TECHNOLOGICAL EDUCATIONAL INSTITUTE OF CRETE

SCHOOL OF APPLIED TECHNOLOGY

DEPARTMENT OF INFORMATICS ENGINEERING

Bachelor thesis

Algorithms and processes for progressive graphics applications

Paschalis Dedousis – R.N.: 2903

Supervisor professor: Dr. Athanasios G. Malamos

Evaluation commission: -

Presentation date: -

Table of Contents

Table of Figures

Table of Tables

Abstract

In previous work an integrated adaptation framework has been proposed for the Web3D, using the X3D and the MPEG-DASH standards. By fusing those two, one can deliver multimedia content adaptively in X3D scenes following the HTML5's plug-in free mind set. Since then, a problem remains of how to have a good network utilization when delivering refined or coarser versions of 3D models by only using open and royalty free web standards and without destroying the X3D's human readable representation form.

When transmitting different levels of detail of a 3D model, we need to do it in a cumulative manner, thus preserving common geometry data. Considering the programmatically created and the hand crafted level of detail techniques, we need a way to support those two by offering an integrated solution based on current or emerging web standards.

Programmatically creating levels of detail, often needs the change of the 3D model's data structure. This means that the content provider and the content consumer must agree for the what and how to implement before runtime, thus driving into case per case solutions. Also, when following the hand-crafted approach, chances are that there is common geometry data in-between the levels of detail. So when delivering them as individual entities will result into poor bandwidth utilization.

To alleviate these issues we will consider a context agnostic approach, namely delta encoding or delta compression, for transmitting levels of detail of 3D models in a unified environment.

Σύνοψη

Σε προηγούμενη δουλειά έχει προταθεί ένα ολοκληρωμένο σύστημα προσαρμόσιμου πολυμεσικού υλικού για το Web3D χρησιμοποιώντας τα πρότυπα X3D και MPEG-DASH. Συνδυάζοντας αυτά τα δύο, μπορούμε να πραγματοποιήσουμε μετάδοση προσαρμόσιμου πολυμεσικού υλικού σε X3D σκηνές ακολουθώντας μια λογική η οποία είναι ελεύθερη από αρθρώματα. Ένα πρόβλημα που παραμένει όμως είναι το πώς μπορούμε να έχουμε μια καλή εκμετάλλευση του δικτύου όταν μεταδίδουμε διαφορετικά επίπεδα ποιότητας 3Δ γραφικών χρησιμοποιώντας μόνο ανοιχτά πρότυπα και χωρίς να αλλάξουμε την αναγνώσιμη από ανθρώπους μορφή περιγραφής του X3D.

Όταν μεταδίδουμε διαφορετικά επίπεδα ποιότητας 3Δ γραφικών πρέπει να το κάνουμε με έναν συσσωρευτικό τρόπο έτσι ώστε να μπορούμε να επαναχρησιμοποιήσουμε τα κοινά δεδομένα γεωμετρίας. Έχοντας ως βάση την δημιουργία επιπέδων ποιότητας με προγραμματιστικό τρόπο αλλά και με το χέρι, χρειαζόμαστε μια ολοκληρωμένη λύση που να μπορεί να υποστηρίξει αυτές τις δύο διαφορετικές μεθοδολογίες και η οποία θα βασίζεται σε σύγχρονα και αναδυόμενα πρότυπα.

Ο προγραμματιστικός τρόπος δημιουργίας επιπέδων ποιότητας συνήθως απαιτεί την αλλαγή της δομής δεδομένων που χρησιμοποιείται για την περιγραφή του 3Δ μοντέλου. Αυτό σημαίνει πως ο πάροχος και ο καταναλωτής του περιεχομένου πρέπει να συμφωνήσουν εκ των προτέρων για το τι και το πως θα υλοποιηθεί. Με αυτόν τον τρόπο οδηγούμαστε σε ανά περίπτωση υλοποιήσεις. Επίσης όταν χρησιμοποιούμε την δεύτερη μεθοδολογία δημιουργίας επιπέδων ποιότητας, οι πιθανότητες είναι πως θα έχουμε κοινή πληροφορία μεταξύ των επιπέδων. Οπότε αν μεταδώσουμε τα επίπεδα αυτά ως ανεξάρτητες οντότητες θα έχουμε χαμηλή εκμετάλλευση του δικτύου.

Για να ξεπεράσουμε τα θέματα αυτά προτείνουμε μια προσέγγιση του προβλήματος στην οποία το μεταδιδόμενο περιεχόμενο θα μας είναι αδιάφορο. Πιο συγκεκριμένα, στο ολοκληρωμένο περιβάλλον μετάδοσης 3Δ γραφικών που θα δώσουμε θα χρησιμοποιήσουμε συμπίεση δέλτα.

Chapter – 1

1.1 - Introduction and previous work

Recent advancements in Web technologies, offer the ability to deliver multimedia content in a heterogeneous environment of platforms and devices. To achieve this, a great number of standards has been introduced, although not all of them are implemented on a large scale or exploited on their full potential.

For many years, multimedia content delivery and interaction over the web was mostly supported by using proprietary, closed source solutions. The most notable example is the Adobe's, Adobe Flash Player, a freeware plug-in that enables audio and video, as well as vector, raster and 3D graphics support for the web browser. Although serving it's purpose well over the years, one of the disadvantages of this approach is that it breaks interoperability. In the list of HTML elements [WHV14] of the latest HTML specification there are some interesting elements, at least from the perspective of the multimedia field, that offer native support of audio, video and graphics in the browser. Specifically, in the subcategory of embedded content lie the audio and video elements and in the scripting subcategory lies the canvas element.

Today's client side scene is formed not only by the well known desktop computers. New portable devices came into, such as smart phones and tablet computers, that are connected to the web using wireless and often unreliable connections. In addition to that they have limited processing power, reliance on battery and limited viewing capabilities mostly due to their size, the goal of achieving a good QoE becomes even harder. One of the approaches and sometimes combined with others to alleviate this issue, adaptive bitrate streaming is used.

Adaptive bitrate streaming is a technique for streaming multimedia content over a network to the client in an adaptive manner. Meaning that while streaming, the content is adapted according to the client's processing power and network bandwidth capabilities. Some implementations of adaptive bitrate streaming include the Adobe HTTP Dynamic Streaming, Apple HTTP Live Streaming and the Microsoft Smooth Streaming. None of them is a standard though, meaning that they fail interoperability wise. To overcome this, companies like Microsoft, Apple, Netflix and others, participated in the standardization of

the MPEG-DASH, an industry oriented, open and international standard. In addition, the MPEG-DASH delivers content using the HTTP protocol, so content can be delivered using the already widely used and well adopted HTTP over TCP [IRS11] [SAC11] .

The directions of how an MPEG-DASH client can switch between different quality media streams are described in an MPD file. The MPD can be obtained usually via the web and is an XML, human readable file. In fact, in the sense that MPEG-DASH and HTML5 technologies can be complementing when consuming media content, the DASH-IF developed a Javascript player for the browser, for supporting adaptive video capabilities.

X3D is an open, royalty free ISO standard managed by the Web3D Consortium, that represents 3D graphics in XML format, readable by both humans and computers, that is supported by stand-alone implementations or browser plug-ins, at least until recently. To overcome the disadvantages of using browser plug-ins, Behr et al. [BEJZ09] presented the X3DOM, a DOM based model that gives a seamless integration between X3D and HTML5 without using plug-ins.

On their first attempt to extend the X3DOM's adaptation methods, Kapetanakis et al. in [KPMZ14] provide a mechanism of adaptive HD video inside 3D virtual reality worlds by merging it with MPEG-DASH. The offered implementation consists of extending the X3DOM's MovieTexture element to work with the DASH video player [GIT14][ML14] .

Although the MPEG-DASH was designed to be primarily used for temporal content, such as audio and video, it does not explicitly restrict the media type that can be used. Given that the X3D is an ISO standard and that the `model/x3d+xml` is a registered MIME type, 3D models written in X3D should be compatible with the MPEG-DASH. Based on this, Kapetanakis in his thesis [KK14], extends the previous mechanism in order to additionally support adaptive 3D model delivery. In his work he also describes how the 3D models should be treated so they can be successfully included in an MPD file. As an overview, the 3D models are segmented into levels of detail and can be sequentially transmitted to the client. Also the client can adapt the 3D model by requesting the corresponding segment, or level of detail.

This work only suggested the first steps into creating such an adaptation system and only supported simple cases where only a single 3D model was included into an X3D scene. Zamploglou et al. in [ZKSMP16] continue the research on the potential of supporting adaptive streaming of complex X3D scenes, where a plethora of geometries, audio, image and video textures are included.

An issue that remains though, is how to have a good network utilization when transmitting different segments, or levels of detail, described in the MPD. To give a solution approach to that, first we have to have an understanding of the term LOD of a 3D model. The first ideas of the LOD were introduced by J. Clark [JHC76] and in a sense is technique for representing an object with different geometry complexities. The goal of this technique is to reduce the representation complexity of an object in order to reduce computation requirements when such resources cannot be met or when the object is viewed from a far distance. There is no restriction to that the LOD technique can only be used for the geometry of a model, it can also be applied to shaders and texture maps, but in this work we are interested in the model's geometry. A qualitative graphical representation, describing the fidelity and the geometry complexity relationship, is given on Figure 1.

There are three basic approaches or frameworks of LOD and these are the discrete, continuous and view-dependent LODs. In the case of the discrete LOD, the object is separated into some number of individual models of different fidelity before the runtime. Usually, those LODs are hand-crafted by a graphics artist(s), if not exported by some mesh simplification algorithm implementation. Using this approach, gives us the ability to control of what will be viewed to the user. On the contrary, in the continuous LOD case, the model is encoded in a data structure such that we can extract the desired LOD from a continuous spectrum of LODs. This approach gives better granularity and minimizes the popping effect when changing LOD. Also, the case of the view-dependent LOD can be considered as an extension of the continuous LOD, in the sense that the geometry simplification varies in different areas of the model, usually according to the camera – area distance. Such an example is when viewing a terrain, where areas closer to the camera have more geometry information than areas that are farther away.

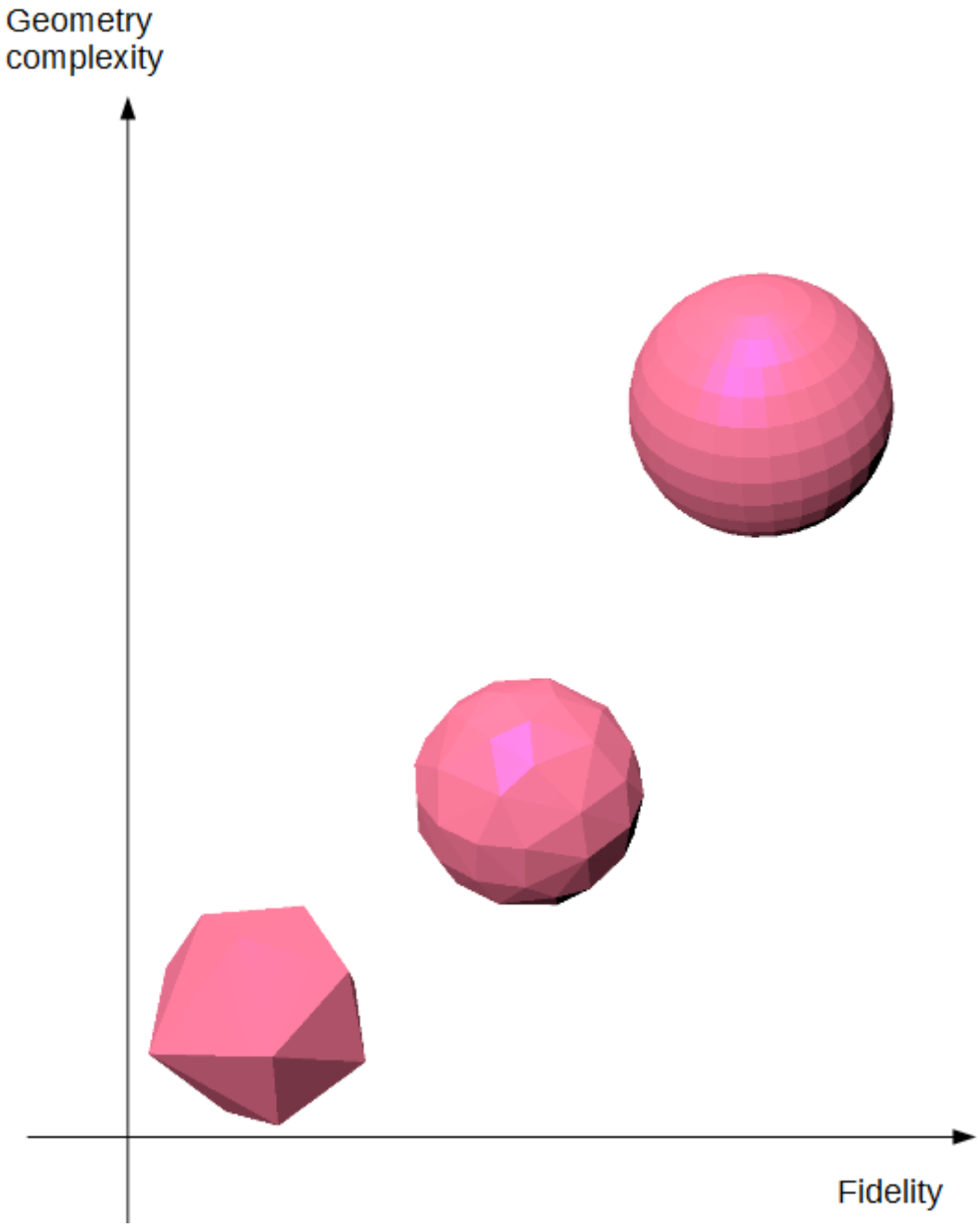

Figure 1: The more the fidelity, the more the cost to transmit

1.2 - Problem definition

The main goal of this thesis is to give a solution approach that changes, in the sense of extending the X3D's already defined nodes, are kept at the most minimal if not non existent all. Further more, the standard uses a representation form about the vertices and the edges of the model, that is human readable and we wish to keep it that way. In addition, we want to support both the discrete and the continuous LOD frameworks in our solution.

Also, the MPEG-DASH player developer is already occupied with solving problems such as content adaptation in real time. So it is quite important to offer a simple API for the delivery of the adaptive models. To achieve this, our solution should support the two different LOD frameworks in a unified way.

As an X3D client, we will use the X3DOM client that was mentioned above, but let's not forget that the X3DOM is only one of the many implementations of the X3D standard that exist. A reference list of the currently available implementations can be found in [X3DP16]. This makes it vital that our solution is based on standards and is not focused only on a single X3D client.

As described in the previous section, continuous LOD frameworks encode the model in a data structure such that it allows the implementation of some geometry simplification algorithm and the extraction of the desired LOD. The issue with this approach is that there are no standard data structures and algorithms that are used among all implementations. A commonly used one though, is the half edge data structure. In addition, some of them, encode the model in some binary form, thus destroying the X3D's human readable representation form. This could not be a quite bad thing, considering the gain in performance, but some binary forms are proprietary and not freely available, so they break the open standard mind set.

The need for using more sophisticated data structures, other than usual X3D's indexed face set, comes from the need that these algorithms depend on queries about the geometry and connectivity of the model. These queries are mostly about the adjacency and incidence of the building blocks of the model. For example, queries such as which faces share a particular vertex, or which edges are adjacent. Of course, such information can be extracted even with the X3D's indexed face set, but this data structure is not designed to store any kind of explicit information to satisfy these queries quickly, without

having to traverse the model's geometry and connectivity repeatedly for every single query during the run time.

In Mauro Figueiredo et al. [FRSV14] a framework that supports interactive topology queries on 3D models is presented. The given open source implementation, named TopTri, allows 3D web client applications to make queries about vertex, edge and face adjacency and incidence on the web server without the need of changing the model's data structure that is already used on the client. Instead the web server is responsible for such operations. Although implemented in the Python scripting language, the web server is satisfying fast enough to serve those queries even for large models and real time applications. While testing their proposed framework, Mauro Figueiredo et al. did not use any kind of continuous LOD algorithm as their test bench. Instead they implemented algorithms that can identify stalactites in a cave, using the web browser as the client runtime environment.

The TopTri toolkit, relieves the client from implementing a sophisticated data structure for continuous LOD, so it seems that is suitable for our needs. A solution approach for our system for transmitting adaptive 3D models, using that framework, is not fully investigated yet. Thus we are in a position that we can not give a clear answer about the level of changes that have to be made to the client, in order to at least support only the continuous LOD framework.

As for the discrete LOD framework, the geometry and connectivity between different levels of detail is disjoint. So there is not an obvious or straightforward approach to support a transmission that is free of redundant geometry information. A first approach that we considered in order to send only the changes that have to be made to reconstruct the target mesh, was to use a technique named geomorping or mesh interpolation or metamorphosis. This technique is widely used where there is the need to express a smooth transition between two models. For example an animation of an infant growing up to an adult or the transition between different facial expressions. In particular, from the perspective of LOD, it is used to eliminate or at least minimize the popping effect when changing the level of detail.

As in [Par05], there are three main steps that are followed to achieve a multi-resolution mesh morphing. First we need to find correspondence between the vertices of the source mesh to the target mesh. Note that vertex to vertex correspondence cannot be always achieved because the two meshes may have different number of vertices. So we compromise with some arbitrary point in 3D space near the source – target mesh. The same step has to be repeated in the opposite direction in order to find the vertices

correspondence between the target and the source mesh this time. Finally we merge the connectivity of the two input meshes in order to produce a new mesh representation that shares the connectivity of these two meshes. This new mesh representation is often called the supermesh. Now using the produced supermesh we can bidirectionally interpolate between the two meshes.

As a rough calculation of the memory size that the supermesh requires, without considering the memory cost for the connectivity, we have:

$$Bytes(\ Geometry(supermesh)\ ) = b\ (\ N(V_s) + N(V_t)\ )$$

Where b is the number of bytes required to store a single vertex, $N(V_s)$ the number of vertices of the source mesh and $N(V_t)$ the number of vertices of the target mesh. So we concluded that this size of overhead leads to an unworkable scenario. This led us to leave the investigation of this solution approach quite early. Even though we did not follow this path in this thesis, we do not completely reject the probability of a solution based on morphing between arbitrary and multi-resolution meshes exists.

Talking about the changes that have to made in order to reconstruct the target mesh, having the current mesh as the reference mesh, we came up with the idea of applying delta encoding. Delta encoding or delta compression, also known as delta differencing in its more general description, is a technique for describing a sequence of input entities in the form of differences between them. By doing that, data that is common in-between the entities is not repeated, thus reducing the requirements of storage space, or bandwidth if we are in the case of transmission. An every day example that delta compression is used is the case of remote file synchronization.

Chun in [Wc12] is dealing with the transmission of 3D models in WebGL scenes over HTTP for the Google Body project. Having in mind that the GZIP algorithm was primarily designed for compressing text input, LZ77 phrase matching will most likely fail with something that is not structured, like in our case lists of vertices and indices. In order to optimize the model for compression, Chun approaches the vertices and indices data as signals and uses delta encoding.

Now in the sense that a web resource may change over time and that the new instance of that resource will most likely be similar with the older one, Mogul et al. [MDFK97] discuss and quantify the potential benefits of using delta encoding and delta compression for HTTP responses and their results are encouraging. In K. Psounis's work [Pks02] dynamically created web content is separated into base documents, named as classes, and content responses are sent in the form of deltas. Experimental results of his

work show that by using the class-based delta model, bandwidth consumption is reduced by a factor of 30.

The Chromium [Chr02] project, the descendant of WireGL [WG01] , offers a framework for scalable cluster rendering. In a sense, the client sends frames of OpenGL commands to be rendered by a cluster of workstations on its behalf. Gasparello et al. in [GMBTB11] deal with the distribution of OpenGL command streams over a network by using their own Chromium-like system as their base framework. In order to have a good network utilization, they propose the use of in-frame and inter-frame compression. Inter-frame compression aims on eliminating or at least reduce the redundant data that exist between consecutively frames. For example and in the case of a scene where there is only one moving object, lets say the camera, only the translation commands need to be streamed. They achieve this by using the open-vcdiff tool, an open source implementation of the VCDIFF delta encoding algorithm and file format.

The RFC 3229 proposed standard as its current state [RFC3229], introduces delta encoding in HTTP. The RFC 3229 tries to deal with problem of serving slightly and frequently modified resources for which the client already has one or more older instances in its cache. Based on the observation that the modification of a resource is much smaller then the resource itself, this RFC proposes delta encoding to be used in such cases, avoiding that way of sending redundant bytes on the response.

Although the RFC 3229 works well for same URL responses, it is not suitable for content that is dynamically created with varying URL query parameters. For example search results pages. To overcome this limitation, Butler et al. in [BLM16] propose the SDCH , pronounced as “sandwich” [Li15] , protocol. In their proposal, a dictionary file is shared between the server and the client, containing strings that have high chances of appearing in future HTTP responses. If both sides support SDCH and the client does not have any dictionary from server, or has an outdated version, the latest dictionary file is sent out of band. If both sides support SDCH and the client has a valid copy of the dictionary, then the HTTP response is represented as references to strings in the shared dictionary. The compression scheme is named as SDCH encoding and is VCDIFF based.

1.3 - Solution Approach

Even though there is some work done about data and mostly web page content transmission over the Internet by using delta encoding, there is no work, at least in our knowledge, that explicitly covers transmission of 3D models and their levels of detail

over HTTP based on this technique. Nevertheless the adoption promise of delta encoded HTTP responses is quite encouraging.

The main advantage of using this kind of technique, from our point of view, is that without extending or modifying the X3D standard we can support the discrete and continuous level of detail frameworks simultaneously. Fulfilling that way the two promises that were given for offering support of the two different LOD frameworks in a unified way and keeping the need for changes minimal. In fact, in this work the X3D standard was kept as is.

We can also fulfill the promise of not altering the human readable representation form that the X3D uses for describing 3D models. More precisely the IndexedFaceSet node. Additionally and given that the delta compression is a context agnostic compression scheme we can also support binary formats and scene assets other than 3D models, even though this was not our goal.

Even though we are aware that by implementing a solution that already exists in the literature of the continuous framework might yield better performance results and smaller network bandwidth requirements, we follow an ubiquity and interoperability over performance solution approach. An overview graphical representation of our proposed solution is given on Figure 2.

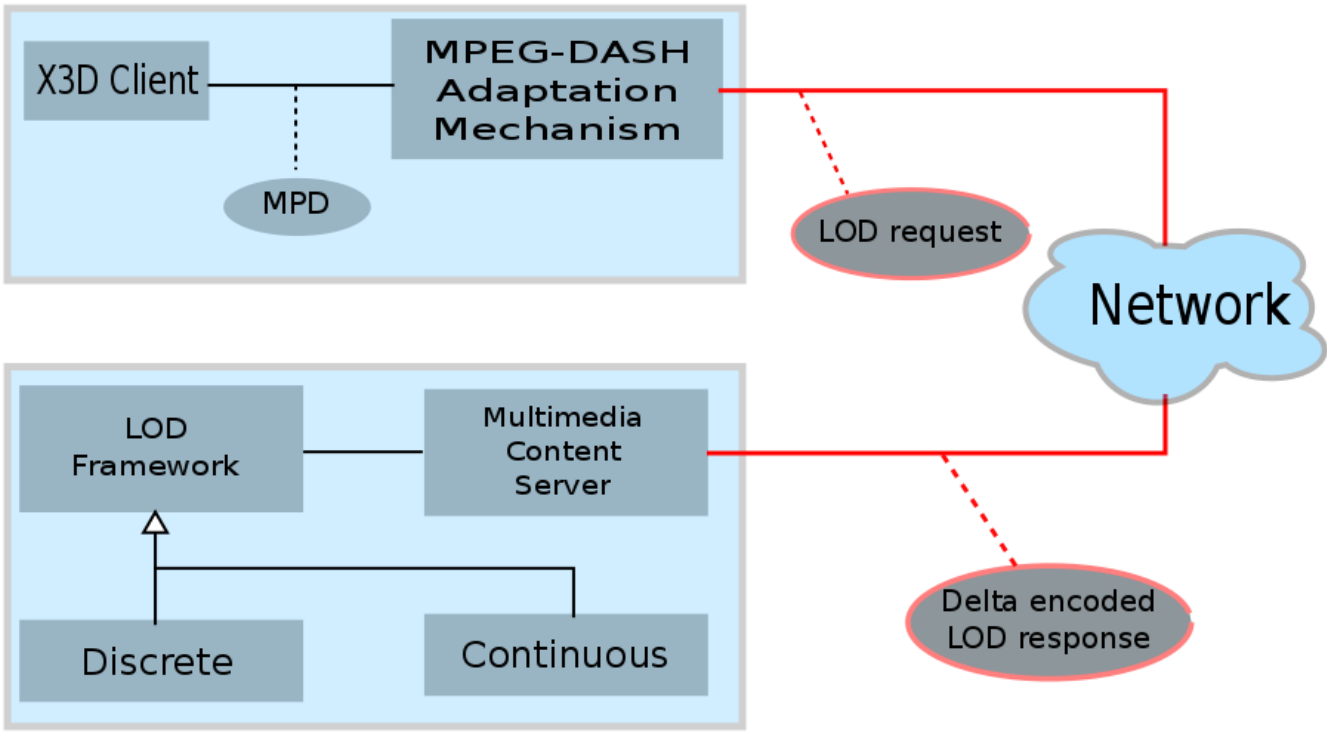

Figure 2: Proposed system overview

As we can see on Figure 2 and on the client's side, we have the X3D client using an MPEG-DASH adaptation mechanism. This adaptation mechanism is responsible for taking performance metrics and adapting the LOD of the models which are described in the MPD file. In the case that the desired LOD does not exist on the client, the MPEG-DASH adaptation mechanism sends the appropriate request to the server using HTTP and additionally advertises the current LOD that already has and on which the delta patch will be applied on.

On the server's side, when that kind of request arrives, the LOD Framework mechanism extracts the requested LOD. Then the delta between the requesting LOD and the current LOD that the client already has is computed and sent to the client as an HTTP response.

Architecture wise, our proposed solution has the advantage of minimal changes on the web server that serves the scene assets. That is because the LOD Framework mechanism can be added in the current configuration in the form of a module.

Chapter – 2

2.1 – Background research

In this chapter we are going to present the background research that has been made. First we will make ourselves familiar with what is a 3D model and how it is represented as a polygon mesh. Next, we will discuss some useful data structures for storing polygon meshes that we can use to extract the desired fidelity of the 3D model using the continuous level of detail framework.

In general, the extraction of the desired fidelity is achieved by using a mesh simplification algorithm that consecutively simplifies the model starting from the original version until the most coarse version is reached. Then we encode the steps that were taken in a way such that we can extract the desired level of detail at any given time.

We will also present the small research that has been made about geomorhping. As mentioned earlier in the previous chapter the geomorphing technique was investigated as a solution approach for delivering the model's levels of detail. But the – roughly - calculated overhead that is required for the transmission is discouraging.

2.2 – 3D Models and Polygon Meshes

A thee dimensional model, or 3D model is a mathematical representation of a three dimensional surface. These models can be created manually by using a 3D designing software for example, or by 3D scanning of real world objects. The most widely representation scheme used for 3D models is the polygon mesh.

The polygon mesh is a mesh of vertices, edges and faces that describe a polyhedron object. The vertices are points in 3D space that are described by their coordinates, for example $V_n = (x_n , y_n , z_n)$. The edges are straight lines that connect two vertices and can be described as $E_n = (V_a , V_b)$. Faces are simple convex or concave polygons. A simple polygon is a polygon that consists of non-intersecting line segments, or in other words there are no pairs of edges that cross each other. In a convex polygon, there is no internal angle that exceeds 180^o , in a concave polygon there is at least one

internal angle that is larger than 180^{o}. In computer graphics, the most used type of faces is the triangle and more rarely quadrilaterals, faces with four vertices and edges. Figure 3 gives an example of simple and complex polygons. Figure 4 gives a graphical representation of the relationships between the vertices, edges and faces.

The connectivity or the topology of a mesh refers to how the vertices are connected in order to form the edges and faces of the mesh. The geometry of a mesh refers to the coordinates of the vertices. Thus for a given 3D conceptual representation of an object we can have different meshes with the same geometry and different topology. Figure 5 gives a graphical representation about the geometry and topology of a mesh.

Meshes can be manifold or non-manifold. A mesh can be considered manifold if every edge touches only one or two faces and faces that are incident to a vertex form a closed or an open fan. Also in a non-manifold mesh, adjacent faces can share o single vertex without sharing an edge. Some mesh simplification algorithms require only manifold meshes as an input. Figure 6 gives a visual representation about manifold and non-manifold meshes.

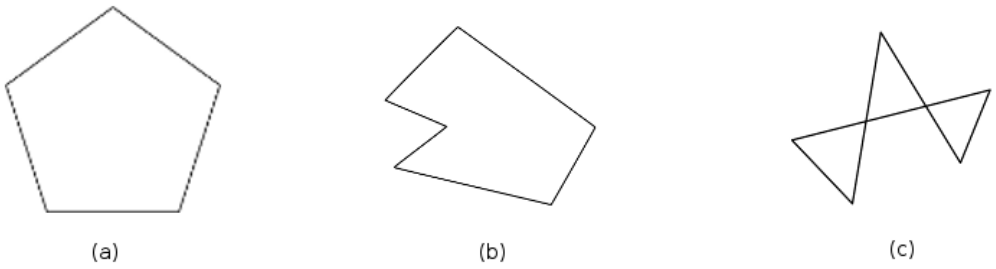

Figure 3:

Polygons (a) and (b) are simple polygons.

Polygon (c) is a complex polygon.

Polygon (a) is convex.

Polygon (b) is concave.

The shapes were taken from:

https://en.wikipedia.org/wiki/Convex_polygon#/media/File:Pentagon.svg

https://en.wikipedia.org/wiki/Concave_polygon#/media/File:Simple_polygon.svg

https://en.wikipedia.org/wiki/Complex_polygon#/media/File:Complex_polygon.svg

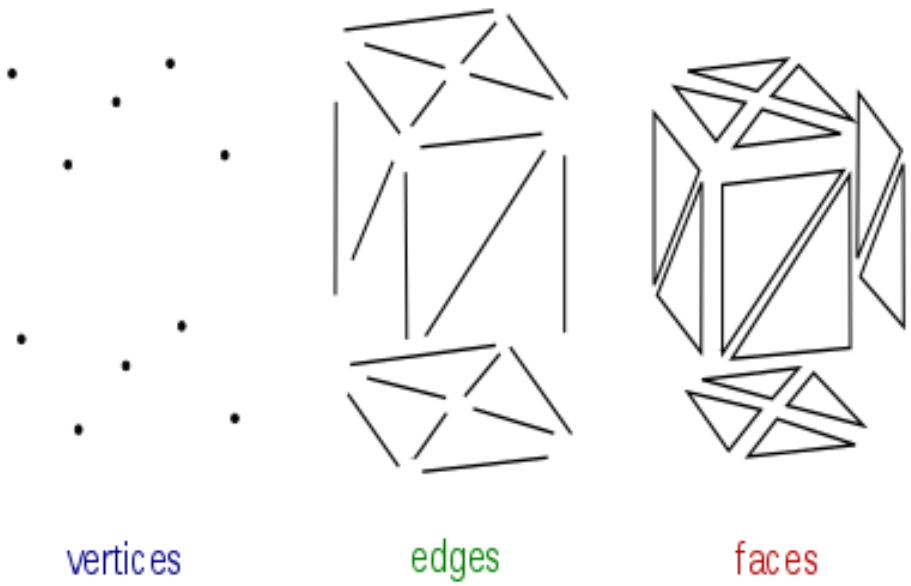

Figure 4: Relationship between vertices, edges and faces. *This figure is a part from https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Mesh_overview.svg*

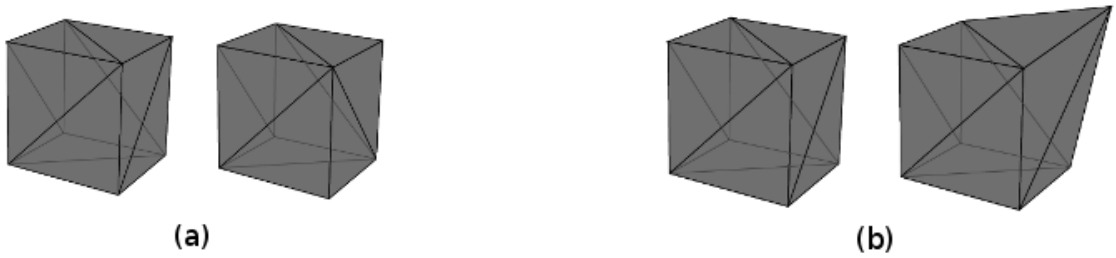

Figure 5:

In (a) we have same geometry with different topology.

In (b) we have same topology with different geometry.

Imagery taken from: http://www.cs.dartmouth.edu/~cs77/slides/07_meshes.pdf

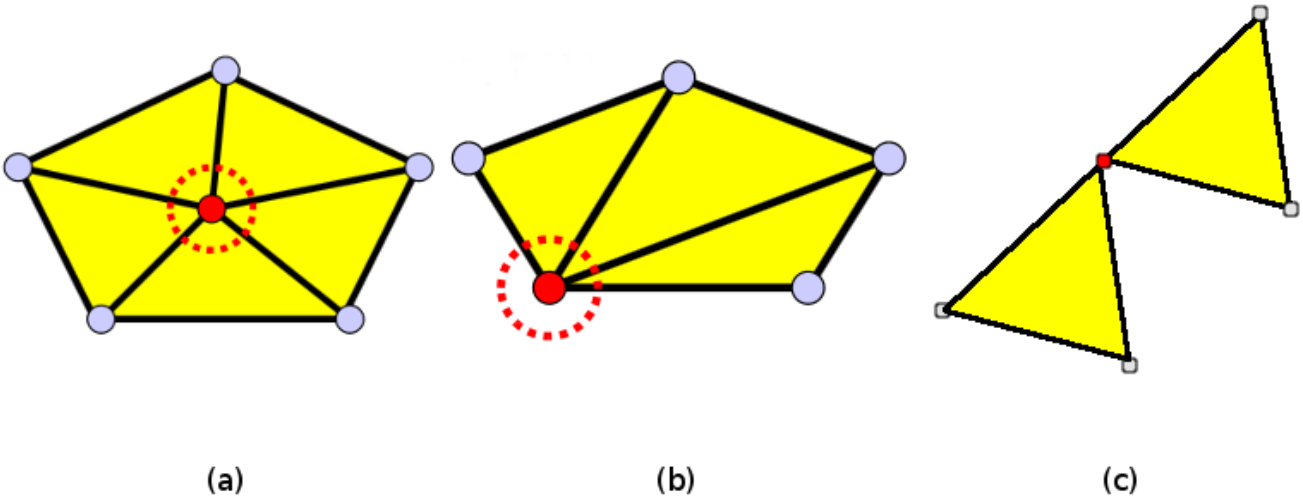

Figure 6:

(a): A manifold mesh forming a closed fan.

(b): A manifold forming an open fan.

(c): A non-manifold where two faces share a single vertex and no edge.

Part of the images was taken from:

https://www.cs.mtu.edu/~shene/COURSES/cs3621/SLIDES/Mesh.pdf

There are two main representation schemes of polygon meshes that are used, among others. First we have the explicit vertex list representation, also known as Vertex-Vertex meshes, where every group of vertices represents a face. The other representation scheme, also known as Face-Vertex meshes, uses two lists, an indexed list of vertices that holds their coordinates and a list of vertex indices. Every group of vertex indices represents a face. The Face-Vertex representation scheme is also used by the X3D to describe polygon meshes. More precisely, the IndexedFaceSet node is of that type. Figure 7 describes these two representation schemes.

Vertex Coordinates(x,y,z)
(0,0,0)
(0,1,1)
(0,1,0)
(0,0,0)
(0,0,1)
(0,1,1)
(0,0,1)
(1,1,1)
(0,1,1)
...continued

(a)

	Vertex Coordinates(x,y,z)
0	(0,0,0)
1	(0,0,1)
2	(1,0,1)
3	(1,0,0)
4	(0,1,1)
5	(1,1,1)
6	(1,1,0)
7	(0,1,0)

Vertex Indices
0
4
7
0
1
4
1
5
4
...continued

(b)

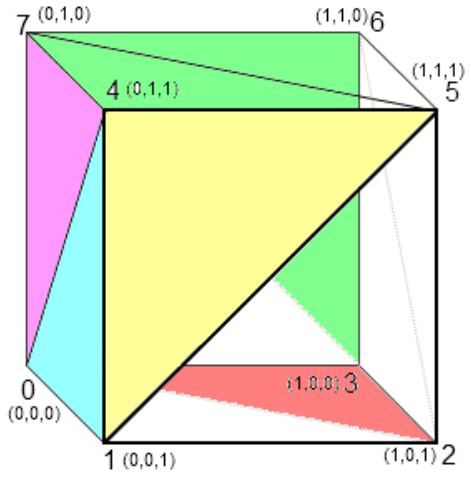

Figure 7 : (a) is a Vertex-Vertex mesh, (b) is a Face-Vertex mesh.

Imagery taken and mixed from: http://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/meshes/polygon_meshes.html

2.3 – X3D's IndexedFaceSet Node

As described in [WX3D], X3D essentially is an XML document, meaning that every entity that composes a 3D world and its interaction is described in a hierarchy of nodes in a parent-child relationship. Among the nodes that can represent 3D objects, the `IndexedFaceSet` node is used to represent 3D objects in a Face-Vertex manner that was previously described.

This node, as in [X3DIFS] extends the `X3DComposedGeometryNode` and uses groups of vertex indices separated by "-1" in order to form the faces. These indices are 32 bit integers that are defined in the node's `coordIndex` attribute field. To define the vertices, this node uses the `Coordinate` node as its child node. The `Coordinate` node as in [X3DCoo] extends the `X3DCoordinateNode` and uses its `point` attribute field to define 3D coordinates.

```
<IndexedFaceSet solid="true"
                coordIndex="46 0 2 44 -1 3 1 47 45 -1 44 2 4 42 -1
                                        ...
                            313 505 503 321 -1 504 506 314 322 -1 319 321 503 389 -1 504 322 320 390 -1 ">

        <Coordinate DEF="coords_ME_Suzanne"
                    point="0.437500 -0.765625 0.164062  -0.085938  0.789062 0.328125
                                        ...
                            -0.125000 0.859375 0.382812 0.382812 -0.859375 0.382812 0.382812 "
                    />

</IndexedFaceSet>
```

Figure 8: A code example of the IndexedFaceSet node

2.4 – More sophisticated mesh data structures

Even thought the Vertex-Vertex and Face-Vertex meshes have simple and straightforward way to represent 3D models, there are some cases where we need to answer questions about the connectivity of a mesh. These questions are related with the adjacency and incidence of the vertices, edges and faces of the mesh. For example, which faces are incident to a given vertex or which faces are incident to a given face or which edges are adjacent. This kind of queries can be answered using the Vertex-Vertex and Face-Vertex representation schemes, but because no such explicit information is held about the connectivity, we have to traverse the whole geometry data many times for every query. Additionally, we need to be able to make operations on the geometry and connectivity in order to add or remove vertices and faces of the mesh. The operations of vertex and face removal are important to mesh simplification algorithms that can reduce the fidelity of a 3D model. These two reasons led to the creation of other data structures in order to solve that problem, but with the cost of more memory usage.

There are quite a few mesh data structure that solve that problem, the most used one though is the half-edge, which is an extension of the winged-edge data structure but they both work only on oriented and manifold meshes. The main idea of these two data structures is that for every vertex of the mesh, we hold references or pointers to the other elements of the mesh.

2.4.1 – Winged Edge data structure

Baumgart in [Bau75] presents the winged-edge data structure. In his work, a polyhedron consists of four types of nodes. These are the bodies, faces, edges and vertices. The body node is a head of a ring of faces, a ring of edges and a ring of vertices. A ring is a doubly linked circular list with a head node.

In this data structure, each face and vertex point to one edge. Each edge points to two faces and two vertices. Finally each edge points to four edges, two in a clockwise direction and two in a counter-clockwise direction. The last four pointers form a conceptual wing and this why this data structure got that name. Figure 9 gives an illustration of the winged-edge data structure.

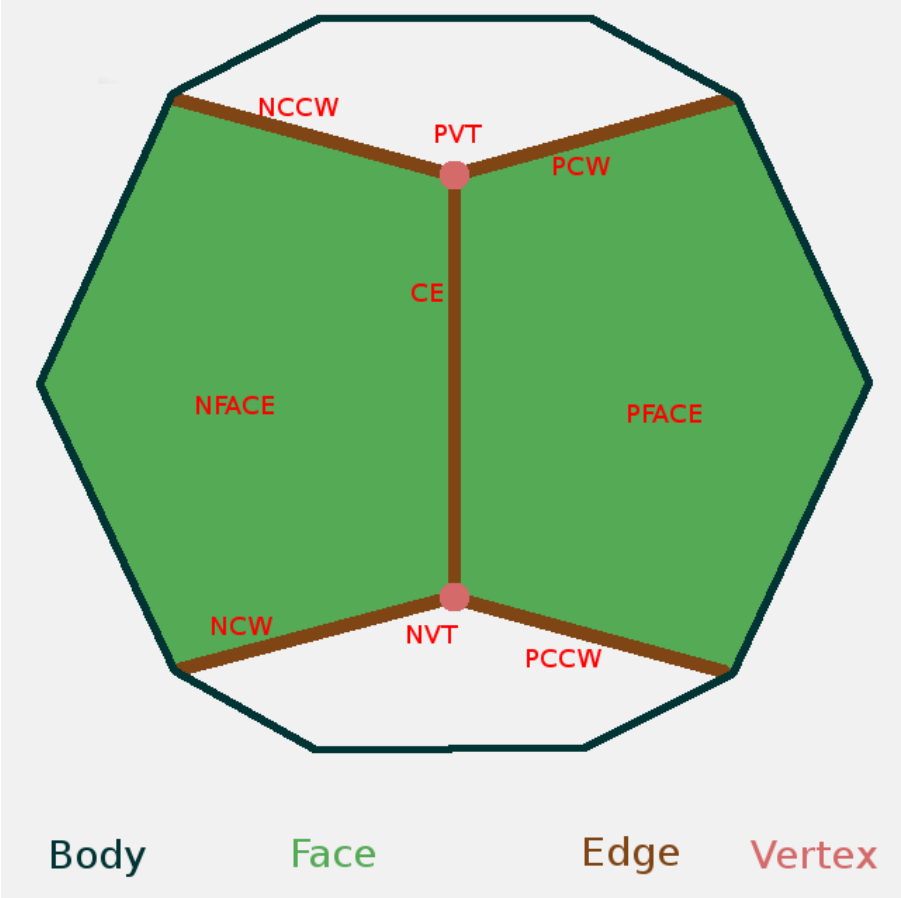

Figure 9: Winged Edge overview

While standing on the CE and while looking up, we can define the following pointers:

- NFACE: The next face.
- PFACE: The previous face.
- NCCW: The next edge in a counter-clock wise order.
- NCW: The next edge in a clockwise order.
- PCW: The previous edge in a clockwise order.
- PCCW: The previous edge in a counter-clock wise order.
- PVT: The previous vertex.
- NVT: The next vertex.

A sample code implementing the data structure in the C language could be the following:

```
struct Edge
{
      Edge *nccw, *pcw, *ncw, *pccw;
      Face *nface, *pface;
      Vertex *pvt, *nvt;
      // Other edge data
};

struct Face
{
      Edge *edge;
      // Other face data
};

struct Vertex
{
      Edge *edge;
      // Other vertex data
};
```

As mentioned above, in order to implement a continuous LOD technique on our input mesh, we must be able to make operations on the mesh such as face, edge and vertex insertion and removal. Baumgart in his work besides describing the data structure, also gives us two reference procedures that we can use. The MKFE procedure, or “Make Face-Edge”, adds a pair of a face and vertex into the surface topology. The KLFE, or “Kill Face-Edge” procedure removes a face-vertex pair. Figure 10 [Bau75] gives an illustration of the effects of applying these two procedures. Below that illustration, the refined pseudo-code for the two procedures is given.

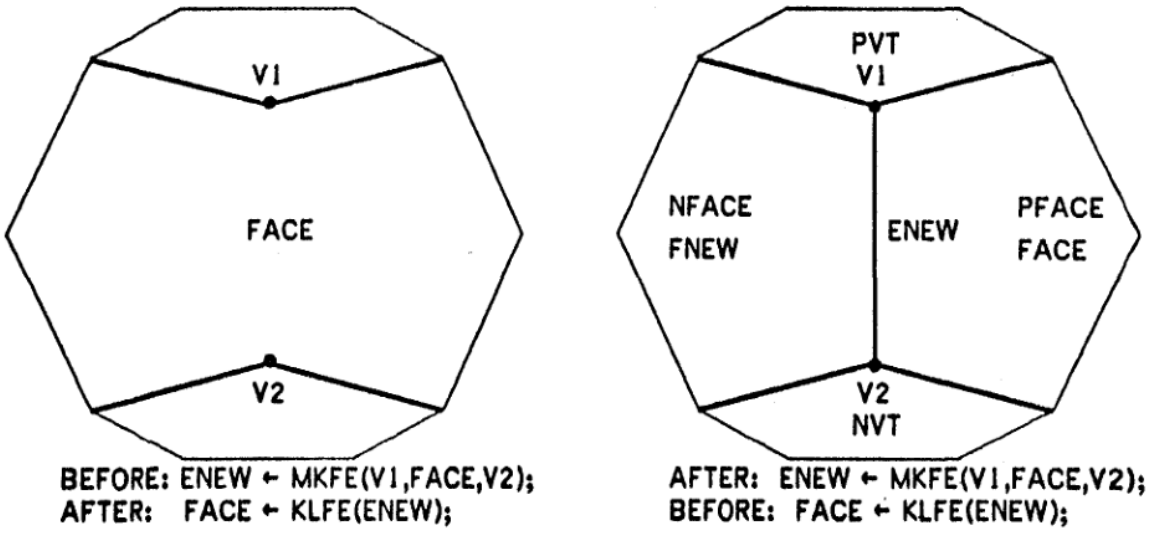

Figure 10: The effects of applying the MKFE and KLFE procedures

```
INTEGER PROCEDURE MAKE_FACE_EDGE (INTEGER V1,V2,FACE);

BEGIN “MAKE_FACE_EDGE”

      // CREATE NEW FACE & EDGE
      FNEW ← MAKE_FACE(FACE);
      ENEW ← MAKE_EDGE(PREVIOUS_EDGE(FACE));

      // LINK NEW EDGES TO ITS FACES & VERTICES
      PREVIOUS_EDGE(F) ← PREVIOUS_EDGE(FNEW) ← FNEW;
      PREVIOUS_FACE(ENEW) ← F;
      NEXT_FACE(ENEW) ← FNEW;
      PREVIOUS_VERTEX(ENEW) ← V1;
      NEXT_VERTEX(ENEW) ← V2;

      // GET THE WINGS OF THE NEW EDGE
      E2 ← PREVIOUS_EDGE(V1);
      DO
            E2 ← NEXT_EDGE_CW( (E1 ← E2), V1 )
      UNTIL
            NEXT_FACE_CW(E1, V1) = FACE;
      E4 ← PREVIOUS_EDGE(V1);
      DO
            E4 ← NEXT_EDGE_CW( (E3 ← E4), V2 )
      UNTIL
            NEXT_FACE_CW(E3, V2) = FACE;

      // SCAN CCW FROM V1 REPLACING FACE WITH FNEW;
      E ← E2;
      IF PREVIOUS_FACE(E) = FACE THEN
            PREVIOUS_FACE(E) ← FNEW;
      ELSE
            NEXT_FACE(E) ← FNEW;

      // LINK THE WINGS
      WING(E1, ENEW);
      WING(E2, ENEW);
      WING(E3, ENEW);
      WING(E4, ENEW);

      RETURN(ENEW);

END;
```

```
INTEGER PROCEDURE KILL_FACE_EDGE (INTEGER ENEW);

BEGIN “KILL_FACE_EDGE”

      // PICKUP ALL THE LINKS OF ENEW
      FACE ← PREVIOUS_FACE(ENEW);
      FNEW ← NFACE(ENEW);
      V1 ← PREVIOUS_VERTEX(ENEW);
      V2 ← NEXT_VERTEX(ENEW);
      E1 ← PREVIOUS_EDGE_CW(ENEW);
      E2 ← NEXT_EDGE_CCW(ENEW);
      E3 ← NEXT_EDGE_CW(ENEW);
      E4 ← PREVIOUS_EDGE_CCW(ENEW);

      // GET ENEW LINKS OUT OF FACE, V1, V2
      IF PREVIOUS_EDGE(V1) = ENEW THEN
            PREVIOUS_EDGE(V1) ← E1;
      IF PREVIOUS_EDGE(V2) = ENEW THEN
            PREVIOUS_EDGE(V2) ← E3;
      IF PREVIOUS_EDGE(FACE) = ENEW THEN
            PREVIOUS_EDGE(FACE) ← E3;

      // GET RID OF FNEW APPEARANCES
      E ← E2;
      DO
            IF PREVIOUS_FACE(E) = FNEW THEN
                  PREVIOUS_FACE(E) ← FACE;
            ELSE
                  NEXT_FACE(E) ← FACE;
      UNTIL
            E4 = (E ← NEXT_EDGE_CCW(E, FNEW));

      // LINK WINGS TOGETHER ABOUT FACE
      WING(E2, E1);
      WING(E4, E3);
      KILL_FACE(FNEW);
      KILL_EDGE(ENEW);

      RETURN(FACE);

END;
```

2.4.2 – Half Edge data structure

The Half-Edge is probably the most widely used data structure when it comes to computational geometry. As stated in [Zcg12] and we agree with that, the origin of the current form of the Half-Edge is hard to find. Nevertheless, in order to find out if two convex polyhedra intersect each other, Muller and Preparata in [MP78] presented the

Doubly Connected Edge List as the base data structure of their algorithm, which its logic is identical.

In the Half-Edge data structure, every edge is split into two parts, the two halves of the edge, that have opposite directions. Those two parts are called half-edges, hence the name of the data structure. This data structure does not explicitly describe any edges, instead the edges are implied by their two half-edges.

Every half-edge points to its opposite twin half-edge. Additionally, every half-edge stores a target vertex but no origin vertex, as opposed to the edge that has one start and one end vertex. Given that the mesh is oriented and that twin half-edges look at opposite directions, if we want to get the origin vertex of a half-edge we need to get the target vertex of its twin. Also and given that the Half-Edge data structure is oriented counter-clock wise, the left half-edge always touches a face and the right always touches its twin. Below there is a sample code in C implementing the Half-edge data structure. Figure 11 gives an illustration of the data structure. The blue arrow, denoted by *h* is one of the edge's half-edge and the arrow on its right is its twin.

```
struct HalfEdge
{
	HalfEdge* heTwin;	// The twin half-edge
	HalfEdge* heNext;	// The next half-edge
	HalfEdge* hePrevious;	// The previous half-edge, this is optional
	Vertex*	vTarget;	// The target vertex
	Face*	face;		// The bordering face

	// Other data
}

struct Vertex
{
	HalfEdge* he; // The half-edge that starts from the vertex

	// Other data
}

struct Face
{
	HalfEdge* he; // One of the half-edges that surrounds the face

	// Other data
}
```

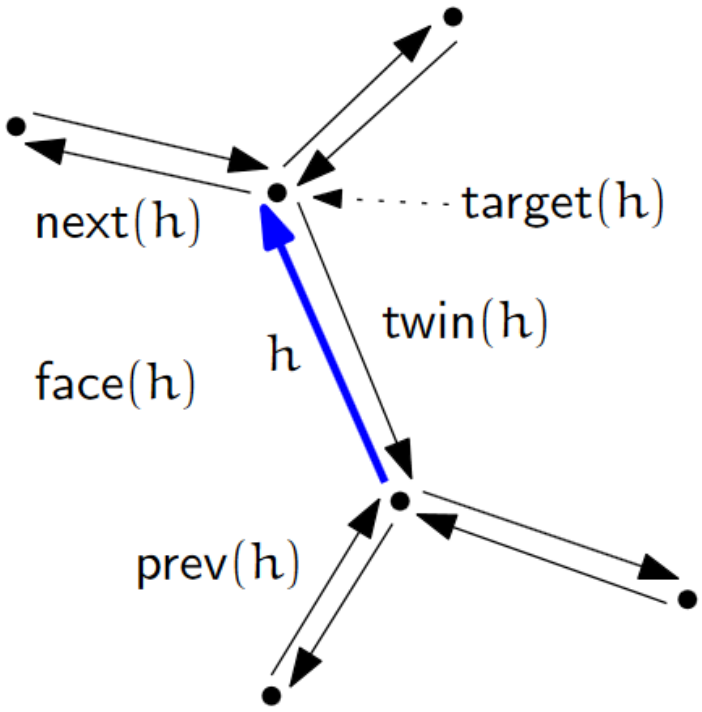

Figure 11: Half-edge overview [Zcg12]

The following enumerated list gives a summary of what the elements of a mesh point to. Figure 12 gives an illustration of this list.

1. Every vertex points to the one outgoing half-edge.
2. Every face points to one arbitrary half-edge that is inside its boundary. A face can be surrounded by many half-edges. In the case of a triangular mesh, a face is surrounded by three half-edges.
3. Every half-edge points to its target vertex.
4. Every half-edge points to its touching face.
5. Every half-edge points to its next half-edge.
6. Every half-edge point to its opposite – twin half-edge.
7. Optionally, every half-edge points to its previous half-edge.

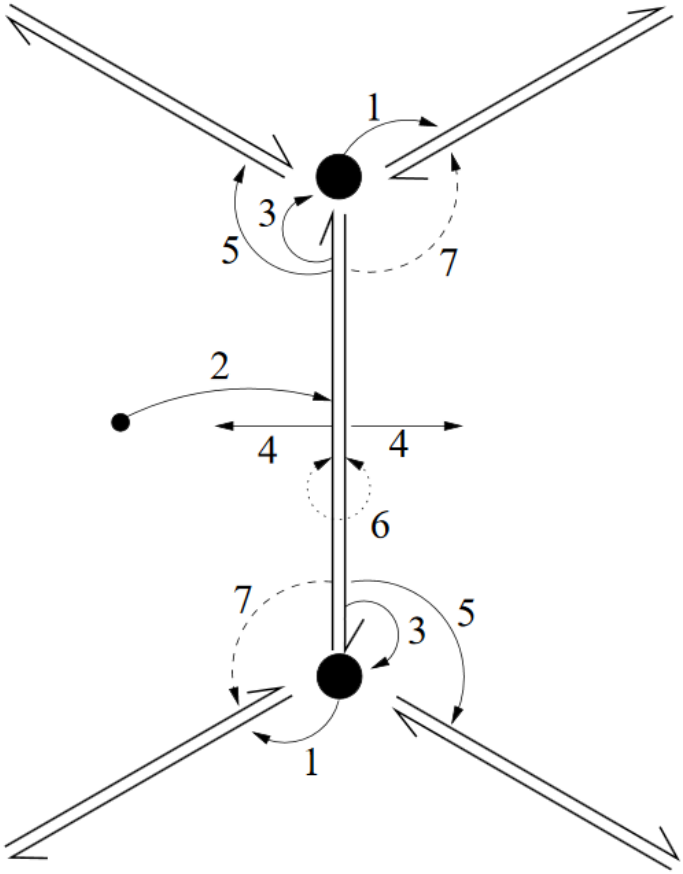

Figure 12: Illustrated Half-Edge enumerated list of references [BSBK02]

The code below is an example of how to find adjacent edges for a given face. The idea is to race through all the half-edges pointed by the heNext pointer of the previous half-edge until we meet the half-edge from which we started.

```
HalfEdge* heStart = face→he;
HalfEdge* heRunner = heStart;

do
{

      heRunner = heRunner→heNext;

} while (heRunner != heStart);
```

2.4.3 – Lath based data structures

Assuming that the geometry can be expressed by vertices, Kenneth I. Joy et al. in [JLM03] present the lath data type. Each lath element can be connected to another lath element and that body of connections express the topology of the mesh. A single lath can be identified by using a record of a vertex, an edge and a face. Also, each of the face-edge, face-vertex, edge-vertex pairs can be associated with a single lath element. Figure 13 gives an illustration of a half-edge mesh representation implemented with the lath data type.

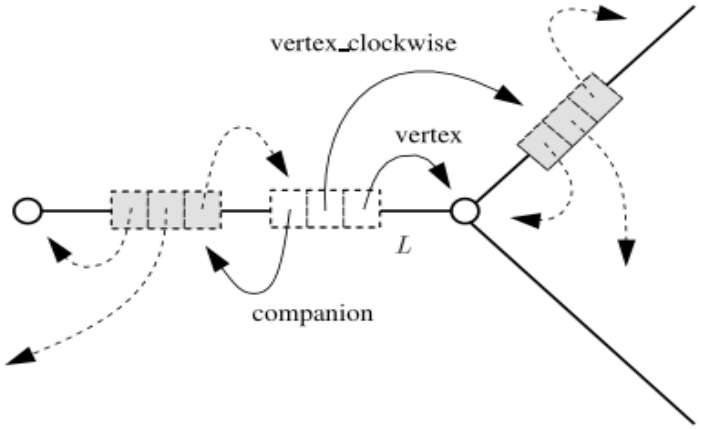

Figure 13 [JLM03]

As we can see, a lath element *L* holds a reference to a single vertex. The *"companion"* field points to the lath L_{comp}. The L_{comp} lays on the same edge as *L* and references the opposite vertex of the edge. Thus an edge can be described as a pair of laths that have this *"companion"* relationship. The "vertex_clockwise" field points to the next lath in a clockwise vertex traversal. Figure 14 shows that the lath's contiguous structure forms two kinds of loops, one in a clockwise direction around a vertex and one in a counter-clockwise direction inside a face.

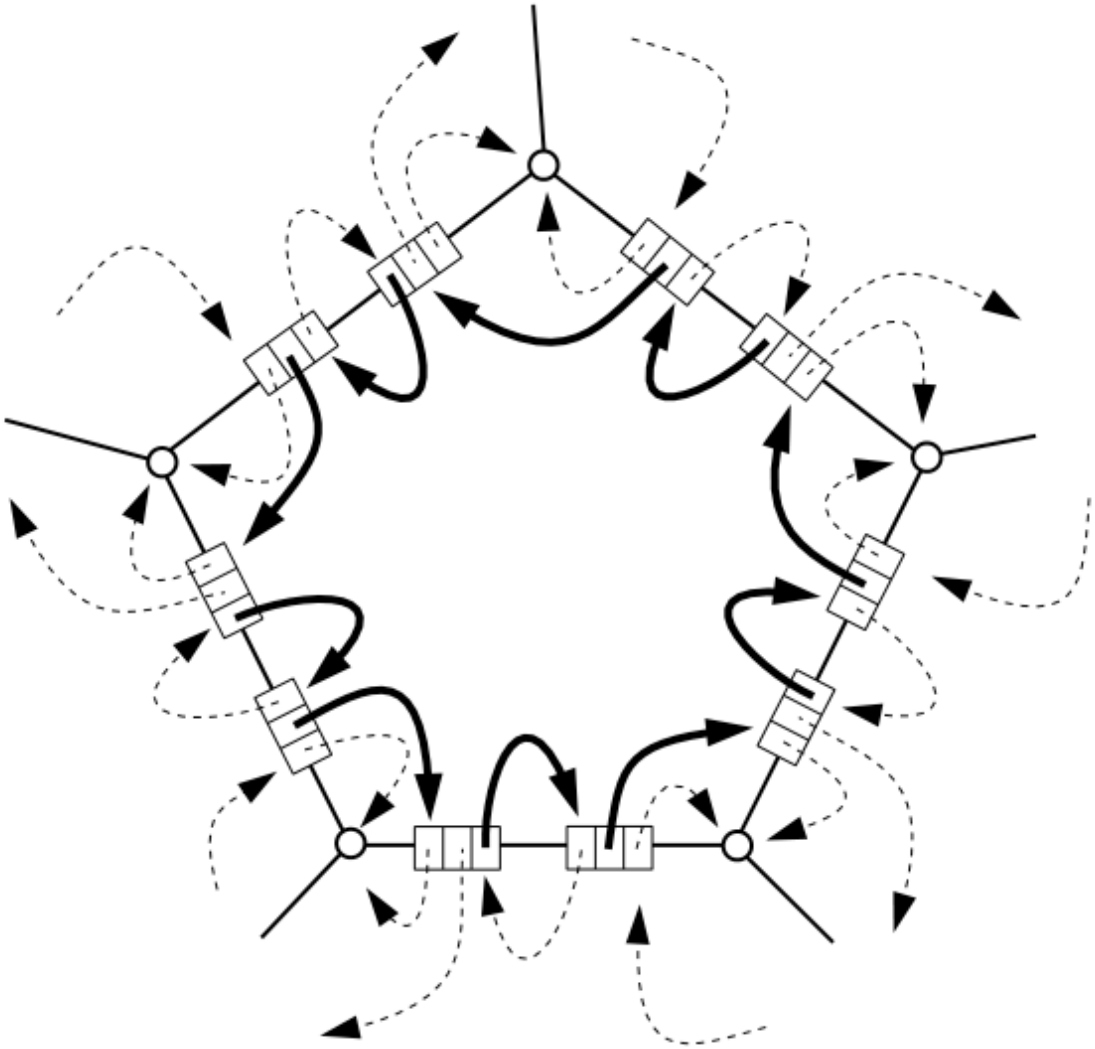

Figure 14 [JLM03] The laths form two kinds of loops, a clockwise around a vertex and a counter-clockwise inside a face.

The traversal of the mesh elements can be done by using the following operations:

- ec(*L*): return the *L*'s edge companion.
- cv(*L*): return the lath that follows *L* that is defined in the *L*'s *"vertex_clockwise"* field.
- ccf(*L*): return the lath that follows *L* in a counter-clockwise traversal of the face that *L* represents.
- cf(*L*): return the lath that follows *L* in clockwise traversal of the face.
- ccv(*L*): return the lath that follows *L* in counter-clockwise traversal around the *L*'s vertex.

2.5 – Level of Detail

The Level Of Detail in applications that use 3D models, such as computer games, is a technique for representing a 3D model in different levels of fidelity. What this means for the geometry of a model is that we can reduce or increase the number of vertices. This can be useful in cases where the available processing power for rendering the model with a given geometry complexity is insufficient or when the model is placed away from the camera. For example, a dodecahedron when viewed from a far distance can be perceived by the human eye as a sphere. For that reason rendering the full geometry of the model would be a waste of computational resources.

The size of 3D models is increasing day by day. Thus they need more memory for storage, as well as more computing power to be rendered. Although the computing and storage capabilities, even in home computing, become noticeably better year by year, the Internet's average speed does not keep up with the same pace. This becomes a problem when 3D models that are above the medium size, need to be transmitted as a Web3D's scene assets. In that case and in order the model to be viewed, the user will have to wait for an undesirably long time until the model is fully loaded.

Nah in [NF03] suggests that the average website user is willing to wait for at most two seconds until the web content is loaded. As in [Rail16] is suggested that interactive content has to be delivered in under one second. To achieve this, we can send to the user

a coarse version of our model and then gradually refine it. This approach has the advantage of keeping the user occupied while the full model is loaded, resulting in that way to a better user experience.

Also and given that the LODs are created by simplifying the input mesh, we need a way to determine if the simplified output is visually pleasant. As mentioned above, there are two ways to simplify a mesh. Either by hand or programmatically. In the first case, we have the opportunity to evaluate the simplified output ourselves but this is not always the case when taking the second approach.

As in Garland's work [Ga99], we need a way to estimate how much similar the input and output meshes are. One approach is to render the two meshes and then calculate the differences of the their produced images. This approach has the advantages of measuring directly the perceptible similarity of the meshes and that not visible details can be discarded. On the other hand we have to render the meshes from all the possible viewpoints. Another approach although is to measure the similarity on the geometry level.

Kapetanakis in [KK14] extends the MPEG-DASH standard in order to support adaptive 3D models. In general, every asset of the scene is described as an *Adaptation Set.* If the asset is separated into different LODs, such as 3D models, these LODs are described as *Representations.* The code below is taken as a part from [ML16] and gives an example of a model and its LODs described in an MPEG-DASH manifesto.

```
<MPD>

    <BaseURL>http://mclab1.medialab.teicrete.gr:8081</BaseURL>
    <BaseURL>http://localhost:8081</BaseURL>
    <BaseURL>http://alternativeHost:8081</BaseURL>
    <BaseURL>http://alternativeHost2:8081</BaseURL>
    <BaseURL>http://alternativeHost3:8081</BaseURL>
    <BaseURL>http://alternativeHost4:8081</BaseURL>

    <Period id="3d_model">

        <AdaptationSet mimeType="model/x3d+xml" codecs="none" minFrameRate="10">


            <Representation id="6" bandwidth="300000" qualityRanking="4">
                <BaseURL>cat3.x3d</BaseURL>
            </Representation>

            <Representation id="7" bandwidth="500000" qualityRanking="3">
                <BaseURL>cat2.x3d</BaseURL>
            </Representation>

            <Representation id="8" bandwidth="1000000" qualityRanking="2">
                <BaseURL>cat1.x3d</BaseURL>
            </Representation>

            <Representation id="9" bandwidth="2000000" qualityRanking="1">
                <BaseURL>catOrig.x3d</BaseURL>
            </Representation>

        </AdaptationSet>

    </Period>

</MPD>
```

2.5.1 – Discrete LOD Framework

In the DLOD framework for every input model, a sequence of gradually coarser and look alike models is created. These output models are individual entities, meaning that the geometry and topology might be similar but they are disjoint. This simplification process is done before runtime. There are mesh simplification algorithms and tools that give an automatically generated hierarchy of LODs, although sometimes this process is

preferred to be made by a human to give a fine tuned result. The video in [Utube16] shows an example of a handmade mesh simplification process. Some tools that can be used to automatically generate LODs can found in [ADM16] and [BDM16].

The X3D standard offers the `LOD` node that enables us to manage a hierarchy of LOD models in camera-to-object distance manner. Every LOD model is included as a child node of the `LOD` node. The selection of which LOD model will be rendered for the current object-to-camera distance is determined by the `range` attribute. Bellow we can find the node's description as defined in [WX3dL]. A live example along with its source code can be found in [XfwaL].

```
LOD : X3DGroupingNode {
  MFNode  [in]     addChildren             [X3DChildNode]
  MFNode  [in]     removeChildren          [X3DChildNode]
  MFNode  [in,out] children       []       [X3DChildNode]
  SFNode  [in,out] metadata       NULL     [X3DMetadataObject]
  SFVec3f []       bboxCenter     0 0 0    (-∞,∞)
  SFVec3f []       bboxSize       -1 -1 -1 [0,∞) or −1 −1 −1
  SFVec3f []       center         0 0 0    (-∞,∞)
  MFFloat []       range          []       [0,∞) or -1
}
```

The DLOD framework is widely used in 3D games because it is very easy to implement. All we need to do is to create a hierarchy of LOD models and then render the most appropriate one. Another advantage is that because the mesh simplification takes place in an offline preprocess, the runtime is free of any mesh simplification algorithms. Thus the cost of processing power for this framework is low.

On the other hand and because these LOD models are individual entities, when transmitting them there will be redundant data in between the LODs. That is because even though the geometry might be similar there is no obvious way to leverage the in between similarities to make a cumulative transmission. This is solved by using the continuous LOD framework, as we will later discuss, because it allows a progressive transmission which unfortunately comes with a complex implementation and higher processing power requirements.

2.5.2 – Level of detail transitions

When using the DLOD framework, the switching between the LODs is abrupt and easily perceptible by the viewer, giving the sense that the 3D object “pops” when the camera is moving near or away from it. This visual artifact is called the *popping effect*. In order to eliminate it or at least reduce it, we must give a smooth transition between the start and target LOD models. For that reason, the geomorphing and alpha blending techniques are used and applied on the mesh level and image level respectively.

In particular in the alpha blending technique, we draw the two LOD models simultaneously one on top of the other and interpolate the transparency values in a short period of time. The main disadvantage of alpha blending is that we need to render the two models at the same time, increasing that way the displayed geometry. This becomes even a bigger problem when we want to switch to a coarser version in order to free up some computing resources. Nevertheless, Scherzer and Wimmer in [SW08] represent an algorithm that renders the two LODs in subsequent frames and in that way we avoid to simultaneously render those two. Figure 15 gives an illustration of the alpha blending transition approach.

Another approach of giving a smooth LOD transition is by using morphing or geomorphing. In general, in the morphing technique, the shape of an object gradually changes from a starting form to another by interpolating between the two input geometries. Figure 17 gives an example of mesh morphing. The problem with this kind of interpolation, is that we need to have a one-to-one vertex correspondence and for that reason the two interpolating models must have the same number of vertices.

To overcome this limitation, Lee et al. in [LDSS99] present a method of morphing between multiresolution meshes. As an overview, they reduce the geometry of both input meshes in order to build the two bijective source-to-target and target-to-source mappings. These mappings are then realized to as what they call the *metamesh*, which is the merged version of the two input meshes. By their estimations although, the size of the metamesh can reach up to 10 times the size of the larger mesh. This makes it inappropriate for our problem’s solution. The Figure 16 is an excerpt of their work that shows the metamesh’s size for four different mesh morphings.

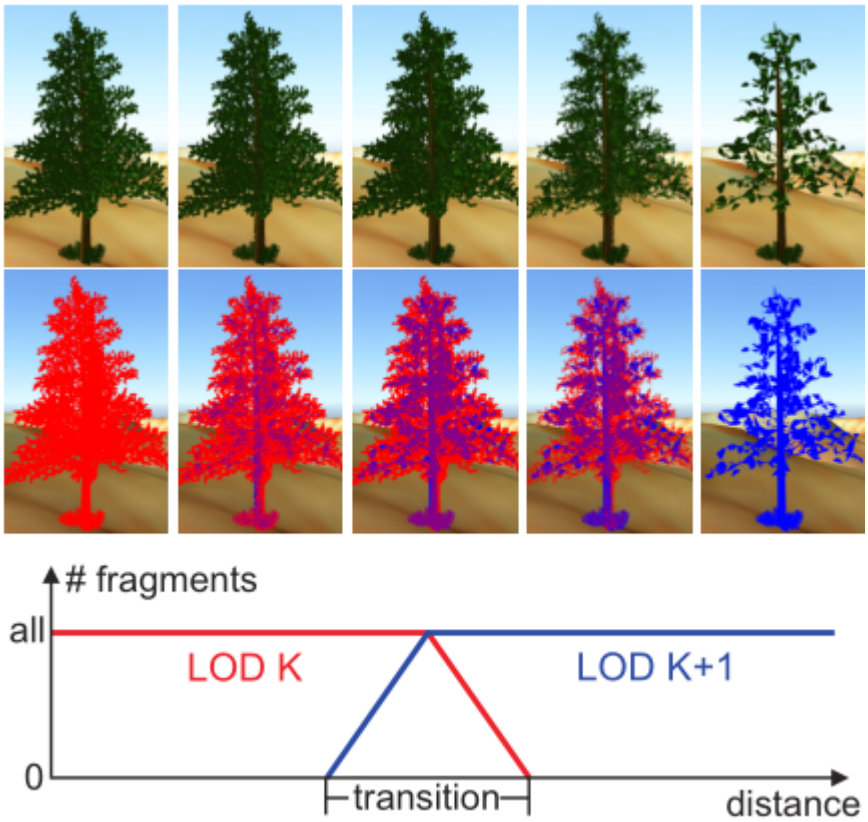

Figure 15 An example of a alpha-blending transition. Taken from [SW08]

Source-Target	Source size (triangles)	Target size (triangles)	Metamesh size (triangles)
mann-venus	5422	90709	225502
cup-donut	8452	2048	43188
mann-spock	5422	14100	75427
horse-rabbit	21130	21582	220201

Figure 16: Metamesh's size [LDSS99]

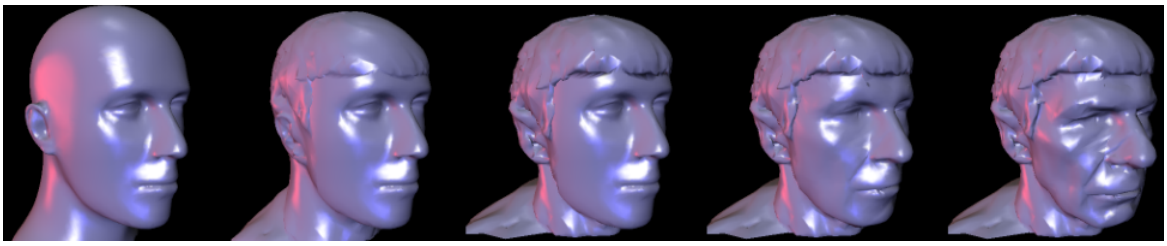

Figure 17: Example of mesh morphing [LDSS99]

The methodology's main idea that is used in [LDSS99] can also be found in many other approaches in this research area. In Parus' work [Par05] we can find a general description of the steps that we have to follow. First, for every vertex in the source mesh we need to find a corresponding vertex on the target mesh. Note that because the two meshes might not have the same number of vertices, some of the source's vertices will be mapped to a point somewhere near the area covered by the target mesh. The next step is the same as the previous one but in the opposite direction. Then the *supermesh* is constructed by merging the two input mesh's topologies. Finally, by using the *supermesh* we can bidirectionally interpolate between the meshes. For that reason and at least intuitively, the size overhead is not appropriate for our needs. Nevertheless, the research in this area is still active and we do not exclude the chance of a solution approach based on mesh morphing.

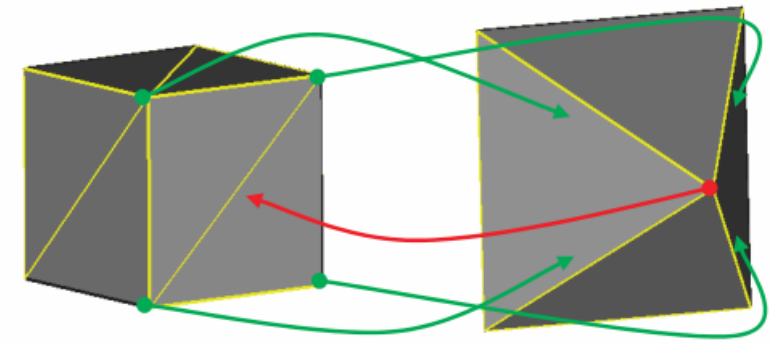

Figure 18: Source to target vertices correspondence (green arrows) and target to source mesh vertices correspondence (red arrow) [Par05]

2.5.3 – Continuous LOD Framework

In the CLOD framework, the model is encoded in a form that it allows us to extract the desired LOD from a "continuous spectrum" of LODs. Hoppe in [Hp96] introduces the progressive mesh representation. In the PM form, an input polygon mesh M is stored as a coarse mesh M^0, along with a series of n refinement records. Thus the sequence $M^0, M^1, \ldots, M^n$ describes a continuous spectrum of LODs, with M^n as the original input mesh. For that reason, the PM representation scheme can support progressive transmission by first sending the base mesh M^0 and later the refinement records.

Hoppe in his work expresses a mesh as a tuple M = (K, V, D, S). Where K and V describe the connectivity and vertex positions. D and S describe the discrete and scalar attributes respectively. The attributes D and S are indicative of visual discontinuities in the mesh's appearance, Figure 19 illustrates that case.

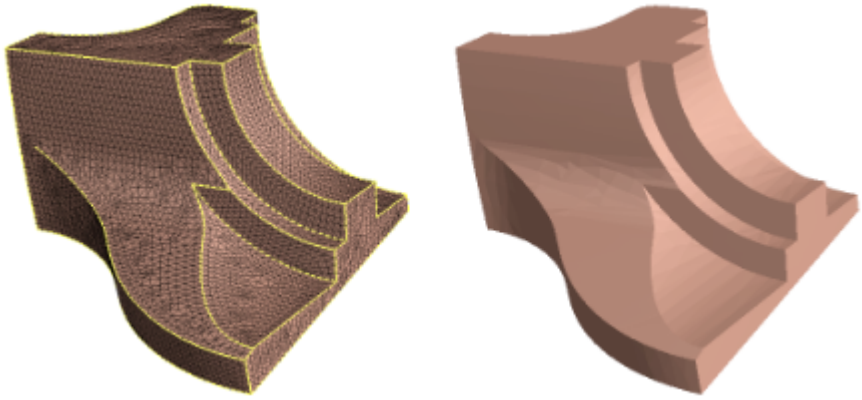

Figure 19: The visual discontinuities are marked as yellow lines [Hp96]

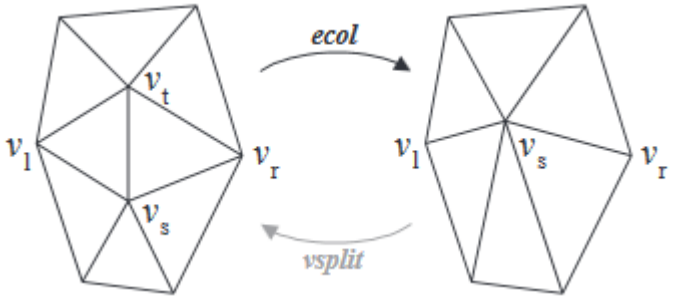

Figure 20 edge collapse and *vertex split* transformations [Hp96]

In order to produce the base mesh M^0 a mesh optimization algorithm iterates the input mesh and at each step an edge is removed. This edge removal transformation $ecol(v_s, v_t)$ called as *edge collapse,* removes the edge by collapsing v_t onto v_s. Figure 20 gives an illustration of the edge collapse transformation and as you can see the incident faces to the edge (v_s, v_t) are removed as well. In addition, the *edge collapse* transformation is invertible. The inverse transformation $vsplit(s, l, r, t\, A)$ called as *vertex split* adds a new vertex at the position *t* and two new faces $\{v_s, v_t, v_l\}$ and $\{v_t, v_s, v_r\}$.

Also, the sequence of edge collapses determines the quality of intermediate LODs. This depends on the mesh simplification algorithm. For example, an easy to implement mesh simplification algorithm, is to remove a random edge at each step, but most likely the result's visual quality will be very low. Although and because this mesh simplification algorithm is executed before the run time, Hoppe in his work takes the approach of investing some time in order to meet a better visual quality.

In a nutshell, there are three steps that we have to follow in order to create a progressive mesh. First, the mesh simplification algorithm iterates the input mesh and produces a sequence of *edge collapse* records. Then the *vertex split* records are created in the reverse order of the *edge collapse*'s records sequence. Finally, we write to a file the base mesh M^0 along with the vertex split records. Now we can transmit the M^0 and later transmit the vertex split records one by one in order to progressively refine the mesh until we reach its original form. Furthermore, at each refinement step we can apply geomorphing to avoid the popping effect.

A technique based on a vertex by vertex refinement scheme offers fine granularity but it has the disadvantage of imposing a big overhead. Pajarola and Rossignac in [PR00] propose an alternative approach to Hoppe's PM representation. In their work, they group the edge collapses into batches. This results into a batch based mesh refinement scheme instead of sending the refinement records one by one. Their approach achieves better compression but compromises with a coarser granularity.

Figure 21 illustrates a comparison between single rate transmission and progressive transformation approaches. a expresses the time needed to send the coarse version of the model. The dashed line curve illustrates the case where after a we send the original model. Note that even though it results in a poor user experience, the overall loading time is the shortest. Approaches based on PM are illustrated by the grey curve, we can see that they offer a fine granularity but in the expense of a long loading time.

Finally the batch based approach is illustrated by the staircase curve which makes a compromise between granularity and loading time.

Limber et al. in [LJBA13] introduce the POP buffer method. The model's coordinates are mapped to a cluster of nested grids of integer coordinates with different quantization levels. Then by using a truncation function they can increase of decrease the grid's resolution. If the two points of an edge are mapped to the same grid point, then the edge is degenerate. Figure 22 illustrates the *cell merge* and *cell split* operations. The triangles marked in red will become degenerate on the grid with smaller resolution. Finally, the triangles are sorted in the reversed order in which they degenerated. They call this reordered sequence of triangles as the *Progressively Ordered Primitive* buffer. That way the progressive transmission in this method is straightforward, all we need to do is to push to the back the incoming vertices and triangles.

Melax in [Sm98] gives a simple, yet quite effective polygon reduction algorithm. In fact, we slightly modified a ported version of his implementation to Javascript [Gzz85] and used it as our CLOD framework on the server side. In his work, the algorithm iterates the mesh and applies an edge collapse operation until the desired number of vertices is removed. The vertex pairs that will be collapsed are selected by calculating the edge's length multiplied by a curvature term. The information of edge collapses is kept and the vertices are sorted by the collapsing order and we can use this sequence of collapses to achieve progressive transmission.

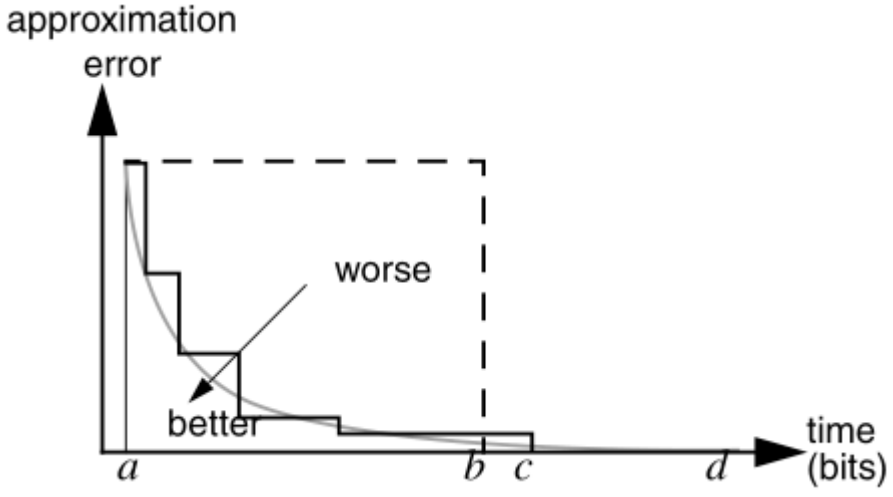

Figure 21 [PR00]

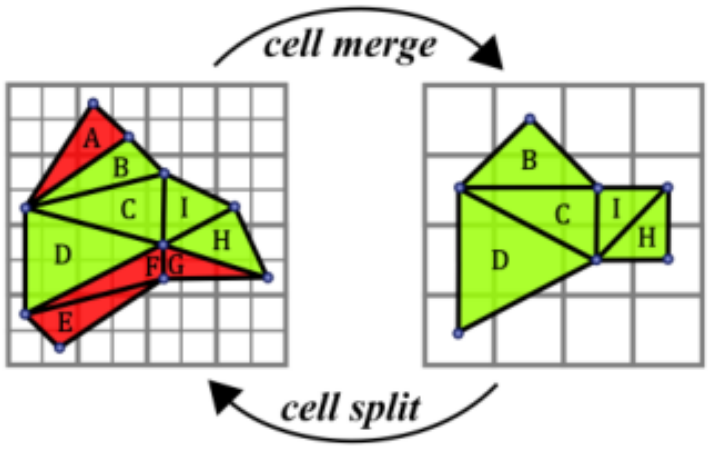

Figure 22: [LJBA13] cell merge and cell split operations

2.6 – Delta Compression

Delta compression or delta encoding is a technique for encoding files in the form of differences. Given the previous and current version of a resource, we can create a patch file that describes how to change the previous version in order to reconstruct the current. Suel and Memon in [SM02] give a more formal definition. Consider two files $f_{new}, f_{old} \in \Sigma$, where Σ is an alphabet, the client C and the server S and the case where C has a copy of f_{old} and S has both f_{old} and f_{new}. We need to compute a delta file f_{δ} such that $sizeof(f_{\delta}) < sizeof(f_{new})$ by which C can fully reconstruct the f_{new}. From their work, some of the cases where delta compression is applied are:

- Software Revision Control Systems, where objects are stored in a way that allows the user to retrieve older versions.
- File system delta compression, where delta compression is applied on the file system level.
- Software distribution, where software updates are transmitted in the form of patches.
- Visualize differences between two files.

- Improving HTTP performance by exploiting the similarities between web resources or different versions of the same resource.

An example of using delta compression in software distribution can be found in [SC12]. This work deals with the distribution of app updates in the Android Market where for each update the full updated version of the app is downloaded. Instead of that, they propose the use delta compression and they achieve an average compression of nearly 50%.

The Git SCM initially saves the objects in its repository in a “loose” object format and compresses them using zlib, a non delta encoding compression library. After that, Git packs the objects into a binary file called “packfile”, where delta encoding is used [PGit]. Other SCMs such as Mercurial and Subversion [Mer16] [Sv16] follow different approaches of how and when to use delta encoding but the goal is the same. As for the visualization of differences between edits, Figure 23 is a screenshot of our implementation’s git repository that gives such an example.

Although text based collaborative and version control systems such as SCMs are fairly mature, in the field of CAD and 3D modeling the development of such systems with capabilities of the same quality level did not catch up. Nevertheless, Doboš in [Do15] introduces the 3D Repo, a cloud based version control and collaboration framework for 3D assets that uses a NoSQL database for data storage and retrieval. Again, for reducing storage requirements it uses delta compression.

Gumhold et al. in [GGS99] deal with mesh compression. First they quantize the vertices’ coordinates by splitting each coordinate into four packages of four bits. Then, to achieve even better compression they use delta encoding on the vertex coordinates. Hoppe in [Hp96] mentions that the vertex split is a local operation and for that reason it results to a coherent output where we could use delta encoding. For example, when splitting the vertex $v_{S_i}^{i}$ into the two new vertices, we can predict their positions and then use delta encoding to reduce the required storage space. Also, Limper et al. in [LWSJS13] mention that we should exploit the browser’s existing compression capabilities by using delta encoding along with GZIP compression.

Gasparello et al. in [GMBTB11] deal with compression schemes of real-time streaming of OpenGL command sequences. As an overview, the command streaming system consists of a master computer that sends OpenGL commands to a pool of slave computers to be rendered on its behalf. The master computer runs a custom device driver

that can intercept any OpenGL call and creates a ghost command code. That way the slave computers can replicate the OpenGL calls. Every intercepted OpenGL call is passed through a *packetizer* module that encodes and stores them into a *command buffer*. Next, the delta between the current and previous command buffers is produced and then is compressed with a general purpose compressor. Finally the compressed delta is sent through the network. Figure 24 gives an illustration of the communication of the master and slave computers.

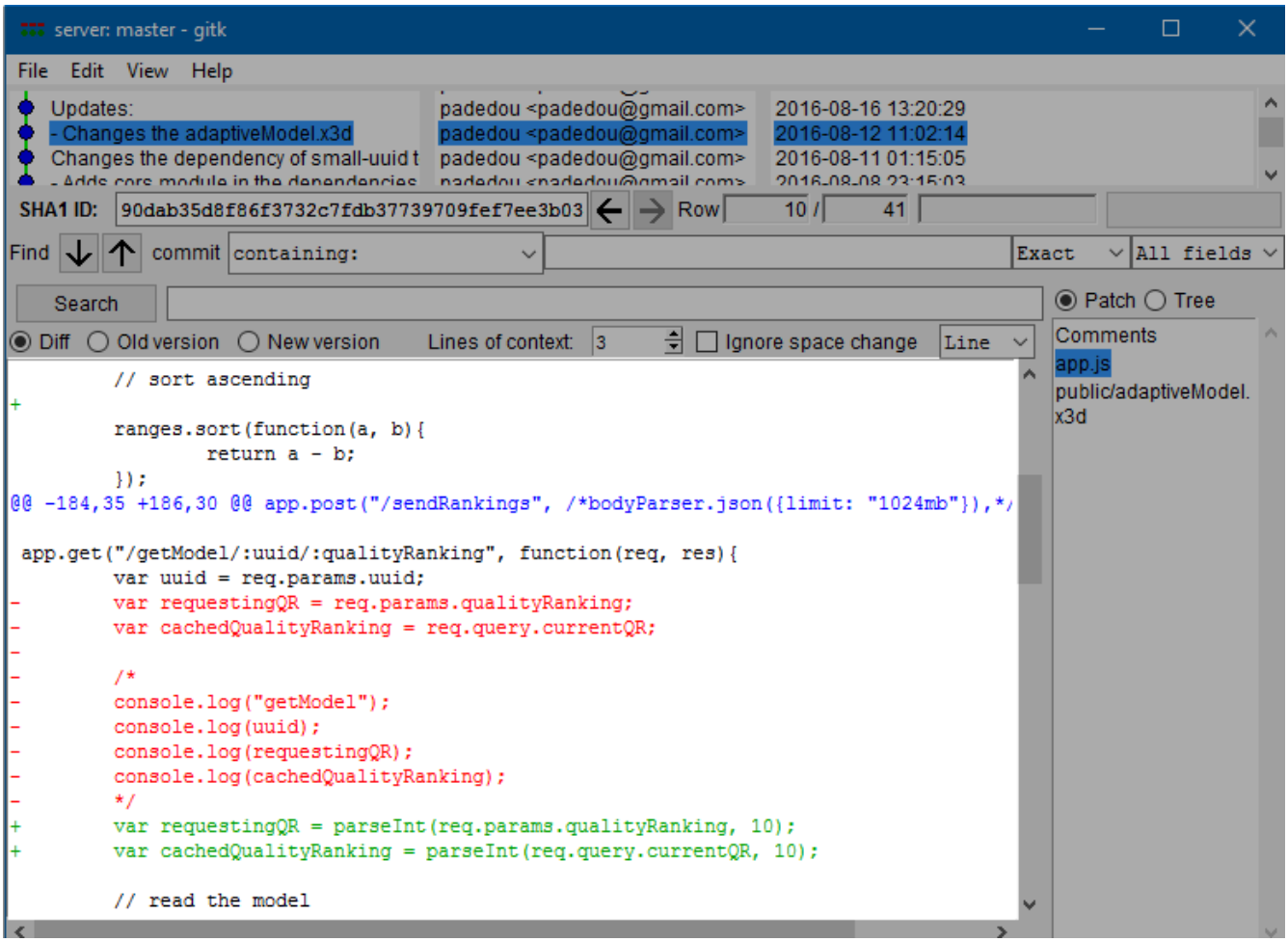

Figure 23: Git GUI - Visualization of edits of a file

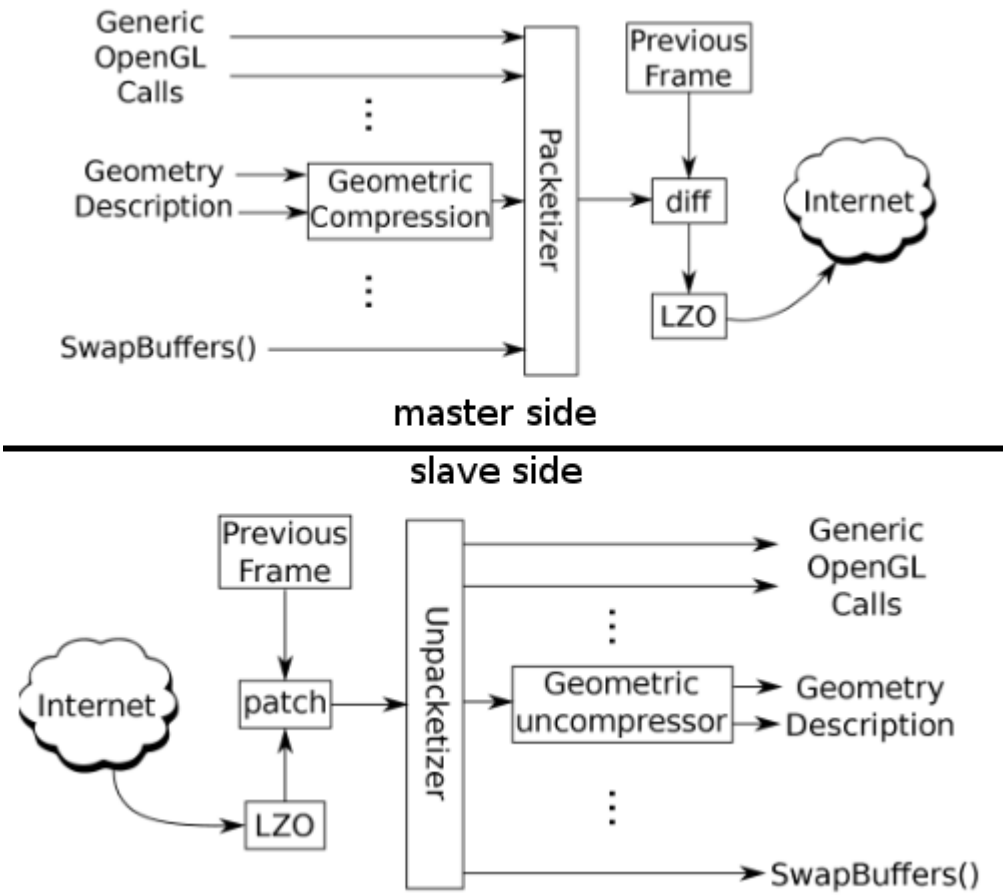

Figure 24: OpenGL commands transmission from master to slave computer [GMBTB11]

Mogul et al. in [MDFK97] try to quantify the potential benefits of delta encoded HTTP responses. In their work they sampled HTTP requests whose URL does not include -practically- any multimedia or binary file extensions. They produced the deltas by using the UNIX command *diff -e* , the compressed output of *diff -e* and *vdelta.* Also they mention from previous work that responses with the same URL prefix are similar, thus making delta encoding effective. In their sampled traces, a fairly big part included URLs containing the "?" character, which suggests a query operation, so we expect effective delta encoding because of same URL prefix. Their results show that the size and delay of HTTP responses is improved when using delta encoding along with data compression. However, using delta encoding is viable only when the imposed overhead is smaller than the potential benefits.

The main goals of the [RFC3229] proposed standard are to reduce the size of HTTP responses, be interoperable with HTTP/1.0 and HTTP/1.1 and optional for the clients and servers. In order to work, it adds optional message headers. The *accept instance manipulation* “A-IM” header for the client and the *instance manipulations* “IM” header for the server. These headers describe which encoding format they are willing to use and which were finally used respectively. Also it proposes that delta encoded responses should be identified with the *226* unassigned code. Figure 25 gives an illustration of the conceptual sequence of transformations that are applied. Figure 26 gives an example of a client requesting the resource `/foo.html` of which it has a cached instance with entity tag “123xyz” and is wiling to accept compressed responses whether or not they are delta encoded.

In RFC 3229 the delta encoded responses only work when they come from the same URL, making it that way unsuitable for URLs with varying querying parameters. The SDCH proposed protocol [BLM16] overcomes this limitation by using a dictionary file that is shared between the client and the server and contains strings that have high chances of appearing in subsequent HTTP responses. The client can retrieve the current dictionary “out of band” and future HTTP responses will include only references to strings in the dictionary, reducing that way the payload size. This compression scheme is referred to as the SDCH encoding and is VCDIFF based.

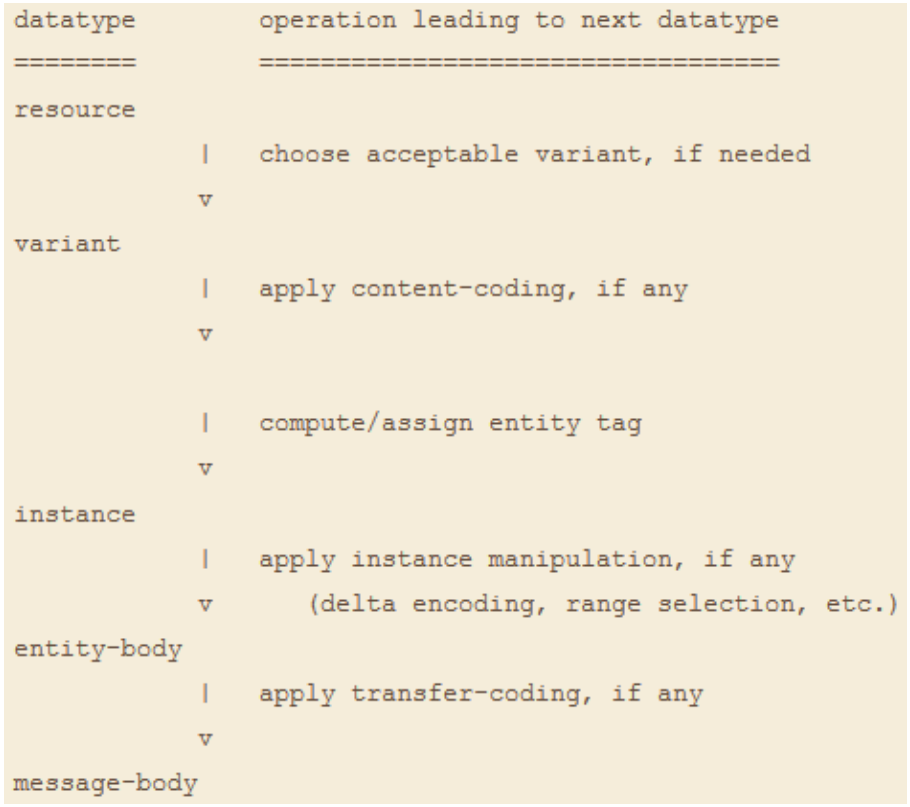

Figure 25: Transformations diagram [RFC3229]

```
GET /foo.html HTTP/1.1
Host: bar.example.net
If-None-Match: "123xyz"
A-IM: vcdiff, diffe, gzip
```

Figure 26: Request example [RFC3229]

Chapter – 3

3.1 – Implementation

In this work, we propose that the LODs should be delta encoded in order to minimize the redundant data and achieve lower payload. On the client side we have the browser which runs the X3DOM as the X3D player. Also, we implemented a simplistic MPEG-DASH adaptation mechanism which is responsible for choosing the appropriate LOD that is available from the given MPD file. This adaptation mechanism is also responsible for sending the appropriate HTTP request to the server and then apply the patch data when the response is received. On the server side, we use the *LOD Framework* module which can extract the desired LOD. When the LOD is extracted, the server computes the delta between the extracted LOD and the client's current LOD and finally responds with the patch data. The overview of our implementation is illustrated on Figure 2. Also, on the same server we host the web application that allows the user to define LODs along with their quality rankings and then produce the MPD file. For the implementation of both the client and server side we used the Javascript language.

3.1.1 – Server overview

For our continuous LOD framework we used and modified a port of [Sm98] from C++ to Javascript [Gzz85] which is based on the Three.js WebGL framework. As for the delta encoding implementation, we used the [plvc] in both the client and the server. For these reasons we chose to implement the server by using the Javascript language and the Node.js [Node] as the runtime environment. As for the web application framework we used the Express framework [Expr]. The project's properties and dependencies are defined in the `package.json` file. To install the defined dependencies we called from the terminal the `npm install` command and the npm [Npm] package manager installed the dependencies from its remote registry. Figure 27 shows the server's `package.json` file.

On the same server we also built and host our web application that allows the user to define the LODs along with the quality rankings by using either the discrete or

continuous LOD framework. After when the quality ranking and LOD pairs are defined, the server produces the appropriate MPD. As a final step of the web application, we display to the user the directions of how to use our MPEG-DASH player. Figure 28 shows the first screen of the web application.

```
{
  "name": "x3dash_server",
  "version": "0.1.2",
  "description": "Serving LODs for MPEG-DASH enabled X3D players",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node ./app.js"
  },
  "keywords": [
    "mpeg",
    "dash",
    "x3d",
    "LOD",
    "melax",
    "three.js"
  ],
  "author": "padedou",
  "license": "MIT",
  "dependencies": {
    "body-parser": "^1.15.2",
    "express": "^4.14.0",
    "xml-wraper": "file:./xml-wrapper",
    "node-uuid": "^1.4.7",
    "pug": "2.0.0-beta4",
    "cors": "^2.7.1"
  }
}
```

Figure 27: The server's package.json file

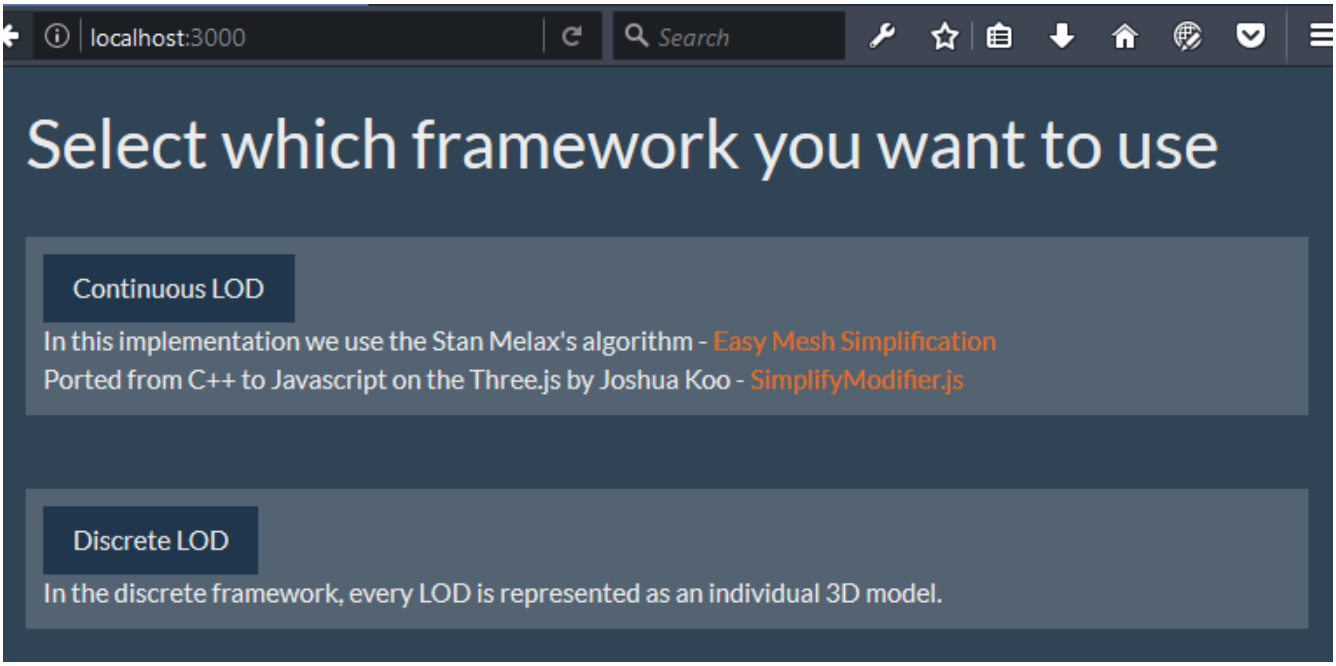

Figure 28: Web application's first screen

3.1.2 – Discrete LOD framework UI

In this screen the user is able to upload the LOD models from his or her filesystem and define for each LOD the network bandwidth that it requires. After that, the screen which contains the directions of how to add the adaptable model into the X3D scene is displayed. Figure 29 shows the upload screen and Figure 30 shows the directions screen.

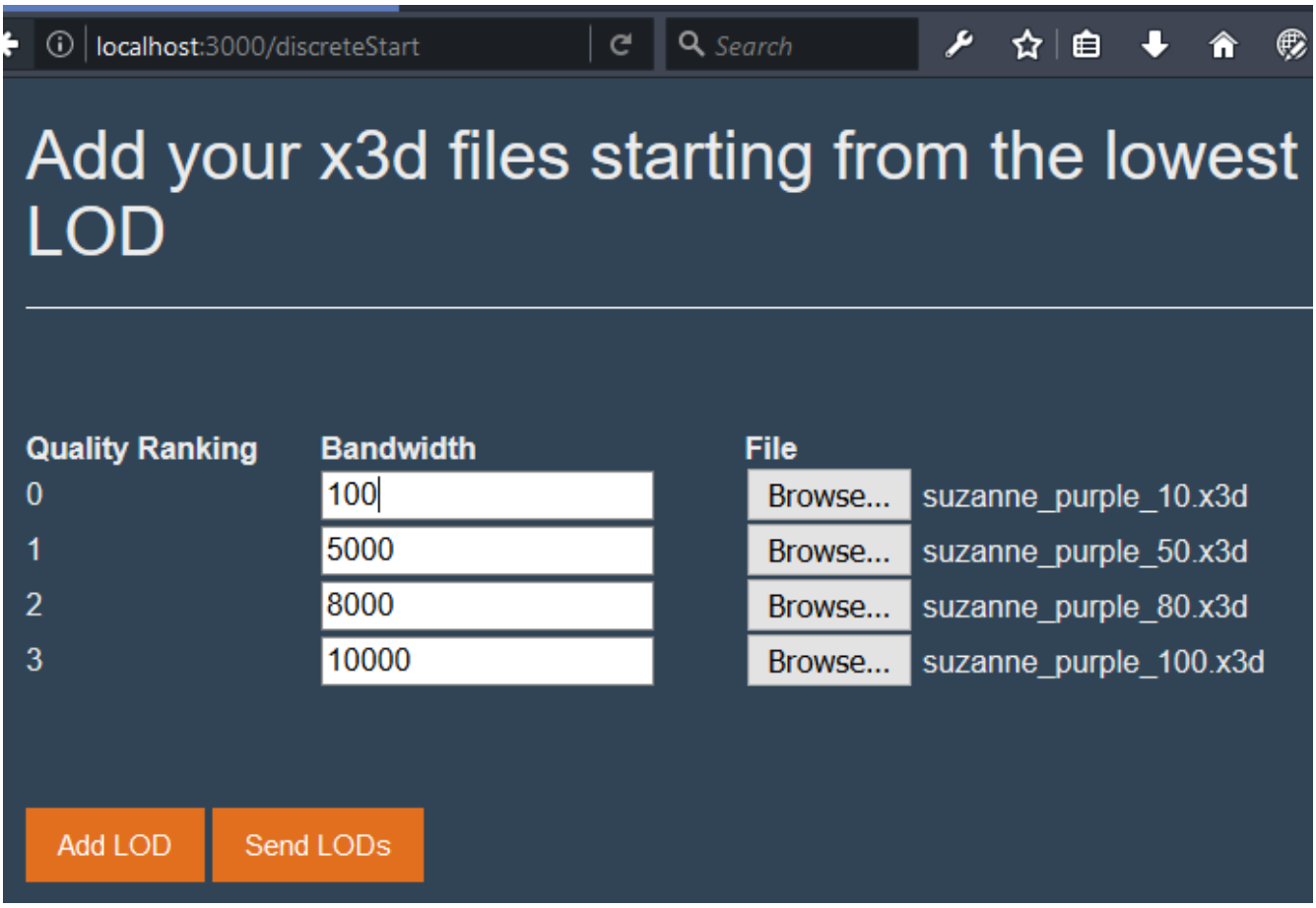

Figure 29: Discrete LOD framework upload screen

Figure 30: Directions screen

3.1.3 – Continuous LOD framework UI

First, we show to the user a menu in which he or she can select a model from a preset list or upload a new one from the filesystem. Then the screen for defining the desired LODs is presented. The selection of the desired LOD is done by moving the slider in the bottom. For each desired LOD the user presses the *Add range* button. When done, the user presses the *Send ranges* button. Figure 31 shows model selection screen and Figure 32 shows the LOD editor screen.

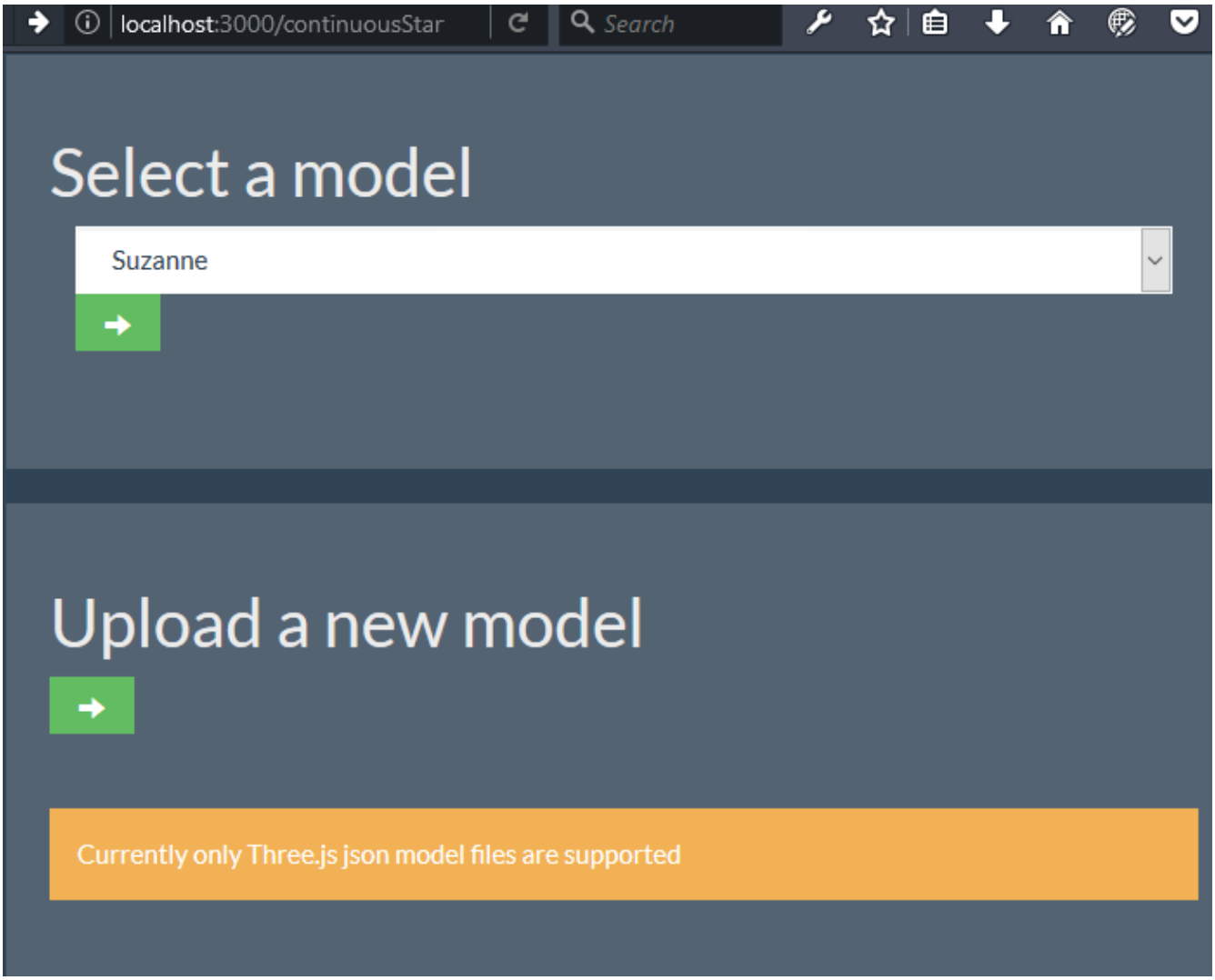

Figure 31: Model selection screen

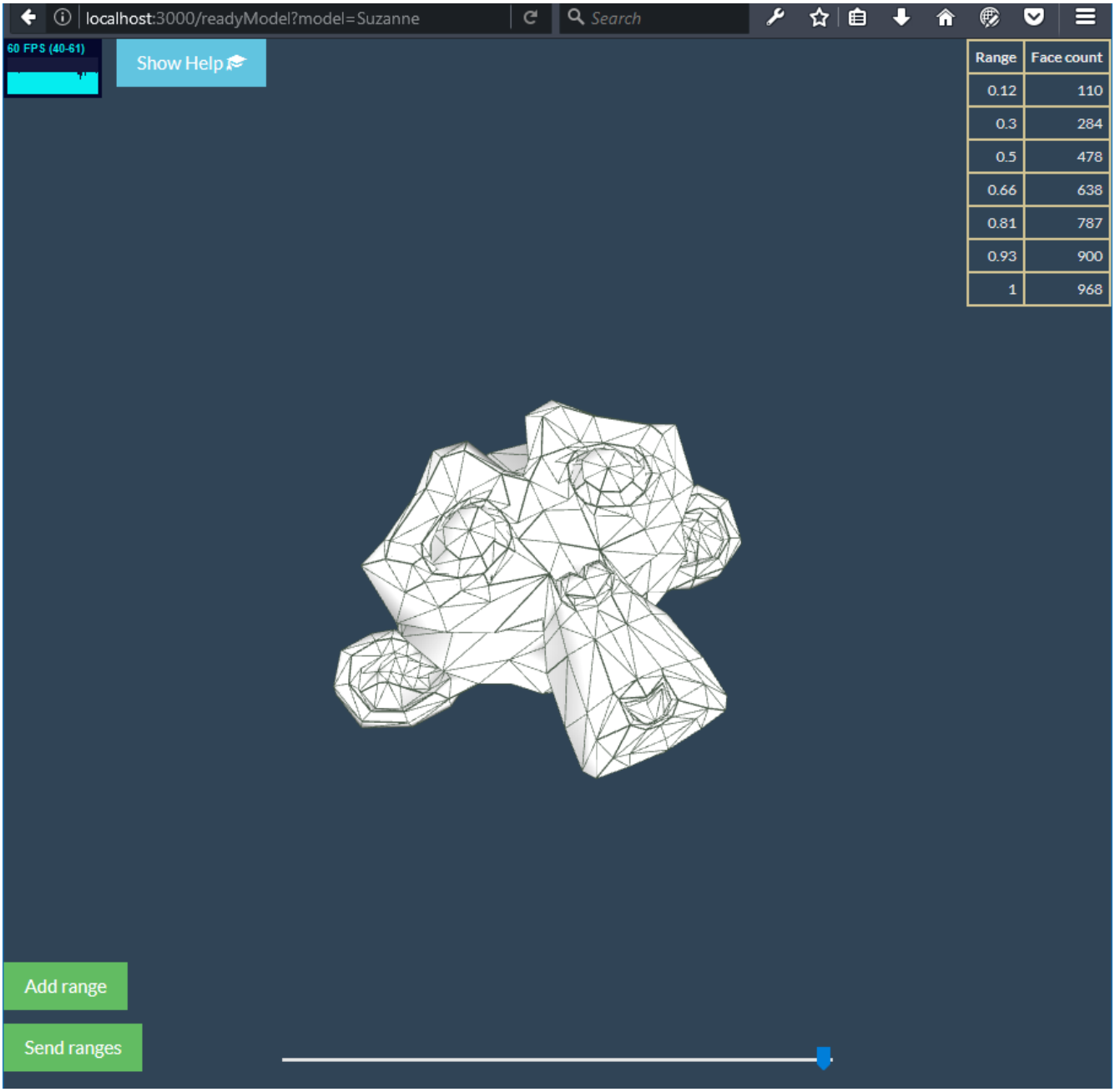

Figure 32: LOD editor

3.1.4 – Client with MPEG-DASH enabled X3D scene

For the needs of demonstration, we implemented a simplistic MPEG-DASH player. Because the implementation of content adaptation mechanisms is not an easy task and fall out of the scope of this work, the selection of the LOD is made in an ascending and descending order of the MPD's available quality rankings in arbitrary time intervals. The X3D scene author needs to include along with our player, the jQuery [jQ] library and the vcdiff Javascript implementation [plvc]. Then, the author must add the references of the adaptive models of the scene to the player by using the DEF attribute. These steps are described in more detail in the instructions page, an example of which is presented in Figure 30. The code that changes the LOD of the model is shown in Figure 33. Figure 34 shows the code that constructs the HTTP request by including the current and the requesting LOD, then when the response is received it applies the patch data. Figure 35 shows an example of a scene where two adaptive models are included. Finally, Figures 36 and 37 are screenshots that were taken from the client's runtime.

```
295     var LODChanger = function(_modelReference){
296       var instance = {};
297
298       var modelReference = _modelReference;
299       var ascending = true;
300       var lodObserver = {};
301
302       lodObserver.notifyDone = function(_status){
303         if(_status === "success"){
304           requestAnimationFrame(showCaseLoop);
305         }
306       };
307
308       instance.start = function(){
309         showCaseLoop();
310       };
311
312       function showCaseLoop(){
313         if(ascending){
314           if(modelReference.maxQualityRanking >= modelReference.currentQualityRanking + 1){
315             modelReference.changeLOD(modelReference.currentQualityRanking + 1, lodObserver);
316           }else{
317             ascending = false;
318             requestAnimationFrame(showCaseLoop);
319           }
320         }else{
321           if(modelReference.currentQualityRanking - 1 >= 0){
322             modelReference.changeLOD(modelReference.currentQualityRanking - 1, lodObserver);
323           }else{
324             ascending = true;
325             requestAnimationFrame(showCaseLoop);
326           }
327         }
328       }
329
330       return instance;
331     };
```

Figure 33: Code that observes and selects the next LOD

```
244         currentModel.changeLOD = function(_requestingQR, _observer){
245           var requestingQRString = "" + _requestingQR;
246           var requestingURL = this.directions.baseURLs[0];
247           var thisModel = this;
248
249           for(var representation of this.directions.representations){
250             if(representation.qualityRanking === requestingQRString){
251               requestingURL += representation.baseURL;
252               requestingURL += "?currentQR=" + this.currentQualityRanking;
253               break;
254             }
255           }
256
257           $.ajax({
258             url: requestingURL,
259             method: "GET",
260             dataType: "json",
261             crossDomain: true,
262             success: function(_data){
263               function updateLOD(){
264                 var currentFaces = $(thisModel.dom.IndexedFaceSet).attr("coordIndex");
265                 var currentVertices = $(thisModel.dom.Coordinate).attr("point");
266                 var updatedFaces = vcdiff.decode(currentFaces, _data.faces);
267                 var updatedVertices = vcdiff.decode(currentVertices, _data.vertices);
268
269                 $(thisModel.dom.Coordinate).attr("point", updatedVertices);
270                 $(thisModel.dom.IndexedFaceSet).attr("coordIndex", updatedFaces);
271
272                 thisModel.currentQualityRanking = _requestingQR;
273
274                 _observer.notifyDone("success");
275               }
276
277               setTimeout(updateLOD, 1000);
278             },
279             error: function(_jqXHR, _textStatus, _errorThrown){
280               _observer.notifyDone("error");
281             }
282           });
283         }
```

Figure 34: Code that requests and applies the LODs

```
<html>
  <head>
    <script
      src="https://code.jquery.com/jquery-3.1.0.js"
      integrity="sha256-slogkvB1K3VOkzAI8QITxV3VzpOnkeNVsKvtkYLMjfk="
      crossorigin="anonymous">
    </script>

    <script type='text/javascript' src='http://www.x3dom.org/download/x3dom.js'> </script>
    <link rel='stylesheet' type='text/css' href='http://www.x3dom.org/download/x3dom.css'></link>

    <script src = "../vcdiff.js"></script>
    <script src = "../x3dashPlayer.js"></script>

    <script>
      var player;
      $(document).ready(function(){
        player = new x3dash.Player();

        player.addModel(
          "58634e10-d2b7-11e6-85c4-6ff5e2713075",
          "http://localhost:3000/mpd/58634e10-d2b7-11e6-85c4-6ff5e2713075.mpd"
        );

        player.addModel(
          "c1644180-d2b7-11e6-85c4-6ff5e2713075",
          "http://localhost:3000/mpd/c1644180-d2b7-11e6-85c4-6ff5e2713075.mpd"
        );

        player.start();
      });
    </script>
  </head>
  <body style="background-color: #2B3E50">
    <x3d width = "800px" height = "600px">
      <scene>
        <transform translation = "-3 0 0">
          <inline nameSpaceName = "58634e10-d2b7-11e6-85c4-6ff5e2713075" mapDEFToID = "true"
            url = "http://localhost:3000/modelStubs/58634e10-d2b7-11e6-85c4-6ff5e2713075.x3d">
          </inline>
        </transform>
        <transform translation = "0 0 0">
          <inline nameSpaceName = "c1644180-d2b7-11e6-85c4-6ff5e2713075" mapDEFToID = "true"
            url = "http://localhost:3000/modelStubs/c1644180-d2b7-11e6-85c4-6ff5e2713075.x3d">
          </inline>
        </transform>
      </scene>
    </x3d>
  </body>
</html>
```

Figure 35: Example of using the player

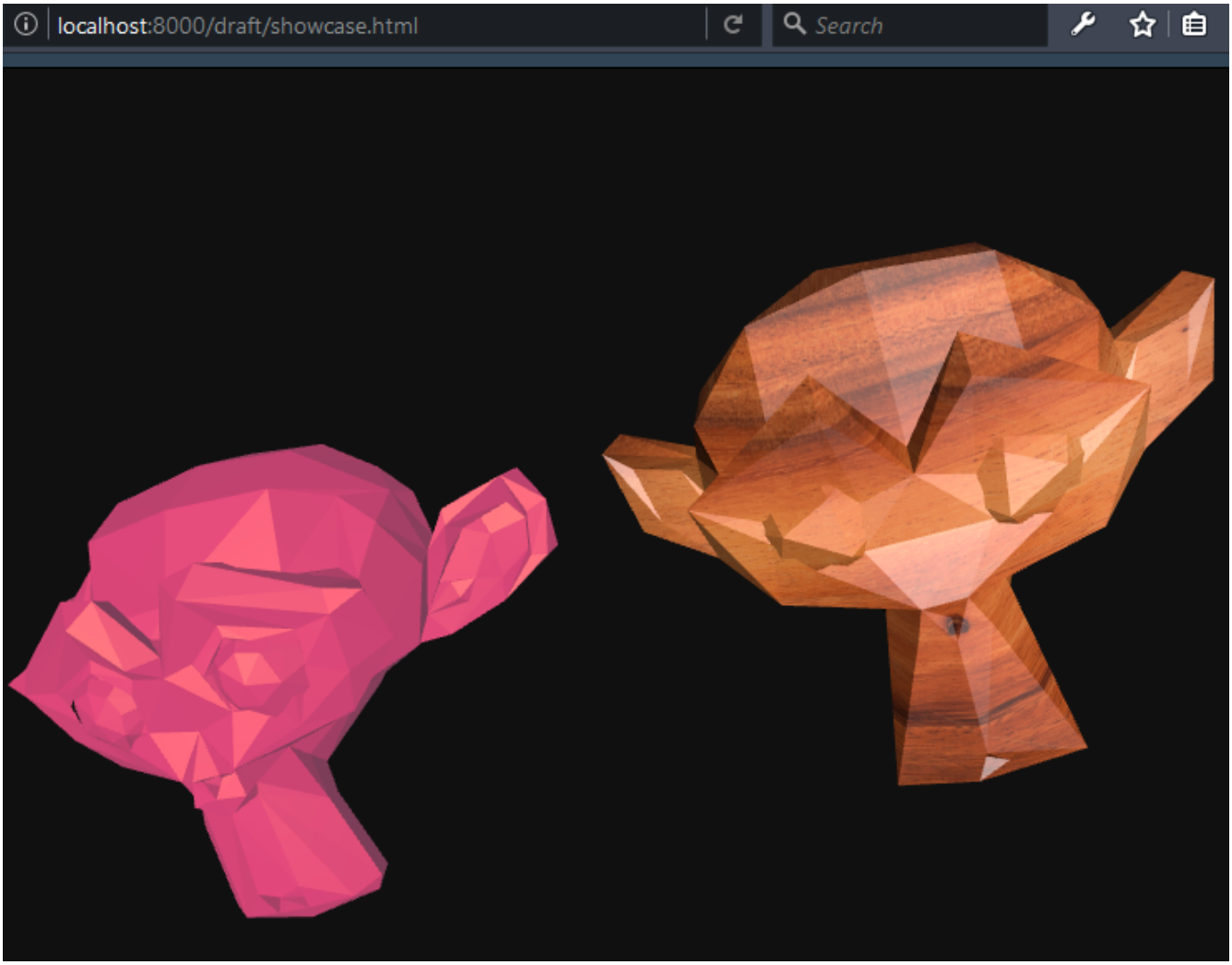

Figure 36: Client during runtime showing two low LOD models

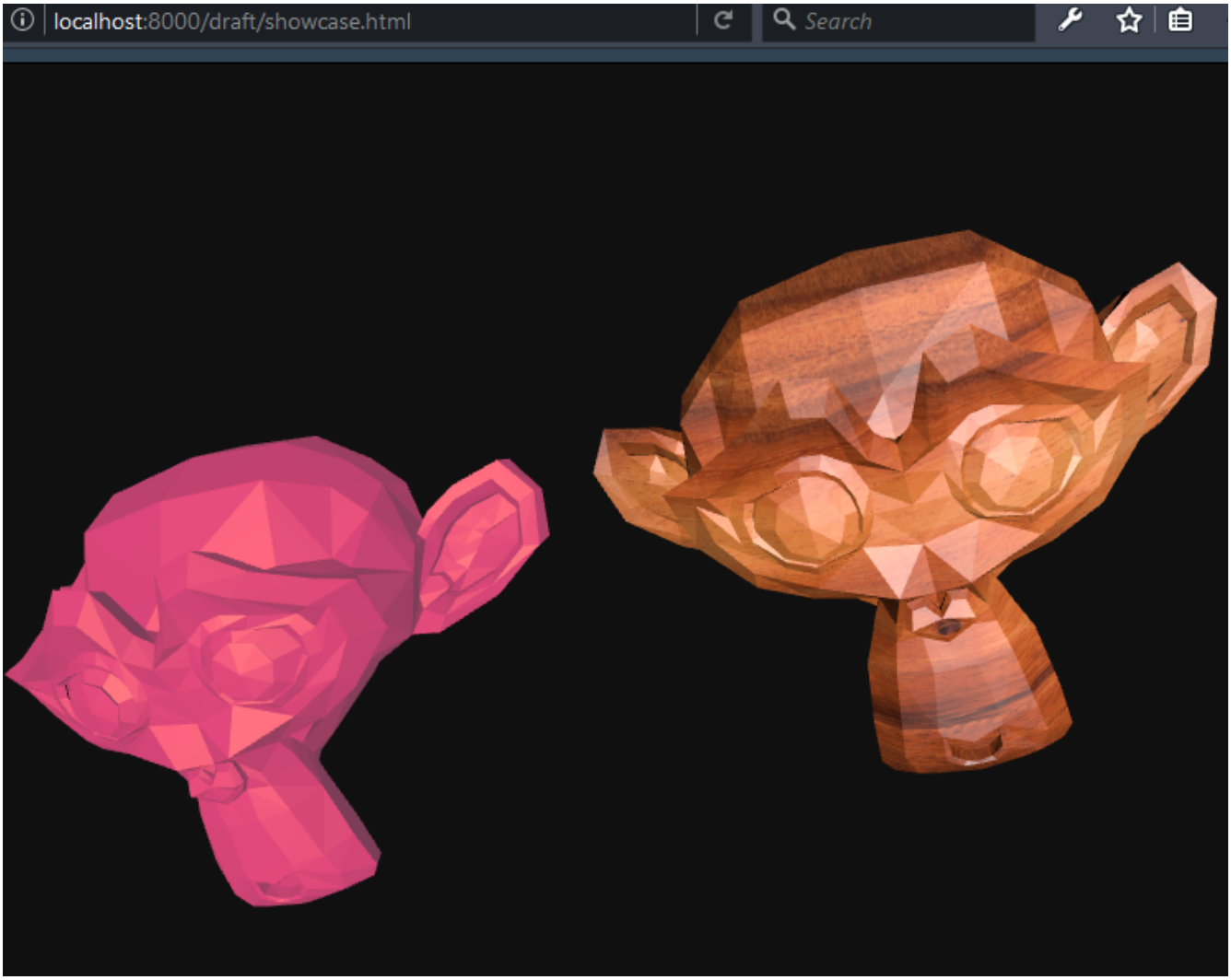

Figure 37: Client during runtime showing two high LOD models

3.1.5 – Results in numbers

Here we are going to present how many bytes were needed to be transmited for each LOD. For the DLOD framework we used five of the most widely used 3D models for testing. More specifically the Bunny, Suzanne, Happy Buddha, Dragon and the Armadillo models which they were converted into the `*.x3d` file format. We produced a hierarchy of six LODs for each model by using the Blender's decimation tool. The CLOD was tested with the Suzzane model. Each LOD, for all the models, includes the 30%, 40%, 60%, 70%, 90% and 100% of the model's faces. Each LOD was delta encoded using the previous LOD as its source.

Faces percentage	Raw data bytes	Delta encoded bytes	Difference	Difference in %
30	706571	706571	0	0.00%
40	948511	734841	213670	22.53%
60	1433984	1208851	225133	15.70%
70	1677299	1199230	478069	28.50%
90	2163635	1712420	451215	20.85%
100	2407398	1651024	756374	31.42%

Table 1: Results table of the Bunny model using the DLOD framework

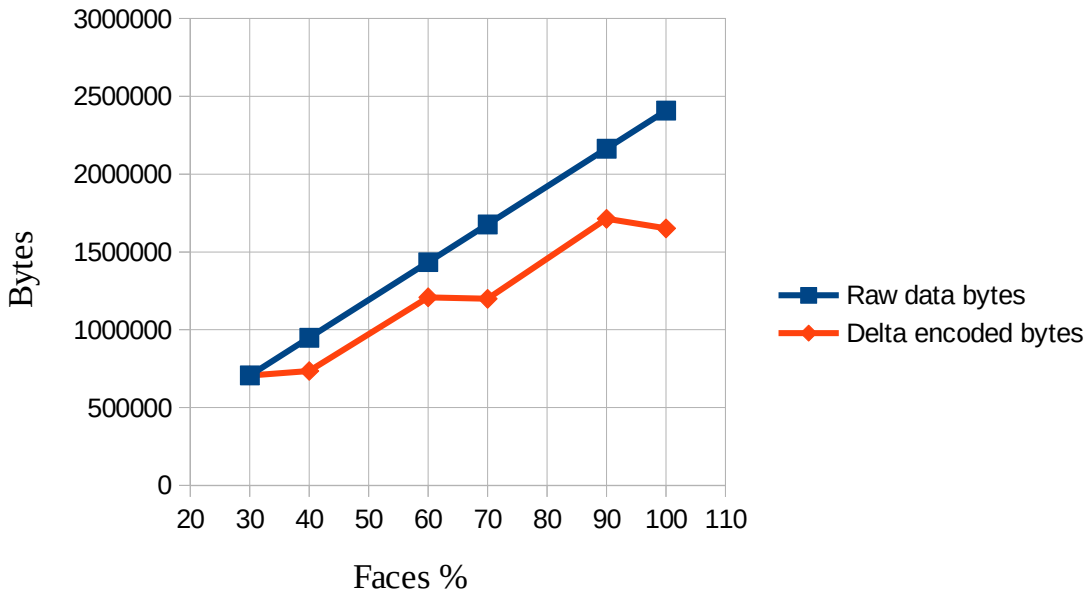

Figure 38: Graph of the Bunny's transmitted bytes using the DLOD framework

Faces percentage	Raw data bytes	Delta encoded bytes	Difference	Difference in %
30	9754	9754	0	0.00%
40	12204	8739	3465	28.39%
60	16863	13268	3595	21.32%
70	19000	11155	7845	41.29%
90	23350	15604	7746	33.17%
100	25374	12908	12466	49.13%

Table 2: Results table of the Suzanne model using the DLOD framework

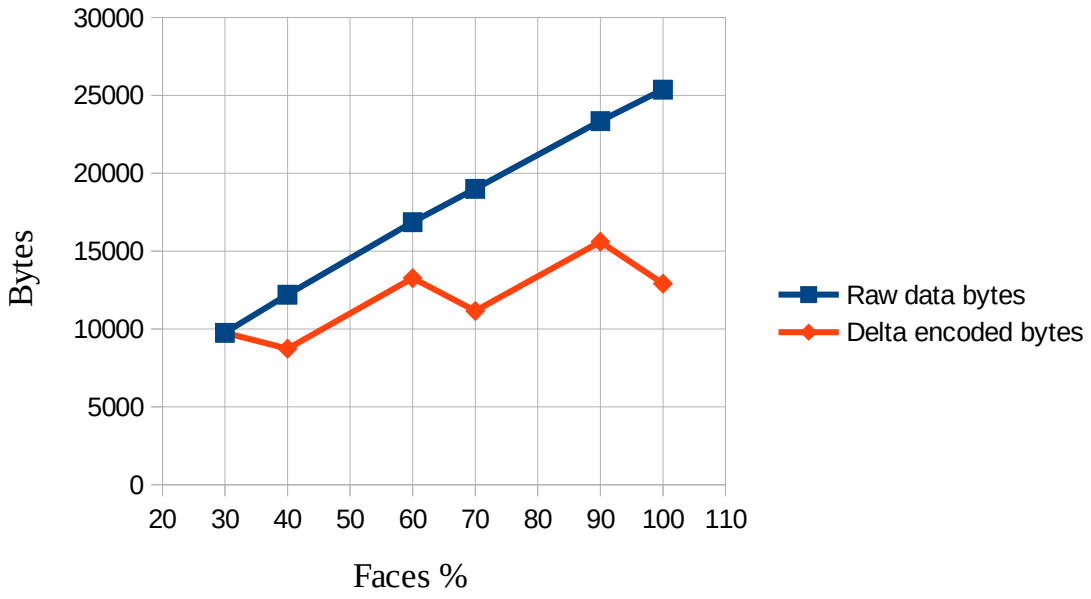

Figure 39: Graph of the Suzanne's transmitted bytes using the DLOD framework

Faces percentage	Raw data bytes	Delta encoded bytes	Difference	Difference in %
30	11766576	11766576	0	0.00%
40	15913259	13706279	2206980	13.87%
60	24209309	22642745	1566564	6.47%
70	28357213	23982883	4374330	15.43%
90	36648112	33496755	3151357	8.60%
100	40790576	32779664	8010912	19.64%

Table 3: Results table of the Happy Buddha's model using the DLOD framework

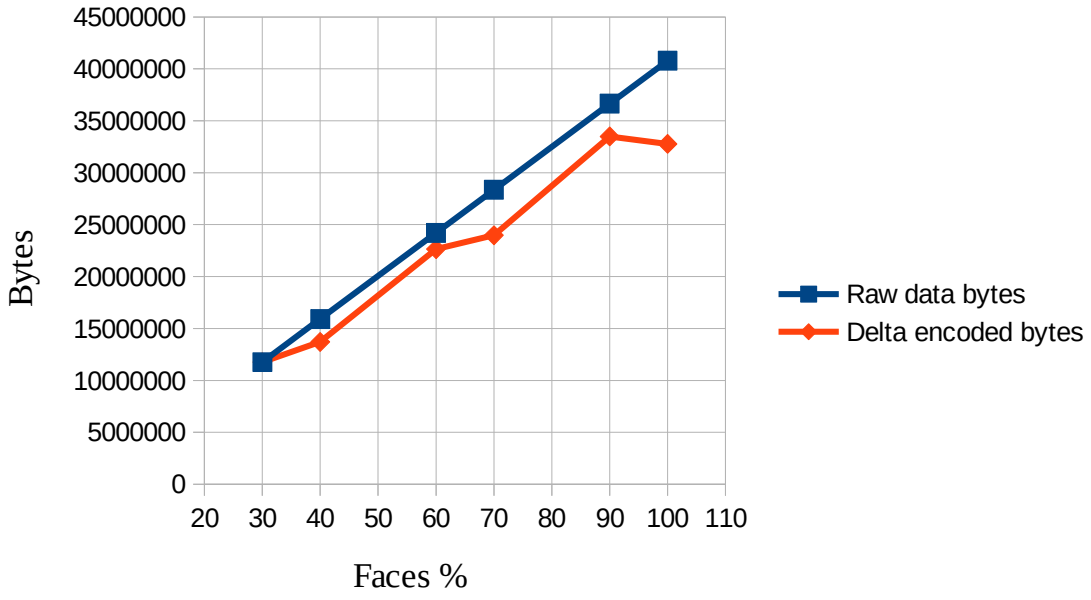

Figure 40: Graph of the Happy Buddha's transmitted bytes using the DLOD framework

Faces percentage	Raw data bytes	Delta encoded bytes	Difference	Difference in %
30	9339862	9339862	0	0.00%
40	12663358	11078500	1584858	12.52%
60	19315641	18044526	1271115	6.58%
70	22641519	18199646	4441873	19.62%
90	29288734	26463757	2824977	9.65%
100	32613082	26035661	6577421	20.17%

Table 4: Results table of the Dragon's model using the DLOD framework

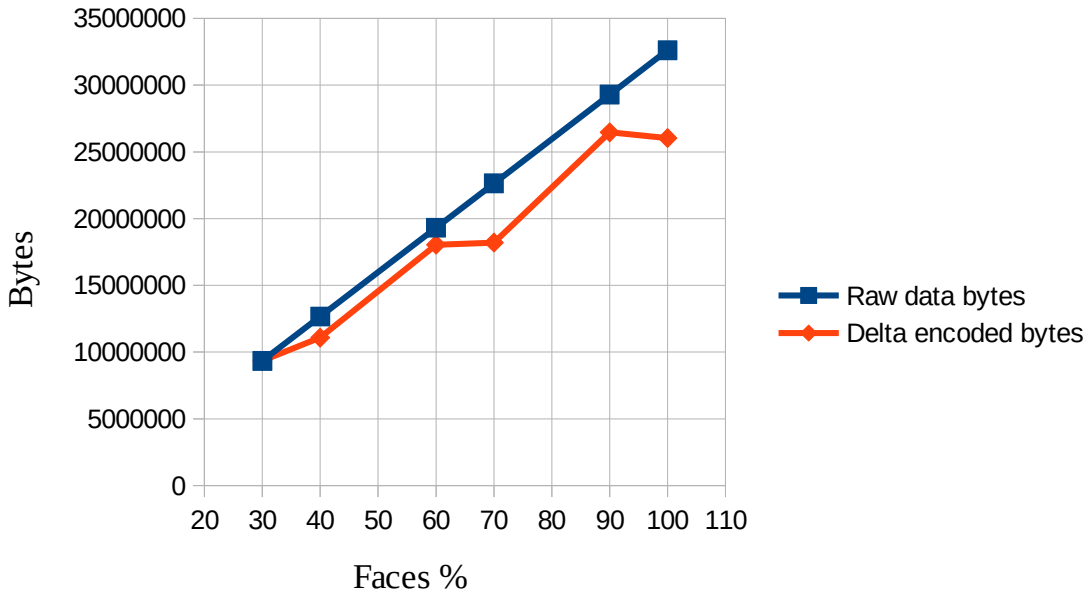

Figure 41: Graph of the Dragon's transmitted bytes using the DLOD framework

Faces percentage	Raw data bytes	Delta encoded bytes	Difference	Difference in %
30	3702060	3702060	0	0.00%
40	4957491	4203655	753836	15.21%
60	7490625	6883533	607092	8.10%
70	8848922	6870041	1978881	22.36%
90	11565303	9971772	1593531	13.78%
100	12923294	9733516	3189778	24.68%

Table 5: Results table of the Armadillo's model using the DLOD framework

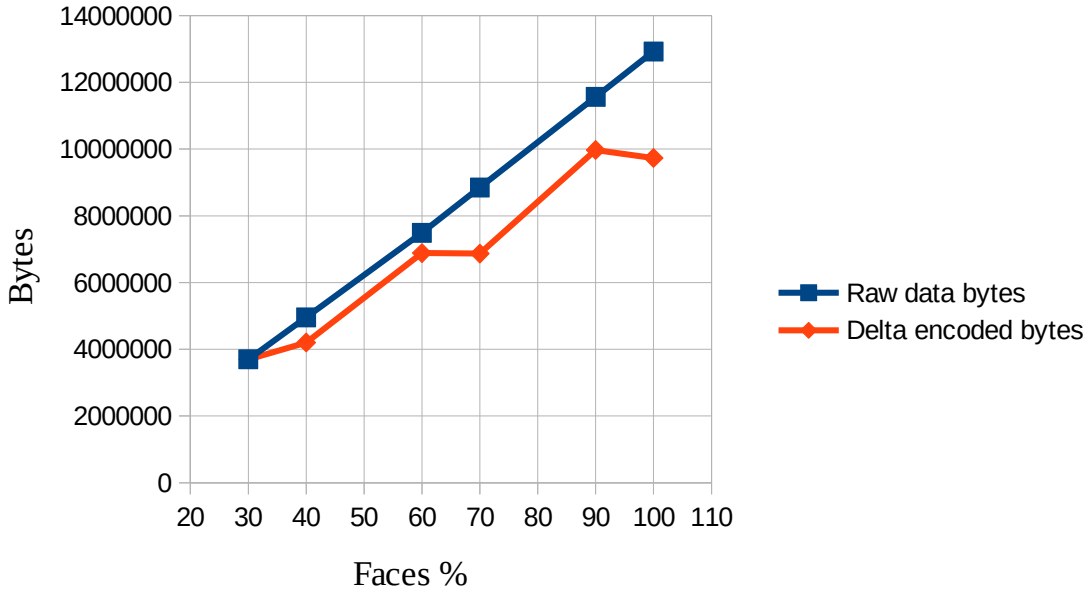

Figure 42: Graph of the Armadillo's transmitted bytes using the DLOD framework

Faces percentage	Average difference in %
30	0.00%
40	18.50%
60	11.63%
70	25.44%
90	17.21%
100	29.01%

Table 6: Average savings of all the tested models using the DLOD framework

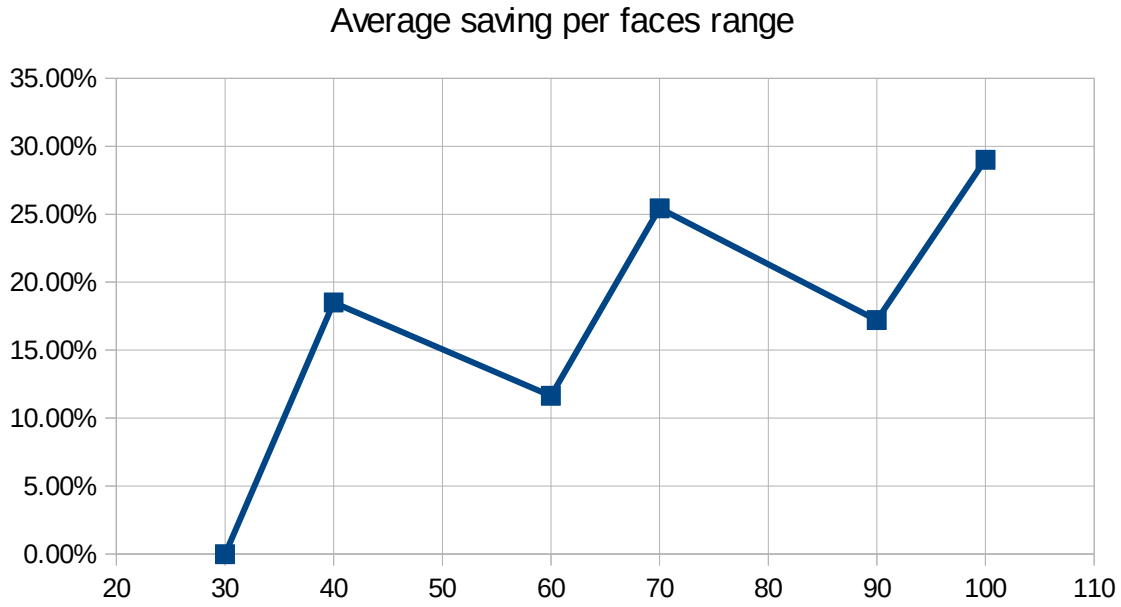

Figure 43: Graph of average savings of all the tested models using the DLOD framework

Faces percentage	Raw data bytes	Delta encoded bytes	Difference	Difference in %
30	147433	147447	-14	-0.90%
40	146821	18136	128685	87.64%
60	144576	34820	109756	75.91%
70	142840	23637	119203	83.45%
90	140572	39709	100863	71.75%
100	139874	27842	112032	80.09%

Figure 44: Results table of the Suzanne's model using the CLOD framework

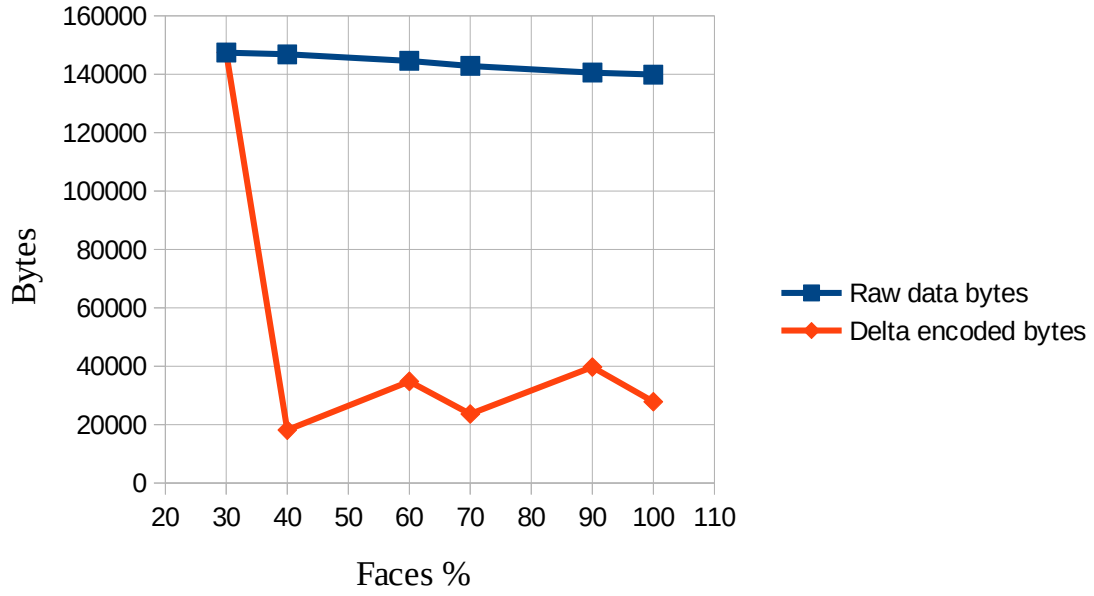

Figure 45: Graph of the Suzanne's transmitted bytes using the CLOD framework

3.2– Summary, conclusions and future work

In this work we focused on the transmission of LODs using the discrete and continuous LOD frameworks. In the first framework each LOD is represented as an individual 3D model. This means that the geometry is disjoint and we don't have a straightforward method for a redundancy free transmission. On the other hand we have the continuous LOD framework in which a 3D model is encoded in a way that it allows us to extract the desired LOD on demand. On the down side, there is no standard encoding scheme that is used among all implementations. To alleviate these issues we propose the use of delta encoding.

Among its many applications, delta encoding is also used in the RFC 3229 and the SDCH protocols in order to minimize the payload size of HTTP responses. The work of [GMBTB11] deals with the size reduction of OpenGL command batches that are streamed through the network. They use data compression along with delta encoding which they call as in-frame and inter-frame compression respectively. We believe, at least in a more abstract level, that their work is close to our solution approach even though they are dealing with a different kind of problem.

On the server side we created a module that can extract the desired LOD which is then converted into a form compatible with the X3D's `IndexedFaceSet` node and the patch data are produced by using the client's current LOD. Then the client produces the target LOD by applying the patch and updates the scene's model by using the jQuery's `.attr()` [jQattr] method for the `point` and `coordIndex` attributes. The given API for the MPEG-DASH client developer is fairly simple. To change the current LOD, he or she will just call the model's `changeLOD` method which takes two arguments. The first one is the requesting quality ranking and the second one is an observer object which is notified if the LOD update was successful or if it failed.

Delta encoding performs well when the differences between the input files are small, which we can confirm that by our results. As we can see, the high compression ratios can be found when we were changing the LOD from the 30% to 40%, from 60% to 70% and from 90% to 100% of the model's faces. Finally, we got the best compression ratios when using the CLOD framework. This is because the data in this framework are homogeneous.

Based on this observation, a possible future research would deal with the development of a mesh simplification algorithm that produces a delta encoding friendly

output. Additionally, we would like to fully investigate the potentials of the SDCH protocol on the transmission of LODs.

References

[WHV14]: WHATWG: Ian Hickson, Google, Inc. W3C: Robin Berjon, W3C, Steve Faulkner, The Paciello Group, Travis Leithead, Microsoft Corporation, Erika Doyle Navara, Microsoft Corporation, Edward O'Connor, Apple Inc. Silvia Pfeiffer, HTML5 A vocabulary and associated APIs for HTML and XHTML, 2014, https://www.w3.org/TR/html5/Overview.html

[IRS11]: I. Sodagar, "The MPEG-DASH Standard for Multimedia Streaming Over the Internet," in *IEEE MultiMedia*, vol. 18, no. 4, pp. 62-67, April 2011. doi: 10.1109/MMUL.2011.71

[SAC11]: Saamer Akhshabi, Ali C. Begen, and Constantine Dovrolis. 2011. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proceedings of the second annual ACM conference on Multimedia systems* (MMSys '11). ACM, New York, NY, USA, 157-168. doi=http://dx.doi.org/10.1145/1943552.1943574

[BEJZ09]: Johannes Behr, Peter Eschler, Yvonne Jung, and Michael Zöllner. 2009. X3DOM: a DOM-based HTML5/X3D integration model. In *Proceedings of the 14th International Conference on 3D Web Technology* (Web3D '09), Stephen N. Spencer (Ed.). ACM, New York, NY, USA, 127-135. doi=http://dx.doi.org/10.1145/1559764.1559784

[KPMZ14]: K. Kapetanakis, S. Panagiotakis, A. G. Malamos and M. Zampoglou, "Adaptive video streaming on top of Web3D: A bridging technology between X3DOM and MPEG-DASH," *2014 International Conference on Telecommunications and Multimedia (TEMU)*, Heraklion, 2014, pp. 226-231. doi: 10.1109/TEMU.2014.6917765

[GIT14]: Kostas Kapetanakis, Github pull request #232, url: https://github.com/x3dom/x3dom/pull/232/files

[ML14]: Multimedia Content Laboratory - X3DOM VR world with MPEG-DASH videostream, url: http://medialab.teicrete.gr/minipages/dash3d/

[KK14]: Kostas Kapetanakis, WEB-3D REAL-TIME ADAPTATION FRAMEWORK BASED ON MPEG-DASH, 2014, url: http://medialab.teicrete.gr/media/thesis/Kapetanakis_thesis.pdf

[ZKSMP16]: M. Zampoglou, K. Kapetanakis, A. Stamoulias, A. G. Malamos, S. Panagiotakis, “Adaptive streaming of complex Web 3D scenes based on the MPEG-

DASH standard", "Multimedia Tools and Applications", Vol. 75, pp.1-24 , 2016, doi: 10.1007/s11042-016-4255-8

[JHC76]: James H. Clark. 1976. Hierarchical geometric models for visible surface algorithms. *Commun. ACM* 19, 10 (October 1976), 547-554. doi=http://dx.doi.org/10.1145/360349.360354

[X3DP16]: Web3D, Applications, Players and Plugins for X3D / VRML Viewing, 2016, url: http://www.web3d.org/x3d/content/examples/X3dResources.html#Applications

[FRSV14]: Figueiredo, Mauro, José I. Rodrigues, Ivo Silvestre, and Cristina Veiga-Pires. "A Topological Framework for Interactive Queries on 3D Models in the Web." *The Scientific World Journal* 2014

[Par05]: Jindřich Parus, Morphing of Meshes, Technical Report DCSE/TR-2005-02, 2005, url: http://hdl.handle.net/11025/21597

[Wc12]: Won Chun, WebGL Models: End-to-End, pp 431 – 453, "OpenGL Insights", July 2012, CRC Press, ISBN: 978-1439893760, url: http://www.openglinsights.com

[MDFK97]: Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. 1997. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication* (SIGCOMM '97), Martha Steenstrup (Ed.). ACM, New York, NY, USA, 181-194. doi=http://dx.doi.org/10.1145/263105.263162

[Pks02]: Konstantinos Psounis. 2002. Class-Based Delta-Encoding: A Scalable Scheme for Caching Dynamic Web Content. In *Proceedings of the 22nd International Conference on Distributed Computing Systems* (ICDCSW '02). IEEE Computer Society, Washington, DC, USA, 799-805.

[Chr02]: Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. 2002. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '02). ACM, New York, NY, USA, 693-702. doi=http://dx.doi.org/10.1145/566570.566639

[WG01]: Greg Humphreys, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. 2001. WireGL: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '01). ACM, New York, NY, USA, 129-140. doi=http://dx.doi.org/10.1145/383259.383272

[GMBTB11]: P. S. Gasparello, G. Marino, F. Bannò, F. Tecchia and M. Bergamasco, "Real-Time Network Streaming of Dynamic 3D Content with In-frame and Inter-frame Compression," *2011 IEEE/ACM 15th International Symposium on Distributed Simulation and Real Time Applications*, Salford, 2011, pp. 81-87. doi: 10.1109/DS-RT.2011.24

[RFC3229]: J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, D. Hellerstein, RFC 3229 - Delta encoding in HTTP, January 2002, url: https://tools.ietf.org/html/rfc3229

[BLM16]: Jon Butler, Wei-Hsin Lee, Bryan McQuade, Kenneth Mixter, A Proposal for Shared Dictionary Compression over HTTP, 2016

[Li15]: Shared Dictionary Compression for HTTP at LinkedIn, 2015, https://engineering.linkedin.com/shared-dictionary-compression-http-linkedin

[WX3D]: What is X3D, 2016, http://www.web3d.org/x3d/what-x3d

[X3DIFS]: Extensible 3D (X3D) - Part 1: Architecture and base components - Geometry3D component, 2016, http://www.web3d.org/documents/specifications/19775-1/V3.3/Part01/components/geometry3D.html#IndexedFaceSet

[X3DCoo]: Extensible 3D (X3D) - Part 1: Architecture and base components - Rendering component, 2016, http://www.web3d.org/documents/specifications/19775-1/V3.3/Part01/components/rendering.html#Coordinate

[Bau75]: Bruce G. Baumgart. 1975. A polyhedron representation for computer vision. In *Proceedings of the May 19-22, 1975, national computer conference and exposition* (AFIPS '75). ACM, New York, NY, USA, 589-596. doi=http://dx.doi.org/10.1145/1499949.1500071

[Zcg12]: Chapter 5 - Plane Graphs and the DCEL, Institute of Theoretical Computer Science, ETH Zurich, http://www.ti.inf.ethz.ch/ew/lehre/CG12/lecture/Chapter%205.pdf

[MP78]: D.E. Muller, F.P. Preparata, Finding the intersection of two convex polyhedra, Theoretical Computer Science, Volume 7, Issue 2, 1978, Pages 217-236, ISSN 0304-3975, http://dx.doi.org/10.1016/0304-3975(78)90051-8. http://www.sciencedirect.com/science/article/pii/0304397578900518

[BSBK02]: M. Botsch S. Steinberg S. Bischoff L. Kobbelt, OpenMesh– a generic and efficient polygon mesh data structure, 2002

[JLM03]: Joy KI, Legakis J, Maccracken R (2003) Data Structures for Multiresolution Representation of Unstructured Meshes. Mathematics and Visualization Hierarchical and Geometrical Methods in Scientific Visualization 143–170. doi: 10.1007/978-3-642-55787-3_9

[NF03]: Nah, Fiona, "A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait?" (2003). *AMCIS 2003 Proceedings*. 285. http://aisel.aisnet.org/amcis2003/285

[Rail16]: Measure Performance with the RAIL Model , 2016, url: https://developers.google.com/web/fundamentals/performance/rail

[Ga99]: M. Garland, Multiresolution Modeling: Survey and Future Opportunities in "Eurographics 1999 - STARs. Eurographics Association", doi:10.2312/egst.19991068

[ML16]: Kostas Kapetanakis, MPEG-DASH for X3D Streaming, 2016, http://mclab1.medialab.teicrete.gr:8081/indexdash.html

[Utube16]: Blender Model Tutorial Polygon Reduction, 2016, https://www.youtube.com/watch?v=ttU6Gz1W0Xw

[ADM16]: AUTODESK 3DS MAX - Level of Detail Utility, 2016, https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2016/ENU/3DSMax/files/GUID-D112D015-8BE6-4172-B816-B5432A50F911-htm.html

[BDM16]: Blender - Decimate Modifier, 2016, https://www.blender.org/manual/modeling/modifiers/generate/decimate.html

[WX3dL]: LOD node definition, 2016, http://www.web3d.org/documents/specifications/19775-1/V3.0/Part01/components/navigation.html#LOD

[XfwaL]: Example for LOD node, 2016, http://x3dgraphics.com/examples/X3dForWebAuthors/Chapter03-Grouping/LODIndex.html

[SW08]: Daniel Scherzer and Michael Wimmer. 2008. Frame sequential interpolation for discrete level-of-detail rendering. In *Proceedings of the Nineteenth Eurographics conference on Rendering* (EGSR '08). Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 1175-1181. doi=http://dx.doi.org/10.1111/j.1467-8659.2008.01255.x

[LDSS99]: Aaron W. F. Lee, David Dobkin, Wim Sweldens, and Peter Schröder. 1999. Multiresolution mesh morphing. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '99). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 343-350. doi=http://dx.doi.org/10.1145/311535.311586

[Hp96]: Hugues Hoppe. 1996. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (SIGGRAPH '96). ACM, New York, NY, USA, 99-108. doi=http://dx.doi.org/10.1145/237170.237216

[PR00]: Renato Pajarola and Jarek Rossignac. 2000. Compressed Progressive Meshes. *IEEE Transactions on Visualization and Computer Graphics* 6, 1 (January 2000), 79-93. doi=http://dx.doi.org/10.1109/2945.841122

[LJBA13]: Limper, M., Jung, Y., Behr, J. and Alexa, M. (2013), The POP Buffer: Rapid Progressive Clustering by Geometry Quantization. Computer Graphics Forum, 32: 197–206. doi:10.1111/cgf.12227

[Sm98]: Stan Melax, A Simple, Fast, and Effective Polygon Reduction Algorithm, 1998, http://www.melax.com/gdmag.pdf

[Gzz85]: Joshua Koo - zz85, SimplifyModifier: initial commit, 2016, https://github.com/mrdoob/three.js/commit/3376a9b00ddb49fac9170b8038b4f34b2770d039#diff-06d2b0be8475b2937e3b66e432271390

[SM02]: T. Suel, N. Memon, "Algorithms for Delta Compression and Remote File Synchronization" in: In Khalid Sayood, Lossless Compression Handbook. Academic Press, 2002

[SC12]: N. Samteladze and K. Christensen, "DELTA: Delta encoding for less traffic for apps," *37th Annual IEEE Conference on Local Computer Networks*, Clearwater, FL, 2012, pp. 212-215.
doi: 10.1109/LCN.2012.6423611

[Pgit]: Scott Chacon, Ben Straub, 10.4 Git Internals - Packfiles, 2014, https://git-scm.com/book/en/v2/Git-Internals-Packfiles

[Mer16]: Bryan O'Sullivan, Mercurial: The Definitive Guide, 2009
Sv16: Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato, Version Control with Subversion, http://hgbook.red-bean.com/read/

[Do15]: Jozef Doboš, Management and Visualisation of Non-linear History of Polygonal 3D Models, 2015, http://3drepo.org/projects/management-and-visualisation-of-non-linear-history-of-polygonal-3d-models/

[GGS99]: Stefan Gumhold, Stefan Guthe, and Wolfgang Straßer. 1999. Tetrahedral mesh compression with the cut-border machine. In *Proceedings of the conference on Visualization '99: celebrating ten years* (VIS '99). IEEE Computer Society Press, Los Alamitos, CA, USA, 51-58.

[LWSJS13]: Max Limper, Stefan Wagner, Christian Stein, Yvonne Jung, and André Stork. 2013. Fast delivery of 3D web content: a case study. In *Proceedings of the 18th International Conference on 3D Web Technology* (Web3D '13). ACM, New York, NY, USA, 11-17. DOI=http://dx.doi.org/10.1145/2466533.2466536

[plvc]: VCDiff Javascript implementation, 2016, https://github.com/plotnikoff/vcdiff.js

[Node]: Node.js, 2016, https://nodejs.org/en/

[Expr]: Express web application framework, 2016, http://expressjs.com/

[Npm]: npm package manager, 2016, https://www.npmjs.com/

[jQ]: jQuery, 2016, https://jquery.com/

[jQattr]: jQuery .attr(), 2016, http://api.jquery.com/attr/