Out-of-Core Progressive Web-Based Rendering of Triangle Meshes

Thiago F. de Moraes[1], Paulo H.J. Amorim[1], Jorge V.L. da Silva[1], and Helio Pedrini[2](✉)

[1] Tridimensional Technology Division, Center for Information Technology Renato Archer, Campinas, SP 13069-901, Brazil
[2] Institute of Computing, University of Campinas, Campinas, SP 13083-852, Brazil
helio@ic.unicamp.br

Abstract. The visualization of large volumes of data has been explored in several knowledge domains, such as remote sensing, medicine, meteorology, biology, among others. In traditional data visualization techniques, data is stored, processed and rendered locally on the client machine, which may require expensive computational resources in terms of storage space and processing power. This work presents and discusses a methodology for out-of-core remote rendering of large three-dimensional triangles meshes. Users are able to interact with the developed visualization tool through requests sent to a server by directly manipulating the data volumes on their own Web browser.

1 Introduction

Technological advances in acquisition and processing of large data volumes have driven the development of applications in several fields of knowledge, such as medicine, biology, remote sensing, entertainment, interactive broadcasting, among others [4,14,16]. The continuous growth of data availability makes the traditional visualization techniques more difficult to be processed locally on a client machine, since they usually demand computationally expensive resources to store, manipulate and display large volumes of data. On the other hand, data remote rendering [5,7,21] has allowed expensive operations to be performed on a server with high processing power and storage capacity, as well as resources for sharing collaboration among multiple users through security mechanisms and version control.

Current browsers implement the WebGL and WebSocket standards to allow visualization of three-dimensional (3D) data and fast information exchange between server and client. However, the implementation of a remote rendering tool on Web browser presents several challenges. The amount of memory available in the client machine may not suffice for storing and manipulating the data. Current WebGL standard (version 1) allows only for 16-bit identifiers [13], which limits the display of meshes with up to 65,535 faces. In order to deal with

© Springer International Publishing AG 2018
J.M.R.S. Tavares and R.M. Natal Jorge (eds.), *VipIMAGE 2017*,
Lecture Notes in Computational Vision and Biomechanics 27,
DOI 10.1007/978-3-319-68195-5_50

large polygonal meshes (e.g. over 1 million faces), out-of-core techniques can be used to partition the meshes into multiple clusters with progressive rendering.

This work describes a novel methodology for 3D mesh rendering on Web using out-of-core strategy. From requests sent to a server, users are able to manipulate and visualize large volumes of data in a fast and easy manner via the browsing environment itself. Several benefits are offered to users from the proposed system: (i) the approach eliminates the installation of special software in the client machine, (ii) the solution uses only free and open source packages, (iii) the server is responsible for executing the operations required by users, which avoids the need for expensive resources on the client machine in terms of memory storage and processing power, and (iv) the visualization results are available from any device with Internet access.

2 Background

With the advances of computer systems and the popularization of Internet, users now have access to multimedia information such as images, videos, audio and texts in a fast and easy way.

The standard WebGL [20] has been proposed, which is an extension of the application program interface (API) OpenGL for Embedded Systems (ES) for browsers, both desktop computers and mobile devices. This standard allows the rendering of various graphics primitives such as lines and triangles, as well as the use of shader for light and shadow effects. It also allows volume rendering through the use of raycasting algorithms.

WebSocket [9] is another important standard in the context of remote rendering, which allows bidirectional communication between two or more computer systems. Client-server interaction can be performed without requiring a request from the client, which is an important requirement for the visualization tool.

Another important issue in the context of visualization is the data structure [8], that is, the representation to maintain the mesh information to be displayed. The mesh is basically composed of interconnected triangular faces.

The indexed mesh is one of the simplest data structures, where a table is used to store the vertex coordinates and another table contains the faces, such that each face indicates its vertex via indices. However, this structure does not contain any information for adjacency queries, in constant time, of vertices and faces, requiring to traverse the entire list of faces and vertices to obtain such information.

The corner table representation [17] is based on the concept of corners, which is a vertex-face association. Each vertex of a face is a corner. Each vertex is referenced by 1 or n corners, where n is the number of faces which the vertex v_i belongs to. Thus, a vertex is referenced by n corners. This structure uses three tables, one for the vertices with their spatial coordinates, one for the corner table, and another for the opposite corners to allow for adjacency queries in constant time.

Another representation, used in our work, is the laced ring [11], which is based on the corner table. This data structure consists of a ring that traverses all or the majority of the mesh vertices. The triangles are classified according to the number of incident edges to the ring: T_0, T_1 and T_2 are triangles with zero, one or two edges incident to the ring, respectively. Triangles T_0 are the minority and are represented through the same structure as the corner table. The other triangles have a more compact representation.

The previously mentioned data structures are known as in-core representations since they need to be fully stored in primary memory. On the other hand, out-of-core representations allow only part of its structure to be in memory, such that, other portions can be brought to memory when required, whereas other ones are removed from memory when are no longer needed.

Yoon and Lindstrom [22] describe an out-of-core mesh representation on clustering. Internally to the clusters, the same representation as the streaming meshes is used. Prior to the clustering construction, a reordering of the mesh is performed. In order to do that, they used the OpenCCL sorting algorithm [23], reducing the problem in two dimensions to a one-dimensional problem. Subsequently, the clusters are compressed through the same operators as the processing sequences [12]. When necessary, the groups are decompressed and the access to faces and vertices is made using similar operators to the corner table. This representation maintains topology information and allows adjacency queries in constant time.

3 Methodology

The proposed methodology can be divided into two components: (i) construction of the out-of-core mesh file and (ii) visualization of the resulting file. These components are detailed as follows.

3.1 Construction of the Out-of-Core Mesh File

The data structure used in our methodology is the laced ring due to its advantages: (i) it is compact, which helps reduce the frequency of page faults, the cost of exchanging mesh portions among processors, and amount of required memory; (ii) it stores adjacency information, which is very important for smoothing algorithms [18] and mesh simplification [19].

Initially, an in-core mesh file is loaded into memory and stored through an indexed mesh structure. The file can be in PLY, STL or OBJ format. From this file, vertices and faces of the mesh are extracted. The indexed mesh structure is then converted to a corner table structure. This step is required for the construction of the laced ring representation to allow adjacency information. The corner table was chosen due to its similarities to the laced ring. From the corner table, the laced ring representation of the mesh is built.

The generation of clusters is performed using the laced ring obtained in the previous step. Each cluster has approximately the same size (the last cluster to be created can be much smaller) and is identified by an integer starting from 0 to $n-1$, where n is the number of clusters. This identifier is both used for indexing and for the retrieval of such clusters. Clusters are useful for maintaining the idea of location, since vertices and adjacent triangular faces are kept in the same cluster, which minimizes the need to transfer mesh blocks from the disk to the memory.

A laced ring sorts its vertices and triangles, such that the construction of clusters is reduced from a problem in two dimensions to one dimension. This means that we can create a cluster for every n triangles, where n is the desired size of each cluster. Triangles T_0 that are not part of the ring will be inserted into the cluster whose first T_1 or T_2 references it as adjacent. If there are triangles T_0 adjacent only to other triangles T_0, they will be inserted in the last cluster(s).

The clusters are then stored in a disk file. All the triangles of the cluster and their vertices are stored in disk according to the laced ring structure. Vertices that are used for triangles of different clusters are stored in more than one cluster. That is, vertices can be duplicated in several different clusters. This is done in order for the rendering of a cluster not to depend on other cluster(s). The position of the clusters in the file and the location of vertices and faces are indexed for further reading and retrieval of information. The indices are stored in a hash table and written to a file along with the mesh of triangles. In parallel, a version with fewer polygons of the mesh is created. This simplified mesh is generated by a decimation algorithm, such as the quadric edge collapse decimation [10].

3.2 Remote Rendering

Based on modern resources available in current browsers, such as WebGL [20] and WebSocket [9], in conjunction with the client-server architecture, we developed a mesh viewer that does not require advanced computing resources. The processing can be performed on the server, such that the client machine is responsible for interaction with the environment and visualization of the results. Figure 1 illustrates the main components of the proposed remote rendering.

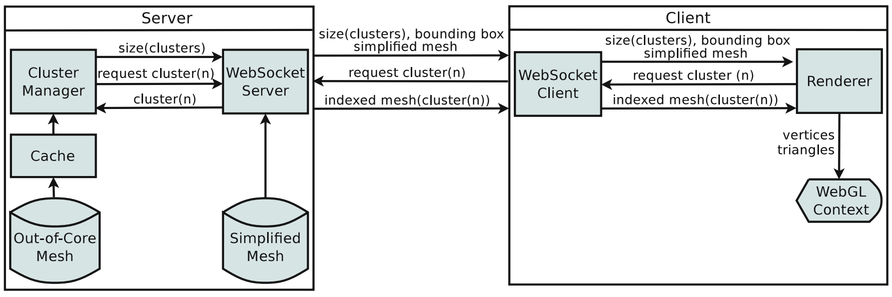

Fig. 1. Main components of the proposed remote rendering.

The server can be subdivided into three parts:

- reading and management of clusters: responsible for reading the out-of-core mesh and loading the cluster as needed.
- HTTP server: responsible for the initial communication, which indicates the model to be rendered. This component was implemented in Python programming language in conjunction with the Web framework Flask [2].
- WebSocket communication: responsible for the WebSocket client communication. It also communicates with the reading component to obtain clusters, convert the triangles and vertices of the clusters to the indexed mesh representation to serialize them in JSON, sending the results to the client. It was implemented in C++ programming language, version 11.

The client can be subdivided into two parts:

- WebSocket communication: responsible for communication with the server. It requests new clusters to the server. After receiving the clusters, it deserializes the clusters and send them for rendering. It was implemented in JavaScript.
- renderer: responsible for rendering the clusters. After displaying a cluster, it requests a new cluster to the WebSocket communication. It is also allows the user to interact with the camera to manipulate the model (for instance, zooming, translation, rotation). It was implemented in JavaScript in conjunction with the threejs library [6], which is based on WebGL.

A server stores the models on disks, according to the methodology proposed here. Client machines, such as desktop computers and mobile devices, connect to the server, requesting the model to be rendered. Then, a connection is started via WebSocket between the client and the server machines. From that moment on, all client-server communication is done via WebSocket. The WebSocket is used to maintain the connection, which optimizes the exchange of messages. Since it involves a bidirectional communication, the server can send messages to a client without a request from it. At the beginning of this communication, the server sends:

- the amount of model clusters: the client needs this information to know how many cluster requests are required.
- the model bounding box: for initial setup of camera, its position and focus.
- the mesh with fewer polygons to be rendered. The rendering process starts with this simplified version of the mesh, such that the user can have a coarse view of the final model. This mesh is also displayed when the user interacts with the view camera. During the progressive rendering of the clusters, details are gradually added to the mesh.

The server uses the initial information from the camera to sort the clusters. Clusters are sorted by the distance from its center to the camera. This ordering is important to first show user the visible portions of the scene, then parts that are farther away and most likely hidden. This is an approximation since what is more distant from the camera is not necessarily hidden. The cluster center does not necessarily indicate whether a cluster is in front or behind others, however, the center is an available information and no access is required to the clusters.

To sort the clusters, first it is necessary to calculate the projection plane. The camera contains the following information used in the projection plane estimation: (i) its position and (ii) focal point position. A plane is mathematically expressed as $ax+by+cz+d=0$, where (a,b,c) is the normal vector that indicates the plane inclination. Considering position p, focal point p_f, normal vector p_n for the plane (a,b,c,d), the projection plane can be calculated as $p_n = v_f - p$, where d is given by $d = -(p_n.p)$. After the calculation of the projection plane, $distance = \frac{|p_n.c_c+d|}{\|p_n\|}$ is applied, where c_c is the cluster center.

The client is a Web browser, which can be a desktop or a mobile browser. It requires only WebGL and WebSocket standards. A client can successively request clusters. The transmission of clusters follows the order described previously. Before transmitting the cluster, the server serializes the cluster using the JSON standard. After receiving the requested cluster i, the client deserializes and renders i and requests the next cluster. The renderer is progressively executed, where details are incrementally added to the low resolution model. For this, the renderer does not clear the screen before rendering. The pixel with lower z-buffer value will overwrite the pixel rendered in the previous cluster. The screen is only cleared when the camera position is changed. When this occurs, in addition to clear the screen, the client sends to the server the new position and focal point of the camera. The server reorders the clusters and the client restarts the request for clusters.

For performance purpose, the server retains a small part of the clusters in cache. The cache keeps a small portion of the clusters in memory to avoiding accessing the cluster in disk. There are several policies for managing cache. The Least Recently Used (LRU) strategy keeps in cache the n most recently used clusters.

4 Results

This section describes and discusses the experimental results obtained with the proposed methodology. Its implementation can be divided into three parts: file creation, server and client.

To validate the proposed methodology, the following experiment was conducted: a browser client requests a model to the server and renders according to the steps discussed in Section 3. During this experiment, the memory consumption of both the client and server was measured. The purpose is to assess the maximum memory consumption and rendering time on the client for each model. This experiment was conducted on models with varying sizes (numbers of vertices and faces). Table 1 shows, in ascending order, information on the number of vertices, edges and faces for each model used in the experiment.

The creation of files was implemented in C++ programming language (version 11). As input, it reads files in PLY format for the models available at Stanford repository [1] and in DICOM format for the models available at OsiriX

Table 1. Number of vertices, edges and faces for each triangle mesh. Buddha, Asian Dragon, Thai Statue and Lucy models are available at Stanford repository [1]. Melanix model, available at OsiriX repository [15], was obtained from the segmentation and extraction of isosurface via marching cubes technique.

Model	Vertices	Edges	Faces
Buddha	543,652	1,631,574	1,087,716
Melanix	1,454,869	4,364,613	2,909,742
Asian Dragon	3,609,455	10,828,359	7,218,906
Thai Statue	4,999,996	15,000,000	10,000,000
Lucy	5,012,704	15,038,106	10,025,404

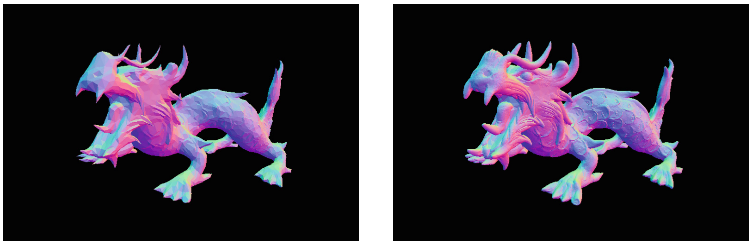
(a) Initial rendering with few polygons (b) Final rendering

Fig. 2. Progressive rendering of the Asian Dragon model.

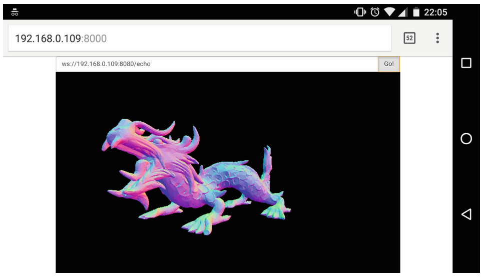

Fig. 3. Asian Dragon model rendered on a smartphone browser.

repository [15]. As output, this module has the out-of-core mesh and the necessary indices.

Figure 2 illustrates the progressive rendering of Asian Dragon model on a desktop browser. The initial rendering with few polygons and less details is shown in Fig. 2(a), whereas the final rendering is shown in Fig. 2(b). Figure 3 illustrates the same model rendered on a mobile browser.

The experiment was conducted by varying the cluster size of each model and the server cache size. The cache size indicates how many clusters are kept in memory in the cache. Clusters with size of 5,000, 10,000 and 25,000 were evaluated. Larger sizes were not used due to the WebGL limitation of only supporting

Table 2. Measurements of maximum memory consumption for the server and the client (browsers) and time to render the entire model on the browser.

Model	Cluster Size	Cache Size	Server max mem (MB)	Browser max mem (MB)	Rendering Time (s)
Buddha	5000	10	76.36	448.84	11.58
		100	180.3	441.3	11.18
	10000	10	125.18	448.22	10.3
		100	216.34	445.42	9.73
	25000	10	278.48	479.97	9.5
		100	336.26	476.3	9.16
Melanix	5000	10	85.16	489.43	34.88
		100	172.55	489.5	35.63
	10000	10	135.42	493.22	30.52
		100	296.44	493.58	30.57
	25000	10	286.4	524.71	28.96
		100	485.72	534.55	28.68
Asian Dragon	5000	10	79.52	494.26	84.03
		100	144.01	488.21	81.65
	10000	10	131.39	510.1	73.78
		100	256.36	500.89	70.37
	25000	10	271.84	532.93	67.26
		100	579.19	536.4	64.23
Thai Statue	5000	10	155.61	491.33	125.82
		100	247.26	493.87	123.14
	10000	10	209.99	507.04	109.84
		100	404.2	500.16	105.06
	25000	10	356.63	524.42	99.64
		100	846.15	536.33	96.04
Lucy	5000	10	163.88	494.13	106.22
		100	258.98	485.98	103.58
	10000	10	209.72	502.05	97.62
		100	405.57	503.39	95.09
	25000	10	400.4	541.82	89.8
		100	897.9	541.32	85.39

16-bit identifiers. Caches with size of 10 and 100 were used to determine their influence on memory consumption and rendering time. The LRU policy was used in the cache management for this experiment.

We used the Firefox browser version 47 in the experiments. To perform a Web browser automation, we used the Selenium tool [3]. The memory consumption assessment was implemented in Python programming language. The experiments were executed on a system with 2 Intel Xeon CPU E5-2640 2.50 GHz with 15 MB cache and 32 GB main memory, running Ubuntu 16.04.

Table 2 shows the measurements of maximum memory consumption for the server and client browsers, as well as the rendering time. It is possible to observe the increased memory consumption on the server in the larger meshes, since they have more index information kept in memory. This large consumption does not occur on the client, because no index information is necessary. The use of larger clusters results in greater memory consumption both for the server and client machines, but less time rendering. The cache size has little influence on the memory consumption on the client, such that the consumption difference may occur due to browser optimizations. The cache size has influence, however not significantly, in the rendering time on the client. The use of larger caches reduces the need for disk reading by the server. On the server, the use of larger caches results in greater memory consumption, reaching double the consumption in the experiments. The rendering time for large meshes (Lucy and Thai Statue models) is high, about 100 seconds. However, the use of progressive rendering and cluster rendering ordering speeds up the results for the user. From the results, it is possible to conclude that the use of larger clusters and smaller caches is more cost-effective, demanding less rendering time and regular memory consumption.

5 Conclusions and Future Work

This paper described an out-of-core methodology for remote rendering of three-dimensional triangle meshes directly on Web browsers. The developed tool requires only a modern browser in the client machine that supports HTML5 language, WebGL and WebSockets, without the need for special software or additional plugins, as in the case of VRML and X3D.

The data rendering performed directly in the client browser makes it simple for users since the process eliminates the use of advanced features on their devices. The use of out-of-core techniques allows the visualization of massive triangle meshes with a low cost memory consumption, which is not possible in the conventional WebGL standard, enabling rendering on mobile devices.

Some directions for future work include the investigation of efficient techniques for cluster compression and effective mechanisms for out-of-core level-of-detail structures.

Acknowledgements. The authors are thankful to São Paulo Research Foundation (FAPESP grants #2014/12236-1 and #2015/12228-1) and Brazilian National Council for Scientific and Technological Development (CNPq grants #300677/2017-0 and #305169/2015-7) for their financial support.

References

1. The Stanford 3D Scanning Repository: http://graphics.stanford.edu/data/3Dscanrep/
2. Flask - A Python Microframework (2016). http://flask.pocoo.org/
3. Selenium - Web Browser Automation (2016). http://www.seleniumhq.org/
4. Amorim, P., Moraes, T., Silva, J., Pedrini, H.: InVesalius: an interactive rendering framework for health care support. In: 11th International Symposium on Visual Computing. Lecture Notes in Computer Science, vol. 9474, pp. 45–54. Springer, Cham (2015)
5. Blazona, B., Mihajlovic, Z.: Visualization service based on web services. J. Commput. Inf. Technol. **4**, 339–345 (2007)
6. Cabello, R.: three.js—JavaScript 3D library (2016). http://mrdoob.github.com/three.js/
7. Evesque, F., Gerlach, S., Hersch, R.: Building 3D anatomical scenes on the web. J. Visual. Comput. Anim. **13**(1), 43–52 (2002)
8. Fazanaro, D., Amorim, P., Moraes, T., Silva, J., Pedrini, H.: NURBS parameterization for medical surface reconstruction. Appl. Math. **7**(2), 137–144 (2016)
9. Fette, I., Melnikov, A.: The WebSocket Protocol. RFC 6455 (Proposed Standard) (2011)
10. Garland, M., Heckbert, P.S.: Surface simplification using quadric error metrics. In: 24th Annual Conference on Computer Graphics and Interactive Techniques, pp. 209–216. ACM Press/Addison-Wesley Publishing Co. (1997)
11. Gurung, T., Luffel, M., Lindstrom, P., Rossignac, J.: LR: Compact connectivity representation for triangle meshes. ACM Trans. Graph. **30**(4), 67:1–67:8 (2011)
12. Isenburg, M., Lindstrom, P., Gumhold, S., Snoeyink, J.: Large Mesh Simplification using Processing Sequences. In: 14th IEEE Visualization, pp. 465–472 (2003)
13. Khronos Group Inc.: glDrawElements (2016). https://www.khronos.org/opengles/sdk/docs/man/xhtml/glDrawElements.xml
14. Moraes, T., Amorim, P., da Silva, J., Pedrini, H., Meurer, M.: Medical volume rendering based on gradient information. In: V ECCOMAS Thematic Conference on Computational Vision and Medical Image Processing, pp. 181–186. Tenerife, Canary Islands, Spain (2015)
15. OsiriX: DICOM Image Library: http://www.osirix-viewer.com/resources/dicom-image-library/
16. Pedrini, H.: An adaptive method for terrain surface approximation based on triangular meshes. Ph.D. thesis, Department of Electrical, Computer, and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY, USA (2000)
17. Rossignac, J.: 3D Compression made simple: Edgebreaker with zip & wrap on a corner-table. In: International Conference on Shape Modeling & Applications, pp. 278–283. IEEE Computer Society, Washington, DC (2001)
18. Taubin, G.: A Signal Processing Approach to Fair Surface Design. In: 22nd Annual Conference on Computer Graphics and Interactive Techniques, pp. 351–358, New York, NY, USA (1995)
19. Vieira, A.W., Lewiner, T., Velho, L., Lopes, H., Tavares, G.: Stellar mesh simplification using probabilistic optimization. Comput. Graph.Forum **23**(4), 825–838 (2004)
20. WebGL: OpenGL ES 2.0 for the Web (2016). http://www.khronos.org/webgl/
21. Wessels, A., Purvis, M., Jackson, J., Rahman, S.: Remote data visualization through WebSockets. In: Eighth International Conference on Information Technology: New Generations, pp. 1050–1051 (2011)

22. Yoon, S.E., Lindstrom, P.: Random-accessible compressed triangle meshes. IEEE Trans. Visual. Comput. Graph. **13**(6), 1536–1543 (2007)
23. Yoon, S.E., Lindstrom, P., Pascucci, V., Manocha, D.: Cache-oblivious mesh layouts. ACM Trans. Graph. **24**(3), 886–893 (2005)