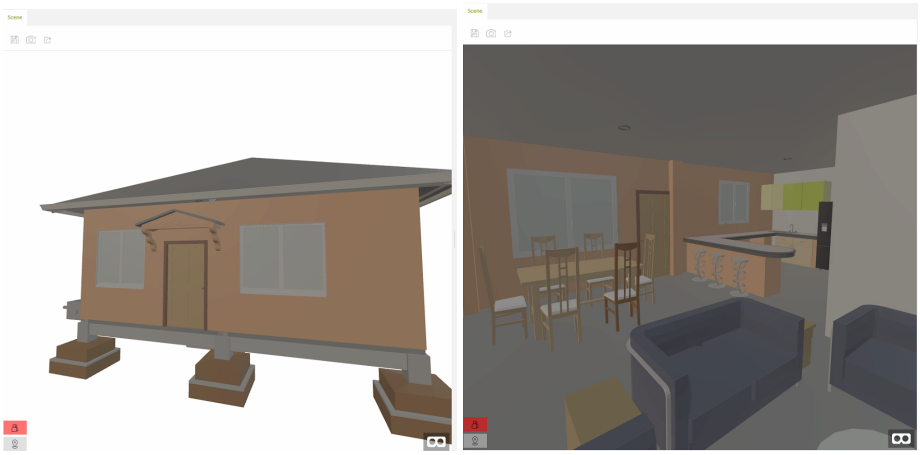
After the *.ifc* 3D format file has being uploaded, it is converted into a supported format (Figure 14). Because *.ifc* 3D file formats do not support textures (Section 2.2) and may have problems in the topology due the conversion, *.obj* 3D file format is a good choice as result from the conversion.



(a) View from the outside (b) View from the inside

Figure 14. Original 3D scene after conversion. Rendered using Blender internal render.

The 3D processing pipeline now takes the *.obj* as input and returns a *.json*, representing the structure of the 3D scene, and a list of files in *.glb* 3D file format; the resulting files are sent to the client and a Javascript parser reads them, building back the original 3D scene (Figure 15).



(a) View from the outside (b) View from the inside

Figure 15. Results in HEGIAS' web app

The second scenario is the upload of an object into the online HEGIAS' web app library (Figure 16) to make it available for being loaded in any 3D scene on the HEGIAS' web application. I choose a model of a desk from a free library[17] of 3D models. The 3D file format is a *.dae* with one texture (Figure 17).

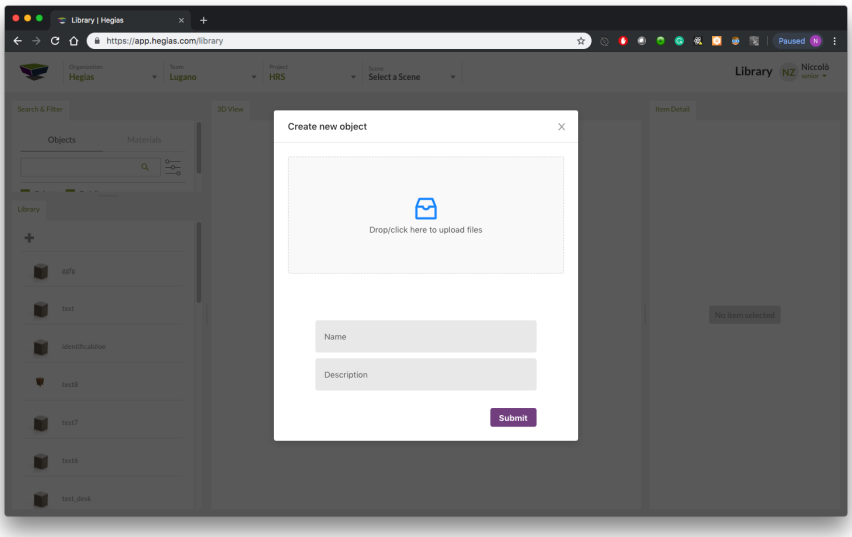


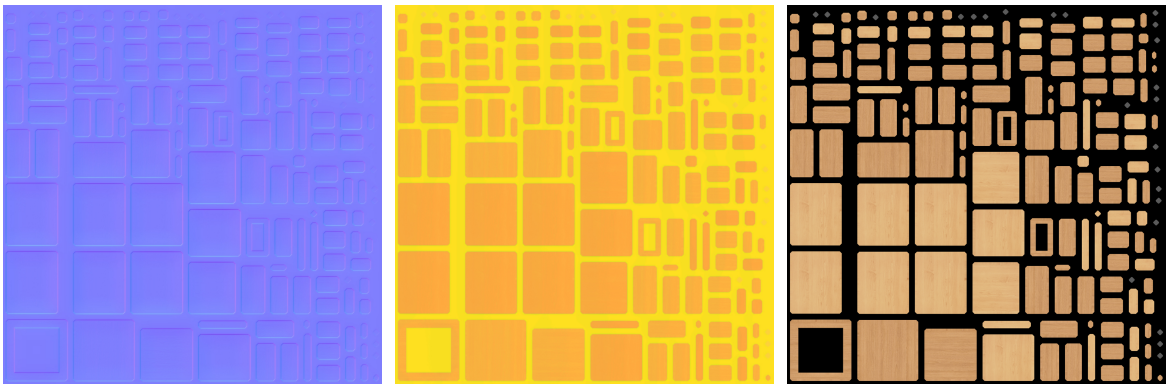
Figure 16. The upload procedure for an object in HEGIAS' web app

[17] https://free3d.com/

(a) View from the outside (b) View from the inside

Figure 17. The original model (on the left) and its texture (on the right)

The result from the 3D processing pipeline in this scenario is slightly different for two reasons. The first one is that the 3D file format has been uploaded as an object, and the second one is that the model contains textures. This object will appear in the middle of the scene, and it will available on the library to be added in a 3D scene. The texture of the object is used to create new texture maps (Figure 18).



(a) Normal map (b) RMA map (c) Albedo map

Figure 18. The resulting texture maps generated from the 3D processing pipeline, going left to right: Normal, RMA(Roughness, Metallic and Ambient Occlusion) and Albedo.

The results from the 3D processing pipeline are a single *.glb* 3D file format, a *.json* containing information about the object and all the new generated texture maps. These files are sent back to the client and made available for visualisation in the library (Figure 19).

Figure 19. The result in the HEGIAS' web app library after uploading an object

5 Implementation

In this section, details and pseudocode about the implementation of the 3D processing pipeline (Figure 20) are shown. The 3D processing pipeline has the role to automatise all the steps and procedure, generally manually done by 3D artists, to make an object ready to be visualised in HEGIAS' web app.

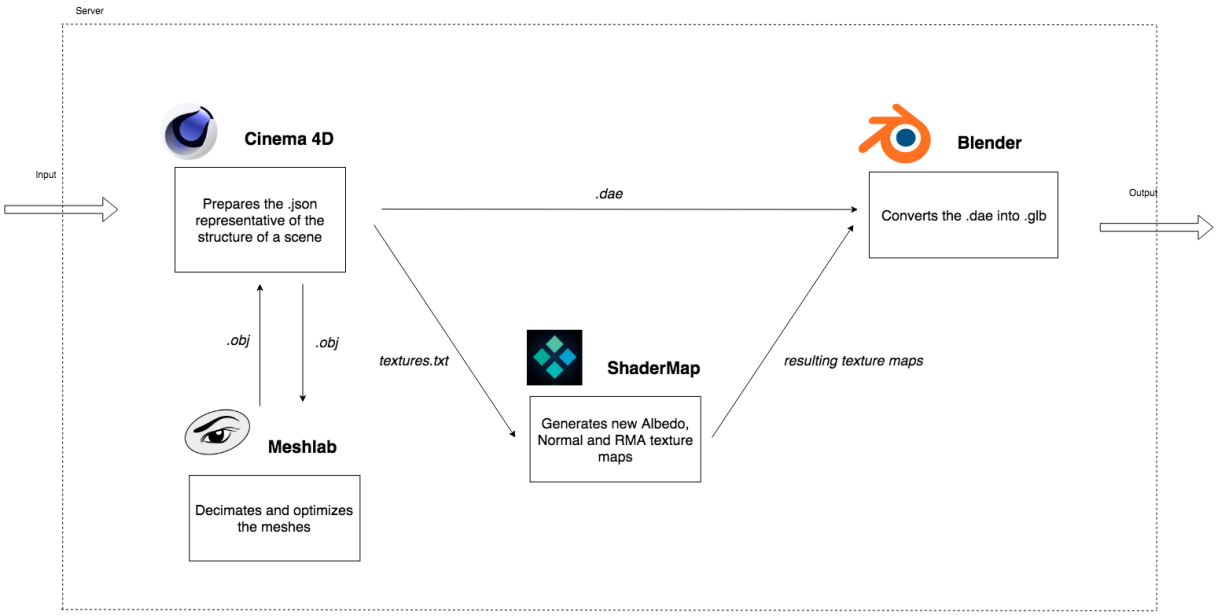


Figure 20. The architecture of the 3D processing pipeline. Cinema4D is the main core and interacts with all the other software. Meshes are sent to Meshlab for optimisation and decimation, textures are sent to ShaderMap to generate new texture maps and Blender takes care of converting the resulting *.dae* 3D file format into *.glb* 3D file format; this is sent back to the server and loaded within HEGIAS' web app.

The section is divided into three parts. The first one covers the pre-processing of files before being sent to the 3D processing pipeline, the second covers the 3D processing pipeline implementation in details and the last is about how to load the resulting files from the 3D processing pipeline into HEGIAS' web app.

5.1 Pre-processing

After a user has uploaded the files she/he wants to visualize in virtual reality, some pre-processing occurs on these files, on the client side. The pre-processing has the role of converting unsupported files into supported ones by the 3D processing pipeline.
The first pre-processing step is to discard invalid and/or unsupported files. Then some conversion occurs. Images in *.tif* and *.tga* formats are converted into *.png*. Because these formats are not much used and they are converted before going through the pipeline, an exhaustive explanation is over the goal of this report. Still, *.tif* is a similar format to *.png* and *.tga* is an old format not much used anymore. It may still be useful in very specific cases, but that is not the case of this project.
The next step is to convert, *.ifc* 3D file format into *.obj* using IfcOpenShell (Section 3.8). As mentioned in (2.2), *.ifc* is a 3D file format meant to be used by architects (i.e. using Autodesk Revit[18]. There is no way to process this 3D file format directly into Cinema4D, Meshlab, and Blender; therefore a conversion to a supported 3D file format is required. Converting *.ifc* into *.obj* using IfcOpenShell generally leads to problems in the topology of the mesh, therefore the 3D processing pipeline has to take care of providing good meshes in terms of topology for any case of input.
As soon as this pre-processing is done, all the files are sent to the server and the 3D processing pipeline can start.

5.2 3D processing pipeline

The 3D processing pipeline is a sequence of scripts written for Cinema4D, Meshlab, Shadermap, and Blender. This sub-section will go through it in detail, both to process a 3D scene and to process an object. The only differ-

[18] https://www.autodesk.com/products/revit/overview

ence between the results from a 3D scene and an object is in the representative *.json* file, for a 3D scene each node will represent an object in it, for an object only one node is present. Moreover, each step done on an object is done for each object in a 3D scene.
This sub-section is divided into four parts. The first one covers Cinema4D, the second one Meshlab, the third one Shadermap and the last Blender. Each software can be thought as an independent module. I structured this sub-section accordingly. In image 20 a schematization of the interactions between each software is shown.
Cinema4D is the main core and the first layer of the 3D processing pipeline. It interacts with all the other software. It sends meshes to Meshlab for optimization and decimating the geometry. It sends the path of textures used in the 3D file to Shadermap to generate new texture maps. And in the last step, it sends the resulting file, in a *.dae* format to Blender to be exported in *.glb*. Cinema4D also takes care of generating a *.json* file with information about the objects in the 3D scene, or of a single object if an object is being processed.

5.2.1 Cinema4D

Cinema4D is the first layer of the 3D processing pipeline. Cinema4D role is to detect objects in a 3D scene, to recompute the origin and gather information of a object and to organise the 3D scene hierarchy in a convenient way inside a *.json* file. Cinema4D interacts with all the other modules. It sends meshes to Meshlab to fix problem in the topology and to decimate their geoemtry. It writes and, eventually, fixes un-correct textures paths in a *.txt* allowing Shadermap to generate new texture maps from these ones, linked by Cinema4D. Lastly, it exports all the unique objects into *.dae* 3D file format and writes their paths into another *.txt* allowing Blender to convert them into *.glb* 3D file format.
If the 3D processing pipeline is processing an object, all the following steps are done only on that object. Instead, if a 3D scene is being processed, first of all, objects within the 3D scene have to be detected. As explained in Section 2.1 an object can be composed of multiple meshes. This means that detecting an object is not as trivial as it seems.
Dividing objects into multiple parts is a really common pattern for 3D modelers, and I wanted the 3D processing pipeline to consider objects divided in multiple parts as a whole object. A convention in 3D modeling, for objects divided in multiple meshes, is to parent them to a null, to keep order in the hierarchy. Given this assumption, the 3D processing pipeline consider objects all the polygon objects (i.e. an object that has geometry) and all the nulls that do not have a null in its children.

Algorithm 1 Detecting Objects with Multiple parts

```
1: procedure DetectObject(object)
2:     if object is a polygon object: then
3:         return true
4:     if object is a null then
5:         childrens = object.GetChildren()
6:         for children in childrens do
7:             if children is a null then
8:                 return false
```

Objects detected using this heuristic which are composed by multiple meshes are joined into a single one. This generally leads to a high number of vertex groups (in Cinema4D, polygon selections). This is solved by joining all the polygon selections sharing the same material into one. An high number of vertex groups will increase the exporting time (from Cinema4D to Blender).
The algorithm to join the polygon selections first creates a dictionary with the material as key and an array of polygon selections as value. The key could be the material itself or the name of it depending on the software. Cinema4D python SDK provides a class for materials, but other software may represent them in different and not hashable ways. In Cinema4D, each material is connected to a single polygon selection. If there are multiple polygon selections, the materials are duplicated, therefore duplicated materials are removed (line 11 of the pseudo-code below) and just a single one is kept.

Algorithm 2 Merging multiple polygon selections

```
1:  procedure MergePolyselection(object)
2:      Generate a dictionary where the key is a material and the keys are the polygon selections
3:      for All materials in the object do
4:          if the material is not in the dictionary then
5:              dictionary[material] = [Material.GetPolySelection]
6:          else
7:              dictionary[material].append(Material.GetPolySelection)
8:              material.Remove()
9:      for each key (material) in the dictionary do
10:         Join the polygon selections
11:         Link the resulting joined polygon selection into the material
```

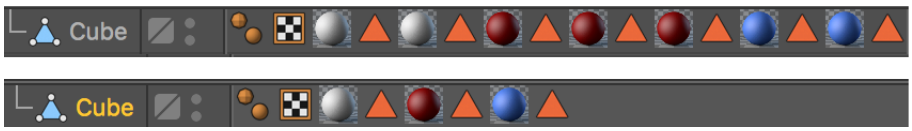


Figure 21. The result of combining polygon selections

After detecting an object, it is checked if this object is already in the database. This is done by a query to the database. If the object is in the database, then there is no need to do anything to it and a reference to the database is written in the *.json*.
Each object not present in the database is sent to Meshlab to perform triangulation, optimization and mesh decimation. After Meshlab has finished, the mesh is imported back in Cinema4D and materials are applied back to it. If an object appears in the scene more than once, it is sent only once to Meshlab.

The object's origin is set in the middle of the bounding box. A bounding box is the smallest rectangle enclosing an object. Moving an object to the center of its bounding box is done to keep consistency between origins. Many objects uploaded by the clients of HEGIAS's, have origins outside the bounding box of the object. This leads to problems in A-Frame's physics engine and when the object needs to be translated and/or moved inside the virtual reality environment.

Figure 22. The bounding box of an office chair. For visualisation, only the edges of the bounding box are shown. The blue dot is the original origin of the object, the red one is its new origin (the middle of the bounding box)

The next step is to write information about each object in the *.json*, which represents the structure of a 3D scene. Each node on it is an object in the 3D scene, containing information about the object such as its position, scale, rotation, its bounding box (bbox parameter) dimensions, and its origin. Below there is the structure of a node in the *.json* file. A 3D scene has many nodes, each one representing an object. If the 3D file format was uploaded into HEGIAS' object uploader web app, the *.json* contains only one node with information about such object.

Algorithm 3 Moves objects origin to the center of the bounding box

```
procedure MoveOriginToBboxCenter(object)
    BboxCenter = objcet.GetBoundingBoxCenter()
    GlobalMatrix = object.GetGlobalMatrix()
    newCenter = GlobalMatrix · BboxCenter
    offset = - BboxCenter
    for all vertices do
        vertex = vertex + offset
    newGlobalMatrix = Matrix(newCenter, GlobalMatrix.v1, GlobalMatrix.v2, GlobalMatrix.v3)
```

```
        {
        "name": string,
            "params": {
                "category": string,
                "objectId": string
                },
            "bbox": {
                "width": number,
                "depth": number,
                "center": [number, number, number],
                "height": number
                },
            "position": [number, number, number],
            "rotation": [number, number, number],
            "scale": [number, number, number]
        }, ...
```

The name is the name of the object, then there are some parameters (params): category, src, and objectId. The category is used for the physics engine, depending on it the physics is dynamic or static. The objectId is the id the object is saved in the database with. There is information about the bounding box (Bbox), used to make a preview in A-Frame of an object being duplicated or added from the library in a 3D scene while a user is in VR. The last information is regarding the position, the rotation and the scale of the object in the 3D scene. If a single object is being processed, the position will be (0,0,0), the center of the scene.

The category is retrieved from the name of the object. A convention is being used to name the objects. There are three kinds of categories: 'standing', 'hanging' and 'attached'. Standing that can move freely, such as items of furniture. Hanging are objects that cannot move freely, such as ceiling lights. The last category, attached, are objects who are connected with an object and cannot move at all, such as a picture on the wall.

The position, rotation, and scale of an object has to be retrieved by applying all model matrices in its way to the root of the scene. A-Frame requires position and scale to be in meters and rotation to be in Euler angles, it is convenient to add them along the path to the root of the 3D scene.

Algorithm 4 Obtaining world position, rotation and scale of an object

```
procedure GetWorldPosRotScale()
    position = object.GetPosition()
    rotation object.GetRotation()
    scale = object.GetScale
    while object has a parent do
        parent = object.Parent()
        position = position + parent.GetPosition()
        rotation = rotation + parent.GetRotation()
        scale = scale · parent.GetScale()
        object = parent
```

This information is gathered for each object and written on each node of the *.json*.

To process the textures in Shadermap, Cinema4D writes all the texture paths into a *.txt*, namely *textures.txt*. If a texture is not linked in any object (the texture is unused) it is not written. Shadermap, after Cinema4D has finished, will process all the texture written in this file.

Cinema4D now is going to export each object into a *.dae* file. Since *.dae* is a neutral 3D file format 2.2 , it is the best way to exchange 3D files from Cinema4D to Blender. The path to all *.dae* exported by Cinema4D are written in a *.txt* file. Every *.dae* linked in the *.txt* file will be opened in Blender to be converted into a *.glb* file.

5.2.2 Meshlab

Each file sent from Cinema4D to Meshlab runs through a script that removes un-referenced vertices, turns eventual non-manifold vertices and/or edges into manifold ones, triangulates and decimates the geometry.
The first Meshlab's step is to remove unlinked vertices and fix overlapping faces. Each non-manifold mesh (Section 2.3), is turned into a manifold mesh. Non-manifold meshes are not problematic for rendering in Virtual Reality, but they may be problematic for physics engines, in HEGIAS' case, for A-frame's physics engine. This is solved by turning non-manifold meshes into manifold ones.
The next step is to triangulate the mesh. It is an important step because, generally, decimation performs better on triangle meshes. Moreover, if no triangulation is performed, the performance in A-frame decreases. Triangulation can be done using Three.js on the client side but this will take more time than doing it on the server. For these reasons, triangulation is done.
The last Meshlab operation is to reduce the number of vertices using Quadric Edge Collapse Decimation.
The last step is to decimate the geometry of the mesh using Quadric Edge Collapse Decimation. The heuristic used as strength parameter of the decimation algorithm is the number of vertices over the surface of a mesh. This heuristic has two main advantages, the first one is that not all objects are sent to Meshlab saving processing time and secondly that the strength of decimating depends on how an object is 'bad'. If an object is really small but it is composed of many vertices it is not a "good" object. The level of detail of an object increases with its number of vertices, small objects do not need a high level of detail.
To give more freedom to the user a parameter (zRatio), which both affects the number of objects to be decimated and the strength of the decimation, is given. The user can set this parameter in the HEGIAS' web app, the higher the number is, the more objects will be decimated and with higher strength.

Algorithm 5 Decimation parameter

```
procedure ComputeDecimationParameter(object, zRatio)
    Compute the area of the bounding box
    number of vertices = len(object.Vertices)
    density of vertices = (number of vertices)/(area of bounding box)
    if density of vertices > zRatio then
        decimate strength = (zRatio / density of vertices)^(2/3)
    else
        Do not perform decimation
```

The user does not have to specify this parameter (the *zRatio*), but it was asked, by HEGIAS' clients, to have control over the decimation procedure. If the user does not specify a *zRatio*, then Meshlab keeps reducing the number of vertices as long as no constrain (the geometry and texture boundaries) is violated on all the objects in the 3D scene. This heuristic is simple, fast to compute and efficient.

5.2.3 Shadermap

Shadermap takes care of improving the appearence of textures. The result is not realistic (in terms of photorealism) but it improves the visual effect of the 3D scene in A-frame.
A script reads all the textures linked in *textures.txt* and through Shadermap generates new Albedo, Normal and RMA (Roughness, Metallic and Ambient Occlusion) maps. The procedure to generate these texture maps is sketched below.

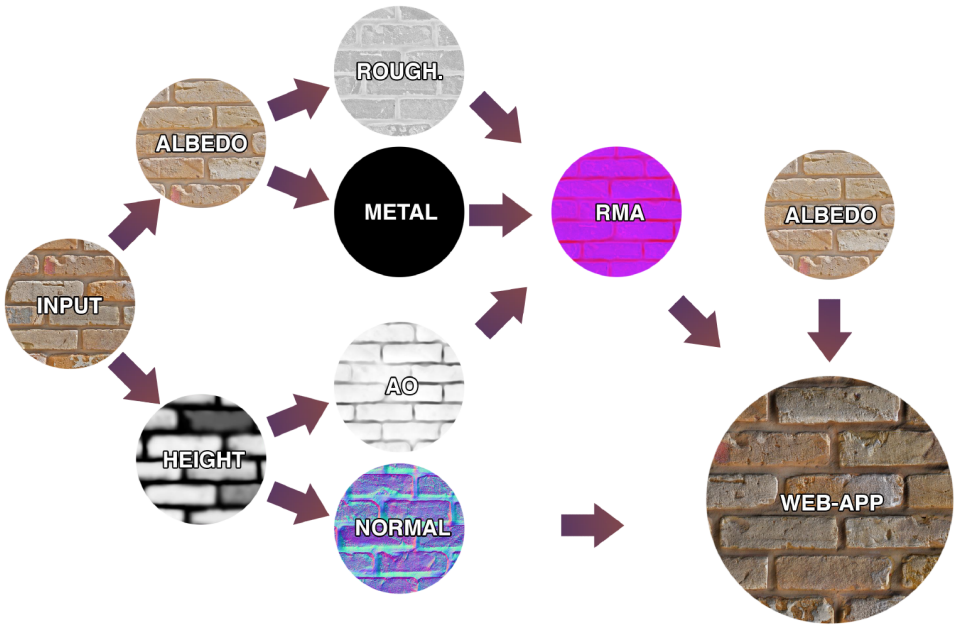


Figure 23. The procedure to generate Albedo, Normal and RMA texture maps in Shadermap. The image is taken from a previous bachelor project by Andrea Franchini

These texture maps will be used in Blender to generate new materials compatible with Blender glTF 2.0 Importer and Exporter (3.4).

It is important to mention that this part of the 3D processing pipeline was developed as Bachelor Project six months ago by *Andrea Franchini*, and it has been fully integrated with the current project.

5.2.4 Blender

Blender has the role of converting the materials into materials compatible with Blender glTF 2.0 Importer and Exporter and to export to .glb format.
A Blender script imports all *.dae* linked in the *.txt* in Blender and all the materials are converted in new materials to be supported for the exporter.
As mentioned in (3.4) Blender Cycles is a node based renderer, the glTF Exporter/Importer partially supports only the Principled BSDF node of Blender Cycles. For each material, a new one is created and all the texture maps from Shadermap (Albedo, Normal and RMA) are linked to it. Once all the materials are converted, Blender exports to *.glb*.

5.3 Visualising results

This section explains how the resulting files are loaded into A-Frame to be finally available to the user in HEGIAS' application.
The results from the 3D processing pipeline are a set of *.glb* files, one *.json* and all the maps generated for each texture. These files are sent back to the client. If the file was uploaded as an object, the resulting *.glb* is saved in the database and available in the library. Then, the *.glb* of the object is added as an entity in A-Frame.
If the file was uploaded as a 3D scene, the *.json* goes through a parser which reads it and builds back the original 3D scene by reading all the objects positions, rotations and scales. If an object is already in the library, the *.json* contains a reference to it and a request is done to retrieve such object.

6 Conclusion

In this report, an architecture to process 3D objects for virtual reality in web visualization is proposed. The 3D processing pipeline presented is currently being used by HEGIAS on its server to support the features offered in their web app. The importance of providing a well structure and to optimise objects in particular through the decimation of the geometry was of primary importance. In the next sub-section some benchmark is given to show the results of the 3D processing pipeline on different 3D files. The work is considered complete, still there are some improvements, presented in the last sub-section, I will do in the future working for HEGIAS.

It is worth saying that the presented architecture could be used in any other application, with the appropriate modifications. For example, it could be used to optimize and to improve textures of objects to be imported in unity[19], or in any other game development platform.

6.1 Benchmark

This section is focused on showing the improvements in the number of FPS in A-Frame, obtained decimating the geometry of the objects and in the loading time of a 3D scene, obtained organizing it in an optimal way. Below, the specification of the computer used for this benchmarking.

Hardware	Model	Memory	Clock	Core
GPU	Nvidia 1080ti	11 GB VRAM	1101 MHz	3548 Cuda Cores
CPU	Intel i7-8700k	-	3.7 GHz	6 core
RAM	Corsair	16 GB	2133 MHz	

The first level of benchmark is a list of 3D files before and after going through the 3D processing pipeline. The number of vertices is shown before and after the decimation of geometry (without any user specifying the strength of decimation).

	Original number of vertices	Processed number of vertices	Percentage of reduction
A house with interiors	946,845	512,444	46%
A building with no interiors	494,884	368,619	26%
A store	1,291,323	878,877	22%
A house with garden	861,793	711,397	18%
A chair	14,017	3,828	63 %
Furniture	9,142	5,617	39%

Looking at the percentage of reduction, it can be seen that it varies from file to file. This depends on how the file has been modeled and, for the *.ifc* format, how well the IfcOpenShell(3.8 did the conversion.

Regarding the loading time of 3D scenes, a similar table is presented. The original loading time refers to loading the whole file, without the *.json* structure file. The processed loading time is loading the file through the *.json*, meaning that each duplicated object is loaded only once.

	Original loading time	Processed loading time
A building with interiors	1m 37s	32s
A building with no interiors	57s	43s
A store	1m 23s	57s
A house with garden	1m 11s	37s

The decrease of the loading time highly depends on how many objects are unique in the scene. If all objects are unique, then the loading time will be the same. This can be seen in the table in the second example, building with no interiors. If the building has no interiors, then it has no furniture, which are, generally, the objects that are mostly duplicated in a 3D scene.

[19] `https://unity.com/`

6.2 Future Work

The project is completed, however some improvements will be done in the future working for HEGIAS.

6.2.1 Multithreading

The 3D processing pipeline is slow. Many operations are single threaded (as instance joining meshes) and there is nothing to do about that. Still, a level of multithreading could be easily achieved by calling more than one instance of Meshlab simultaneously, processing one than one mesh at the same moment.

6.2.2 Hashing

Objects are referenced in the database by name. This is not an optimal solution. An improvement would be to hash the sorted vertices of an object and then reference it by this hash.

6.2.3 Geometry Instancing

Future work to increase even more the performance would be to instantiate the geometry of identical objects in a 3D scene. As said, architectural 3D scenes generally have a lot of objects that are identical, such as chairs, tables, lamps, etc. During this project, objects have been grouped to prevent from loading and retrieving them from the database multiple times. All identical objects share the same geometry, therefore it would be possible to instantiate the geometry, with a technique called geometry instancing (Section 2.7), for each group of identical objects, increasing A-Frame's rendering performance. The basis for this future development have been laid. The *json* structure is made in such a way to support this feature. A level of optimization is already achieved using A-Frame's assets, meaning that in case of multiple equal objects, the object is loaded only once.

6.2.4 Compression

Compressing *.glb* files with Draco[20], an open-source library for compressing and decompressing 3D geometric meshes and point clouds, the loading time of objects will decrease. Compressing the resulting *.glb* files resulting from the 3D processing ipeline would save a lot of time, due to the smaller size they will take less time to be sent back to the client, and as Draco claims in its documentation , their compressed files are faster to load. This will be a nice future development.

[20] `https://google.github.io/draco/`

References

[1] J. J. L. Jr. *A Discussion of Cybersickness in Virtual Environments*. 2000.

[2] T. Jukic. Draw call in a nutshell. https://medium.com/@toncijukic/draw-calls-in-a-nutshell-597330a85381. Accessed: 2019-06-14.

[3] L. Kobbelt, M. Botsch, , and M. Pauly. *Polygon mesh processing*. 2010.

[4] D. P. Luebke. *A Developer's Survey of Polygonal Simplification Algorithms*. 2001.

[5] G. Nalbant and B. Bostan. *Interaction in virtual reality, in 4th International Symposium of Interactive Medial Design (ISIMD)*. 2006.

[6] R. P. Cignonia, C. Montania. *A comparison of mesh simplification algorithms*. 1998.