

A hierarchical framework for large 3D mesh streaming on mobile systems

Jiro Park · Haeyoung Lee

Received: 27 April 2014 / Revised: 27 September 2014 / Accepted: 10 November 2014 /
Published online: 29 November 2014

© The Author(s) 2014. This article is published with open access at Springerlink.com

Abstract Large 3D meshes are emerging as the new media for various applications but are still hard to use in mobile applications, due to the limited resources of mobile systems. This paper introduces a large 3D mesh streaming framework which flexibly deals with the limited resources of mobile systems and also provides a user with interactive controls and random accessibilities. To reduce resource usage, our framework presents a uniform mesh partitioning algorithm, in which each partition of a large mesh has the same number of vertices. Our uniform partitioning is based on a k-d tree clustering and extended for out-of-core meshes. A median search with heaps in the main memory is designed for faster external sorting. Large 3D meshes are transformed into a set of partitioned and simplified meshes in the server. For interactive 3D browsing, on mobile devices, our framework presents a mobile 3D viewer which is hierarchically designed with intuitive interfaces. A user can experience rapid 3D searches with 3D previews of simplified meshes. Multi touch inputs can control zooming and the level of detail in meshes automatically. Double tap touches enable a user to randomly select a region of a large mesh, and a set of partitions for the selected region will be streamed and displayed on a mobile client. As a result, our framework enables a user to browse large 3D meshes on a mobile system interactively, while optimizing system resource usage and protecting the original data in the server.

Keywords Mobile 3D · 3D mesh streaming · Vertex clustering · Out-of-core mesh partitioning

1 Introduction

As a result of the rapid spread of mobile systems and cloud computing, various media such as texts, images, sounds, or videos are widely used in many mobile applications. 3D meshes are being widely utilized in mobile applications such as 3D games, however, there are challenges

H. Lee (✉)
Computer Engineering Department, Hongik University, 72-1 Sangsu-dong Mapo-gu, Seoul zip: 121-791,
South Korea
e-mail: leeh@hongik.ac.kr

J. Park
Wemade entertainment, Seoul, South Korea
e-mail: qkrwlfh@gmail.com

to be overcome. Unlike PCs, mobile systems have relatively limited resources such as low power, low memory, slow CPU, and small screens. The advance in 3D mesh processing technologies now enables us to create large sophisticated 3D meshes. 3D meshes utilized in medical imaging and cultural heritage preservation, usually include up to millions of vertices. Therefore, browsing large 3D meshes is still difficult on mobile systems with the limited resources.

The large size of 3D meshes has inspired various studies such as mesh compressions [1, 5, 17], simplifications [6, 22], chartifications [5], and out-of-core methods [5, 6, 22]. These previous methods were developed for PCs with more resources. Progressive mesh streaming techniques [4, 13, 15, 23, 25] have also been suggested for 3D applications mostly on PCs. Most previous streaming techniques transmit data for each vertex in a predefined order. The transmission of vertex-based information takes too much time to process because of the volume of information involved with millions of vertices in large 3D meshes. Furthermore, no user interactions to control progressivity or accessibility are allowed, either.

3D visualizations on mobile devices have been actively studied recently. Generally they can be classified into two categories: remote rendering and local rendering. Remote rendering [9, 16, 24] usually displays only images which are generated and transmitted by the remote server. Local rendering [10, 11, 15, 20] receives 3D mesh data from the server and then renders it directly on mobile systems. Generally remote rendering utilizes plentiful resources on the server and protects the originality of the data and the copyright. Local rendering provides interactive user controls. However, these previous visualization methods are hard to apply to large 3D meshes. For local rendering, receiving all the mesh data on mobile systems may take too much time, or be impossible due to insufficient memory. For remote rendering, images may not fully display the fine details of large 3D meshes, or require too much time for dynamic view changes, due to repeating procedures of rendering a large 3D mesh on the server, and then transmitting and displaying the images on the client.

Therefore, in this paper, we propose a new framework for streaming large 3D meshes on mobile systems. We propose to consider the fact that users cannot see every detailed part of a large 3D mesh at one time. As shown in Fig. 1, we suggest that a large 3D mesh should be transformed into a set of small meshes partitioned and simplified in the server. A mobile 3D viewer should also be created allowing a user to interactively control models, levels of detail, and random streaming.

Generally, a mesh partitioning algorithm performs vertex or face clustering. Previous partitioning methods concentrated on functional or visual qualities [3, 5, 22] and resulted in non-uniform partitioning where each partition mesh has various numbers of vertices in non-compact shapes [12, 18]. Our research suggests that a uniform partitioning, where each partition has the same number of vertices is more effective [14]. The shape of each partition should also be as compact as possible to facilitate good visual quality of streamed meshes. Our method builds a k-d tree structure of a large 3D mesh. Our k-d tree is constructed recursively by dividing a mesh into two sub-meshes, each having half the vertices of the original mesh and consequently results in uniform partitions of a large mesh. The k-d tree is used for multi-resolution simplifications and is also used as an index to find a set of partitions to be streamed in the server for a part of the mesh, which a user randomly selects on the client. Our uniform partitioning will help to create a standard processing time for each partition, utilize cache memory efficiently, minimize I/O or power usage, and ultimately optimize the usage of the limited resources on mobile systems. With our mobile 3D viewer, a user can start rapid 3D searches from a small preview of a mesh, simplified in a low resolution and then select a mesh to see in a full screen. Multi touch inputs can naturally control a hierarchy in multi-resolution simplifications. When requested by a user's double tap on a random part of the simplified mesh, a set of small partitions of a large 3D mesh is streamed from the server to a mobile client and rendered locally.

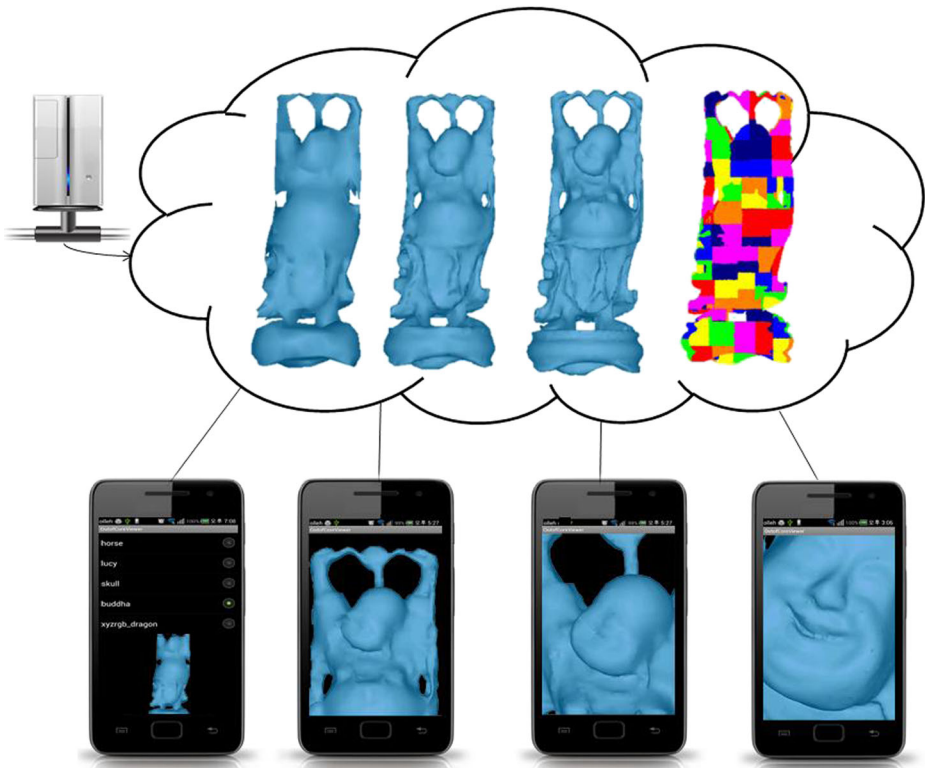


Fig. 1 An example of our 3D mesh streaming with a mesh Buddha

As a result, our framework enables a user to interactively browse large 3D meshes on a mobile client and also protects the original data while optimizing system resource usage. We have published a short paper of our work presented at a conference [14]. In this paper, we present the details of our method and extend our partitioning to utilize out-of-core memory for large 3D meshes. We also elaborate on our mobile viewer extensively, primarily with the goal of designing its intuitive interfaces.

The rest of the paper is organized as follows. In [Section 2](#), the related works are explained. [Section 3](#) presents an overview of our mesh streaming framework. Our mesh partitioning algorithm is explained in detail in [Section 4](#). Our mobile 3D viewer is introduced in [Section 5](#). Finally, [Section 6](#) concludes this paper with future works.

2 Related works

Our 3D mesh streaming framework is closely related to two research areas: 3D mesh partitioning and 3D visualizations on mobile systems.

2.1 3D mesh partitioning algorithms

Most mesh segmentation methods have focused on visual or functional qualities. A benchmark study, on mesh segmentations has already been published [3]. However, mesh segmentations

cannot be applied to large 3D meshes on mobile systems because they usually require: (a) many user interactions, (b) complex computations to determine where a vertex or a face belongs, or (c) repetition for optimization. Mesh partitioning has been performed by clustering vertices or faces. Previous methods can be classified into two categories: space divisions and incremental additions.

Space division methods first define a bounding box for a mesh. The bounding box is then recursively subdivided until user-specified conditions are satisfied. Various tree structures with a hierarchy have been presented for space divisions. Octree is widely used for various mesh processing algorithms, such as mesh simplification [6, 22] and compression [11, 15, 17], since it can provide fast space subdivision.

Incremental clustering methods which use cost functions, have been published [5, 12, 18]. The most widely used algorithm is k-means clustering illustrated by Lloyd [18]. The algorithm starts with k vertices or k faces randomly selected as seeds. A user specifies the cost function to be used by selecting the next vertex or face to add among neighboring vertices or faces. After every vertex or face is grouped to the k partitions, new seeds are recalculated and incremental clustering is repeated again to the user's specifications. K-means clustering has also been used for mesh simplifications [21] and compressions [5].

The octree method guarantees fast clustering due to simple subdivisions of the given region. However, the numbers of vertices in sub-regions or partitions are not the same because the division is performed, not by the number of vertices, but by the size of the regions. The k-means clustering method cannot generate uniform partitions either [14]. K-means clustering may take a great deal of time, perhaps many hours, due to optimization requiring many repetitions for large 3D meshes. They cannot provide progressivity in resulting partitions, and require a separate index mesh to allow random accessibility [5].

2.2 3D visualizations on mobile systems

3D applications, such as games, have recently inspired many studies on 3D visualizations on mobile devices. Generally previous studies have focused on remote rendering or local rendering.

Local rendering techniques [10, 11, 15, 20] require mesh data to be placed on mobile clients. The server simply sends the mesh data to mobile clients and the mobile clients are responsible for displaying the mesh itself. Previous local rendering studies have suggested various ways to send the mesh data to deal with the limited resources. A point-based rendering method [11] proposed that the server should progressively send vertex data. This method determined it was best to omit sending face data to mobile clients. The progressive local rendering method [15] suggested that the server should send compressed vertex data only during intermediate levels and later add face data at the last level. These previous methods build an octree of the mesh for compression, simplification, and progressivity. For a 3D tree model, a shape-based progressivity was proposed as incrementing the number of branches and leaves on mobile clients [10]. Simplified and compressed mesh data is transmitted to a mobile device with preview images [20]. For preview images, pre-defined viewpoints are stored with the related view matrix in the server for image-based navigation. Progressive mesh streaming techniques [4, 23, 25] can also be considered for local rendering if the client is a mobile device. These streaming methods send codes to split a vertex for refinement in a pre-computed order to the client [13]. They mainly try to resolve network related problems such as congestion or lost packets. In local rendering, after mobile clients finish receiving all of the data required, users can interactively manipulate the camera view or the model with local 3D viewers. But users cannot control the progressivity or the view during downloading because the order of

streaming was predetermined with a fixed camera view. For large 3D meshes, receiving all the data may be impossible due to insufficient memory (i.e. 14,027,867 vertices and 28,055,742 faces in a model Lucy) or take too much time (i.e., about 2 min for progressive transmission of a model Horse having about 20,000 vertices [13]). It may also be hard to protect the copyright since mobile clients should store all the data locally.

Remote rendering techniques [9, 16, 24] allow mobile devices to receive images or videos from a server. The server stores the mesh data and generates images for mobile clients. Remote rendering also protects the originality and the copyright of the mesh data. There is usually no need for separate viewers for images or videos on mobile clients. In [9], large 3D scenes of a city are rendered in the server and the clients use these images for cube maps as backgrounds of a simulation game. However, remote rendering usually cannot provide dynamic view changes or sophisticated details of 3D meshes in images. To change views, user requests should be transferred to the server, and then the server re-renders the mesh, and generates images which are later transmitted and displayed on mobile clients. Repeating these procedures for large 3D meshes takes a lot of time resulting in slow responses. Network bandwidth and the number of concurrent clients may also be bottlenecks for interactivity in remote rendering.

Mobile systems usually provide touch-based interactions. Various multi-touch interfaces have been studied and presented [7, 19, 20]. Our mobile viewer provides an intuitive interface such as double tap touch for selection or multi-touch for zooming.

3 An overview of our hierarchical mesh streaming framework

3D meshes created by 3D scanning are usually very large. For example, the model Xyzrgb_dragon in Fig. 2 has 3,609,405 vertices and 7,218,906 faces. The model Lucy is composed of 14,027,867 vertices and 28,055,742 faces (See Table 1). For many applications such as games, cultural heritage preservation, and medical imaging, mesh modification may not be permitted to protect the copyright and the originality of the created meshes. With previous methods, it is still difficult to interactively experience the exquisite details of these large 3D meshes on mobile systems which have limited resources. Therefore a new mesh streaming framework is presented in this paper. The goals of our framework are as follows: we wish to

- ① Reduce resource usage on mobile systems.

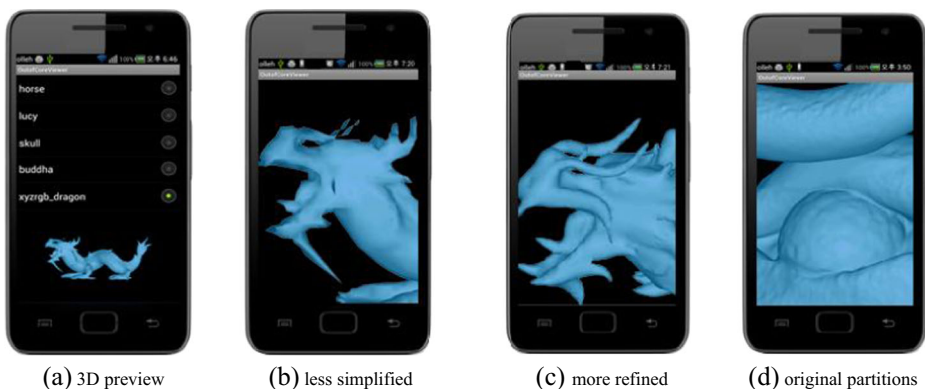


Fig. 2 An example of our hierarchical 3D viewer browsing a large 3D mesh Xyzrgb_dragon

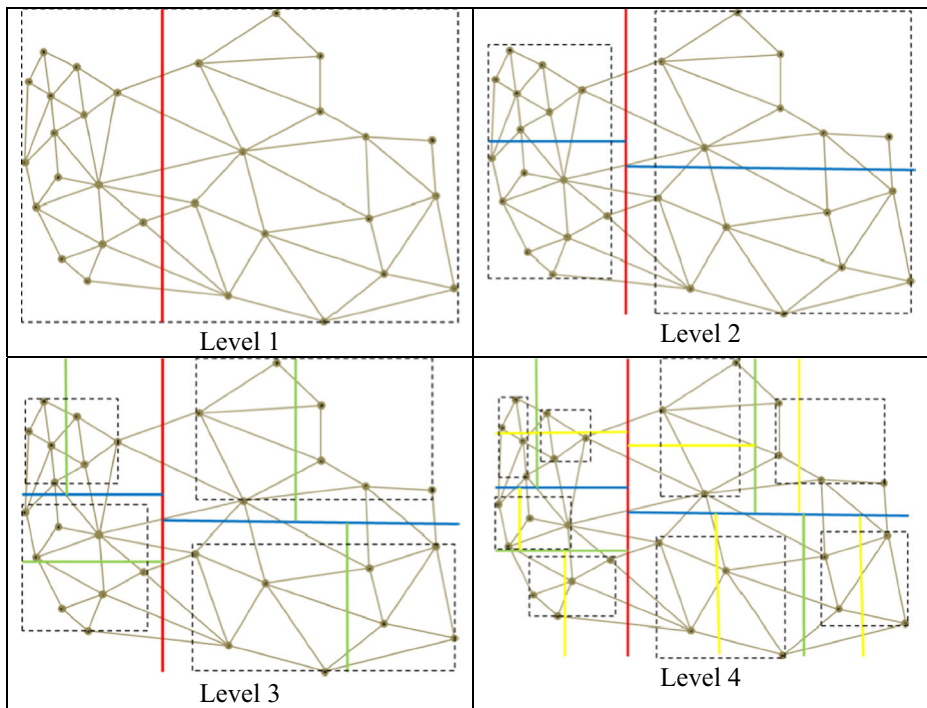


Fig. 3 An example of our k-d tree construction in 2D

- ② Provide mobile users with interactive 3D browsing.
- ③ Allow mobile users to experience the sophisticated originality of large 3D meshes.
- ④ Protect copyright and preserve the original data.

We noted the fact that users cannot see every detail of a large 3D mesh at one time on the small screen of a mobile device. Therefore, our framework is first designed to divide large 3D meshes into a set of equally sized partition meshes in compact shapes and perform simplifications in multi-resolutions in the server. Original 3D mesh data is protected in the server. Only when requested by a mobile user, simplified or partitioned meshes are to be transmitted and rendered locally. Our framework provides a mobile 3D viewer which allows users to interactively control local rendering and random streaming on mobile clients. As a result, when using our framework a mobile user can start fast 3D browsing with 3D preview of a simplified mesh in a small window, as shown in Fig. 2(a). Users can also rotate or translate the mesh to facilitate the display of a mesh with more detail. A mesh can be displayed in detail as shown in Fig. 2(b), (c), and (d). The level of detail in simplified meshes is automatically determined by

Table 1 Large 3D meshes tested in our experiments

| 3D mesh | # of vertices | # of faces |
|---------------|---------------|------------|
| Buddha | 541,336 | 1,087,716 |
| Xyzrgb_dragon | 3,609,405 | 7,218,906 |
| Lucy | 14,027,867 | 28,055,742 |

multi touch inputs. The final level of detail rendered displays a set of partitions for a region of a large mesh, randomly specified by a mobile user. The mesh streaming order is therefore determined by a mobile user.

4 Uniform mesh partitioning

Our mesh streaming framework proposes to divide a large 3D mesh into a set of equally sized small partitions. This uniform partitioning reduces mobile resource usage and also provides users with random accessibility in mesh streaming. Our algorithm uses a vertex clustering since it requires less space for data structures and less computational overhead than face clustering, which has duplicate vertices in faces on the boundary of partitions. Our algorithm constructs a k-d tree for the mesh in which each cell represents a vertex cluster and subsequently forms a single partition of the mesh. Our uniform partitioning algorithm was already presented and showed better performance than previous k-means clustering and octree method [14]. In this paper, out-of-core mesh partitioning is newly presented with mobile interfaces for large 3D meshes.

4.1 Adaptive k-d tree construction for vertex clustering

As introduced in our previous paper [14], our k-d tree divides a cell into two sub-cells each having half the vertices of the original cell. Instead of regular cycling from the x, to the y, to the z-axis, our k-d tree flexibly determines the longest axis of a bounding box as the axis for a perpendicular splitting plane. The five steps for building our k-d tree are as follows:

- Find the bounding box B_M for all the vertices V_M of the mesh M .
- Determine the longest axis A_M among three axes in B_M .
- Find a median MV_M , from V_M sorted by the order of A_M
- Divide V_M with MV_M into two halves for two partitions M_L and M_R .
- Repeat steps a through d for M_L or M_R if the half V_{ML} of M_L or V_{MR} of M_R is bigger than N , the user-specified number of vertices in a partition.

As presented in [14], Fig. 3 shows an example of the above steps from level 1 to level 4 of the k-d tree in a simplified 2D format. The dotted-line is the bounding box of vertices in a cell. The solid dividing lines are perpendicular to the longer of the x-axis or the y-axis of

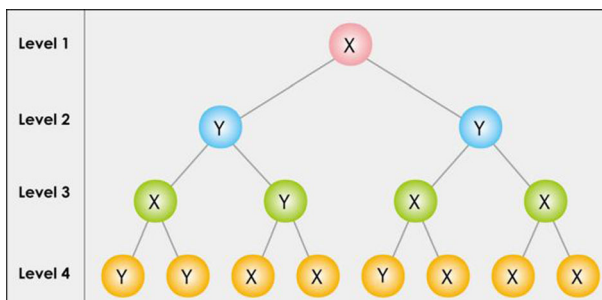


Fig. 4 An example of the adaptive selection of splitting axes in each level of our k-d tree built in Fig. 3

the bounding box. The dividing axes are colored red for 2 cells in level 1, blue for 4 cells in level 2, green for 8 cells in level 3, and yellow for 16 cells in level 4. The axes for cells at the same level may be chosen differently depending on the shape of the bounding boxes as depicted in Fig. 4. Median values are computed to split vertices in half. Finally, 16 uniform partitions of the mesh are created from 16 clusters of vertices in 16 leaf-cells in the k-d tree.

Our k-d tree is designed to create uniform partitions. If there are an odd number of vertices in a mesh, the numerical difference between partitions is one. The octree divides a cell into eight sub-cells including empty cells. As the level of the octree goes higher, the cells are rapidly increased. Therefore, it is hard to control when to stop subdividing cells to generate a user-specified number of partitions. However our k-d tree divides a cell into two sub-cells without empty cells. Moreover, it is relatively easy to control how many partitions are being generated. Our k-d tree selects a splitting plane dynamically for the compactness of the partition shapes. Compactness is a quantity which measures the degree to which a shape is compact. A shape having a smaller surface perimeter is more compact if the areas of all the shapes are the same [2]. Given a partition with area w and perimeter p , we use the compactness c as a ratio of its squared perimeter p^2 to its areas w [12].

$$c = \frac{p^2}{4\pi w} \quad (1)$$

A square figure has better compactness than a long thin rectangular figure. To avoid long thin shaped partitions, our algorithm considers compactness when determining an axis for perpendicular splitting planes to subdivide cells in the k-d tree. The lower the quantity is the better the compactness is. Moreover, a k-d tree is intrinsically a hierarchical structure in which leaf cells provide clusters of vertices. Non-leaf cells can provide clusters of vertices which are grouped from child cell vertices. Also the number of partitions and the number of vertices in a partition can easily be controlled by a user.

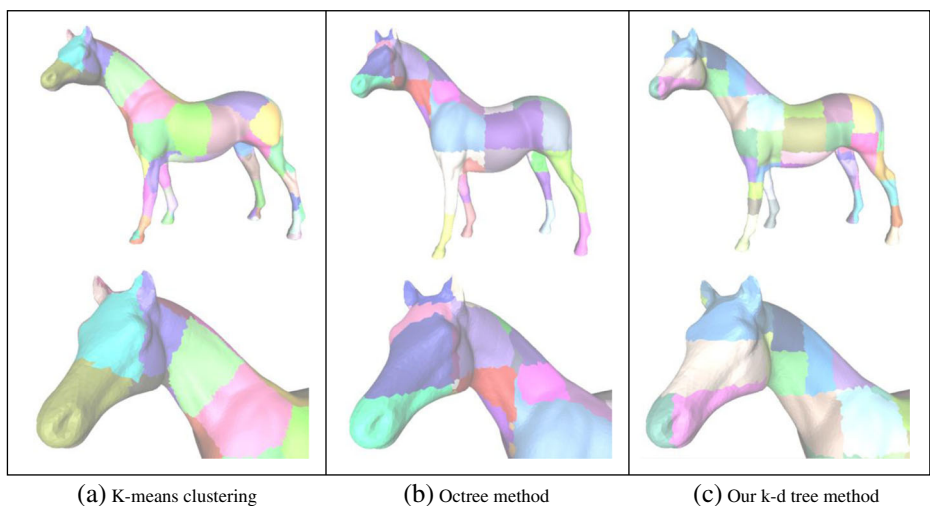


Fig. 5 Examples of mesh partitioning

4.1.1 Experimental results of our k-d tree clustering

A computer with a 2GHz CPU and 3GB RAM was used to test algorithms with relatively small 3D meshes. We implemented two leading methods, octree-based clustering [22] and k-means clustering [5].

Our k-d tree can split cells into two sub cells of different sizes with the same number of vertices. The octree method splits cells into eight sub cells of the same size but with different numbers of vertices, possibly creating empty cells [22]. K-means clustering starts with a user-specified number of random seeds, which is also the number of partitions. The number of vertices in each partition is dynamically determined depending on the positions of seeds and the number of iterations. If two seeds happen to be too close with no faces between them, then no face is added to one of the two seeds resulting in a single face forming a partition. Therefore our version of k-means clustering, adds a hierarchical face clustering [14] as preprocess to spread out the positions of random seeds. This preprocess contributes to the generation of partitions with narrower ranges of variance in the number of vertices, than the octree method [5]. The cost function is designed to consider the compactness based on the geodesic distance [5] and the number of vertices in a cluster. Five repetitions were executed for optimization. K-means clustering requires the most processing time. For each seed, every vertex should be accessed to determine whether it satisfies the cost function. Since k seeds create k partitions, basically n^k times to process vertices are required per iteration, where n is the number of vertices in a mesh. Octree and k-d tree methods process n vertices in each level so $n*d$ times are needed where d is the depth of the tree. The depth of octree is shorter than k-d tree so octree requires less time than k-d tree method. On average, our k-d tree method takes 5.73 s while the octree method takes 5.68 s and k-means clustering takes 9.21 s for 3D meshes tested as listed in Table 2. For the compactness measure using (1), partitions made by the octree are the most compact since cells of the octree are all cubic figures. Our adaptive choice of an axis in the k-d tree construction helps provide better compactness than k-means clustering. In Fig. 5, examples of mesh partitioning are presented. A mesh horse is divided into 64 partitions using k-means clustering in (a), octree method in (b), and our k-d tree method in (c). Each partition in a mesh is colored differently. Shapes and the number of vertices in each partition are irregular with k-means clustering and octree method. Only our k-d tree based clustering creates uniform mesh partitions relatively quickly. It also enables users to control both the number of partitions and the number of vertices in each partition. The visual results of our partitioning are depicted in Fig. 6.

4.2 Out-of-core vertex clustering

Our k-d tree construction requires sorting vertices to find median values. For out-of-core meshes which cannot be loaded into the main memory, external sorting needs to be applied;

Table 2 Processing time and compactness of mesh partitioning algorithms

| Model | Number of vertices | Compactness | | | Processing Time (seconds) | | |
|---------|--------------------|-------------|--------|---------|---------------------------|--------|---------|
| | | k-d tree | octree | k-means | k-d tree | octree | k-means |
| Bunny | 35,947 | 2.068 | 1.856 | 2.358 | 3.18 | 3.15 | 4.10 |
| Feline | 49,864 | 2.938 | 2.299 | 2.440 | 4.43 | 4.38 | 5.85 |
| Igea | 67,180 | 1.787 | 1.816 | 2.435 | 6.16 | 6.12 | 9.27 |
| Skull | 98,306 | 1.857 | 1.761 | 2.331 | 9.12 | 9.08 | 13.14 |
| Average | | 2.163 | 1.933 | 3.049 | 5.73 | 5.68 | 9.21 |

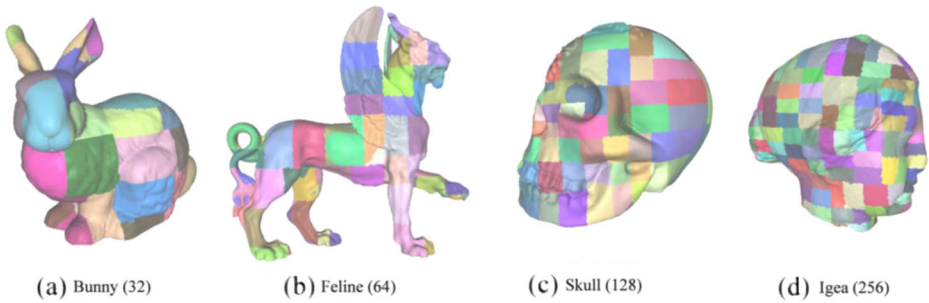


Fig. 6 Uniformly partitioned meshes using our k-d tree clustering with the number of partitions in a parenthesis

however, external sorting takes a lot of time. Therefore, we introduce a new selective sorting to reduce I/O and iterations during external sorting.

4.2.1 Median search for out-of-core data

Supposing that data A of n elements is already sorted, the median value m is defined as follows:

$$m = \frac{\left(A \left[\frac{n}{2} \right] + A \left[\frac{n}{2} + 1 \right] \right)}{2} \quad (2)$$

A median can easily be found from a sorted set of data. For out-of-core meshes, vertex data cannot be maintained in the main memory and must be kept in external files. Therefore, the third and fourth steps c and d in our k-d tree construction of Section 4.1 must be modified as follows:

- c. Find a median of MV_M , from V_M in an external file in the order of A_M .
- d. Divide V_M with MV_M into two halves and save them to two external files for two partitions M_L and M_R .

External sorting generally takes extensive amounts of time. However, there is no need to sort all the data since we only need to divide V_M into M_L and M_R , two files of unsorted vertices before and after a median MV_M . Many algorithms utilized for a median search which do not require sorting have been introduced in [8]. Among them, only Torben's method is capable of handling such large amounts of data. The process can be summarized as follows:

- a. Find the minimum (min) and the maximum (max) of the input data.
- b. Calculate an average of min and max to set a pivot value.
- c. Classify values lower (L) than the pivot value and values greater (G) than the pivot value.
- d. Find the total number of values in L, numLT, and the maximum value in L, maxLT; find the total number of values in G, numGT, and the minimum value in G, minGT.
- e. If numLT is equal to half of the total input data, then the median is the average of maxLT and minGT.
 - else if numLT > numGT, then set max to maxLT and repeat steps b through d.
 - else if numGT > numLT, then set min to minGT and repeat steps b through d.

Torben's method compares each element with the pivot value while counting the number of elements that are bigger and smaller than the pivot. When the number of smaller elements is

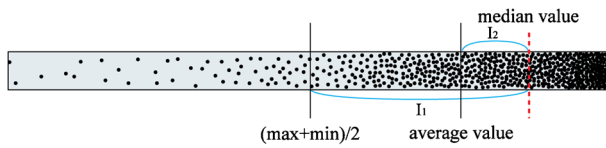


Fig. 7 Our average of all the data is closer to the real median than Torben’s average as $I_2 < I_1$

half the input data, the average of the minimum of the bigger elements and the maximum of the smaller elements is set to the median of the input data. The number of iterations is $O(\log N)$, so it may require extensive time if there are many I/O’s. In this paper, in order to improve the performance of the out-of-core median search, our method alters the setting of the pivot and introduces a minimized sorting technique with the use of heaps.

4.2.2 *New pivot for median search*

The choice of the pivot is critical to the performance of our algorithm. The closer the pivot is set to the real median of the input data, the fewer the number of iterations are required. In our method, the average of all the input data is used for the pivot. With this pivot, the number of iterations is reduced since the average of the data is always closer to the real median than Torben’s average of the minimum and the maximum. For example, when most vertices are skewed to the right maximum, as shown in Fig. 7, our average of the vertices is closer to the real median on the right while Torben’s average is in the middle, away from the real median. Notice that $I_2 < I_1$ in the Fig. 7. When finding the bounding box B_M of the mesh initially, the average of the vertices can also be calculated so that no additional I/O is required.

4.2.3 *Selective sorting with min and max heaps*

For a faster median search, our method introduces new selective sorting through the utilization of main memory. Our data structure is illustrated in Fig. 8a. The left and right files are stored as external files with min and max heaps in the main memory. Our method finds the position of the average by classifying the vertices into two groups, the

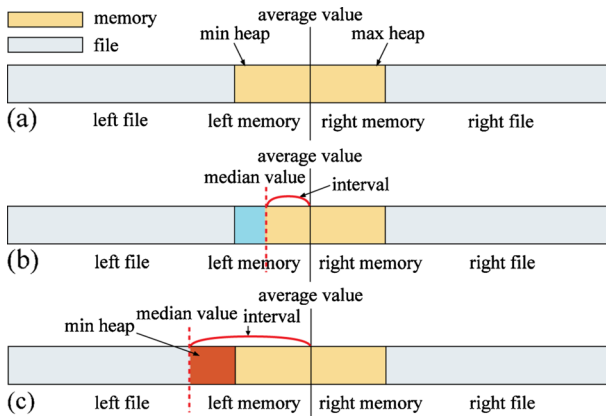


Fig. 8 Data structures for our out-of-core selective sorting

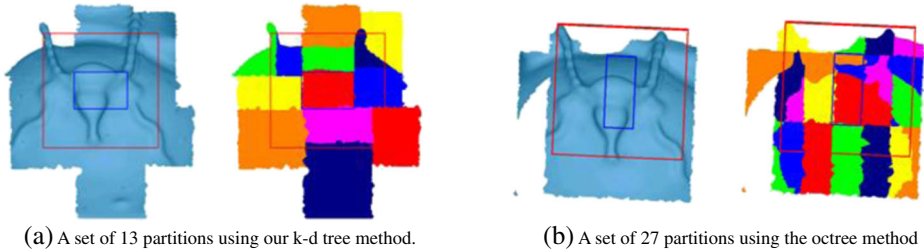


Fig. 9 Captured images of partitions of a mesh Buddha on a mobile client

left, L, if smaller than the average and the right, G, if bigger than the average. Then, only vertices located in the interval, between the positions of the average and the median are selected and subsequently sorted for the purpose of finding the median value. These vertices are stored in the min heap and the max heap and later form a minimized set of vertices to be sorted. As a result, our method reduces the number of I/O operations and iterations.

Our method is described as follows:

1. Classify vertices to find the position of the average.

For a vertex v from the input file,

If $v \leq$ the average,

go to the left.

If $v \leq$ the minimum of the min heap,

save v to the left file.

else if $v >$ the minimum of the min heap,

If the min heap is full, remove the minimum of the min heap and save it to the left file.

Save v to the min heap.

else if $v >$ the average,

go to the right.

If $v \geq$ the maximum of the max heap,

save v to the right file.

else if $v <$ the maximum of the max heap,

If the max heap is full, remove the maximum of the max heap and save it to the right file.

Save v to the max heap.

2. Calculate the interval.

After finishing classifying vertices, the position of the average can be found from the number of elements in the two groups. Then the interval I is calculated as follows with the position of the median from (2):

$$I = \text{position of the median} - \text{position of the average}$$

3. Find the median from a reduced number of vertices.

If $I < 0$, the median is in the left.

else if $I > 0$, the median is in the right.

If $I <$ the size of the heap, the median is instantly found in the sorted min heap or max heap.

If $I >$ the size of the heap, the median is in the left file or the right file.

So for the left file or the right file, repeat steps 1) through 3).

Figure 8(b) illustrates the median in the left min heap in blue. Figure 8(c) shows the median in the left file in red. Generally it is enough to repeat steps 1) through 3) for the left data. When the median is in the left file and the interval is bigger than the memory size, such as the red area in Fig. 8(c), the heap size can be adjusted to the interval. The size of the heaps determines the performance time. The bigger the heap is, the faster our method finds the median. Our algorithm is designed to adjust the memory allocated to heaps. From our experiments, statistically the median is located in the range of -5% to $+5\%$ of the average of all the vertices. Therefore, the median can be found in one cycle without repetition when the size of the heap is set to about 10% of the total number of all vertices. The results of our out-of-core algorithm are presented and also compared to other methods in Table 3.

4.2.4 Performance evaluation of our out-of-core mesh partitioning

The computer with a 2GHz CPU and 3GB RAM was also used to test our out-of-core mesh partitioning. The size of the heaps was set to 256 MB for all tests. We only implemented octree-based clustering [22] to compare the performance of our out-of-core method. We did not implement k-means clustering for out-of-core meshes because this previous method cannot generate uniform partitioning [14], progressivity, and requires too much time up to several hours [5]. Processing times of k-means clustering for out-of-core meshes were simply cited from [5], which were tested on a system with a 2.4 GHz Dual CPU and 2GB RAM. Similar to previous experimental results with relatively small 3D meshes, for out-of-core meshes the time required for our k-d tree method is a little more than the octree method but much less than the k-means clustering. For example, for partitioning a large mesh of Lucy consisting of 14,027,867 vertices, our k-d tree took about 33 min or 2.04 s per partition, while the octree took about 21 min or 1.07 s and k-means clustering required about 6 h or 17.07 s per partition as shown in Table 3. Our selective sorting contributed to reduce processing time which may

Table 3 Processing times for out-of-core mesh partitioning (s for seconds, m for minutes, and h for hours)

| Method | Model # of vertices | Buddha 541,336 | Xyzrgb_dragon 3,609,455 | Lucy 14,027,867 |
|-------------------------|------------------------|-------------------|----------------------------|--------------------|
| K-means clustering [5] | # of partitions | 584 | 576 | 1,195 |
| | Initial chartification | 410 s | 6,900 s | 11,700 s |
| | Chart update | 390 s | 2,220 s | 8,700 s |
| | Total time | 13 m 20 s | 2 h 32 m | 5 h 40 m |
| | Sec./partition | 1.37 s | 15.83 s | 17.07 s |
| Octree clustering | # of partitions | 587 | 571 | 1,196 |
| | Initialization | 65 s | 453 s | 1,187 s |
| | Partitioning | 10s | 15 s | 95 s |
| | Total time | 1 m 15 s | 7 m 48 s | 21 m 22 s |
| | Sec./partition | 0.13 s | 0.82 s | 1.07 s |
| Our k-d tree clustering | # of partitions | 512 | 512 | 1,024 |
| | Initialization | 65 s | 453 s | 1,887 s |
| | Partitioning | 12 s | 42 s | 205 s |
| | Total time | 1 m 17 s | 8 m 15 s | 34 m 52 s |
| | Sec./partition | 0.15 s | 0.97 s | 2.04 s |

take hours for recursive external sorting. Note that the octree clustering cannot produce uniform partitions even if the processing time is faster than our k-d tree method.

Mesh partitioning is executed only once in the server. Frequent events are streaming partitions to satisfy user requests. We examined streaming times for partitions of a mesh Buddha using our k-d tree method and the octree method. In Fig 9a, 13 uniform partitions using our k-d tree clustering are rendered with a total 16,600 vertices on a mobile device. 27 non-uniform partitions using the octree clustering are rendered, with a total 15,295 vertices on the same mobile client in Fig 9b. The blue rectangle is the region a user touched and the red rectangle is the rendering region on a mobile screen. The size of each partition file is charted in Fig. 10. The blue line displays the size of 13 uniform partitions using our k-d tree method. The red line shows the size of 27 non-uniform partitions using the octree method. For face reconstruction, we added vertices on the boundary to both partitions sharing the same boundary so the actual file size of our partition is not exactly the same. Even if the total number of vertices in the set of octree partitions is less than our k-d tree partitions (See Fig. 10), the total streaming time for our 13 uniform partitions took 0.6 s while 0.8 s for 27 octree partitions as charted in Fig. 11. As a result, our k-d tree clustering generates uniform partitions relatively quickly. Our method provides uniform processing times, critical for responsive and scalable mobile interactions.

4.3 Mesh simplification using our k-d tree clustering

In our mesh streaming framework, simplified meshes are to be used as 3D previews so they should reflect the original shape as much as possible. A uniform number of vertices in a partition play a key role in the quality of the mesh simplification. The octree based simplification [22] was implemented along with our kd-tree based simplification for comparison. A single representative vertex was calculated for a cluster of vertices in a partition. Triangulations were performed with representative vertices according to the original connectivity. The simplified mesh by our method preserves the original shape better than the simplified mesh created by the octree method [14]. As illustrated in Fig. 12, a red vertex represents each partition. In our k-d tree simplification, 16 red vertices represent 16 uniform partitions at the top of Fig. 12. Even if 16 cells in octree are the same size, 11 non-uniform partitions are generated with various numbers of vertices. In Fig. 13, a 3D mesh Venus is hierarchically simplified by our kd-tree method and the octree method. Our kd-tree method illustrated at the top of Fig. 13 produces better feature-preserving results. With the smaller number of partitions our method presents better visual qualities than the octree method. In Fig. 14, two simplified faces of a mesh David are presented. The result of our simplification with 2,048 partitions is on the left and the result of the octree method with 2,052 partitions is on the right. In our method, simplified vertices are averages of uniform number of vertices in partitions, while in octree method, simplified vertices are calculated with varied number of vertices in non-uniform partitions. Therefore our simplification generates more regular triangle faces and better visual quality.

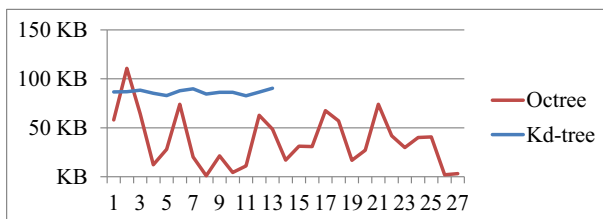


Fig. 10 Comparison of the size in KB of 13 partitions using our k-d tree and 27 partitions using octree

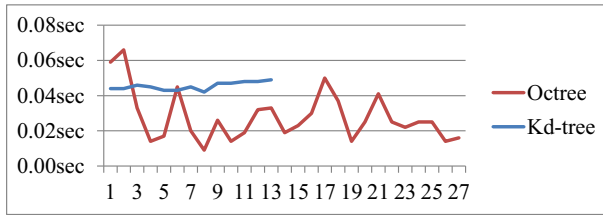


Fig. 11 Comparison of transmission times for 13 partitions using our k-d tree and 27 partitions using octree

5 Mobile 3D viewer with an intuitive user interface

Our mobile 3D visualization has been designed to provide users with the advantages of previous local and remote rendering: interactive controls and original data protection. For data protection, original mesh data is kept in the server; and simplified or partitioned data will be transmitted to mobile clients. For interactive controls, a mobile 3D viewer is presented. Our mobile 3D viewer is hierarchically designed so that users on mobile clients can interactively select levels of detail and specify random streaming. Therefore our mobile 3D viewer eventually minimizes mobile resource usage by avoiding excessive downloading and rendering unwanted parts.

5.1 3D preview for fast 3D searches

Since previews are usually provided for 2D images, fast image searches are possible. However, it is still hard to find 3D previews in mobile viewers. Therefore, our mobile viewer is designed to provide 3D previews as well as filenames for rapid 3D searches. A simplified mesh with a low resolution will be transmitted and rendered instantly as shown in Fig. 15. 3D previews in our experiments required 0.1 to 0.8 s. This has been shown in Tables 4, 5, and 6. A user can interactively control the view by rotating or moving a mesh, or zooming. A fixed window on the bottom of the screen can automatically display previews when a user browses filenames.

| | Original mesh | Space division | Simplified mesh |
|-----------------------------|---------------|----------------|-----------------|
| Our k-d tree Simplification | | | |
| Octree simplification | | | |

Fig. 12 Examples of simplifying meshes in 2D











| | | | | | |
|-----------------------------------|--|--|---|--|--|
| Our k-d tree simplification |  16 partitions, 28 faces |  32 partitions, 62 faces |  64 partitions, 134 faces |  128 partitions, 272 faces |  256 partitions, 554 faces |
| Octree simplification |  19 partitions, 36 faces |  32 partitions, 62 faces |  67 partitions, 142 faces |  130 partitions, 276 faces |  255 partitions, 542 faces |

Fig. 13 Hierarchical simplifications of a mesh Venus

To save mobile resource more, a user can choose a pop-up window to see 3D previews when requested without a constant rendering in a fixed window at the bottom.

5.2 Simplified meshes in full screen

When a user selects a mesh from a list of filenames or a 3D preview, simplified meshes in multi resolutions will be transmitted and rendered locally in full screen on mobile systems as described in Fig. 16. Our interface is designed to be intuitive to enable a user to control levels of detail in simplified meshes through multi touches with the thumb and index finger together. The interval between the thumb and index finger automatically determines zooming in or out and controls the hierarchy or the resolution in simplified meshes. In Fig. 16, the left two images are captured from a mobile device. They display simplified meshes of Lucy and the right two images display simplified meshes of Buddha. We can choose a simplified mesh in a single file, so that the transmission and rendering would take about 0.3 to 2.6 s depending on the resolution as listed in Tables 4, 5, and 6. Once a file is received from the server, the file is saved on a sd memory card on mobile systems, avoiding retransmitting the same file. If the mesh is uploaded, a user can control the view interactively.

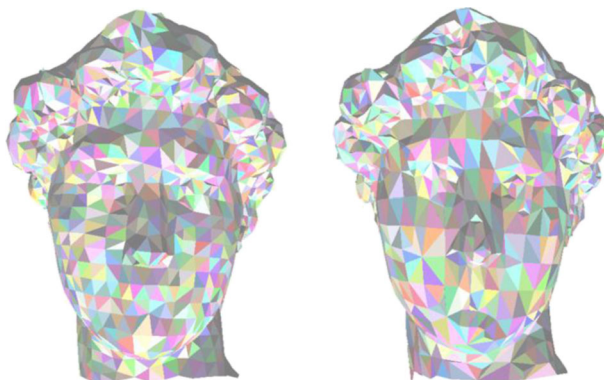


Fig. 14 Simplified faces of David (our method on the left and the octree method on the right)

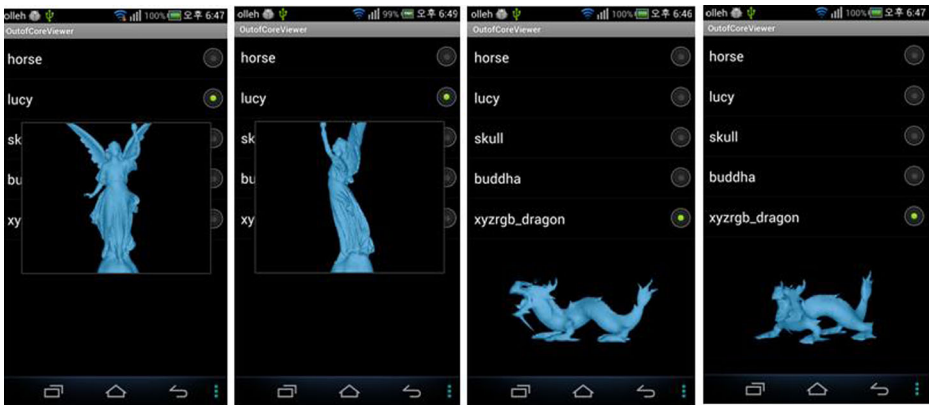


Fig. 15 Examples of 3D previews in our mobile viewer

5.3 Original partitions of large 3D meshes

In previous mesh streaming methods, a user was passive since the streaming order was predetermined. Our framework allows a user to randomly specify streaming order with our mobile viewer. A double tap on a random part of a simplified mesh will trigger the streaming of a set of partitions of a large 3D mesh. The original 3D mesh data is protected in the server and only partitioned files covering the part a user selected are rendered on mobile clients. The k-d tree is kept both in the server and the client for indexing partitions to be streamed. Our mobile viewer requests partitions as needed from the server so resource usage can be reduced by avoiding streaming unnecessary partitions. As a result, a mobile user can see the sophisticated original mesh. In Fig. 17, partitions for the belly of Buddha and the face of Lucy are streamed and rendered. Mobile resource usage and transmission times are determined according to the number of partitions required. The rendering range focuses around the part a user selects and can be determined adaptively according to computing environments. When the rendering range is small, the number of partitions is also small, so resource usage can be reduced but the result may be visually unsatisfying rendering. If the range is large, many partitions requiring more resource usage will be streamed; even so, the results will be visually satisfying.

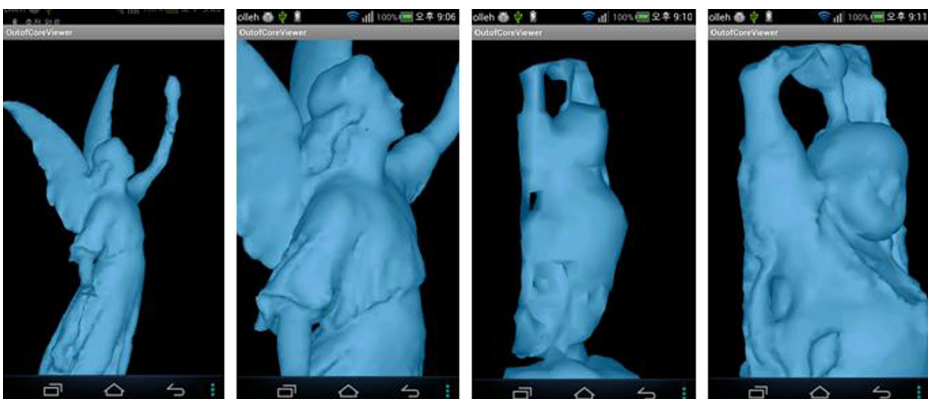


Fig. 16 Examples of simplified meshes in multi resolutions (Lucy on the left and Buddha on the right)

Table 4 Processing times in our mobile 3D viewer for a mesh Buddha

| Buddha | # of vertices | # of faces | Transmission (seconds) | Parsing (seconds) | Rendering (seconds) | Total (seconds) |
|--------------------------------|---------------|------------|------------------------|-------------------|---------------------|-----------------|
| Low-resolution simplification | 690 | 1348 | 0.06 | 0.04 | 0.03 | 0.13 |
| Mid-resolution simplification | 2651 | 5188 | 0.10 | 0.12 | 0.12 | 0.34 |
| High-resolution simplification | 9751 | 19,332 | 0.34 | 0.52 | 0.43 | 1.28 |
| 11 partitions of the original | 26,544 | 45,933 | 1.19 | 0.98 | 1.00 | 3.17 |

5.4 Experimental results of our mobile 3D viewer

This section explains in detail the experimental results of our mobile 3D viewer. We used a Pantech smartphone. The model is IM-A850K. It has Qualcomm Quad Core 1.5GHz (APQ8064). Android 4.04, OpenGL ES 1.0, JNI, and Android-NDK are used to program our mobile 3D viewer. Our network specification is 802.11b/g Wi-Fi.

After large 3D meshes are transformed into a set of small files in the server, a smartphone user can browse large 3D meshes with our mobile viewer as shown in Figs. 1, 2, and 18. The number of partitions of a large 3D mesh can be determined adaptively according to the number of vertices in each partition. When a partition includes many vertices, the size of the partition is larger. When the rendering region is covered by a small number of partitions, the visual quality may be low. For example, we experimented rendering two sets of partitions of a mesh of Lucy. In Fig. 19(a), each partition has about 2,000 vertices. A set of 11 partitions are rendered having about 20,000 vertices in total. In Fig. 19(b), each partition has about 4,000 vertices and 6 partitions are rendered having about 20,000 vertices in total. Even if the total vertices are similar, the size of a partition determines visual quality displayed. The smaller the size of each partition, the easier it is to control the density in the rendering region. The hexagon is the cell a user specified by double tap touches. 11 partitions in Fig. 19(a) can result in more dense visual quality, than the 6 partitions in Fig. 19(b). Therefore, we set 2,000 to be the number of vertices in a partition in our experimental environment.

When a user specifies an arbitrary part of a mesh to see in more detail, the cell of the k-d tree is identified and sent to the server so that the server can identify the same cell. The region of 1-ring neighbors is an extended region. It is extended by adding the averaging length of the x, y, and z-axis of the cell to the region of the cell. In Fig. 9(a) the blue region is the selected cell and the red region is the extended 1-ring neighboring region. A set of partitions belonging to the extended region will be streamed and rendered on mobile clients. For better visual quality, as depicted in Fig. 17, partitions in 2-ring or 3-ring neighbors can be selected even if they require more time and resources. The results of our 1-ring region are displayed in Figs. 1, 2, and 18.

Table 5 Processing times in our mobile 3D viewer for a mesh Xyzrgb_dragon

| Xyzrgb_dragon | # of vertices | # of faces | Transmission (seconds) | Parsing (seconds) | Rendering (seconds) | Total (seconds) |
|--------------------------------|---------------|------------|------------------------|-------------------|---------------------|-----------------|
| Low-resolution simplification | 690 | 1348 | 0.05 | 0.04 | 0.03 | 0.12 |
| Mid-resolution simplification | 2455 | 4778 | 0.10 | 0.11 | 0.11 | 0.32 |
| High-resolution simplification | 9168 | 18,176 | 0.36 | 0.52 | 0.41 | 1.29 |
| 20 partitions of the original | 42,816 | 70,784 | 2.05 | 1.57 | 0.70 | 4.33 |

Table 6 Processing times in our mobile 3D viewer for a mesh Lucy

| Buddha | # of vertices | # of faces | Transmission (seconds) | Parsing (seconds) | Rendering (seconds) | Total (seconds) |
|--------------------------------|---------------|------------|------------------------|-------------------|---------------------|-----------------|
| Low-resolution simplification | 5161 | 11,856 | 0.21 | 0.31 | 0.24 | 0.77 |
| Mid-resolution simplification | 9770 | 19,386 | 0.32 | 0.54 | 0.43 | 1.29 |
| High-resolution simplification | 18,625 | 36,984 | 0.72 | 1.06 | 0.84 | 2.62 |
| 9 partitions of the original | 18,166 | 30,824 | 0.97 | 0.69 | 0.77 | 2.43 |

Tested meshes and processing times are listed in Tables 1, 4, 5 and 6. Transmission times are variable according to network environments. These variances include network bandwidth and the number of concurrent users. The averaged time of 10 trials is listed in Tables 4, 5, and 6. A WRL file format is used for a mesh which is composed of vertex and face data. Normal vectors and texture data are excluded in our experiments to reduce the size of files. Parsing is a process to read vertex and face data from a file, build a mesh, and calculate normal vectors for every face. Rendering is a process which allows the display of a mesh on the screen. We tried to avoid high resolution simplification and use each simplified file as a small single file so that processing times for simplified meshes were 0.85 s on average for fast responses to user requests. For streaming partitions, processing time is dependent on the number of partitions. A single partition having about 2,000 vertices needs 0.26 s on average to process. The number of partitions in 1-ring neighbors is dependent on the complexity of the original mesh. A part of Lucy is less complex than a part of Xyzrgb_dragon: 20 partitions in 1-ring of Xyzrgb_dragon are equal to 9 partitions in 1-ring of Lucy. Our current coding is not an optimized version, so processing times for parsing and rendering can be reduced further if optimized with the GPU later.

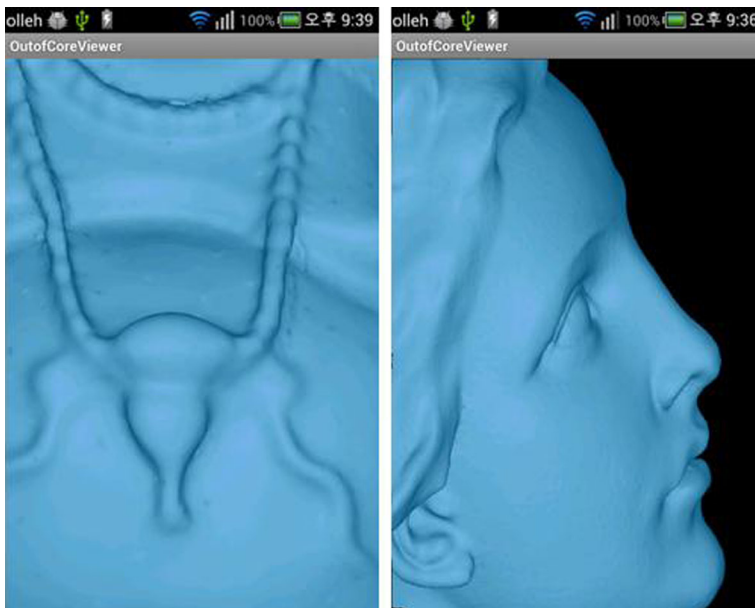
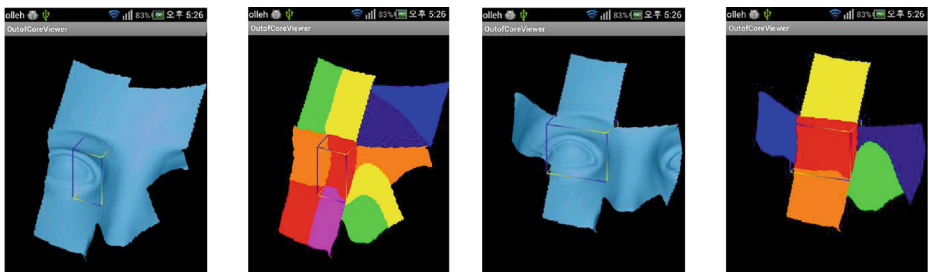
**Fig. 17** Examples of streaming sets of partitions of large 3D meshes, Buddha on the left and Lucy on the right



Fig. 18 An example of a hierarchical browsing with a mesh Lucy in our mobile 3D viewer

6 Conclusion and future work

Mobile systems have relatively limited resources; such as low memory, low power, small screens, or slow CPU. Large 3D meshes require a lot of resources. This paper introduces a hierarchical framework to stream large 3D meshes into a mobile system. In our framework, a 3D mesh is divided into uniform partitions which have an equal number of vertices in compact shapes, using k-d tree clustering. For an out-of-core mesh, min-heap and max-heap in main memory are newly created and select a minimized set of vertices for faster external sorting. Our uniform mesh partitioning also results in better simplification than the previous octree method, since representative vertices for clusters of vertices in uniform partitions preserve the original shape better than those in non-uniform partitions. Our mobile 3D viewer has been designed with an intuitive interface. Our viewer provides 3D preview on mobile clients facilitating rapid 3D searches. Automatic transitions in zooming using multi touches also enable a user to control the level of detail in meshes rendered. Double tap touches on a random part of a mesh allow a user to experience the sophisticated large 3D mesh on a mobile client by triggering partition mesh streaming. As a result, our framework enables a user to browse large 3D meshes interactively on mobile devices while optimizing resource usage and simultaneously protecting the original data. Our future research will include a user study on our mobile 3D viewer. Our research will also focus on the study of how to optimize our programming with the GPU and extend our framework to include more mesh data such as texture information.



(a) 11 partitions with a partition having 2,000 vertices

(b) 6 partitions with a partition having 4,000 vertices

Fig. 19 Comparisons of rendering sets of partitions, having about 20,000 vertices in total

Moreover, texture mapping for simplified meshes could also be studied since we are interested in creating systems which render better visual quality.

Acknowledgments This work was supported by the 2013 Hongik University Research Fund. The authors would like to thank Prof. Marc Levoy at Stanford University for his out-of-core meshes; Prof. Seungyong Lee at Postech and Junho Kim at Kookmin University for their simplified meshes; Dr. Sungyul Choe at Samsung Electronics and Dr. Daeyoung Kim at 3D Systems for their assistance implementing k-d tree clustering.

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

1. Alliez P, Gotsman C (2005) Recent advances in compression of 3D meshes. In: Dodgson N A, Floater M S, Sabin M A (Ed) *Advances in Multiresolution for Geometric Modelling*, Springer-Verlag, pp. 3–26
2. Bribiesca E (2008) An easy measure of compactness for 2D and 3D shapes. *Pattern Recogn* 41(2):543–554
3. Chen X, Golovinskiy A, Funkhouser T (2009) A benchmark for 3D mesh segmentation. *ACM Trans Graph* 28(3):73:1–73:12
4. Cheng W (2008) Streaming of 3D progressive meshes. In: *Proceedings of the 16th ACM international conference on Multimedia*, pp. 1047–1050
5. Choe S, Kim J, Lee H, Lee S (2009) Random accessible mesh compression using mesh chartification. *IEEE Trans Vis Comput Graph* 15(1):160–173
6. Cignoni P, Montani C, Rocchini C, Scopigno R (2003) External memory management and simplification of huge meshes. *IEEE Trans Vis Comput Graph* 9(4):525–537
7. Daiber F, Araujo B R D, Steinicke F, Stuerzlinger W (2013) Interactive surfaces for interaction with stereoscopic 3d (ISIS3D): tutorial and workshop at its 2013. In: *Proceedings of the 2013 ACM international conference on Interactive tabletops and surfaces (ITS'13)*, pp. 483–486.
8. Devillard N (1998) Fast median search: an Ansi C implementation. <http://ndevilla.free.fr/median/>
9. Doellner J, Hagedorn B, Klimke J (2012) Server-based rendering of large 3D scenes for mobile devices using g-buffer cube maps. In *Proceedings of the 17th International Conference on 3D Web Technology*, pp. 97–100
10. Doran A, Mondet S, Grigoras R, Morin G, Ooi W, Boudon F (2009) A demonstration of MobiTree: progressive 3D tree models streaming on mobile clients. In: *Proceeding of ACM international conference on Multimedia*, pp. 955–956
11. Duguët F, Drettakis G (2004) Flexible point-based rendering on mobile devices. *IEEE Comput Graph Appl* 24(4):57–63
12. Garland M, Willmott A, Heckbert P (2001) Hierarchical face clustering on polygonal surfaces. In: *Proceeding of ACM Symposium on Interactive 3D Graphics*, pp. 49–58
13. Hoppe H (1996) Progressive meshes. In: *Proceedings of SIGGRAPH'96*, pp. 99–108
14. Kim D, Lee H (2012) Interactive Rendering of Huge 3D Meshes in Cloud Computing. In: *Proceedings of ICAS 2012 The Eighth International Conference on Autonomic and Autonomous Systems*, 8(2):38–41
15. Kim D, Lee S, Lee H, Cho S (2008) A distance-based compression of 3D meshes for mobile devices. *IEEE Trans Consum Electron* 54(3):1398–1405
16. Lamberti F, Sanna A (2007) A streaming-based solution for remote visualization of 3D graphics on mobile devices. *IEEE Trans Vis Comput Graph* 13(2):247–260
17. Lee H, Desbrun M, Schröder P (2003) Progressive encoding of complex isosurfaces. In: *Proceeding of SIGGRAPH'03*, pp. 471–476
18. Lloyd S (1982) Least square quantization in PCM. *IEEE Trans Inf Theory* 28(2):129–137
19. Rekić Y, Vatavu R-D, Grisoni L (2014) Match-up & conquer: a two-step technique for recognizing unconstrained bimanual and multi-finger touch input. In: *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces (AVI'14)*, pp. 201–208.
20. Rodríguez M B, Agus M, Marton F, Gobbetti E (2014) HuMoRS: huge models mobile rendering system. In: *Proceedings of the Nineteenth International ACM Conference on 3D Web Technologies (Web3D'14)*, pp. 7–15
21. Sander PV, Wood ZJ, Gortler SJ, Snyder J, Hoppe H (2003) Multi-chart geometry images. *Proc Eurographics 2003*:146–155
22. Schaefer S, Warren J (2003) Adaptive vertex clustering using octrees. In: *SIAM Geometric Design and Computing*

23. Tang Z, Guo X, Prabhakaran B (2011) Receiver-based loss tolerance method for 3D progressive streaming. *Multimed Tools Appl* 51:779–799
24. Tang Z, Ozbek O, Guo X (2011) Real-time 3D interaction with deformable model on mobile devices. In: *Proceedings of 19th ACM international conference on Multimedia*, pp. 1009–1012
25. Zhao S, Ooi WT, Carlier A, Morin G, Charvillat V (2014) Bandwidth adaptation for 3D mesh preview streaming. *ACM Trans Multimed Comput Commun Appl* 10(1s):13:1–13:20



Jiro Park received his B.S. degree in Game Software at Hongik University School of Games in 2009 and his M.S. degree in Computer Engineering at Hongik University, Seoul, Korea in 2013. He is working currently at Wemade Entertainment as a mobile 3D programmer. His research interests include mobile 3D games and 3D mesh processing.



Haeyoung Lee received her B.S. degree in Mathematics at Yonsei University, Seoul, Korea in 1986. She received her M.S. degree in Computer Science at San Jose State University, San Jose, California, in 1992. She has worked at Korea Telecom R&D Group as a researcher for about 7 years from 1993 to 1999. She received her Ph.D. in Computer Science at University of Southern California, Los Angeles, California and became a faculty member in the Department of Computer Engineering at Hongik University, Seoul, Korea in 2003. She is currently an associate professor. Her research interests include 3D mesh processing, mobile 3D applications, and mobile multimedia.