Image-based Network Rendering System for Large Sized Meshes

Yasuhide Okamoto, Takeshi Oishi, and Katsushi Ikeuchi
University of Tokyo
Institute of Industrial Science 3rd Dept., Ikeuchi Laboratory,
4–6–1 Komaba, Meguro-ku, Tokyo, 153–8505, JAPAN
{okamoto, oishi, ki}@cvl.iis.u-tokyo.ac.jp

Abstract

Recent advancement in sensing and software technologies enables us to obtain large scale, yet fine 3D mesh models of cultural assets. However, such large-scale models cannot be displayed interactively on consumer computers because of the performance limitation of the hardware. We propose an interactive rendering system for large scale 3D mesh models, stored on a remote machine through relatively small capacity of networks. Our system uses both model and image based rendering methods for efficient load balance between a server and clients. On the server, the 3D models are rendered by the model-based method using a hierarchical data structure with multi-resolution. On the client, it reconstructs an arbitrary view by using a novel image-based method, referred to as the Grid-Lumigraph, with blending image colors from sampling images received from the server. The resulting rendering system can efficiently render any image in real-time.

1. Introduction

Sensing and modeling technologies have advanced drastically. One of the most important applications of these techniques is digitization of heritage assets, referred to as e-Heritage. These can be used for preservation, planning restoration, and PR and education. Some of the representative projects include: [8, 13].

Typically, e-Heritage consists of billions of triangles with high complexities. So far, it has still been difficult to view such e-Heritage in real-time on current consumer computers. First of all, current internet does not have the capability to download such mesh models in real-time. Secondly, usual PC at the client side cannot render such e-Heritage in real-time.

This paper proposes an efficient rendering system by both model and image based rendering as shown in Figure 1. Our system locates original mesh models of e-Heritage on a remote server, in order to avoid the channel limitation between the server and the client. The server pre-renders the mesh models from various viewing positions, and stores these images in a repository. In run time, the client sends a request of displaying of the mesh model, at a certain view position, with the viewpoint parameters. The server sends back the pre-rendered images, necessary to calculate the view as well as the sparse mesh model. The client calculates and displays the new view, by using the image-based rendering with the set of images and the sparse mesh model from the server.

This paper has the following outline. Section 2 surveys the related work and discusses the benefits and drawbacks of the methods. In section 3, we describe the construction of a repository composed of sampling images by model-based LOD rendering. Moreover in section 4 we describe Grid-Lumigraph, which is the image based method for reconstruction of arbitrary views from sampling images. We explain the detail of our proposed server-client rendering system in section 5, perform and evaluate the system in section 6, and we conclude in section 7.

2. Related works

The Level of Detail (LOD) method is proposed for displaying large scale mesh models in [14]. LOD methods represent 3D objects with mesh models in multi-resolution. The Progressive Mesh presented in [7] records the sequence of the reduction that merges smaller triangles to form larger ones in the mesh structure. The Adaptive Tetrapuzzles proposed in [1] converts the input mesh into a hierarchy structure composed of nodes, containing smaller meshes, referred to as patches with multi-resolution. Sending mesh data over the network is very expensive, though.

The LOD methods are also used in point-based representations. QSplat [15] and Layered Point Cloud [4] are point-based rendering systems, which use points as the rendering primitive with a hierarchical data structure. Far Voxels proposed in [5] uses points and voxels with view-dependent

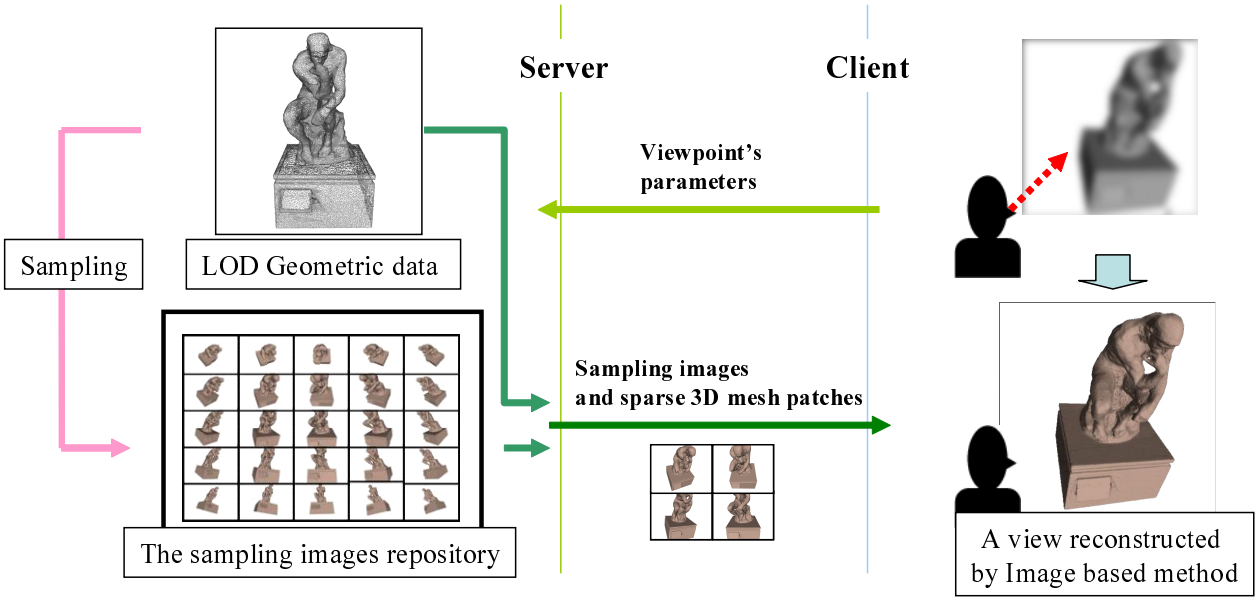


Figure 1. The overview of proposed rendering system. The server has geometric data recorded in format of LOD hierarchy, and the repository of sampling images of the input 3D model. The client system displays arbitrary views reconstructed by using received images and sparse 3D mesh patches.

color as the rendering primitives in the LOD hierarchy. Those methods can be extended to server-client rendering system as proposed in [1, 4, 5, 7, 16]. Although this point-based representation is much more compact than the mesh-based, depending on the complexity of the input model, the communication traffic can still be high.

In contrast to those geometry based methods using such as mesh or point as described above, the image based methods are also extended those in the server-client method in [11]. In this approach, the server has the geometric model, and renders then sends images corresponding to the requests from clients. Although the server only needs to send an image in real-time, rendering at the server is a costly operation, and, if too many requests occur, the server may break down.

Impostor, presented in [2, 9, 17] is the rendering method on the network using geometric and image information. This system is mainly designed for walkthrough's environments such as urban scenes. It assumes that a 3D model is already established on the client, which is not the case in our paradigm.

3. Generating sampling images

Our rendering system generates and stores the pre-rendered image, to be sent to clients, in the repository on a server machine. The Level of Detailed (LOD) rendering method extended from the Adaptive Tetrapuzzles [1] is employed for rapid construction of the image repository.

3.1. Constructing LOD hierarchy

Our method to construct LOD is different from the Adaptive Tetrapuzzles. The Adaptive Tetrapuzzles recursively divide the space and construct a hierarchical structure. The approach of this method is very simple to implement but very efficient to process. However, the number of triangles of our target 3D models is very huge, so the recursive splitting process is very time-consuming. To solve it, we perform the splitting process not by triangles but by small meshes. Our method defines a voxel space of pre-determined resolution and generates a group of small meshes (Step 1), forms a graph of divided meshes for constructing a tree structure from them (Step 2), and then simplifies the tree structure (Step 3). We prefer this method for space efficiency adapted to the object shape in hand.

Step 1: Decomposition into voxel space We define the voxel space at the finest level. Then, we decompose an input mesh model into smaller meshes using the voxel space. We sort each triangle to a single voxel which contains at least one of its vertices. If the vertices of a triangle belong to multiple voxels, it is sorted to the voxel which contains the most vertices of the triangle. Finally, we assign a group of meshes to the corresponding voxel. We control granularity of the voxel space so that each voxel contains less number of triangles than the pre-defined value N_v.

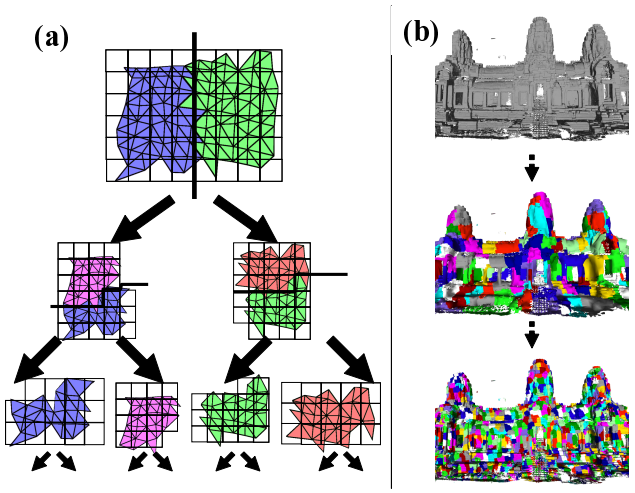


Figure 2. Construction of Level-of-Detail hierarchy. (a): Recursive splitting of mesh by graph-partitioning of voxel space. (b): Results of partitioning into small mesh patches.

Step2: Recursive graph partitioning We convert a group of meshes, defined in the Step 1, into a graph representation, $G(V, E)$, for the construction of a LOD structure in the next step. One vertex in the vertex set, V, of the graph corresponds to one voxel and one edge in the edge set, E, corresponds to the adjacent relation between two voxels. Thus, in this graph, at most, one vertex has six edges corresponding to six adjacent voxels.

For splitting the graph evenly, and adaptively for shapes, we assign a weight to each vertex and each edge. Here, a vertex weight is defined as the number of belonging triangles. To define edge weight, first, we calculate the mean surface normal vector at each vertex. Then, an edge weight is defined as the inner product of two averaged vectors of the two vertices on the edge.

We recursively split the graph into a pair of sub-graphs, shown in Figure 2. This splitting procedure is continued until the size of each sub-graph, defined as the number of the belonging triangles, is less than a pre-defined number N_l. This graph partitioning procedure is implemented by using the Metis library [10].

After this procedure, we can obtain a hierarchy structure, whose each leaf node has a small sub-graphs of the size N_l. Non-leaf nodes also contain larger sub-graph. At each node in the graph structure, we connect all meshes of that node into a continuous mesh patch.

Step 3: Simplification at non-leaf nodes We simplify a mesh patch at each non-leaf node in the constructed hierarchy. The number of triangles at each node is reduced to a pre-defined number N_n. We implemented this simplification method by using a quadric error metric [3]. This method iteratively collapses edges from the lowest edge, in ascending order of quadric errors, until the number of triangles becomes the desired number. Here the quadric error represents a rough approximation of the distance between original and simplified mesh patches.

Each node holds the node parameters and the geometric data of a simplified mesh patch. The node parameters, given by a mesh patch, are eight corner positions of the bounding box, the range of surface normal and the minimum quadric error in simplification. The geometric data consists of positions and connectivity relations of polygonal vertices in the mesh patch. Node parameters will be used for traversing the hierarchical structure, while the geometric data is used for rendering the mesh patch.

Simplification procedure ensures consistency of boundaries between mesh patches. Inconsistency between mesh patches causes holes and artifacts along the patch boundary on rendering. Simplification is not conducted on edges either along boundaries or directly connected to boundaries.

3.2. Efficient LOD rendering

In order to generate a set of pre-rendered images our system uses the LOD structure. Rendering process traverses the constructed hierarchy from the root node along the tree structure following the depth-first search strategy. If the process successfully finds a mesh patch which satisfies necessary resolution, the process tracks back following the depth-first search strategy, while it adds the node to a list of rendering nodes. The necessary resolution is given by the projected quadric error $\frac{\epsilon}{r}$, where ϵ and r are the quadric error at the node, and the distance between a viewpoint and the center of the mesh patch. If the procedure reaches a leaf node, it is also added to the list. After traversing the entire tree structure, the resulting list of rendering nodes is given to the rendering pipeline.

Some exceptional cases occur in the tree traverse, which are out of screen and back-face. An out of screen case is given by a node, of which bounding box is projected outside of the screen. In a back-face case, a traversed node contains triangles, in the mesh patch, which turn away from the viewing direction. If one of these cases occurs, the traverse operation immediately backtracks along the tree structure following the depth-first strategy.

3.3. Building a sampling repository

The previous section described the method to traverse the LOD structure. In this section, we construct a set of pre-rendered images, referred to as a sampling repository. Later, a group of images from this repository will be sent to a client for image-based rendering.

We sample the viewing space, twice larger than the input 3D model in our implementation, into regularly located

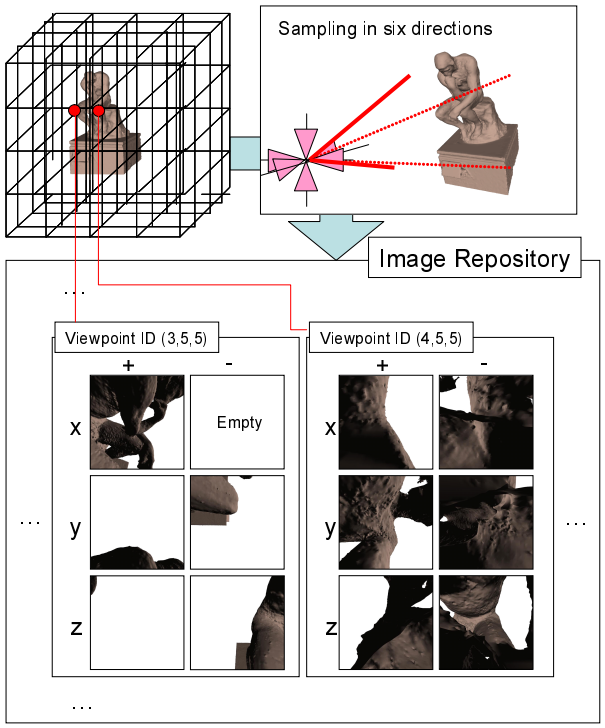


Figure 3. Viewpoints for sampling are located on each grid points of the voxel space surrounding the input 3D model, and the view directions are set along axes x, y, and z. The image repository manages those sampling images in a hash table.

sampling view points. The granularity of sampling is manually decided depending on the density and complexity of the input model. At each viewpoint, we generate an image of the object using the LOD structure, in six directions along the axes, positive and negative directions of x, y, and z described in Figure 3. To capture all rays in the space, the viewpoint is covered by six cubic faces. Each face is an image plane, whose aspect ratio and the angle of the field of view, are 1 and 90 degree, respectively. If there are no mesh patches to be rendered in a sampling image, we can skip recording the image and set the empty flag instead of it. Sampling images are stored as JPEGs in the image repository. Those images are retrieved using a hash table with the combination of grid point id and direction key.

4. Grid-Lumigraph for image reconstruction

A client machine in our system can display any arbitrary view of the input 3D model, from the direction that a user chooses, using a new image-based method, referred to as the Grid-Lumigraph. Our Grid-Lumigraph reconstructs a view, with only sampling images near current viewpoints and a rough 3D mesh model.

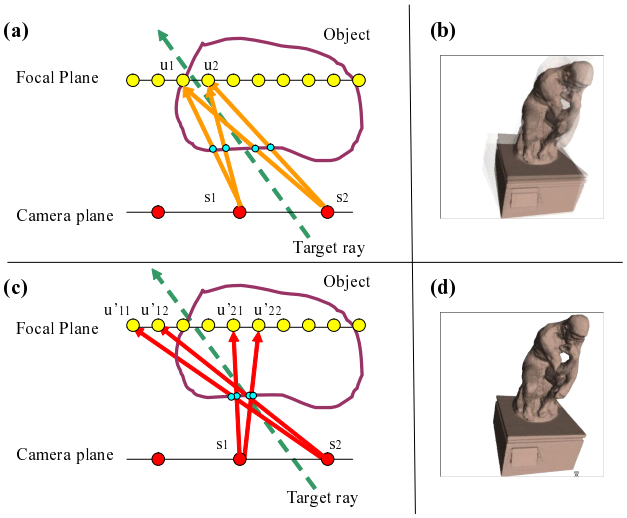


Figure 4. Correct sampling with geometric information. (a): Extraction colors from sampled colors without geometric information. (b): The result without correction. (c): Extraction with geometric information. (d): The result with correction.

4.1. Grid-Lumigraph

Our Grid-Lumigraph has the same basic idea as the Light Field Rendering [12], which reconstructs a view from color values of all rays going through the view. The collection of all rays for this operation is referred to as Light Field.

Constructing a view using the Light Field method is depicted in Figure 4(a). The dotted line indicates one of the new rays necessary for rendering a updated image. The color values on the nearest rays s_1-u_1, s_1-u_2, s_2-u_1 and s_2-u_2 are blended into the new color value along the dotted ray. This Light Field method requires dense sampling of s-u pairs, otherwise, the reconstructed image may have the blurs and ghosting, as shown in Figure 4(b).

The Lumigraph [6], shown in Figure 4(c), utilizes geometric information to remedy this issue. The dotted line also indicates a necessary ray which need to be generated. The geometric information provides the intersection position between the ray and the object surface, which help use more appropriate rays. This correction does not need precise 3D model. Since our case has a coarse-level 3D mesh model, we prefer this Lumigraph type operation.

Our Grid-Lumigraph texture maps the sampling images on the 3D mesh model. The image repository stores sampling images at each viewpoint in 3D space. Once a viewpoint is selected by a user, the Grid-Lumigraph chooses the nearest sampling images around viewpoints and then projects those images onto the 3D mesh model. The Lumigraph and our Grid-Lumigraph have the same principle to use the image-based rendering with a rough 3D mesh model. The Lumigraph calculates the nearest rays from the

Light Field, while our Grid-Lumigraph determines the nearest images from the image repository. Our Grid-Lumigraph only needs texture mapping capabilities and is very efficient for rendering.

4.2. Grid-Lumigraph on GPU

The Grid-Lumigraph can be easily implemented on a GPU. For generation of a new image, the Grid-Lumigraph uses sampling images, typically four to twelve, around the view point selected by a user. Then, the Grid-Lumigraph projects each sampling image, one by one, using the projected texture mapping method onto the 3D mesh model. This procedure can be efficiently done by GPU's texture mapping capability, shown in Figure 5. In this projection, shadow mapping of the GPU is also utilized to avoid mapping rays to back faces of the 3D mesh from the sampling view point. These mapping results are projected and normalized on the current viewing image plane of the user, using the built-in GPU capability.

5. Rendering system through the network

Our system has a server for image repository and some clients for view generation. This section describes the detail of data exchange between the server and the clients over the network.

5.1. Protocol for rendering

The protocol between a server and clients for rendering is described in Figure 6.

Step 1: When a user requests to display the 3D mesh model from one particular viewpoint, the client system sends the parameters of the current viewpoint to the server system. Those parameters include the current view position, the current viewing direction and the rotation angle around the viewing direction.

Step 2: On receiving the current viewpoint, the server determines the set of the nearest sampling points. Then it retrieves those images, from the image repository, using a hash key as a grid point's ID.

Step 3: The server sends sampling images to the clients. Before sending images, the server checks whether the client has already had the same image in the cache or not. The server sends it, only if the client does not. In addition to it, the server sends extra images to the client. In one process to send images, the server sends not only images nearest to viewing directions at nearest sampling point (say the x direction at a point), but also images in other directions at the points (say the y and z direction at the point).

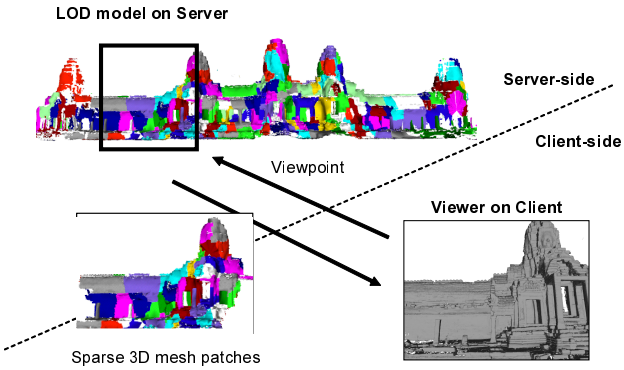


Figure 7. Communication of the geometric data. The server provides very sparse mesh patches depending on the client's viewpoint. The sparse mesh patches are chosen from the LOD hierarchy structure.

The client saves received images as cache in the local memory. The stored sampling images in the cache are managed in the manner of least recent used (LRU).

Step 4: The server sends the sparse 3D model to the clients. The server has the sparse 3D mesh model in the multi-resolution hierarchy. In the initial request, the server sends the entire model which is composed of coarse mesh patches extracted from hierarchy data structure. In the later requests, it sends the corresponding mesh patches, visible from the current viewpoints, described in Figure 7. The clients request new mesh patches, if necessary. In particular, when zooming in toward the object, the number of vertices displayed is reduced. The client requests new mesh patches in a finer resolution, if the number of vertices is less than a predefined value R_v.

5.2. Additional capabilities for better performance

Under a rapid change in viewpoints, the server and the client work asynchronously to avoid stall. The client continues to send viewpoint's parameters, while it renders a new image using other images whatever available in the cache. It updates the rendered results whenever new data is available from the server. The server continues to send sampling images corresponding to the received requests.

Some holes may occur in reconstructed images from sampling images. The main reason is that sampling granularity is lower than complexity of the input object's shape. If the holes are small, we cover them by calculating shading effect using the surface orientation of the triangles and the light source direction. In the case the size of a hole is larger than a threshold, the client requests the server to generate the current view from the current viewpoint.

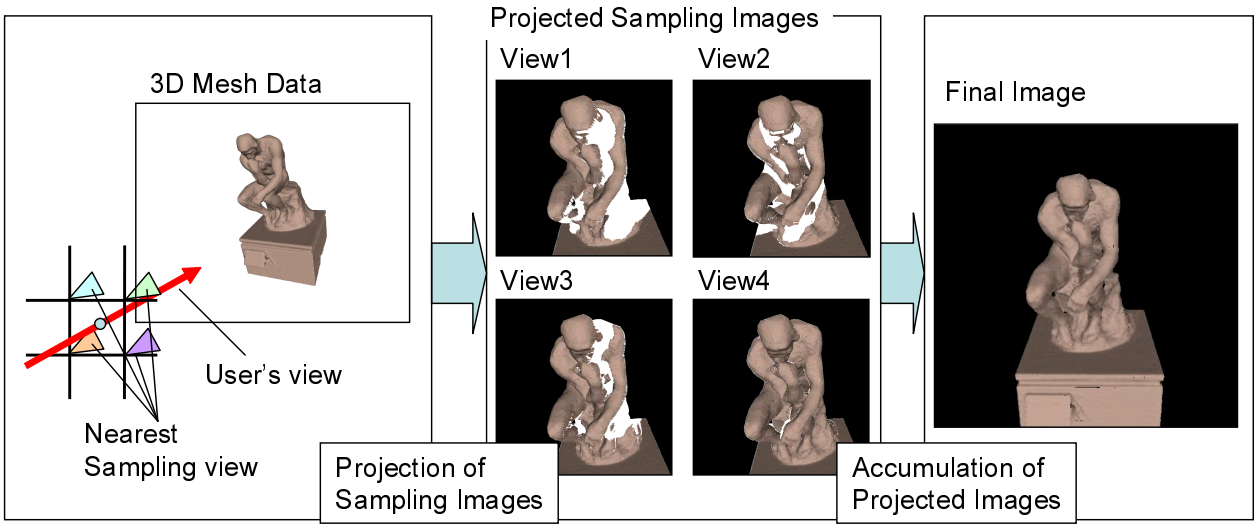


Figure 5. The reconstruction of images by projective texture mapping using the GPU capability. We can efficiently extract each pixel's correct color, from sampling images, by texture projections. Each nearest sampling image is project from the sampling point to the geometric data. By accumulating the results on the final buffer, we can obtain the reconstructed image.

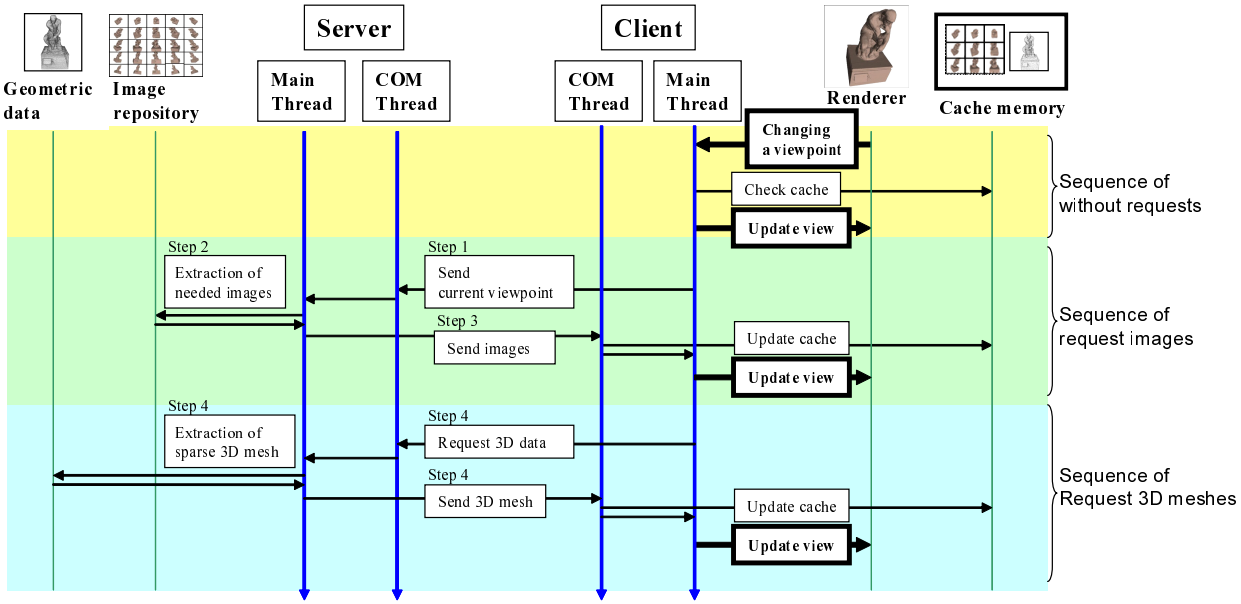


Figure 6. The sequence chart for rendering pipeline on our system. If the client's cache has enough data to reconstruct the current view, the client's thread can display the view without requests to the server. Otherwise, the client make requests for images and 3D meshes to the server. The data communication is managed by COM threads, and it is asynchronously done. The renderer updates the view whenever the data communication is finished.

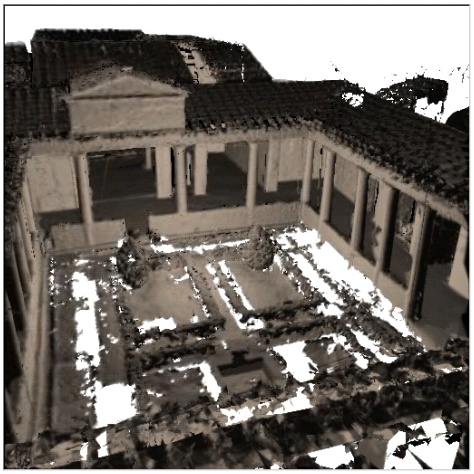
6. Implementation and results

We implemented and performed the server and client on 2.4GHz AMD Athlon64 X2 PCs with 4GB RAM, which has GeForce 8800GTS with 512MB of video memory. Our system runs on Windows XP. We used 1GBit LAN between the server and a client. We use NVIDIA Cg Toolkit for implementation of details in GPU processing on the server and clients.

We constructed the LOD structure on the server, setting N_v to 500, N_l to 4000, and N_n to 2000. The size of cache on clients is 40MB for images and 10MB for sparse meshes. The minimum number of rendered meshes' vertices on clients, R_v, is set to 10000. The dimension of sam-

Model	Thinker
Input Triangles	1,742,122
Sampling Grid	16
Average fps	59.5

Model	Bayon Face
Input Triangles	5,922,790
Sampling Grid	16
Average fps	48.1



Model	Menandro's house
Input Triangles	10,388,120
Sampling Grid	32
Average fps	33.7

Model	Bayon 3 towers
Input Triangles	18,132,893
Sampling Grid	32
Average fps	38.5

Figure 8. The parameters of input 3D models and rendering results on clients' renderer.

pling images is set to 512 pixels.

The rendering results on clients are shown in Figure 8. The input models have large numbers of triangles from one million to twenty million. We constructed the image repositories for those models, whose sampling number per dimension is 16 or 32, described as Sampling grid in Figure 8. In the experiment, all models were efficiently rendered in real-time, over 30 fps, on clients, even if the input model is very large. Additionally, we also evaluated the size of received data from the server in rendering process for the model of "Bayon 3 towers", shown in Figure 9. The size of received data is almost always less than 500kB while the rendering process.

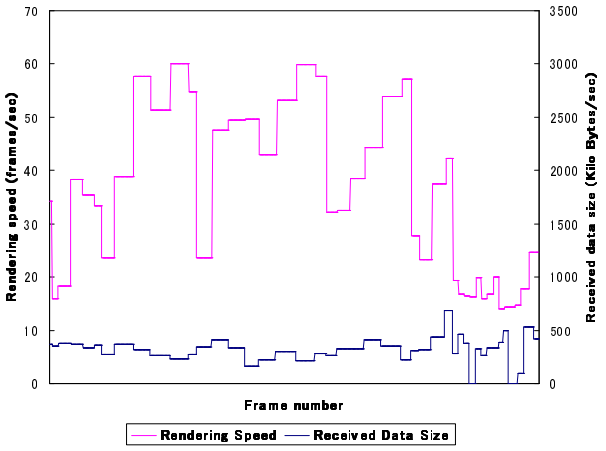


Figure 9. Rendering speed (frames per second) and received data size (kilo bytes per second).

7. Conclusion

We proposed the view-dependent rendering system for large-scale 3D models located on the remote server. In our approach, we use a model-based rendering method for repository generation at the server and a novel image-based method, referred to as the Grid-Lumigraph, for view generation on the client. A model-based rendering system, on the server, generates rendering images from various view positions using the LOD structure and stores these sampling images in the image repository.

The rendering system, on the client, displays a requested view by using the Grid-Lumigraph using a sparse 3D mesh model and sampling images nearest to the viewpoint. The Grid-Lumigraph, a variation of light-field method, projects blended sampling images using the projective texture mapping method. The Grid-Lumigraph can be implemented effectively on the GPU.

Our system can display huge 3D models, which have a huge number of triangles, in real-time with efficient data communication.

References

[1] P. Cignoni, F. Ganovelli, E. Gobetti, F. Marton, F. Ponchio, and R. Scopigno. Adaptive TetraPuzzles - efficient out-of-core construction and visualization of gigantic polygonal models. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2004)*, 23(3):796–803, 2004. 1, 2

[2] S. Dobbyn, J. Hamill, K. O'Conor, and C. O'Sullivan. Geopostors: A real-time geometry/impostor crowd rendering system. *ACM Transaction on Graphics (Proceedings of ACM SIGGRAPH 2005)*, 24(3):933–933, 2005. 2

[3] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of ACM SIGGRAPH 97*, pages 209–216, 1997. 3

[4] E. Gobbetti and F. Marton. Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers and Graphics*, 28(6):2004, 2004. 1, 2

[5] E. Gobbetti and F. Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Transaction on Graphics*, 24(3):878–885, 2005. 1, 2

[6] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. In *Proceedings of ACM SIGGRAPH 96*, pages 43–54, 1996. 4

[7] H. Hoppe. Progressive meshes. In *Proceedings of ACM SIGGRAPH 96, Computer Graphics*, pages 99–108, 1996. 1, 2

[8] K. Ikeuchi, K. Hasegawa, A. Nakazawa, J. Takamatsu, T. Oishi, and T. Masuda. Bayon digital archival project. In *Proceedings of Virtual Systems and Multimedia 2004*, pages 334–343, 11 2004. 1

[9] S. Jeschke, M. Wimmer, H. Schumann, and W. Purgathofer. Automatic impostor placement for guaranteed frame rates and low memory requirements. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pages 103–110, 2005. 2

[10] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998. 3

[11] D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia, P. Cignoni, and R. Scopigno. Protected interactive 3d graphics via remote rendering. In *Proceedings of ACM SIGGRAPH 2004*, pages 695–703, 2004. 2

[12] M. Levoy and P. Hanrahan. Light field rendering. In *Proceedings of ACM SIGGRAPH 96*, pages 31–42, 1996. 4

[13] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital michelangelo project: 3d scanning of large statues. In *Proceedings of ACM SIGGRAPH 2000, Computer Graphics*, pages 131–144, 7 2000. 1

[14] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2002. 1

[15] S. Rusinkiewicz and M. Levoy. QSplat : A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352, Jul. 2000. 1

[16] S. Rusinkiewicz and M. Levoy. Streaming QSplat : A viewer for networked visualization of large, dense models. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 63–68, 2001. 2

[17] J. Shade, D. Lischinski, D. H. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of ACM SIGGRAPH 96*, pages 75–82, 1996. 2