POLITECNICO DI TORINO

-

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON

Master's degree course in Computer Engineering

Master's Degree Thesis

Implementation and evaluation of 3D graphics compression for optimizing the quality of user experience in networked virtual reality

Supervisor

Professor Andrea Giuseppe Bottino

Associate Supervisors

Professor Guillaume Lavoué
Professor Jean-Philippe Farrugia

Candidate

Fabrizio Lo Presti

A.Y. 2020-2021

Politecnico di Torino – Institut National
des Sciences Appliquées de Lyon

ABSTRACT

Implementation and Evaluation of 3D Graphics Compression for Optimizing the Quality of User Experience in Networked Virtual Reality

by Fabrizio Lo Presti

A thesis on the implementation of a 3D Graphics Compression interface between an already available compression algorithm developed in C++ and Unity. The objective of this work is to design a method to provide better quality of user experience for VR simulation through the utilization of a decoder in Unity to execute at runtime and decompress data while downloading it from a remote server, saving resources from the user's terminal and lowering the complexity of the terminal's computations. Lastly, an evaluation between the state-of-the-art solutions for treating compressed data in Unity and this new method is performed to verify the consistency of the improvement. The thesis explores the key passages needed to create the final model, among them the choice of the right communication protocol for an optimized data transmission, the choice of the most suitable inter-process communication method for data exchange between C++ and C#, the implementation of concurrent programming and the choice of compliant 3D objects to provide a coherent simulation. The final simulation consists of a user-friendly environment (visit to a museum), navigable either via HMD or Desktop VR (WoW), in which heavily dense 3D meshes are progressively retrieved and rendered as they are downloaded from a remote server.

By analyzing the statistics related to the delay of each LOD retrieval the results show that the proposed solution brings an optimization in terms of efficiency in the tangibility of the scene, also providing excellent outcomes on environments with limited network availability.

TABLE OF CONTENTS

LIST OF FIGURES

Number *Page*

LIST OF TABLES

ACKNOWLEDGMENTS

I dedicate this thesis to every person who has been present during my educational path and helped me growing both professionally and personally.

My deepest thanks to my on-site supervisors, Professor Guillaume Lavoué and Professor Jean-Philippe Farrugia, who directed me effectively and carefully through the work, paying particular attention to my ideas and providing me with all the support I needed and even more, considering the peculiar condition in which this work has been carried on.

I would also like to thank my Italian supervisor, Professor Andrea Bottino, for providing me with the guidelines to build and refine this paper and for following my work during the period abroad.

I wish to reserve a special thank you to my family, and especially to my mother and father, who supported me along each step of this journey and have always been present, through good times and tough times.

Finally, I dedicate this work to my closest, irreplaceable and much-loved friends, and to the new ones I made in Lyon, who made me feel at home since the very beginning.

Fabrizio Lo Presti

- *3D Mesh:* A collection of geometric information that describes the shape of a tridimensional object. It is composed of vertices (single points), edges (straight lines that connect two vertices) and faces (flat surfaces enclosed by edges). Meshes are empty volumes, therefore they only consist of the 3D grid formed by the combination of the geometric elements mentioned before. A 3D mesh can be classified as manifold or non-manifold; they are manifold if, for every edge, there are exactly two faces that contain it.

- *Data compression:* process of reducing the size of data to fewer bits while maintaining a coherence with the original information. The compression is either lossy, meaning that some information is lost during the process, or lossless, meaning that the bits are reorganized so that less space is needed to store the same amount of information. Data is often compressed at the source of a transmission or storage process, to lessen the resources that are required to perform these operations. For 3D objects, data compression usually focuses on basic assumptions on the shape of the mesh: the subsequent faces of a certain model are mostly placed in a coherent position with respect to the previous ones, so it is possible to exploit these information to efficiently code the geometry of the mesh. Another important issue to address, when dealing with 3D compression, is the behavior of *textured* meshes when compressed: how the UV information is preserved and if the texture itself may be compressed to further reduce the size of the whole model are some of the questions to answer to optimize this process.

- *.OBJ* : geometry definition file format, it represents the coordinates for each vertex, the vertex normal, the UV position of each texture coordinate vertex and the faces.

  Example of line representing a vertex: v 1.2 0.44 0.12 1.0

  Vertex coordinates are expressed in the form (x, y, z, [w]), with a default value of w of 1.0.

  Example of line representing texture coordinates: vt 0.200 0.800 [0]

  Texture coordinates are expressed in the form (u [, v, w]), with each value varying between 0 and 1; the two optional values are set to 0 by default.

  Example of line representing vertex normals: vn 0.312 0.000 0.312

  Vertex normals are expressed in the form (x, y, z), the normals are not necessarily unit vectors.

  Example of line representing a face: f 4//4 7//7 19//19

  Faces are defined as lists of vertex, texture and normal indices with the format *vertex_index/texture_index/normal_index*.

- *.MTL* : Material Template Library format, always associated to an .OBJ, describes the material properties of objects [9]. The .OBJ file has a reference to one or more .MTL files that define material properties such as color diffusion and others, according to the Phong model [10].

Example of .MTL file format:

newmtl material_0

Ka 1.000 1.000 1.000

Kd 1.000 1.000 1.000

Ks 1.000 1.000 1.000

Ns 1000

map_Kd obj_map.jpg

Each .MTL file can define more than one material with *newmtl* command, and each material may have different properties. An explanation of the basic properties of a material follows: the ambient color is defined using *Ka*, the diffuse color is defined using *Kd* and the specular color using *Ks,* each one of them refers to RGB model with values between 0 and 1. The specular color is weighted through the specular exponent *Ns*. If the material is textured, the definition of the texture maps is done using *map_XX*, accordingly to which material property must follow the texture.

1. Introduction

3D models are getting more complex and detailed everyday thanks to the development of computer graphics applications. This increase of quality, obtained by adding more and more geometric information and appearance attributes (such as texture maps) inevitably leads to the growth of the 3D object in terms of size and to an augmented complexity in reading and managing the model. While this represents a big step towards the final aim of rendering an hyper realistic environment, nowadays applications are mostly based on network communication (i.e. web 3D applications), therefore it may be inefficient to produce large-scale information, due to the fact that the application requires low-latency visualization in order to provide a satisfactory user experience. This issue can be addressed by applying efficient compression techniques that both reduce the storage size and equally organize data so that it could be efficiently read and easily used by the endpoint of the communication.

When choosing to compress a 3D mesh, one must consider different choices: the first one is between single-rate approaches and progressive approaches. The single-rate approach usually gives a very high compression rate, and the decompressed mesh is identical or only slightly different from the original one. Its main drawback is that, in order to decompress and visualize the mesh, all the data must be received at the decompression stage, and if any network issue occurs during the communication, the amount of data received up to that point cannot be used to retrieve a partial model. The progressive approach has two main benefits: it provides a stream of data that consists in a constant refinement of a coarse version of the model, so that the receiver is able to visualize at least a lower resolution model while the communication is ongoing, and it also allows to produce different levels of detail [16], adapting the complexity of the model to the resource capacity of the client.

The main drawback of progressive techniques is the complexity of the algorithms that are needed to implement them, plus, there are still some limitations in the categories of 3D models that can exploit these approaches: as stated by Caillaud et al. [1] most of them only deal with triangular manifold meshes and just a few can compress either polygonal manifold or triangular non-manifold meshes.

Until now, the focus of this research branch has been to develop an algorithm which could provide a progressive codec of 3D meshes in a simple and efficient way. The main objectives of this thesis are to evaluate the performance of the lossless progressive compression algorithm provided in MEPP2, a platform developed by LIRIS (Laboratoire d'InfoRmatique en Image et Systèmes d'information) to manage 3D models, and, finally, to design and implement a VR simulation via Unity, in which the compressed objects are retrieved from a remote server and made available to the user as soon as enough data is ready. The coarse version of the object will be in sight in a very short time and as the data flows from the server to the client, the 3D mesh will be more and more detailed up until its original complexity.

In 2005, two reviews about the innovative work on mesh compression were published. Since then the 3D mesh compression was limited to a very specific set of meshes, it involved single-rate compression only and didn't take into account large mesh compression or random accessible algorithms, in which only some requested portions of the input mesh are compressed. The future study of these algorithms set the base for *progressive* random accessible algorithms, capable of decompressing different parts of the input at different levels of detail.

Compressing a 3D mesh is different from encoding known structures as sound and images: the model is not certainly regular; thus, a mesh encoder must work on the structure itself and encode the connectivity instead of the geometry. The general assumption behind the connectivity compression techniques is to perform a traversal of the mesh and emit symbols that vary on the patterns met.

One of the first work in mesh compression is the generalized triangle mesh format of Deering (1995), which exploited generalized triangle strips, a mesh representation useful to transfer data from CPU to GPU in an efficient way: a triangle strip is a sequence of vertices where each additional vertex describes a new triangle with the two former vertices of the strip. This is considerably more efficient than the standard index representation, in which, for coding one triangle, the three vertex indices are needed. Deering also noticed that many of the internal vertices are encoded twice, so he proposed to push their positions in a queue and refer them by their position in it; the experiments showed that, even with this configuration, Deering's algorithm achieves compression rates from 3.3 to 9.8 bpv (bits per vertex).

Another development that is worth mentioning is the one provided by the algorithm of Touma and Gotsman(1998), based on valence encoding: a manifold triangular mesh has approximately twice less vertices than triangles, so creating an algorithm that generates one symbol per vertex to describe its local connectivity is more efficient than a triangle traversal approach. By encoding the list of vertex valences with an entropy encoder, they obtained a compression rate of 2.3 bpv, which is still today an admirable result in the world of connectivity compression methods.

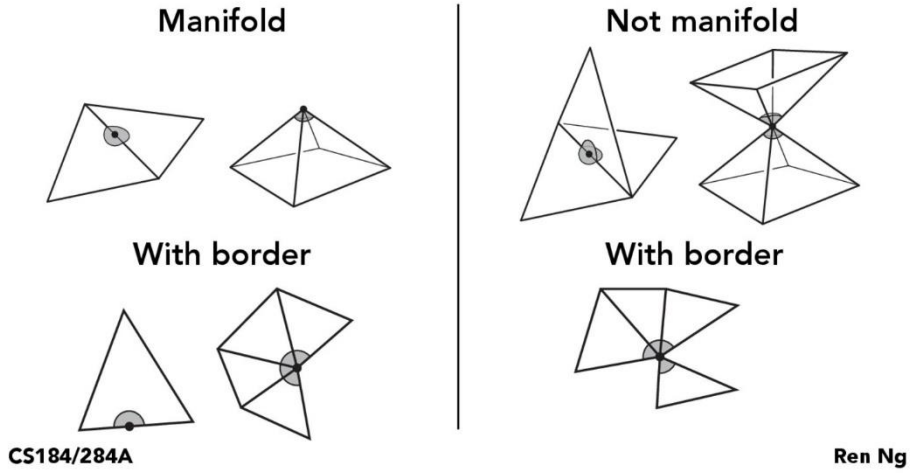

Figure 1. Visual example of manifold/non-manifold meshes. Image retrieved from [2]

In 2001, Ho et al. proposed a solution for handling large meshes that could not fit into the main memory. This method consisted in partitioning the input model and compress each part independently. In 2003 Ueng came up with a recursive technique in which the mesh connectivity data is divided into blocks called octans, with an octree data structure based on the geometry.

The techniques that have been analyzed so far are lossless, meaning that the reconstructed data is the same as the original, but some applications didn't require to rebuild the initial connectivity of the model after decompression, for this

purpose a set of algorithms that further enhanced compression rates, exploiting this new degree of freedom, were developed: the standard mesh simplification that can be found in almost every 3D manipulation program is based on a lossy compression.

For what concerns progressive mesh compression, the first time the concept of progressive mesh has been introduced was in 1996. The researcher Hoppe built the idea of it around an incrementing decimation of the model using the edge collapse operator [3], driven by an optimization formula that minimizes an energy function. The main strength of this algorithm is the possibility to select the amount of refinement during the decoding phase, but the main drawback is a very low compression rate (37 bpv using a 10-bit quantization). To address this disadvantage, some algorithms in which the vertex split operations are performed in batches were developed.

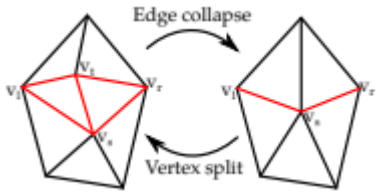

Figure 2. Edge collapse and Vertex split. Image retrieved from [3] page 39:16

Another progressive compression technique is based on vertex removals, followed by a local patch retriangulation; the first time this method has been adopted was in 1998 by Li and Kuo, alongside with the idea of adapting the vertex quantization along the different levels of detail.

As of today, due to the necessity of compressing larger meshes and the limitations on the delay for streaming applications, two new branches of 3D mesh compression are being developed: random accessible mesh compression (used to decompress only a part of the mesh), which could be either single-rate or

progressive, and for which few approaches have been proposed, and dynamic mesh compression, for which the current trend is to consider local scalable methods, that fit well with low-delay applications[3].

Another important mention for the purpose of this work is Draco[1], an open source library for compressing and decompressing 3D meshes developed by Google. Draco uses a single-rate approach to process the meshes, with very low compression-decompression delay, furthermore, several parameters can be customized to achieve the desired compromise between efficiency and cost. This library is released as C++ source code and it also provide a Unity plugin to easily perform the decompression inside the platform.

The compression algorithm that will be analyzed and used in this work has been developed by Ho Lee, Guillaume Lavoué and Florent Dupont. It consists in a lossless progressive compression algorithm based on R-D optimization, that is suited for non-manifold meshes with color attributes. The quantization precision is adapted to the intermediate levels, and can be chosen between two tactics: the first one, more precise but computationally heavier, consists in a mesh distortion measurement; the second one, suited for applications in which the computational time must be limited, is a quasi-optimal decision based on the analysis of the complexity of the model. For the purposes of this work it is not necessary to further explain the inner mechanisms of the algorithm, which can be found in [4], but it is worth to notice that the results show an outperform with respect to the state-of-the-art algorithms regarding colored meshes, and a very satisfactory result with respect to the most efficient algorithms for non-colored meshes.

The first Unity simulation to test the quality of the compression algorithm and to evaluate the impact on the application's usability has been developed by former

[1] *https://github.com/google/draco*

researchers in LIRIS. Their work consisted in an open space environment in which the player could move around freely and select a 3D model that would be created from his HTC Vive controller. The retrieval of the object was implemented with an FTP communication protocol, the data transfer was between the client a private server inside the laboratory. The most noticeable issue was that, during the time in which the object was downloaded from the server, the whole application froze, and the user could not move or interact in any way with the environment. This problem was related to the lack of multiprogramming.

The work of this thesis starts from analyzing the code of the previously mentioned application and addressing the main issues. Consequently, the focus will be to develop a new Unity program that could easily provide both objective quality data for the evaluation of the progressive compression algorithm and subjective quality data for the user experience in a VR/WoW environment in which the object appear more detailed as soon as enough information is available.

3. Environment setup

The idea behind this project is to create a museum environment, in which the player could move and take a closer look to the set of art pieces that are rendered in the scene as they are downloaded from a remote server. To achieve so, it has been selected, among a group of possible alternatives, an asset[2] from the Unity Asset Store that fits as the right compromise between its price and its quality, in terms of visuals and orderliness.

This chapter explains the environment setup for the Unity simulation. The application can be used as a Window on World (WoW) [11] with mouse and keyboard inputs, or via Immersive Virtual Reality. For this purpose, and for the available technologies at the laboratory, the headset that has been used is the HTC Vive.

The HTC Vive is a tethered VR headset, therefore, the computation needed for the VR application are performed inside the computer, making its architecture become the bottleneck of the. On a brighter side, the movement limitations caused by the wiring do not result in a tangible issue because the player is not supposed to perform complicate gestures or motions in order to explore the museum.

The headset is configured using VIVEPORT™ Software, and every input of the controller is customizable through the application *SteamVR*.

To obtain an adequate level of immersivity it is necessary to choose between a roster of metaphors of navigation, that are models to explore the Virtual Reality environment. For this purpose, two choices have been made:

[2] https://assetstore.unity.com/packages/3d/props/interior/gallery-showroom-environment-92146

-The first one is Teleportation [12], with this the user can quickly teleport in a selected area by pointing the controller towards a Teleport Area (an object to whom the script "TeleportArea" is attached) while holding the Trigger button; once released, the player will immediately move to his new position. While this metaphor of navigation is very simple, understandable and fast, it could lead to a very common problem in VR simulations: motion sickness. Motion sickness is experienced when there is a contrast between the movement perception in the human body (sight, vestibular system and somatosensory system) and the external (mostly visual) stimuli that are provided by the virtual environment. In this specific case suddenly switching position without a proper transition, i.e. fade-in/fade-out, may cause nausea, confusion and disorientation. This metaphor has been implemented by using the SteamVR asset in Unity.

-The second one is Directional movement using the Trackpad. This metaphor is less immersive than teleportation, but it has been implemented to address the motion sickness and to allow the user to move smoothly inside the museum. This metaphor has been implemented by customizing the task of the Left Trackpad via SteamVR.

During the first phase of development the idea was to start the download of the 3D model once a virtual button, close to the pedestal where the art piece was supposed to appear, was pushed. This could be accomplished whether using the controller through a selection metaphor (in immersive VR) or by pressing the "I" keyboard button when the player is in the proper range (in Window on World). When the object is hovered, its silhouette is outlined by the means of the *Outline* script attached to it.

The Virtual Hand [13] has been the selection metaphor chosen for the purpose: the user could reach the button moving the controller (which had the resemblance of a hand in the VR environment) and then press the Grip on the controller to

push it. The main drawback of this implementation, despite its simplicity, is the object reachability: to properly activate the interaction, the virtual hand needs to be as close to the button as it would be necessary on a real scenario. The natural manipulation has not been an issue since there was no actual manipulation other than the event of pushing the button.

For what concerns the Desktop VR, the navigation has been implemented through a customization of the *FPSController*, an asset available in the Unity Standard Assets library. The interaction with the environment occurs by using the *Raycast* [14] technique: the camera, representing the sight of the player, casts a ray that is orthogonal to its plane of view, characterized by a maximum length and colliding against all the colliders in the scene. When the ray hits an interactable object the object will be highlighted, and if the "I" key is pressed, the event will occur.

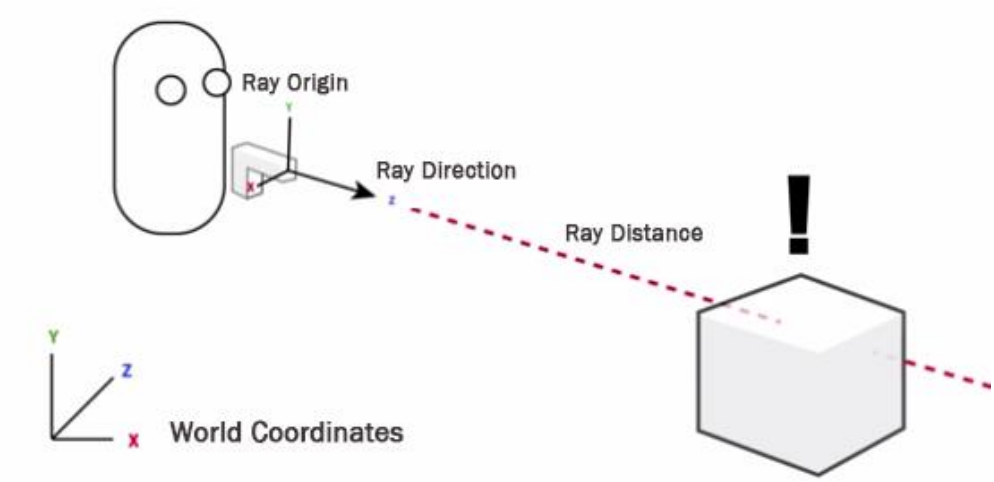

Figure 3. Visual explanation of Raycasting. Retrieved from[3]

The next chapter will discuss about the mechanism that takes place once the user interacts with the pedestal.

[3] *https://medium.com/@miguel.araujo/raycast-what-the-hell-is-that-6d36b3c8dd8b*

4. Analysis of uncompressed data retrieval

For the first part of the work a collection of test models has been retrieved locally and rendered in the scene at runtime: the format of these models is *.obj*, paired with an *.mtl* file which contains the material and texture information (see appendix *a* for more information). The functions needed to import the file are available in the *ObjReader DLL*[4]. *ConvertFile*, which performs a conversion from an *.obj* file (present into the project folder under a specified path) into a Unity mesh that will be pictured inside the scene, has the following parameters: the object file path, a Boolean that let the function know if a material will be used, and finally the material itself. The first experiment was performed with object retrieved a local project folder. The sample objects provided inside the assets plus a set of objects from the laboratory database have been used to test the velocity of the algorithm:

File	**Size**	**Rendering Time**
Napoleon_obj.obj	651879KB	2m 56.73s
Dragonfly_obj.obj	199413KB	1m 4..51s
Bugatti_obj.obj	84159 KB	22.62s
BorderlandsCosplay_obj.obj	56667 KB	15.78s
Frog_obj.obj	26023 KB	8.3s
Car_obj.obj	1808 KB	1.25s
Pig_obj.obj	347 KB	0.88s
Spot_obj.obj	265 KB	0.85s

Table 1. Local .OBJ retrieval results

[4] https://starscenesoftware.com/objreader.html

Since the principal issues with the previous work were the lack of multithreaded programming and the prohibitive timing for a real-time application, before collecting data on the performances given by downloading and processing the uncompressed data at runtime, it was necessary to choose the right network protocol to minimize the delay and to implement a concurrent programming solution. All of the following tests in this chapter have been performed via Wi-Fi connection to the roaming service *eduroam* (the rendering time may significantly lower if a more performant internet connection is used). The first experiment regarding remote communication protocols has been made following the former work's choice and results have been compared to the local retrieval simulation. The objects have been uploaded in the laboratory's private FTP server using *Filezilla*[5]. The functions available inside the *System.Diagnostic* library have been used to measure the latency between the input of the user and the end of the execution.

File	**Size**	**Rendering Time**
Napoleon_obj.obj	651879KB	14m 23.36s
Dragonfly_obj.obj	199413KB	5m 13.21s
BorderlandsCosplay_obj.obj	56667 KB	1m 57.8s
Frog_obj.obj	26023 KB	43.3s
Car_obj.obj	1808 KB	33s
Pig_obj.obj	347 KB	24.9s
Spot_obj.obj	265 KB	23.5s
Dog_obj.obj	1KB	14.88s

Table 2. Remote .OBJ retrieval results (FTP)

[5] *https://filezilla-project.org/*

It has been noticed that, despite the different sizes of the objects taken into account a fixed delay was present: the File Transfer Protocol is a connection-oriented protocol, so the client must perform a TCP handshake with the FTP server before starting to communicate, causing a flat delay in the operation.

The choice therefore moved to HTTP because, being a connection-less protocol, the delay caused by connection establishment mechanisms is absent; additionally, this protocol is widely used and a consistent number of supporting libraries (both in C# and C++) are available.

File	**Size**	**Rendering Time**
Napoleon_obj.obj	651879KB	13m 36.76s
Dragonfly_obj.obj	199413KB	4m 51.21s
ClassicSideTable_obj.obj	70656KB	2m 19.93s
TigerFighter_obj.obj	58675KB	1m 9.63s
BorderlandsCosplay_obj.obj	56667KB	1m 36.85s
Deathstroke_obj.obj	55398KB	1m 16.57s
Giraffe_obj.obj	26215KB	42.88s
Frog_obj.obj	26023KB	29.79s

Table 3. Remote .OBJ retrieval results (HTTP)

For what concerns the texture inside the *.mtl* file, the experiments showed that the amount of time needed to apply the material to the object within the *ObjReader* script is irrisory with respect to the total computation time, thus, for every experiment until here, the texture files are picked up from a local folder in the Unity Project.

With the obtained results, taken into account the non-deterministic behavior caused by the workload variance on the CPU and the GPU and by the high variability of network traffic, it is possible to compare the trends of the algorithm's performance for local import, remote download via FTP and via HTTP:

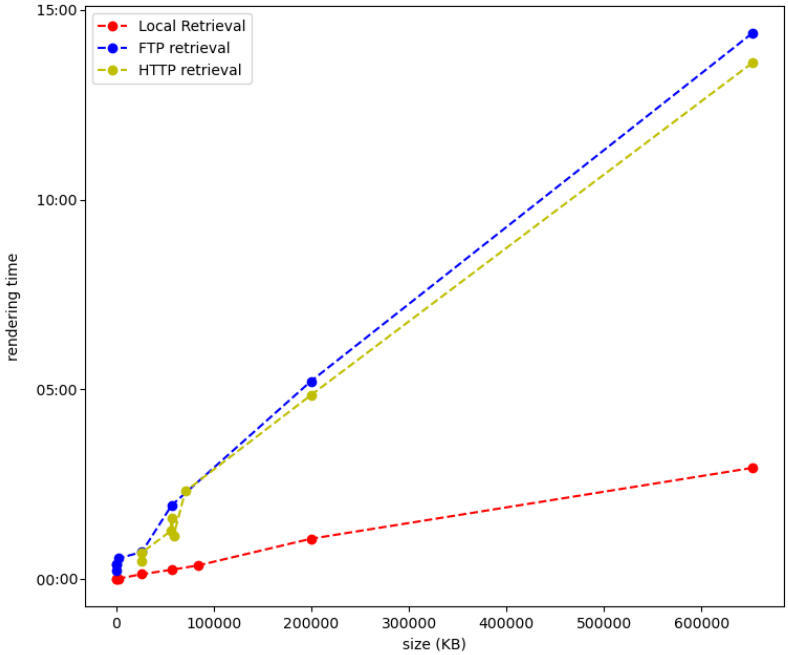

Figure 4. Graphs comparison of local .OBJ retrieval, FTP download and HTTP download.

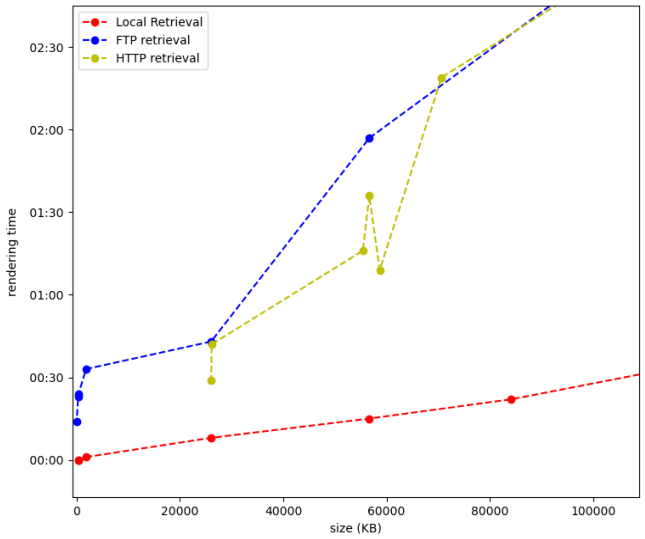

Figure 5. Closeup on smaller values from graph at Figure x.

The resulting trend is as expected: noting that the fluctuation for lower size values is caused by the variability of the experiment itself, the HTTP protocol provides a flat reduction of the time needed to render the mesh inside the scene.

The two remote retrieval simulations result in a drastic increase of time for the visualization of the meshes with respect to the local retrieval, and they are not suitable for being used in a real-time application; furthermore, despite the fact that the growth trend of the local simulation is not sharp, the first scenario is also not suitable for a real-time application, since the size of the .OBJ file is still too large to allow a smooth runtime visualization. This scenario is also weighed down by the impossibility of performing any action during the importation, so the next chapter will focus on implementing a multithreaded environment to address this issue.

As specified in Chapter 2, one of the most critical points to be addressed is the implementation of multithreaded programming. A program is considered *concurrent* when there are two of more simultaneous execution flows in the same addressing space that work to achieve a common goal. The available cores of the CPU are divided and assigned from the scheduler in a non-deterministic fashion. The main advantages of this approach are the superimposition between computation and IO operations, the reduction of the complexity derived from the process communication and the effective use of the multicore CPU; on the other hand, the program is more complex, it is necessary to coordinate the memory access, there are new sources and new typologies of errors and the execution is non-deterministic.

The goal of this task is to allow the player to move around the environment and perform other actions while the application computes the occurred calculations for rendering the 3D meshes. A few different approaches have been tested before the final, optimal solution:

The first approach has been to exploit the well-known low level structures for synchronization: as soon as the client requests the 3D object, the main flow creates and launches a thread who is devoted to the import operation, while the main thread still manages the movement and the interactions of the user. The mechanism for letting the main thread understand when the importation was concluded was a semaphore: a semaphore is a low level synchronization primitive that restricts the number of threads that could concurrently access to a resource; it has two functions and a counter that usually starts at 1: the first function is called by a thread who wants to access to the common resources and, if the counter is different than 0, it decreases its value and access the data; the other function is

called when the thread has finished its operations with the common resources, and it increments the counter value by one. The secondary thread would acquire the semaphore and populate the empty object with the requested one, while the main thread would wait for the semaphore to be available and, subsequently, render the object in the scene.

The solution above was incomplete: even though the two tasks are managed in two different working threads, the main one is forced to wait for the completion of the second one and no action can be performed while the computation is ongoing. It has subsequently been chosen to use Coroutines: a coroutine is a function that can pause execution and return control to Unity and then continue where it left off on the following frame[6]. The point at which execution will pause/resume is the *yield return x* and any variable or parameter will be preserved between yields. For this purpose, two coroutines have been used: the first one (*Load()*) loads the object by calling the methods of the *ObjReader* class and sets the value of a Boolean global variable, representing the availability of the object, to true; the other one (*check()*) is a simple implementation of polling, a cyclic verification of the state of an event. The main disadvantage in the polling approach is the *busy wait*, that causes unnecessary computations for the CPU.

The solution that has been selected in the final version is using *UnityEvents*: they are a way for efficiently allowing user driven callback to be persisted from edit time to run time[7]. This approach leads to a better result because there is no polling mechanism, as soon as the event is triggered the needed functions are called. In the next chapters it will be explained the usage of these structures for the simulation.

[6] *https://docs.unity3d.com/Manual/Coroutines.html*

[7] *https://docs.unity3d.com/Manual/UnityEvents.html*

6. Evaluation of Draco performance

Before evaluating the performances of the progressive compression algorithm developed by the LIRIS team, it has been proven useful to perform a deeper analysis on one of the most performing single-rate compression algorithms available. The main reason why this approach has been a consistent portion of this study is because it has been possible to create a simple and explanatory version of the final product.

The Draco encoder has a set of parameters for customizing the compression:

- quantization parameter (*qp*): it represents the number of bits for the position attribute.

- quantization for textures (*qt*): it represents the number of bits for the texture coordinate attribute.

- quantization for normal (*qn*): it represents the number of bits for the normal vector attribute.

- quantization for generic attribute (*qg*): it represents the number of bits for the generic attribute.

- compression level (*cl*): it represents the compression level, with 10 as the most compressed and 0 as the least compressed.

Different compression levels mean different algorithms for compression techniques such as entropy coding. It is possible that the maximum compression is achieved by means of arithmetic coding and the fastest compression is achieved by means of Huffman coding. The different quantization values are strictly related

to geometry data: a 3D grid incorporates the mesh and all the points belonging in one of the cells are collapsed into one and will be mapped with a specific value of the index. The denser the grid is the more information are coded.

For the purposes of this work it has been chosen a value of *qp* equal to 16, which has proven to be the right compromise between visual quality (no visible compression artifacts) and compression efficiency, and the result have been satisfactory: there was an increase in speed of about 24:1 (what is locally loaded in roughly 24 seconds is now both loaded from the File System, decompressed and displayed at runtime in about 1 second).

Draco's Unity plugin does not include a script for downloading *.drc* files from internet, so a method that downloads the file from via *http* in the form of a bitstream (*.bytes*) and acts as an interface between the data stream and the decompression algorithm has been implemented. The obtained results were also positive: the gain obtained for downloading, decompressing and displaying a *.drc* object on a web server, compared to the same process for an *.obj* file, is equal to approximately 96:1, considering that the size of the compressed file is, on average, around 50 times smaller than the original.

The tests concerning the http retrieval of Draco compressed meshes have been carried out with a cabled connection and a nominal bandwidth of 1Gbps.

The results table follows:

File	Size (Uncompressed)	Size (Texture)	Size* (Draco)	Time to Encode	Local Import (Draco)	http Import** (Draco)
Napoleon_obj.obj	651879KB	17612KB	9164KB	11.26s	5.69s	5.86s
Dragonfly_obj.obj	199413KB	25292KB	4300KB	9.24s	2.54s	2.69s
ClassicSideTable_obj.obj	70656KB	6768KB	1771KB	2.3s	1.08s	1.11s
TigerFighter_obj.obj	58675KB	2846KB	1116KB	1s	0.68s	0.8s
BorderlandsCosplay_obj.obj	56667KB	9758KB	1085KB	1.72s	0.6s	0.95s
Deathstroke_obj.obj	55398KB	6656KB	1054KB	1.7s	0.77s	0.85s
Giraffe_obj.obj	26215KB	4587KB	497KB	0.46s	0.35s	0.58s
Frog_obj.obj	26023KB	4505KB	537KB	0.45s	0.36s	0.59s

*Quantization parameter qp=16 bit; **tested on different connections, result may vary on network state

Table 4. Local and remote retrieval of *.drc* files

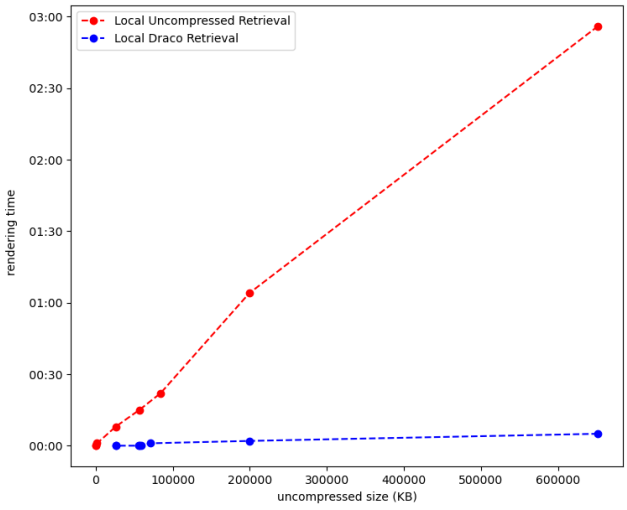

Figure 6. Graphs comparison of local uncompressed and *.drc* retrieval

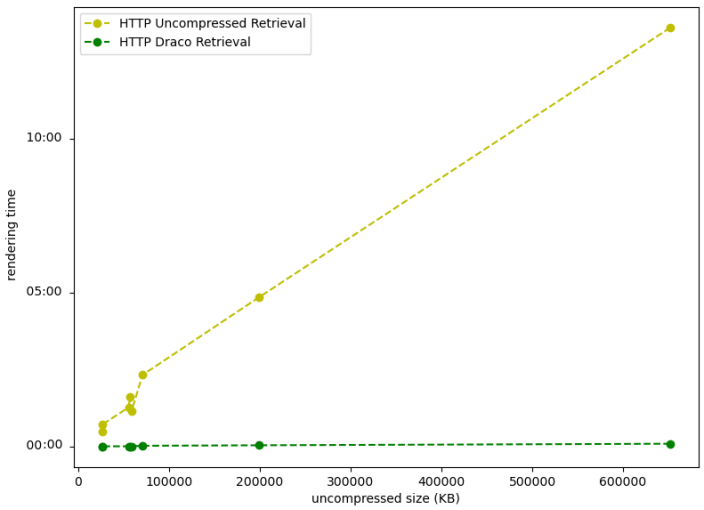

Figure 7. Graphs comparison of remote uncompressed and *.drc* retrieval

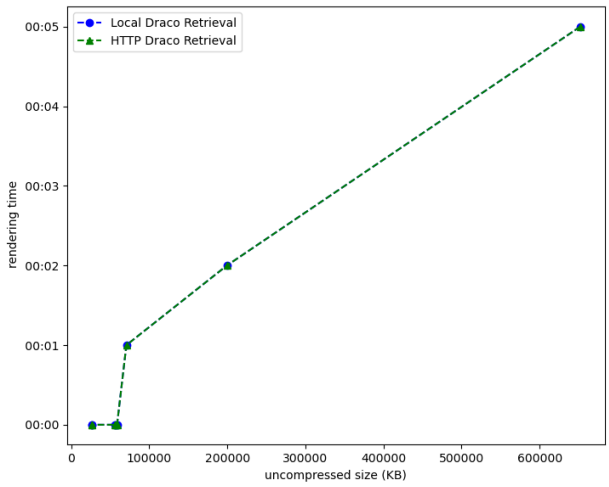

Figure 8. Graphs comparison of local and remote *.drc* retrieval

The results show that there has been an enormous improvement with respect to the retrieval time for uncompressed objects: the object is rendered in the scene quickly and the difference with the original version are not noticeable. As said before, this experiment has been performed with a highly performant cabled internet connection; in this scenario, a compressed object with a size of approximately 9 MB is rendered in the scene in circa 6 seconds. This is a grand improvement with respect to the roughly 13 minutes of the same mesh without any compression applied, but, for bigger meshes (and for different network conditions), this outcome will not scale well, since it is necessary to wait for the whole data to be downloaded to finally have the model inside the simulation. This result is amendable with the exploitation of progressive codec techniques, that will be discussed in the next chapter.

MEPP2[8] is a C++, cross-platform, software development kit for processing and visualizing 3D surface meshes and point clouds. The program is structured so that it is possible to use both an application programming interface to create personalized *filters* and use them without accessing to the program interface itself, or it is possible to implement a *plugin* into the graphical user interface to produce results inside the application. MEPP2 has been built on *Qt*, *OpenSceneGraph*, *Boost* and *Eigen*, and optional dependencies include *FBX*, *Draco* and *Clmg*.

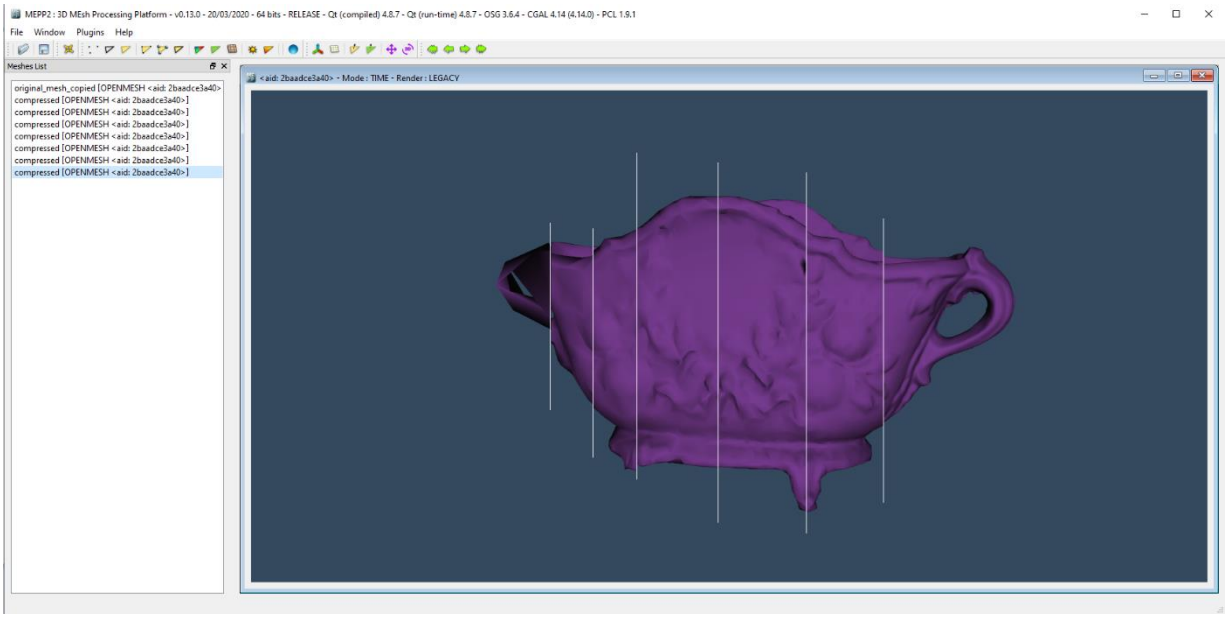

Figure 9. MEPP2 Interface: flowerpot compressed at different Levels of Detail

As specified in [4], the algorithm extends the concept of the valence-driven single rate approaches for progressive encoding. The base algorithm proposed by Alliez and Desbrun [5] consecutively applies two vertices conquests which remove a set of vertices, generating the different LODs. The vertex conquest is strictly connected to vertex decimation and retriangulation to maintain the shape of the

[8] https://projet.liris.cnrs.fr/mepp/mepp2/index.html

original mesh as faithfully as possible. Two types of conquest are distinguishable: the *decimation* conquest travers the mesh in a deterministic way and, as the valence code of the current front vertex is inferior or equal to 6, it is removed and a retriangulation occurs. The *clean* conquest is similar, but the threshold is set at 3 instead of 6. For what concerns the geometry coding, first a global and uniform quantization is applied to the coordinates, after that, assuming that the mesh is smooth and regular, the resulting coordinates of the vertex conquest are predicted from the average position of the 1-ring neighboring vertices and the difference is encoded. The further development of this base algorithm has led to some improvements inside the compression plugin: the quantization precision is fitted to each LOD in consonance with its complexity to optimize the rate-distortion performance. It is important to state that the choice of the quantization parameter is extremely important when dealing with high resolution meshes, in fact, many artifacts are visible when a limited number of quantization bits are chosen for the compression of a 3D model with a large number of elements; on the other hand, the visual distortions caused by the same number of quantization bits on the same mesh with lower resolution are hardly noticeable. A visual example follows:

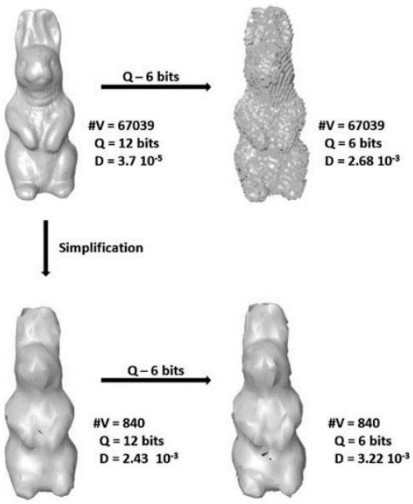

Figure 10. Model at different resolutions behave differently when compressed with the same quantization precision. Retrieved from [4]

This examination demonstrates that each intermediate mesh that is the result of the compression can be quantized at different precision without compromising the geometry quality. Since the quantization parameter has to be chosen for each LOD, the next operation between vertex decimation and adjustment of the quantization parameter must be optimally determined at each iteration, furthermore, the decrease of the parameter has to be efficiently encoded to avoid a significant overhead that could slow down the computation. The main drawback of this approach is its high computational time, so it is possible to use another faster sub-optimal approach: the basic assumption on which this approached is built is that a single global optimal quantization precision exists for each iteration independently of precedent operations, and it is possible to calculate that analyzing the geometry properties of the mesh, the more complex it is the more precision is needed; this operation takes into account the volume of the bounding box surrounding the mesh, the surface area and the total number of vertices. The sub-optimal quantization precision is determined as:

$$q_{so} = \text{round}\left(-1.248 * \log\left(\frac{volume\ of\ bounding\ box}{surface\ area * number\ of\ vertices}\right) - 0.954\right)$$

Retrieved from [4]

The algorithm calculates q_{so} at each iteration and if the current number of quantization bits is higher than this value the decrease of quantization precision is performed, otherwise, the decimation is performed.

MEPP2 supports various 3D data types including static, dynamic meshes and point clouds, furthermore, it integrates different data structures. For the purpose of this work the *OpenMesh* structure has been chosen, considering its efficiency and simplicity in manipulation and processing 3D data.

The core of the platform is the *Face Edge Vertex Volume* template library, which offers an abstraction layer over the different 3D data types. The libraries that provide the key features to process the mesh are *CGAL* and *Boost*: the first one provides vertex and edge manipulation features, while the second one extends these concepts with half-edge and face features. [6]

As previously stated, MEPP2 provides an interface using plugins, that are selected at compilation time and automatically loaded at runtime. A plugin incapsulates a filter, that performs the desired operation on the mesh.

The first task that has been performed using MEPP2 is the creation of a filter that retrieves the compressed object from the network and decompress it in a single-rate fashion. The details about the decompression have not been furtherly analyzed since the platform already had a plugin that decompress a local file, hence the critical point has been to perform the *http* request in C++.

The idea of designing the request from scratch was not suitable, hence the remaining option was to choose a packaged library that could provide the needed components.

The library *libcurl*[9] is the most common and most used among the different options available: it supports a wide variety of communication protocols, supporting secure connections through *SSL,* it is *thread-safe* and thoroughly documented. Nevertheless, since the MEPP2 platform relies on a heavy-structured code that guarantees its portability and flexibility in choosing from a various set of mesh data and parameters, it was deemed to be more efficient to integrate the communication functionalities with other branches of the already present libraries: *Qt* and *boost*.

[9] https://curl.se/libcurl/

Boost is a portable C++ set of source libraries that has been implemented to correctly perform alongside C++ Standard Library. The module that will be added for this work is *beast* [10], a header-only library that provides the fundamentals for creating networking protocol communications inside a C++ application.

The *boost* version that was built inside the software did not support secure connections through *ssl*, so it has been necessary to update the library set. Since the process of updating the dependencies inside the platform could have been of use for other projects that used MEPP2, the libraries have been updated to the newest *boost* version by building the new components with *CMake* [11].

It has not been necessary to implement a plugin, considering that the MEPP2 function will be invoked by a C# script, therefore a simple filter has been implemented, *http_client_ssl.cpp*, that runs the setup of the communication between the remote server and the client itself. Later on this filter has been updated to an asynchronous version (*http_client_async_ssl.cpp*) to exploit the benefits of a multithreaded program for downloading the object and sending the batches to Unity at the same time.

In the next chapter the design of the communication between the C++ code and the C# script will be discussed, together with the other possible attempts that have been made.

[10] https://www.boost.org/doc/libs/1_74_0/libs/beast/doc/html/beast/introduction.html

[11] https://cmake.org/

8. Communication between different processes

A process is an instance of an executing program, identified by its Process ID (PID) and defining its own addressing space, in which various independently scheduled execution flows can operate. It may occur that the isolation level provided by the standard structure of a process sophisticates the achievement of a goal that is common to more processes, for this reason, it is possible to reduce this isolation in a controlled manner.

The first attempt of setting up a communication bridge between the two interfaces was made creating a C++ DLL: by including the dynamic library inside a C# script one could retrieve some of the data structures available inside the MEPP2 program, however, it was not possible calling the complex functions needed to perform the remote download and manipulation of the 3D object. It has lastly been chosen to use an IPC mechanism.

Inter Process Communication (IPC) is an isolation level reduction mechanism that allows safe data-exchange and activity-synchronization. The exchanged information has to be adapted to be comprehensible to the recipient: this may result easier if the two programs are already written in the same programming language, but when the programs are more different the issue is not trivial. Internal representations are not suited to be exported in any case (e.g. pointers only have meaning inside the addressing space they're created in), external representations for arbitrary data structures exist, but in this case, since the recipient will expect a bitstream to work on, the flow from the sender to the receiver will not have its format manipulated.

The IPC mechanism that has been implemented for the simulation is the *NamedPipe* [15], which allows the transfer of byte sequences of arbitrary size. The communication is 1-1 with markers that delimit the single messages.

The first sample code that has been implemented to test the effectiveness of the *NamedPipe* performed as following: the Pipe server, hence the one who provides the data, waits for a request from the Pipe client (i.e. the pipe-end inside the C# script), once it receives the initialization message from the client, the pipe-end in the C++ program creates a sample message and sends it as a binary stream; the pipe-client just performs the request, receives the stream and saves it as a binary file.

After successfully implementing this sample, the filter that performed the object retrieval has been updated from a blocking *read* to an *asynchronous* one: while the C++ program is downloading the progressive-compressed object and filling a local buffer, another thread sends the available chunk via IPC so that the C# could start performing the rendering of the first Levels Of Details and keep receiving data until the decompressed mesh at the maximum quality is rendered.

The first idea that came up together with the supervisors of this work was the following: the MEPP2 side is in charge of both downloading the data from the remote server, recognize when a batch, i.e. a portion of the bitstream that contains a set of comprehensible information so that the decompressing process could take place, has been downloaded and finally send the decompressed intermediate object via IPC. The main issue about this approach was that, in order to know how many bytes correspond to a single, complete batch, it was necessary to access to the header of the file and extract these information among the whole data that is written in it; another critical issue is that an object which is compressed in a progressive fashion has a peculiar structure in which, given *n* total batches, hence *n* total LODs, for each *i-th* element with $i \in n$ and $i != 0$, the data inside *i* has meaning only if preceded from all the previous batches.

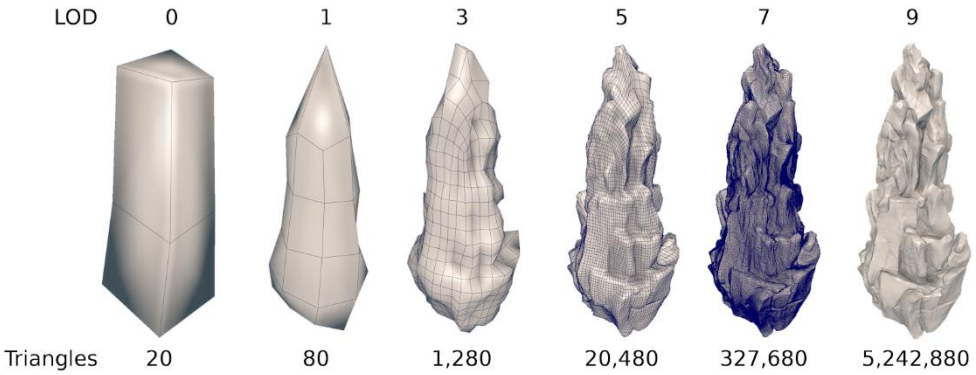

Figure 11. Visual explanation of Levels Of Detail. Each batch is complete when all the information for the geometry of that LOD is available. Retrieved from[12]

[12] *https://developer.nvidia.com/blog/using-turing-mesh-shaders-nvidia-asteroids-demo/*

During the development of the work it has been noticed that the extrapolation of information from the header of the compressed file was not trivial: the structure that act as a wrapper for the compressed mesh was made to be portable and efficient with different data structures, a direct consequence is that it is complex and hardly readable; additionally, the whole platform has been designed to draw multiple functions from different source libraries and possesses a vast number of filters and functionalities, therefore creating a new method that locates and extracts these values would have strayed from the original goal of the work. Another complication is tied to the fact that, as stated before, every new batch needs all the data from the previous ones to provide a new Level Of Detail (it can be seen as a data refinement), so appending the new bytes into a non-volatile buffer without any optimization technique would have caused extra workload for the processor.

Confronting with these constraints, it was decided to continue with a more abstract model that mimics in a reliable way the actual behavior of the original simulation, allowing to obtain visible improvements and easily appreciable with respect to the estimated model based on the utilization of MEPP2 codec.

This new model is based on a revisitation of the progressive approach exploiting the previously mentioned codec algorithm *Draco*: a set of gradually decimated *.obj* files is generated from a heavy and complex 3D model, also in the *.obj* format; then, each one of these pseudo-LODs is compressed with the *Draco* encoder and merged together to form a unique binary file, that will be uploaded on the remote server and eventually retrieved using the already implemented mechanism inside MEPP2. While this model does not exploit the standard progressive approach, it actually achieves another goal for the optimization of the process: in the previous version the compressed object would be retrieved and decompressed in the C++ application, then the decompressed batch will be sent to the application, that will only render it inside the scene by translating the format of the mesh into one that

is understandable by Unity; with this solution, the decoder is directly implemented inside the C# script, so the amount of data to send is considerably inferior since it consists of the compressed version and not of the uncompressed one, lessening the workload for the IPC process and exploiting the cutting-edge algorithm behind the *Draco* decoder.

MeshLab is an open source platform for processing and editing 3D triangular meshes; it offers different methods for the decimation of triangulated surfaces and preserving the geometrical detail and the texture mapping.

Two methods have been taken into consideration for the mesh simplification: the first one is a *Clustering Decimation*, in which a 3D grid with a determined cell density is created so that it fully encloses the model, then the vertices are discretized so that each group (cluster) of vertices that belongs to a certain cell is merged into one single vertex. This decimation method allows different approaches in the Cluster Generation (hierarchical approach or top-down / bottom-up), but, most importantly, the quality of the result is strictly connected to the chosen approach in computing the representative vertex that will substitute each cluster.

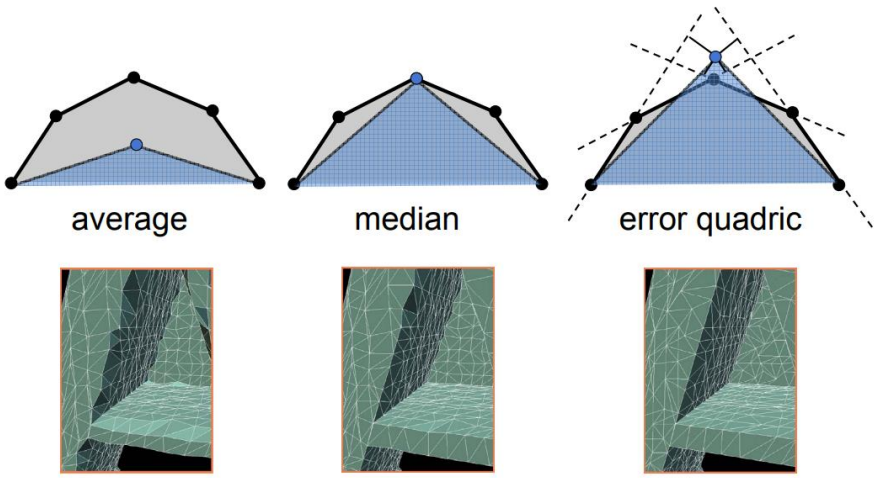

Figure 12. Three possible alternatives for computing the representative vertex. Retrieved from[13]

Choosing the representative vertex by computing the *average* value means calculating the average value of each dimension and creating a point in that specific

[13] *http://graphics.stanford.edu/courses/cs468-10-fall/LectureSlides/08_Simplification.pdf*

position (the point may not belong to the original mesh); choosing the *median* approach means that the representative vertex belongs to the original mesh, and it is the one that lies at the midpoint of the distribution of the vertices inside the cluster. The *error quadrics* approach assumes that the patch is expected to be piecewise flat, therefore the representative vertex is chosen so that it minimizes the distance to neighboring triangles' planes, minimizing the squared distance error. The edges are calculated during the mesh generation, given two clusters X and Y, and its representative vertices, respectively x and y, the edge (x, y) is created if there was at least an edge (x_i , y_j) with $x_i \in X$ and $y_j \in Y$.

The second method taken into account is the *Quadric Edge Collapse Decimation (with texture)*, which is based on decimation via pair contraction and approximation of error with quadrics. While vertex decimation are careful to preserve model topology and assume manifold geometry, edge contraction algorithms' iteration has to be carefully designed in order to address the non-manifoldness of the resulting mesh or the flip of a face caused by such operation [7]. In a quadric edge collapse decimation, the vertices belonging to an edge are collapsed, and this operation is performed iteratively until the desired simplification is reached. The edges to decimate are chosen at initialization time with the assumption that these simplification will not heavily compromise the topology of the mesh; the edge to be contracted in the iteration is chosen taking into account its *cost*, that is based on the approximation of error with quadrics, whose example of derivation is given by Ronfard and Rossignac [8]. The pairs are placed in a data structure and ordered by their local cost and, for each iteration, the pair with the lowest cost is collapsed, while the other pairs' cost is updated.

The Clustering method is known to be faster than the Quadric Edge Collapse, but with equal sizes the second approach resulted in a better quality, additionally, this filter preserves the UV parametrization, generally providing an even better quality of the decimated mesh, thus it has been chosen to use the second method.

To improve the quality of the final simulation a set of greatly detailed meshes has been chosen from the *Sketchfab* database. Each model has been processed in the following way:

- The mesh is imported into MeshLab and decimated with *Quadric Edge Collapse* into 5 different Levels Of Detail, respectively composed of 1k, 10k, 30/50k ,100k ,300/400/500k resulting faces, in dependency with the original geometry.

- The original *.obj* file is compressed via *Draco encoder*, with a quantization parameter *qp* of 16, then every *.obj* file that is the result of the precedent operation is also encoded with the same quantization parameter.

- The last file, *XLod.drc.bytes*, is created by appending in an orderly manner the previous *.drc* files, from the coarse one to the non-decimated one.

The textures have not been manipulated and they have been locally imported inside the Unity project.

Figure 13. Horse. Different levels of detail through the MeshLab decimation filter. 3D model attribution to[14]

Horse	**Vertices**	**Faces**	**OBJ Size**	**Draco size**
Full size	304131	608373	72953 KB	1440 KB
I	444	1000	109 KB	8 KB
II	4944	10000	1056 KB	48 KB
III	14944	30000	3159 KB	118 KB
IV	49944	100000	10489 KB	322 KB
V	149944	300000	31698 KB	803 KB

Table 5. Horse 3D model decimation table

[14] https://sketchfab.com/3d-models/pferdestatue-0bd8088f92be4a39a3c8a7faafbf897f

Figure 14-15 Bronze Cat and Lion. 3D models attribution [15] [16]

Figure 16-17 Sphinx and Putti. 3D models attribution [17] [18]

[15] https://sketchfab.com/3d-models/bronze-cat-e4c8a7ec94ea4f4bb0a1170f275071bd

[16] https://sketchfab.com/3d-models/lowe-von-asparn-57a57a99ce1f4e45adee5ae37a91f51b

[17] https://sketchfab.com/3d-models/sphinx-d94a2dfbf413465395fde1bd17981b85

[18] https://sketchfab.com/3d-models/putti-gruppe-49342e3b3c7d4e3c92bb5525ffeb397f

Lion	Vertices	Faces	OBJ Size	Draco size
Full size	358928	714772	73617 KB	1702 KB
I	478	1000	114 KB	8 KB
II	4978	10000	1132 KB	54 KB
III	24978	50000	5594 KB	209 KB
IV	49978	100000	11083 KB	372 KB
V	199978	400000	43794 KB	1128 KB

Table 6. Lion 3D model decimation table

Putti	Vertices	Faces	OBJ Size	Draco size
Full size	315159	630380	75385 KB	1518 KB
I	469	1000	110 KB	8 KB
II	4969	10000	1059 KB	50 KB
III	14969	30000	3165 KB	122 KB
IV	49969	100000	10520 KB	334 KB
V	149969	300000	31734 KB	828 KB

Table 7. Putti 3D model decimation table

Sphinx	**Vertices**	**Faces**	**OBJ Size**	**Draco size**
Full size	308254	616584	74125 KB	1513 KB
I	462	1000	111 KB	8 KB
II	4962	10000	1067 KB	51 KB
III	14962	30000	3194 KB	123 KB
IV	49962	100000	10609 KB	337 KB
V	149962	300000	31951 KB	828 KB

Table 8. Sphinx 3D model decimation table

Bronze Cat	**Vertices**	**Faces**	**OBJ Size**	**Draco size**
Full size	650778	1301367	87682 KB	2311 KB
I	533	1000	92 KB	5 KB
II	5035	10000	921 KB	33 KB
III	25041	50000	4820 KB	132 KB
IV	50046	100000	9761 KB	243 KB
V	250094	500000	51677 KB	1053 KB

Table 9. Bronze Cat 3D model decimation table

The *Museum* project has been finally put together as the result of the studies previously discussed. Performances and impact of three different versions of the simulation are evaluated:

- Uncompressed object retrieval scenario: a batch of *.obj* files are gradually downloaded while the character explores the scene. In this scenario, the meshes will appear after a long delay, only when they are retrieved in their entirety.

- Single-rate compression object retrieval scenario: the batch of *.obj* files is firstly compressed via Draco, then, they are uploaded in an *http* server, retrieved by a dedicated C++ filter and decompressed with a C# script. In this scenario, the meshes will appear in a short delay, only when they are retrieved in their entirety.

- Progressive visualization of compressed object retrieval scenario: each *.obj* file is split into 6 different-sized versions, each one of them is compressed via *Draco* and glued together into a unique file, that will be gradually retrieved from the remote *http* server. In this last scenario a first coarse version of the mesh will appear in a noticeably shorter delay and it will consequently be enhanced as the download progresses, finally turning into the most detailed version.

During the simulation, the user walks inside the museum, eventually colliding with a set of invisible objects, one for each mesh, that will trigger the operation of retrieval for a certain art piece. The user is free to move around the area throughout all the loading phases of the object(s), and he may activate more than one trigger.

Figure 18. Visual explanation of the trigger process in WoW. Screenshot from the final version of the application.

If the simulation is played in VR through the mounted headset, the triggers will be activated as previously mentioned if the user is moving with the Trackpad. Additionally, a mesh covering an area around the pedestal of each art piece has been created and marked as a trigger so that, in case the user *teleports* to a new location, if this new location is contained in one of the said areas, the loading operation will immediately take place.

The key functions behind the operation are explained in the following paragraphs:

For each object that will be downloaded there is an *invisible trigger* (Fig. 18) that will invoke an *Event*. The *startSimulation.cs* script is the one delegated to the progressive compression simulation: it will initialize the C++ process with the task of establishing an http-secure connection, then it will invoke the Event so that the scripts attached to the empty object can run, and finally it waits for some data available to read from the pipe. The *startFullSimulation.cs* script is the one devoted to start the second scenario of the application, the one in which the compressed most detailed version of the mesh is retrieved.

Each empty object in the scene that represents the "shell" of the 3D model has two important scripts attached to it:

- *PipeBud.cs* initializes the client of the piped communication, creates a buffer in which data will be written, then performs multiple *read* until the number of reading is equal to 6 (5 LODs and the original compressed object). This script also invokes an event in order to alert the script *DracoDecodingObjects.cs* that a batch is ready to be processed.

- *DracoDecodingObjects.cs* is a customization of the original decoding script provided by *draco*: it works for piped communication with the C++ code, decoding locally retrieved objects and decoding 3D objects once downloaded via http. The textures of the meshes are also processed inside this script.

On C++ side, the MEPP2 filter *http_client_async_ssl.cpp* works as follows:

In the function *Pipis* the *namedPipe* is created and it is called every time a batch of a specific object is ready to be sent; this function calls the *InstanceThread* method, that instantiates and specifies the task of a new thread once a client connects to the *namedPipe*. All the necessary data to perform the request, such as the size of the different batches, the *host,* the *port* and the *target,* is provided as a set of parameters by the C# script devoted to start the execution of the C++ program and it is collected as the server receives the first request message from the client, along with the ID of the object to retrieve. The method *async_read_some* provided by the *boost/beast* library is the one who performs a gradual read, allowing to access data while the *http* retrieval is still performing. The *on_read* method is called every time the previous function returns: every time a chunk of data is read and saved in the buffer, this method saves the body of the http reply and, finally, every time the chunk is big enough to contain a batch, it sends the data through the pipe.

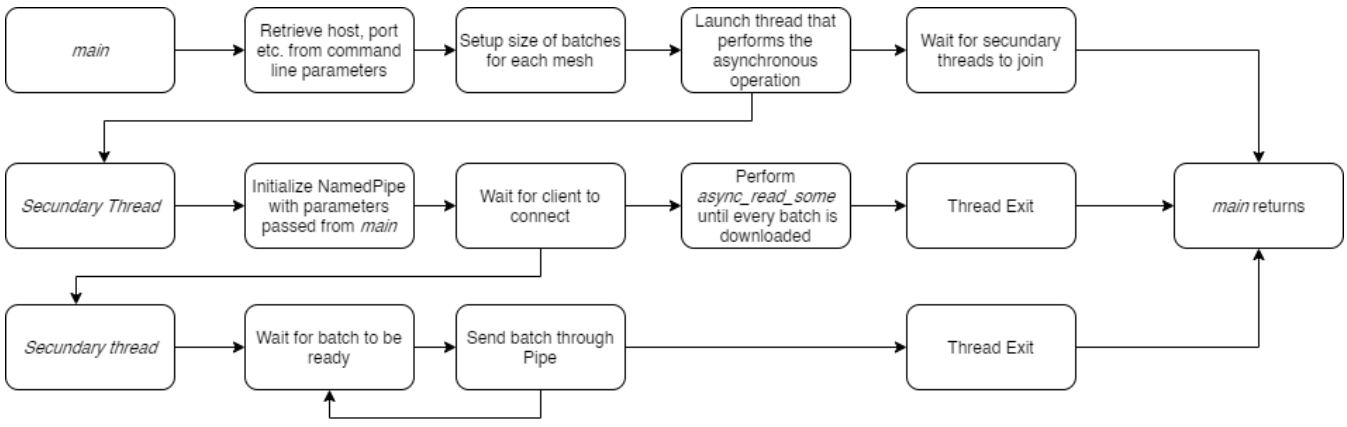

Figure 19. Flowchart of the C++ process.

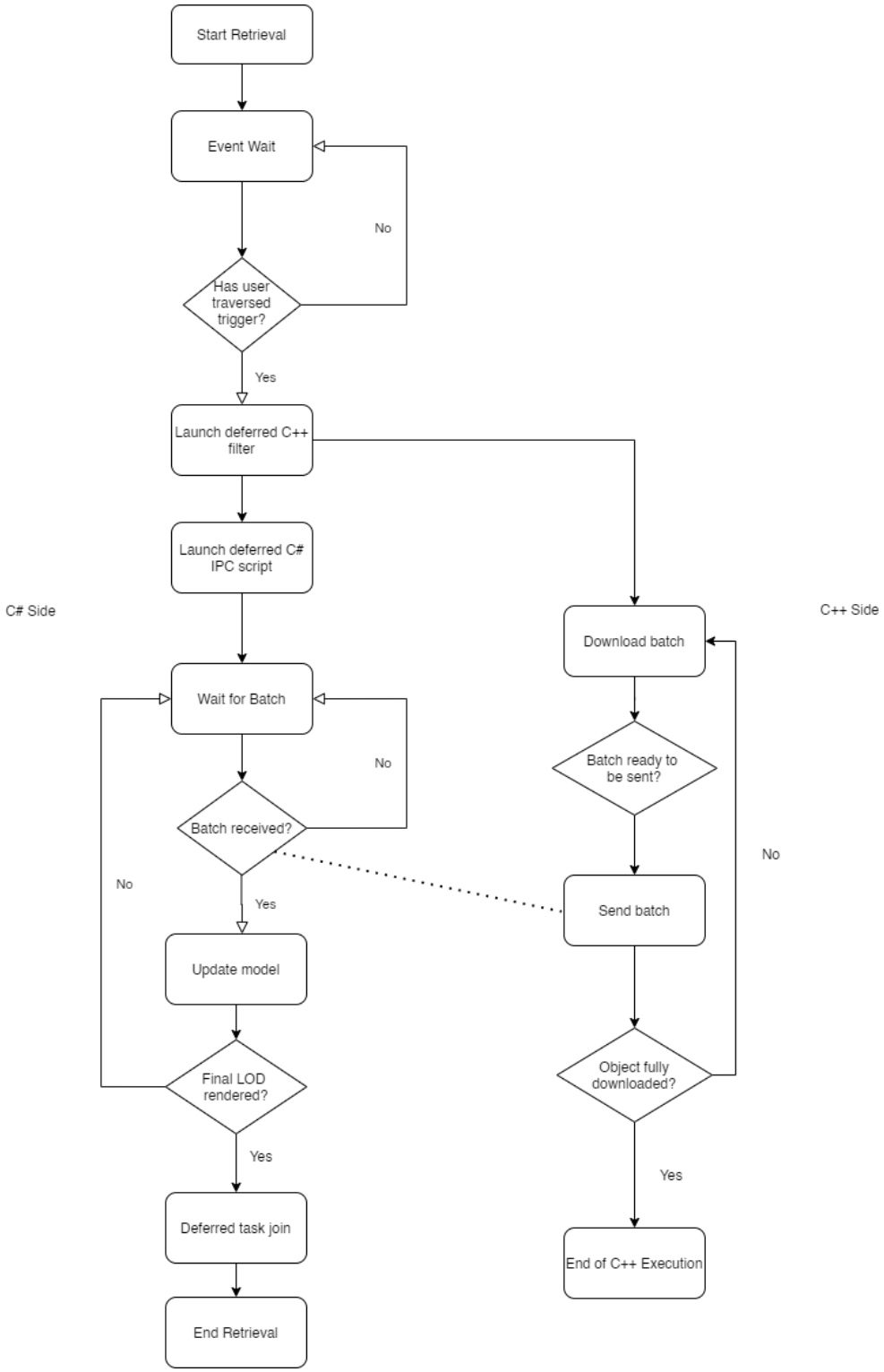

Figure 20. Flowchart of the object retrieval process.

An example of a running simulation follows:

The player starts his experience at the entrance of the museum. At this specific moment, no object is present inside the museum. The player starts exploring the area by walking across the main entrance and triggering the "Lion" art piece to show up. From here he has free access to the rest of the area, so he can walk towards the other separators to trigger the activation of another event or just wait for the first art piece to complete its rendering.

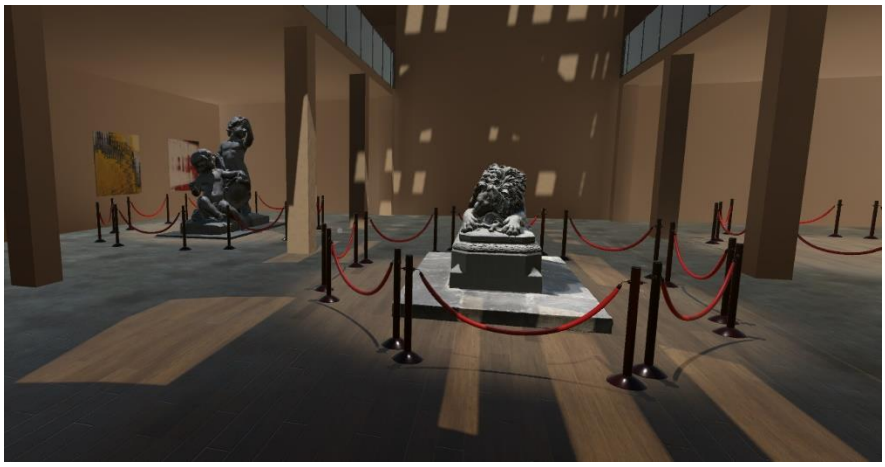

Figure 21. Screenshot of the Simulation #1.

As he discovers the different rooms, the museum takes life by being populated with different sculptures.

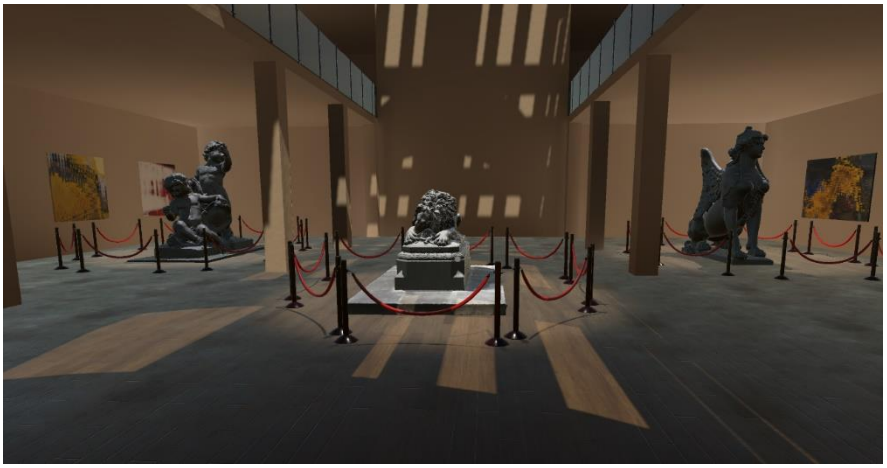

Figure 22. Screenshot of the Simulation #2.

The next table summarizes the rendering times of each object in the scene by taking into account the second scenario (first column) and the third scenario (second to last column):

Mesh	**Full Reso Rendering Time**	**Batch #1**	**Batch #2**	**Batch #3**	**Batch #4**	**Batch#5**	**Full Reso Batch**
Bronze Cat	2.32s	0.01s	0.03s	0.11s	0.25s	0.94s	2.19s
Lion	2.01s	0.03s	0.07s	0.17s	0.36s	1.01s	2.05s
Horse	1.82s	0.05s	0.17s	0.43s	0.92s	1.69s	2.79s
Sphinx	1.80s	0.01s	0.04s	0.10s	0.29s	0.72s	1.68s
Putti	1.95s	0.01s	0.03s	0.09s	0.30s	0.77s	1.77s

Table 10. Results table. The first column is referred to the single-rate scenario, the other columns to the progressive scenario.

It is worth noticing that the difference in rendering time, related to the variability of the sizes of each object, is nullified during the first steps of the algorithms, since the batches have similar sizes. As a consequence, an object will always be visible in short time, although the time for completing its retrieval will vary on the long term.

The rendering times of the objects in the second scenario (dots) and in the third scenario (crosses) are compared in the following graph:

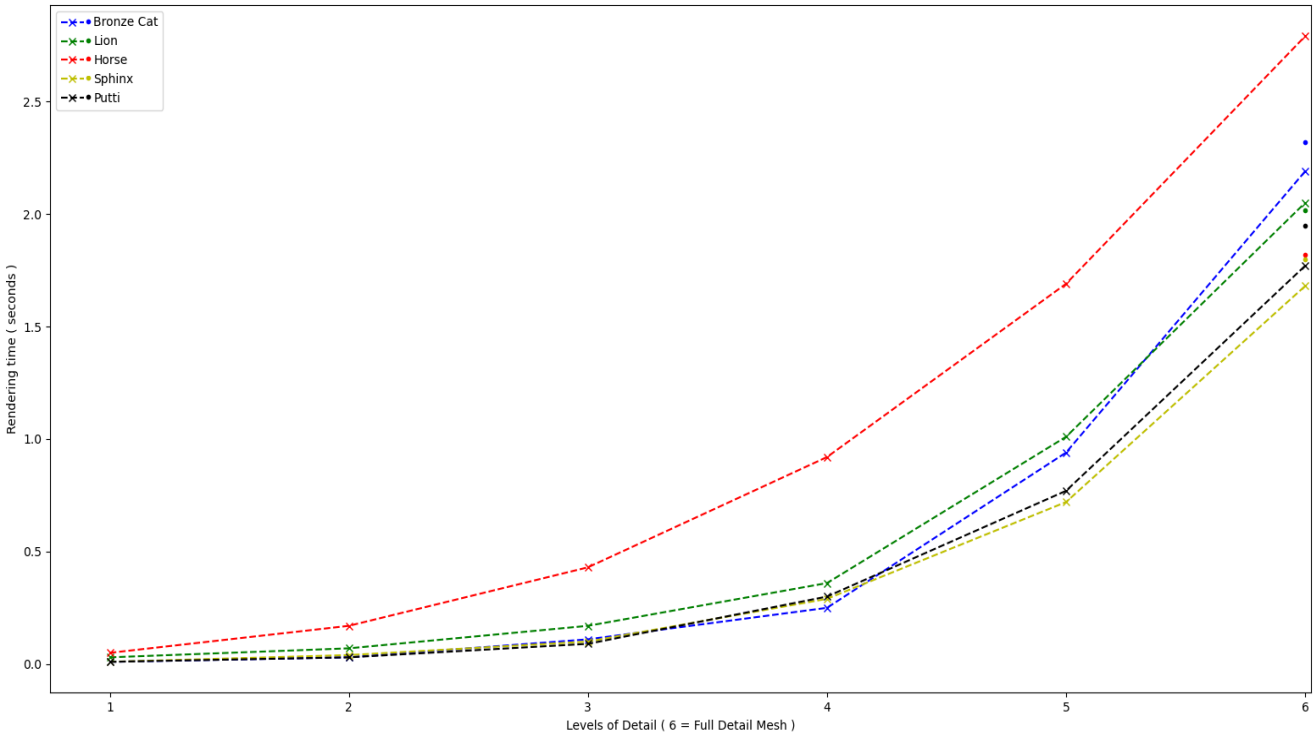

Figure 23. Performance comparisons between second and third scenario.

As expected, the third simulation provides an almost-immediate visualization of a low-detailed version of the meshes; in some cases, the final batch of a 3D model will be rendered with a slight delay with respect to the same object retrieved using the single-rate approach: as it has been stated before, this may be caused by the complexity of the process behind the progressive simulation. Overall, the behavior of the simulation confirms that the progressive approach brings a tangible optimization regarding user experience while not excessively overcharging the systems' resources.

Another important experiment has been conducted to broaden the perspective of this work. A test of efficiency in case of different network scenarios has been developed as follows: two additional simulations have been performed and compared to the standard scenario, that was carried out with an effective bandwidth of approximately 40 Mbps. For the first experiment the wireless band of the device in which the experiment takes place has been limited by setting a *low priority* on Quality of Service router regulation, next, the band has been kept busy by downloading and streaming content on different devices connected to the same router. As a result, the effective bandwidth on the terminal was around 1 Mbps. For the second experiment the terminal has been connected via hotspot to a mobile 4G data network, with an effective bandwidth of roughly 25 Mbps.

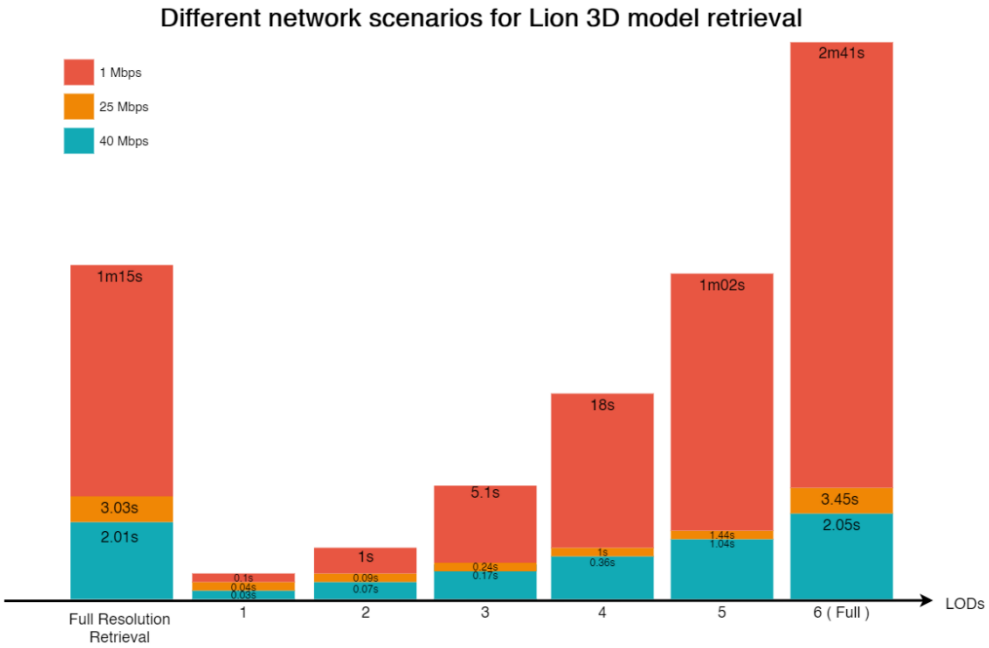

Figure 24. Different network scenarios for Lion 3D model retrieval.

It is firstly noticeable that, lowering the bandwidth, the most detailed version of the mesh (Level Of Detail #6) will render with a consistent delay with respect of directly retrieving the Full Resolution mesh (as performed in scenario #2). On the other hand, the experiment proved that there is a remarkable difference between the retrieval of the Full Resolution mesh in a band-limited environment (the user must wait more than one minute before seeing anything), and the LOD-based approach, in which the object is present in the scene with a 100ms delay. Furthermore, the fourth Level of Detail is rendered with a delay of 18 seconds, and its level of detail is generally acceptable within a dynamic scenario: the user will hardly notice the following improvements unless he gets very close to the object.

In conclusion, this experiment led to satisfactory results and it successfully models the proposed scenario, with a remarkable difference with reference to the raw retrieval of a 3D uncompressed object.

The proposed solution also helps maintaining an acceptable degree of immersivity: since one of the most known application areas of LODs is saving up memory for distant objects (i.e. when their details are still not noticeable), the trigger is placed at an adequate distance so that the user is not capable of distinguishing the imperfections of the intermediate model, but he can immediately notice that there's a certain object with a certain shape; additionally, in case he walks up to take a closer look, the object will have the time to acquire more detail. From these considerations, the third scenario results to be the most effective one both in terms of immersivity and requisites for a real time application, in coherence with the thesis that has been carried out in this work; furthermore, this solution better performs in environments with limited or unstable bandwidth: the first batch usually has a limited number of vertices and, consequently, a size of roughly a dozen of KB to be downloaded.

Some possible improvements and integrations may be the setup of a subjective quality measurement experience, in which the users are asked to give their impressions on the different scenarios; another possible development could be the insertion of distractor elements (i.e. masquerading) for diverting the user's attention while meshes are being loaded. A further study may be conducted on how remote retrieval of adaptive textures may change the result in terms of efficiency.

This work paves the way for future developments in the optimization of user experience in VR environments in many scenarios (videogames, staff training, instruction etc.), opening doors to new possible researches in the field, focusing on optimizing visual experience without further computational cost, up until the almost integral shift of graphic applications' computation load from the user's terminal to a dedicated remote machine.

BIBLIOGRAPHY

[1] CAILLAUD F., VIDAL V., DUPONT F., LAVOUE G. : Progressive compression of arbitrary textured meshes. *Computer Graphics Forum* 35, 7(2016), 475-476.

[2] NG REN: CS184/284A *Computer Graphics and Imaging Course*, Berkeley University.

[3] MAGLO A., LAVOUE G., FLORENT D., CELINE H. : 3D mesh compression: survey, comparisons and emerging trends. *ACM Computing Surveys,* Vol. 9, No.4, Article 39(2013), 39 :16 - 39 :31.

[4] LEE H., LAVOUE G., DUPONT F. : Rate-distortion optimization for progressive compression of 3D mesh with color attributes. *Springer-Verlag*(2011).

[5] ALLIEZ P., DESBRUN M. : Progressive compression for lossless transmission of triangle meshes. *Proceedings of SIGGRAPH,* (2001), 195-202.

[6] VIDAL V., LOMBARDI E., TOLA M., DUPONT F., LAVOUE G. : MEPP2: a generic platform for processing 3D meshes and point clouds. *EUROGRAPHICS 2020,* (2020).

[7] GARLAND M. HECKBERT S. P. : Surface Simplification Using Quadric Error Metrics. *SIGGRAPH'97: Proceedings of the 24th annual conference on Computer Graphics and interactive techniques,* (1997), 209:216.

[8] RONFARD R. ROSSIGNAC J. : Full-range approximation of triangulated polyhedral. *Computer Graphics Forum,* 15(3), Eurographics 96 (1996)

[9] RAMEY D., ROSE L., TYERMAN L. : MTL material format (Lightwave, OBJ) *FILE FORMATS, Version 4.2* (1995)

[10] PHONG B. T. : Illumination for Computer Generated Pictures. *Communications of the ACM 18, No. 6* (1975)

[11] FEINER S., MACINTYRE B., HAUPT M., SOLOMON E. : Windows on the World: 2D Windows for 3D Augmented Reality. *UIST '03 (User Interface Software and Technology)* (1993)

[12] BOZGEYIKLI E., RAIJ A. KATKOORI S. : Point & Teleport Locomotion Technique for Virtual Reality. *CHI PLAY '16: Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play* (2016)

[13] PIETROSZEK K. : Virtual Hand Metaphor in Virtual Reality. *Encyclopedia of Computer Graphics and Games. Springer Publishing International* (2019)

[14] PIETROSZEK K. : Raycasting in Virtual Reality. *Encyclopedia of Computer Graphics and Games. Springer Publishing International* (2019)

[15] KHAMBATTI M. : Named Pipes, Sockets and other IPC. *Arizona State University Paper.*(2001)

[16] SEO J., JOUNGHYUN KIM G., CHUL KANG K.. : Levels of Detail (LOD) Engineering of VR Objects. *Pohang University of Science and Technology Paper.*(1999)