View-dependent Streaming of Progressive Meshes

Junho Kim[1,2] Seungyong Lee[1] Leif Kobbelt[2]

[1]Dept. of Computer Science and Engineering, POSTECH, Korea
[2]Computer Graphics Group, RWTH Aachen, Germany

Abstract

Multiresolution geometry streaming has been well studied in recent years. The client can progressively visualize a triangle mesh from the coarsest resolution to the finest one while a server successively transmits detail information. However, the streaming order of the detail data usually depends only on the geometric importance, *since basically a mesh simplification process is performed backwards in the streaming. Consequently, the resolution of the model changes globally during streaming even if the client does not want to download detail information for the invisible parts from a given view point.*

In this paper, we introduce a novel framework for view-dependent streaming of multiresolution meshes. The transmission order of the detail data can be adjusted dynamically according to the visual importance *with respect to the client's current view point. By adapting the truly selective refinement scheme for progressive meshes, our framework provides efficient view-dependent streaming that minimizes memory cost and network communication overhead. Furthermore, we reduce the per-client session data on the server side by using a special data structure for encoding which vertices have already been transmitted to each client. Experimental results indicate that our framework is efficient enough for a broadcast scenario where one server streams geometry data to multiple clients with different view points.*

1. Introduction

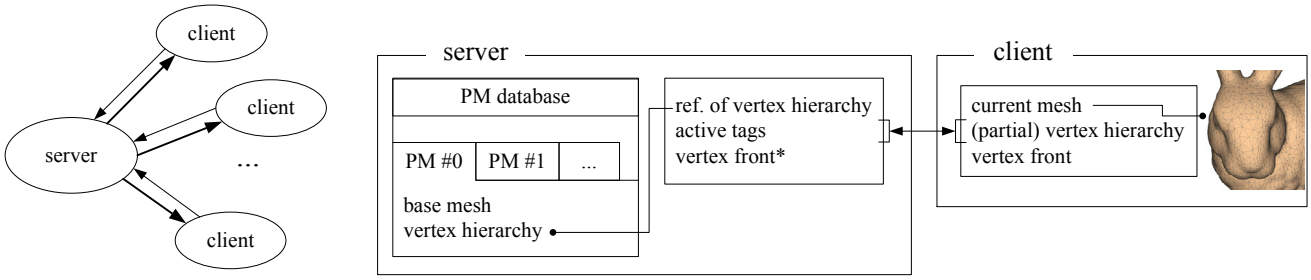
With the ever increasing complexity of polygon mesh models, the need for hierarchical and adaptive techniques becomes more and more obvious. In the graphics literature, much research has been done on mesh decimation techniques, which can effectively reduce the complexity of a given mesh while taking some prescribed error tolerances into account [7, 2, 16]. If we store the sequence of elementary decimation steps, we can later reverse the sequence and perform refinement steps in order to reconstruct the original mesh.

This observation has motivated the term *progressive mesh* (PM) [8] which refers to a mesh data set that is represented by a coarse base mesh plus a sequence of details that can eventually reconstruct the original high resolution mesh. This representation turns out to be particularly useful in the context of a distributed server-client network where a polygon mesh has to be transmitted over a data channel with a limited bandwidth. If we transmit the coarse base mesh first, followed by the detail data, the client can display a low quality version of the object right away and then progressively improve the quality as more and more refinement steps are received.

Unfortunately, however, the standard PM representation only provides *view-independent* streaming of an object. The transmitted details globally change the model on the client side although the client does not need to download data that do not contribute to its screen-space image quality at that time. This limitation is due to the fact that the streaming order of detail data usually depends only on the *geometric importance* since the multiresolution representation is produced by a simplification process that does not consider any information about viewing directions. In the case of transmitting a very large-scale mesh, it would be more effective to transmit the detail data *view-dependently*, based on its current *visual importance* on the client side.

View-dependent refinement of PMs provides the functionality of selecting and adaptively applying detail data with respect to the visual importance. For each vertex in the current mesh, a view-dependent refinement criterion is evaluated to indicate the parts of the mesh which should be locally refined or simplified in order to obtain a certain visual quality. When the regions affected by two different decimation or refinement steps overlap, the partial ordering is handled by a selective refinement scheme.

The contribution of this paper is to combine the two concepts of progressive transmission and view-dependent refinement. The idea is to store a mesh representation for view-dependent refinement on the server side and progressively transmit detail information according to the visual importance with respect to the current viewing parameters on the client side. The major problem to be solved for this is to find a mesh representation that allows the client to re-



(a) network model (b) data structures on the server and a client

Figure 1. Overview of the proposed view-dependent streaming framework: We assume that the bandwidth of the down-link is wider than that of the up-link as depicted in (a). The vertex front* on the server side can be eliminated by windowing active tags, as discussed in Section 6.1.

construct a proper mesh even if the server sends the detail information in random order. This is similar to the view-dependent refinement setting with the important difference that the client does not have access to the complete vertex hierarchy. We propose a solution to this problem that is based on the truly selective refinement approach [13].

To reduce the communication overhead between the server and the client, we design the system architecture in a way that minimizes the redundancy in the up-link as well as the down-link communication. Also, the amount of information maintained in the server is minimized in order to make the framework applicable to a scenario where a single server communicates with a number of clients.

Much research has been devoted to the view-dependent streaming of geometry [21, 1, 19, 20, 24]. However, only a few techniques can provide the original mesh connectivity after downloading the entire PM with the view-dependent streaming process. Similar to [21, 20, 24], this paper concentrates on view-dependent mesh streaming without any loss of mesh connectivity. However, our framework provides a much better performance than those previous techniques in terms of the transmitted data size and the stored data size at the server.

The contributions of this paper can be summarized as follows.

- Our framework provides the original mesh connectivity on the client at the end of the streaming process while only the base mesh and the vertex hierarchy are stored at the server. In contrast to [21, 20], no additional data structures are needed on the server side to provide the original mesh connectivity.
- Since our framework is based on the truly selective refinement scheme of a progressive mesh [13, 14], we can guarantee that the minimal amount of streaming data is transmitted from the server to the client for a given viewpoint.
- We propose a special data structure that compactly represents the vertex front [9] for selective refinement of a progressive mesh. With this data structure, the memory overhead at the server can be minimized.

2. Related Work

Streaming of multiresolution geometry

Streaming of multiresolution geometry is closely related with multiresolution geometry representation and its compression. Due to its intrinsic property, any type of multiresolution representation can be naturally extended to a view-independent geometry streaming framework. Basically, the progressive loading of a multiresolution model is already a streaming process when we consider the external memory as a server and the main memory as a client. Moreover, from the streaming point of view, we can reduce the required network bandwidth between a server and a client with a compressed multiresolution representation.

Hoppe introduced the progressive mesh (PM) representation that consists of a base mesh and a sequence of detail data, which indicates how to rollback to the original mesh data [8]. The resolution of the model is changed by adding details with vertex split ($vsplit$) transformations or subtracting details with edge collapse ($ecol$) transformations. With PM representation, multiresolution streaming of an irregular mesh can be easily achieved by transmitting the base mesh and the details sequentially to the client side. Moreover, the size of the transmitted details can be dramatically reduced when we use a compressed progressive mesh (CPM) representation introduced by Pajarola and Rossignac [18].

Labsik *et al*. [15] proposed a progressive transmission method for subdivision surfaces. Khodakovsky *et al*. [12] presented a compression technique for semi-regular meshes.

View-dependent streaming of multiresolution geometry

Rusinkiewicz and Levoy proposed a view-dependent streaming based on QSplat [19]. They provide a network based visualization for very dense polygon meshes but the splatting approach is not suitable when the client requires the mesh connectivity. Bischoff and Kobbelt introduced an error resilient streaming approach [1]. They define the normal forms of meshes and progressively construct a mesh by Delaunay triangulation with several topological operations. Therefore, a small number of errors during communication does not affect the global shape of the reconstructed mesh on the client side. However, a loss of mesh connectivity can occur since the technique ignores the original mesh connectivity.

Yang *et al*. [24] introduced a patch-based view-dependent streaming technique. They divide a mesh into several patches and compress each patch offline. In the streaming of a mesh, the entire connectivity information of the mesh is first transmitted to the client and then the compressed patches are selected and streamed with respect to the client viewing information. With the approach, the resolution of the mesh cannot be changed smoothly on the client side.

To *et al*. [21] presented a view-dependent streaming based on the view-dependent refinement method proposed by Xia *et al*. [23]. Since the selective refinement scheme of [23] has an 1-ring neighborhood precondition, a fan of 1-ring faces is reserved for each node of the vertex hierarchy on the server side. Southern *et al*. [20] introduced a view-dependent streaming based on Hoppe's view-dependent PM refinement framework [9]. In order to check the precondition of [9], Directed Acyclic Graphs (DAG) are used, each of which is combined with a node in the vertex hierarchy on the server side.

The previous view-dependent streaming frameworks inherit the fundamental limitations of the view-dependent refinement schemes they adopt. Since the preconditions of selective refinement schemes used in [21, 20] invoke complicated dependencies among vertex split transformations, they should reserve an additional data structure such as a 1-ring triangle fan or a DAG graph for each node in the vertex hierarchy on the server side. In contrast, our framework only needs a base mesh and a vertex hierarchy on the server side for view-dependent streaming of a PM.

3. Overview

One of the possible applications of our view-dependent streaming framework would be a flight simulation game with multiple participants. The service company may want to attract customers by providing several new map data occasionally. However, with the typical downloading process, a customer cannot play the game until the whole terrain data have been received. In contrast, our view-dependent streaming technique allows any participant to join and play the game immediately without waiting for the download of the whole terrain data, even when several terrain maps have been newly created.

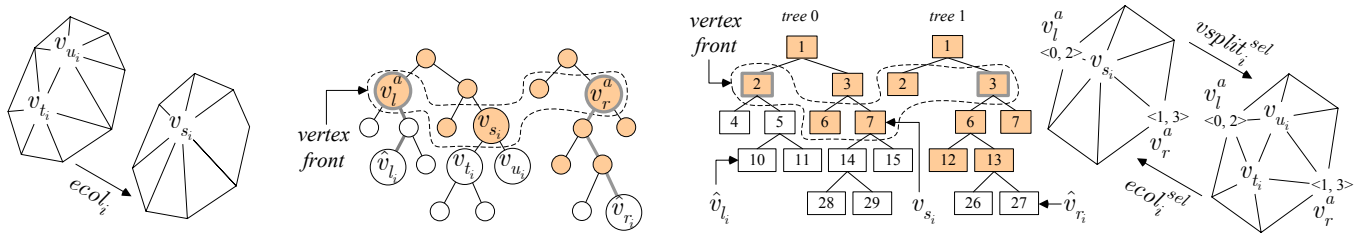
3.1. Setting

The network model of our view-dependent streaming framework consists of a server and multiple clients (see Figure 1). The server has a database of several different meshes represented in the form of a view-dependent PM (i.e., a base mesh and a vertex hierarchy), and we assume that the server is powerful enough to deal with requests from multiple clients. Each client downloads a mesh from the server view-dependently and visualizes the downloaded mesh with different view points not only during the streaming process but also after downloading the entire PM data. We assume that the network bandwidth of a down-link from the server to a client is much wider than that of an up-link in the reverse direction (see Figure 1(a)).

3.2. Our Approach

Let a given triangle mesh model M be stored in a view-dependent progressive mesh data structure on the server side. To provide maximum flexibility in terms of the order in which the vertex split operations can be performed, we choose the particular data structure underlying the truly selective refinement approach [13].

At any point in time the client has received a sub-set M' of the original finest resolution mesh M. Initially M' will be just the coarsest base mesh. With a change of the viewing parameters, the client generates a view-dependently refined mesh $\widetilde{M}$ from M'. If some vertices of $\widetilde{M}$ do not need to be displayed to satisfy the error tolerance requirements, the client can remove them from the set of active vertices without contacting the server. Moreover if these vertices have to be re-activated later, the client can include them again into the list of active vertices without contacting the server. Only when the view-dependent refinement criteria requires the inclusion of new vertices that have not yet been transmitted, the client sends a request to the server by transmitting its current viewing parameters.



(a) (b) (c)

Figure 2. Truly selective refinement scheme: A PM is constructed with several edge collapse transformations shown in (a). Active cut vertices can be located with a climbing-up approach in (b) or a 1-ring test approach in (c). The gray lines in (b) indicates the climbing-up path. In (c), each vertex is denoted by the $<$*tree-id, node-id*$>$ notation.

The server on the other hand maintains a list of binary flags indicating which vertices it already transmitted to the client in the past (i.e., the vertices of M'). When a client request arrives, the server checks based on the view-dependent refinement criterion which vertices have to be refined and transmits the corresponding vertex split operations to the client. Notice that the server only checks if refinement is necessary and is not concerned about coarsification since this is handled autonomously by the client. This guarantees that each vertex split operation is sent exactly once which minimizes the communication overhead.

By this approach we are able to design a server-client system for view-dependent progressive geometry streaming which has several important features:

- **Minimum redundancy:** Each vertex split operation is sent from the server exactly once and only when the view-dependent refinement criterion first requires it. On the other hand, the client sends requests to the server only when it cannot satisfy the criterion using the information that it received in the past.
- **Maximum efficiency:** At each moment in time the server sends exactly those vertex split operations that add the most to the *visual quality* on the client side. This reduces the bandwidth requirements by a factor of about two if the client displays the complete object from a fixed perspective. This factor decreases if the user changes the viewpoint during downloading, but it even increases if the client's display only shows a part of the object.
- **Minimum server load:** The server only needs to store a comparably small amount of data for each client. Only a list of binary flags for the vertex status (transmitted/not transmitted) is necessary. As we will show in Section 6.1 the server can use a dynamic data structure that grows proportionally to the number of already transmitted vertices rather than requiring an amount of memory proportional to the finest level of detail. This reduction of the per-client costs on the server side makes our approach applicable to multi-client scenarios where a server broadcasts geometry data to a multitude of clients.

4. Basic Idea

We use the truly selective refinement scheme [13] to provide the desired properties for our view-dependent streaming framework. In this section, we briefly review the scheme and give the basic idea of how to handle the view-dependent streaming problem with the scheme.

The key ingredients of the truly selective refinement scheme are the fundamental cut vertices and a special index notation for nodes in the vertex hierarchy. Let v_{s_i} be the vertex introduced by collapsing an edge $e_{t_i u_i}$ in the PM construction (see Figure 2(a)). With the truly selective refinement scheme, the vertex v_{s_i} and the edge $e_{t_i u_i}$ can be adaptively split and collapsed with $vsplit_i^{sel}$ and $ecol_i^{sel}$ transformations, respectively.

$$\begin{aligned} vsplit_i^{sel} &= vsplit(v_{s_i}, v_{t_i}, v_{u_i}, v_l^a, v_r^a) \\ ecol_i^{sel} &= ecol(v_{s_i}, v_{t_i}, v_{u_i}), \end{aligned}$$

where

$$\begin{aligned} v_l^a &= \textit{ActiveAncestor}(\hat{v}_{l_i}) \\ v_r^a &= \textit{ActiveAncestor}(\hat{v}_{r_i}). \end{aligned}$$

The vertices $\hat{v}_{l_i}$ and $\hat{v}_{r_i}$ are the fundamental cut vertices of v_{s_i}, and every fundamental cut vertex corresponds to a leaf node in the vertex hierarchy. The *ActiveAncestor*() procedure returns the active ancestor of a vertex which exists in the current vertex front (see Figure 2(b)).

In [13], it was shown that the *active* ancestors of $\hat{v}_{l_i}$ and $\hat{v}_{r_i}$ are the *valid* cut vertices of v_{s_i}. The proof is based on the dual perspective of a progressive mesh. Since the active ancestors of fundamental cut vertices are always contained in the current mesh, any $vsplit_i^{sel}$ or $ecol_i^{sel}$ transformation can be immediately performed without triggering other transformations. This is why the refinement scheme is called *truly* selective.

Now we extend the scheme to the server-client network streaming problem. Suppose a given triangle mesh M is stored as a view-dependent PM data structure (i.e., a base mesh M^0 and a vertex hierarchy H) at the server. During the streaming of M, the client has received only a subset H' of the original vertex hierarchy H. Assume that the shaded nodes in Figure 2(b) represent the partially constructed vertex hierarchy H' on the client side and the client wants to split a vertex v_{s_i} into an edge $e_{t_i u_i}$ due to the view-dependent refinement criteria. Then, the server sends information of two children of v_{s_i} in H, v_{t_i} and v_{u_i}, as well as the fundamental cut vertices, $\hat{v}_{l_i}$ and $\hat{v}_{r_i}$. On the client side, we should determine the active cut vertices, v_l^a and v_r^a, from the fundamental cut vertices in order to maintain valid mesh connectivity between the edge $e_{t_i u_i}$ and the current 1-ring neighborhood of v_{s_i}. However, it is non-trivial to find the active cut vertices with the *partially* constructed vertex hierarchy on the client side since the fundamental cut vertices may not be present in the current vertex hierarchy H' in most cases (see Figure 2(b)).

We can resolve this problem by using the <*tree-id, node-id*> notation for a node in the vertex hierarchy, proposed in [13]. In the notation, each tree in the vertex hierarchy is assigned a *tree-id* and each node in a tree has a proper *node-id* similar to heap indexing (see Figure 2(c)). Based on the <*tree-id, node-id*> notation, the active cut vertices can be located with the partially constructed vertex hierarchy as follows. Basically, the *ActiveAncestor*() procedure can be implemented by climbing up from a leaf node until we meet an active node in the vertex hierarchy. With the <*tree-id, node-id*> notation, the climbing-up can be replaced by the binary right-shift operation of *node-id*. For example, as shown in Figure 2(c), an active cut vertex v_l^a indexed by <0,2> can be located from <0,10>, the index notation of $\hat{v}_{l_i}$, by right-shifting twice until we meet the vertex front. With this technique, the client can properly update the current view-dependent mesh from the transmitted fundamental cut vertices even though only a partial vertex hierarchy has been constructed. Note that locating the active cut vertices with an *incomplete* vertex hierarchy was not considered in the original scheme [13] and this is a key observation that allows us to extend the scheme [13] to the network streaming domain.

5. View-dependent Streaming

5.1. Overall process

In our view-dependent streaming framework, the overall process of network communication between the sever and a client goes as follows.

- Upon the initial request for a mesh from a client, a session is established between the server and the client (session creation).
- When the session has been created, the server sends the base mesh to the client and waits for the current viewing parameters being sent from the client (base mesh transmission).
- If the client realizes that some additional detail data are required to refine the current mesh view-dependently, the client sends its viewing parameters to the server (view information transmission).
- With the viewing parameters from the client, the server selects proper detail data and sends them to the client. On the client side, the transmitted detail data will be used for improving the screen-space image quality (view-dependent *vsplit* packet transmission).
- If sufficient detail data have been received for the current view, the client updates the current view-dependent mesh using the data (view-dependent refinement).
- The session can be closed upon a request from the client (session close).

The steps of the viewing parameter transmission and view-dependent *vsplit* packet transmission are repeated in the main loop during network communication. The server performs the *vsplit* packet transmission step only when a new view information has arrived from the client. At the client, the view information transmission, receiving the *vsplit* packets from the server, view-dependent refinement are performed in parallel with a multi-threading or time-sharing technique.

5.2. Data structures

Figure 1(b) shows the data structures used in our view-dependent streaming framework. The server contains several view-dependent PMs in a common database, where each view-dependent PM consists of a base mesh and a vertex hierarchy. The database on a server is static and does not change during network communication. The server may establish session connections with multiple clients. For each session, the server stores a reference to a vertex hierarchy of the mesh being transmitted and active vertex tags to represent the current vertex front. An active tag is set to '1'

only when the corresponding node in the vertex hierarchy is contained in the vertex front. The vertex front at the server keeps track of the transmitted vertices to the client, while the vertex front at the client consists of the vertices of the current view-dependently refined mesh. Since we do not resend previously transmitted *vsplit* packets, the vertex front on the server side corresponds to the leaf nodes of the partially constructed vertex hierarchy on the client side. Therefore, the vertex front on the server side is updated only downward in the vertex hierarchy as view-dependent *vsplit* packets are transmitted.

The data structures on the client side for each session is similar to those for the view-dependent refinement; a mesh (e.g., halfedge data structure), a vertex hierarchy, and a vertex front. The vertex hierarchy on the client side is partially constructed from previously transmitted detail data. The vertex front on the client corresponds to vertices in the current mesh. In contrast to the vertex front on the server side, it freely moves within the partial vertex hierarchy on the client during the view-dependent streaming process.

5.3. Detailed steps

Now we describe the detailed steps of network communication between a server and a client. In our experiments, we use TCP sockets for network communication.

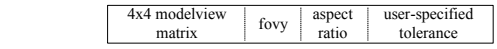
Session creation: A session is established by a connection request from a client to the server. After the session has been created, the client chooses a PM from the catalogue of PM database stored in the server.

Base mesh transmission: After the desired PM has been specified, the server streams to the client the base mesh of the PM followed by information for view-dependent refinement of base mesh vertices, which consists of a radius, a cone of normals, and parameters to calculate the screen-space errors [9]. Next the server sets a reference to the vertex hierarchy of the PM. Then, active tags are created with the number of nodes in the vertex hierarchy. Initially, we set the tags that correspond to the roots of the trees in the vertex hierarchy as 'active', which means the vertex front at the server consists of the root nodes.

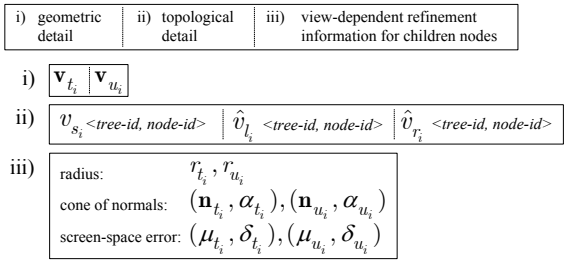
After downloading the entire data related to the base mesh, the client creates a base mesh, a vertex hierarchy, and a vertex front. At that time, the vertex hierarchy at the client is composed only of root nodes. Similarly, the vertex front only contains the roots of the vertex hierarchy.

View information transmission: With the changes of the its viewing parameters, the client may need additional detail data related to the newly visible parts. To obtain the additional details, the client sends the current viewing parameters to the server. The packet for the client's viewing parameters contains a 4×4 modelview matrix, a fovy (field of view) of the view frustum, an aspect ratio of the client screen, and a user-specified tolerance (see Figure 3(a)).

View-dependent *vsplit* packet transmission: With the viewing parameters received from the client, the server determines *vsplit* packets to be transmitted to the client. During this process, we visit each node in the vertex front and perform a query procedure *qrefine*() to test whether the vertex should be split or not with respect to view-dependent refinement criteria. In our framework, we use the criteria proposed by Hoppe [9], which tests the viewing frustum, the cone of normals, and the screen-space error. If the *qrefine*(v_{s_i}) returns true, then the server transmits a *vsplit* packet, which contains the fundamental cut vertices of v_{s_i} and information for view-dependent refinement for two newly created vertices, v_{t_i} and v_{u_i}, as depicted in Figure 3(b). Then, the vertex front stored at the server is updated by deactivating the node v_{s_i} and activate its two children, v_{t_i} and v_{u_i}.



(a) view information packet (client → server)



(b) *vsplit* packet of v_{s_i} (server → client)

Figure 3. Communication packets: view information (a) and *vsplit* packets (b) are transmitted through the up-link and down-link, respectively.

Although the truly selective refinement scheme is used in our framework, the partial ordering of $vsplit_i^{sel}$ must be checked on the server side due to limitations of the $vsplit/ecol$ operators.[1] If the active ancestor v_l^a of $\hat{v}_{l_i}$ is equal to the active ancestor of v_r^a of $\hat{v}_{r_i}$, we enforce transmission of the *vsplit* packet for v_l^a prior to v_{s_i} until v_l^a is different from v_r^a. We can simply determine the partial-ordering among *vsplit* packets without the mesh structure, as summarized in the following pseudocode.

1 $vsplit$ and $ecol$ operators can only deal with transformations between 2-manifold meshes [14, 4, 10].

Algorithm SelectiveStreaming(*vertex hierarchy* of a PM)
for each $v \in$ *vertex front* **do**
  if *qrefine*(v) = true **do**
    SelStreamVSplit(v)
  end
end

Algorithm SelStreamVSplit(v_{s_i})
$v_l^a \leftarrow$ *ActiveAncestor*($\hat{v}_{l_i}$)
$v_r^a \leftarrow$ *ActiveAncestor*($\hat{v}_{r_i}$)
// check the partial ordering
while $v_l^a = v_r^a$ **do**
  SelStreamVSplit(v_l^a)
  $v_l^a \leftarrow$ *ActiveAncestor*($\hat{v}_{l_i}$)
  $v_r^a \leftarrow$ *ActiveAncestor*($\hat{v}_{r_i}$)
end
StreamVSplit(v_{s_i})

In the pseudocode, the *qrefine*() procedure returns false for a real leaf vertex of the complete vertex hierarchy for any viewpoint of the client since there is no screen-space error for the real leaf vertices.

When the client receives a *vsplit* packet, it updates its vertex hierarchy using the data. All newly added nodes are dangled as the leaves of the partially reconstructed vertex hierarchy on the client side. With the <*tree-id, node-id*> notation of v_{s_i}, we first find the corresponding node in the vertex hierarchy and set the fundamental cut vertices of v_{s_i} from the transmitted packet. Then, we create two nodes v_{t_i} and v_{u_i} as the children of v_{s_i} and copy the information for view-dependent refinement of the nodes from the packet. This process is summarized in the following pseudocode.

Algorithm UpdateVHierarchy(*vsplit* packet of v_{s_i})
$v_{s_i} \leftarrow$ GetVHierarchyNode(<*tree-id, node-id*> of v_{s_i})
v_{s_i}.*fund_lcut_index* $\leftarrow$ <*tree-id, node-id*> of $\hat{v}_{l_i}$
v_{s_i}.*fund_rcut_index* $\leftarrow$ <*tree-id, node-id*> of $\hat{v}_{r_i}$
$[v_{t_i}, v_{u_i}] \leftarrow$ MakeChildren(v_{s_i})
FillChildrenInfo(v_{t_i}, v_{u_i}, *vsplit* packet of v_{s_i})

View-dependent refinement During the view-dependent refinement on the client side, the vertex front can be freely moved in the partially constructed vertex hierarchy. Since each vertex at the client has the <*tree-id, node-id*> notation, we can perform view-dependent refinement by the following pseudocode.

Algorithm SelectiveRefinement(*selectively-refined* PM)
for each $v \in$ *vertex front* **do**
  if *qrefine*(v) = true **do**
    if *IsLeaf*(v) = false **do** SelVSplit(v) **end**
    else do StreamViewingInfo() **end**
  end
  else if *IsRoot*(v) = false **and** *IsSiblingActive*(v) = true **and**
    qrefine(v.parent) = false **do**
    SelECol(v.parent)
  end
end

In the pseudocode, *IsLeaf*(v) returns false if a vertex v is not a leaf node in the partially reconstructed vertex hierarchy at the client. SelectiveRefinement() is similar to the typical view-dependent refinement procedure except for the view information streaming part. SelVSplit() and SelECol() procedures selectively splits a vertex into an edge and collapses an edge into a vertex with the truly selective refinement scheme, respectively. Note that the client does not send its view information to the server until the additional detail data are required which are not available in the current vertex hierarchy. Recall that the *qrefine*() procedure for a real leaf vertex of the complete vertex hierarchy always returns false. Hence, the client does not transmit its view information to the server when the current vertex is a real leaf vertex of the complete vertex hierarchy.

Session close: The session can be closed when all nodes in the vertex hierarchy have been transmitted from the server or when the client is satisfied with the current resolution of the mesh. In the case when the session is stopped by a network problem, the session can be continued later from the stopped point. In that case, to re-initialize the the vertex front at the server, the client transmits the <*tree-id, node-id*> notations of the leaf nodes in its partially constructed vertex hierarchy to the server side.

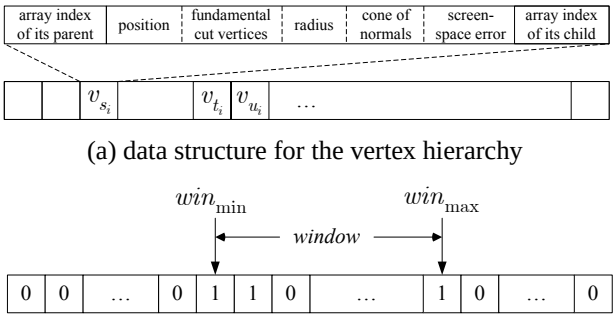
6. Optimization

6.1. Windowing active tags

If we store a vertex front in each session on the server side the most significant size of the data structure on the server may come from the vertex front. The maximum length of a vertex front is equal to the number of vertices in the original mesh. When the server has several sessions with different clients to visualize very large meshes, the memory requirement for vertex fronts can be a serious problem on the server side.

To minimize the memory overhead, we discard the vertex front data structure and just use the active tags to represent which nodes in the vertex hierarchy are contained in a vertex front. The vertex hierarchy is packed into an array while preserving the partial order that parent indices should be smaller than their children's indices (see Figure 4(a)). Then, the active tags for the nodes in the vertex hierarchy can be represented by a bit array (see Figure 4(b)). To determine the *vslpit* packets to be transmitted from the server, we *sequentially* visit the elements in the active tag array. If the current bit is '1', the *qrefine*() procedure is evaluated to determine if the corresponding node should be split into its children. Since the indices of newly active nodes are

greater than the one of the currently visited node, this sequential search would not miss any vertices that should be considered in determining *vslpit* packets.



(a) data structure for the vertex hierarchy

(b) active tag windowing

Figure 4. Windowing active tags for the vertex hierarchy

This procedure can be further accelerated with a windowing technique. The idea of our windowing technique is similar in spirit to [22, 3, 11] which handles very large sized data by concentrating the computation or the storage only on the currently active parts. Since the vertex hierarchy stored in an array is a linear data structure, we can bound a non-zero area with two array indices, $win_{\min}$ and $win_{\max}$, as shown in Figure 4(b). The left part of $win_{\min}$ and the right part of $win_{\max}$ are all zero bits, and the in-between part contains '0' and '1' bits. Note that the window only grows from the left to right direction because the partial order among the nodes in the vertex hierarchy is preserved in the array. While the window is moved from the left to the right by performing $vsplit$ operations, the left and right parts of the window will be deallocated and allocated, respectively. Although there remain several '0' bits within the window, we can reduce the traversing time for a vertex front by confining the search among the indices from $win_{\min}$ to $win_{\max}$.

In our experiments, we pack the vertex hierarchy in a breadth-first manner. As reported in Section 8, the windowing technique is sufficiently fast and uses much less memory than storing a vertex front itself.

6.2. Construction of a balanced vertex hierarchy

To identify a node in the vertex hierarchy, we use the $<$*tree-id, node-id*$>$ notation in the network transmission. Since the number of bits for *node-id* is equal to the height of the vertex hierarchy, we should construct the vertex hierarchy as balanced as possible in order to reduce the number of bits used for representing the node index.

Two approaches exist to construct a balanced vertex hierarchy. One approach is to reflect subtree depths onto the error metric for edge collapses [9]. Another is to perform level-wise edge collapses by choosing maximally independent edge sets [23]. In this paper, we use a hybrid approach. We choose maximally independent edges with the error metric that considers subtree depths in the vertex hierarchy.

For each node index, we give $\lceil log_2(\#v^0) \rceil$ bits for *tree-id* and $depth_{\max}$ bits for *node-id*, where $\#v^0$ and $depth_{\max}$ represents the number of vertices in the base mesh and the maximum depth in the vertex hierarchy, respectively. With the construction of the balanced vertex hierarchy, 32 bits were sufficient to represent the node indices in the vertex hierarchy for any large-scaled meshes used in our experiments (see Table 1).

model	$\#v$ of base mesh	# of details	bits for $<$*tree-id, node-id*$>$
bunny	62	34,772	$<$6, 16$>$
horse	75	19,776	$<$7, 14$>$
feline	12	49,852	$<$4, 28$>$
skull	146	98,160	$<$8, 23$>$
Buddha	65	545,557	$<$7, 25$>$

Table 1. Statistics of PMs with balanced vertex hierarchies: Note that each $<$*tree-id, node-id*$>$ notation can be stored in one 32 bit integer.

7. Discussion

We can consider several variations of the approach proposed in this paper. A straightforward variation would be that the client sends the vertex indices to be split, instead of viewing parameters. This can be seen as more efficient than our framework in that the server does not have to evaluate *qrefine*() procedure on its vertex front. However, in this case, the network traffic on the up-link will increase with the size of the mesh data to be displayed. This approach could be suitable for the peer-to-peer network model, but it would be inefficient for the single server multiple clients network model.

Another possible variation is to change the component of the selective refinement scheme in our framework with one of the other refinement schemes [23, 9, 5, 17], as in the previous work [21, 20]. However, in this case, the streaming data size from the server to the client should be larger than in our technique. The first reason is that the sizes of their topological detail information are larger than the truly selective refinement scheme, as discussed in [13]. The second reason is that the vertex hierarchy on the client side must be synchronized with that on the server side in order to perform

selective refinement with their topological details stored at the client. To accomplish this, the *vsplit* packet must contain some information for synchronization. In contrast, the truly selective refinement scheme adopted in our framework does not require any synchronization information due to the nice properties of fundamental cut vertices and the indexing scheme of the vertex hierarchy.

Some clients may want to download the details of a mesh in a view-dependent way but need not to perform later the view-dependent refinement with the downloaded data. In this case, the server can determine the cut vertices with respect to the client view information and substitute them for the fundamental cut vertices in the *vsplit* packets. With this approach, the *vsplit* packet size is comparable to the standard view-independent transmission since we can eliminate the information for view-dependent refinement (i.e., iii) in Figure 3(b)) from the *vsplit* packets.

8. Experimental Results

To measure the performance of our framework, we experimented with a fixed navigation path where each client downloads the base mesh and then looks around the model from top to bottom twice.

Figure 5 shows the memory requirement of the server in the session where a Buddha model is transmitted to a client. As shown in Figure 5, when the vertex front is stored, the memory usage at the server increases with time since the number of active nodes in the vertex hierarchy at the server are monotonously increasing during transmission. The jump in the graph for the vertex front technique is incurred by the client's request for a higher visual quality just before the client looks around the model for the second time. In contrast, our windowing technique is rarely sensitive to the change of the client's visual tolerance and consumes approximately only 10% of memory required for storing a vertex front. The gap between an ideal case and our implementation is introduced by our memory allocation policy, which doubles the buffer size for windowed active tags when the window size exceeds the current buffer size.

Figure 6 illustrates how much data are transmitted between the server and the client in the case of the bunny model. The accumulated packet size of the up-link is linearly increasing since the client sends a view information packet each time when it has vertices to be split. In contrast, the accumulated packet size of the down-link is irregularly increasing since the number of transmitted *vsplit* packets varies from the view information of the client. Note that after the client has looked around the bunny once, there are no packets transmitted through the up-link as well as the down-link, since the client has already downloaded enough detail data to visualize the model view-dependently.

Figure 7 shows the measurements of the number of faces and required times for rendering and view-dependent refinement when we download the Buddha model with our streaming framework. Several peaks appear when we suddenly change the viewpoint, but the performance variations are small when the viewpoint is smoothly changed.

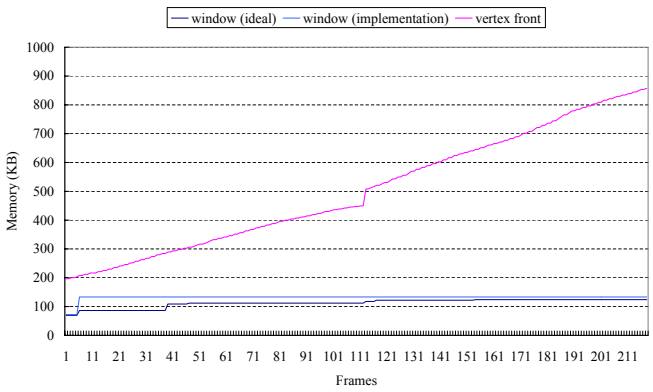


Figure 5. Memory requirement on the server side for a session: Our windowing technique with active tags uses less than 10% of the memory size required in the case when the vertex front data structure is used.

We compared the performance of our framework with the view-independent streaming framework in terms of visual quality and data size. Firstly, we measure how much communication data should be transmitted in a session in order to satisfy a fixed tolerance of the screen-space error on the client side. Secondly, we fix the data size of the session communication and compare the visual quality achieved with the transmitted mesh data.

Figure 8 shows the resulting meshes from our framework and view-independent streaming when the detail data have been transmitted until a preset tolerance of the screen-space error is satisfied. As shown in Figure 8(a), our technique achieves visual tolerance by transmitting smaller sized data to the client than the view-independent streaming technique. The rendered image with the view-independent streaming technique over-satisfies the tolerance in some areas, as shown in Figure 8(b), since detail data should be transmitted in a fixed order.

Figure 9 illustrates the visual qualities of the refined meshes under the restriction of the communication data size. We limit the transmitted data size to 0.5Mbytes and stream the Buddha model with our approach and view-independent streaming. As shown in Figure 9, the screen-

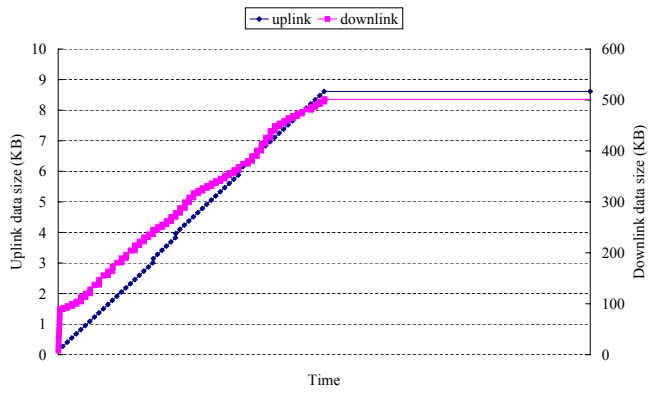


Figure 6. Accumulated communication data size: The client looks around the bunny model twice with the same navigation path and visual tolerance. On the second round, no data are communicated between the server and the client because all necessary data have already been transmitted from the server to the client on the first round.

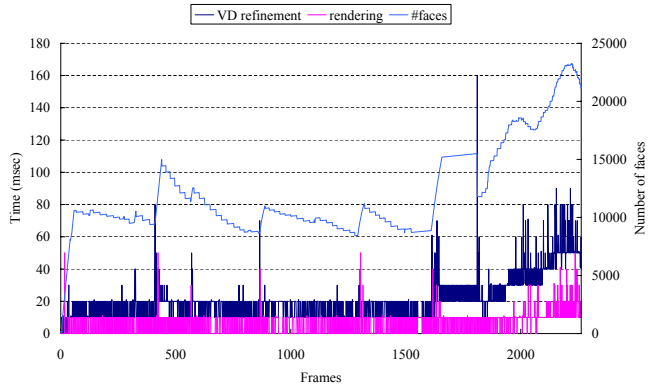


Figure 7. Statistics for streaming the Buddha model: The times required for the view-dependent refinement and the rendering are dominated by the number of faces in the view-dependently refined Buddha model.

space visual quality with view-independent streaming is better than one with view-dependent streaming when the viewpoint is far from the model. This is because we have to transmit additional information, such as radius, cone of normals, and parameters to calculate the screen space error for view-dependent refinement on the client side. However, when the viewpoint is close to the model, our framework provides superior screen-space images even though we transmit the additional information. Furthermore, if later performance of view-dependent refinement is not desired on the client side, as discussed in Section 7, the visual quality of our technique is always better than the view-independent streaming under a fixed transmitted data size. In that case, the packet size for the view-dependent streaming would be the same as the view-independent streaming, while most of the transmitted triangles really help to improve the visual quality.

9. Conclusion and Future Work

In this paper, we presented a novel view-dependent streaming framework for irregular meshes. By adopting the truly selective refinement scheme, we could achieve the minimal size of the transmitted data from the server to the client while satisfying the specified visual quality. Furthermore, the data size stored at the server is optimized using the windowing technique.

Future work includes compression of the *vsplit* packet with quantization and a smart *vsplit* packet transmission strategy for utilizing the session idle time.

Acknowledgement

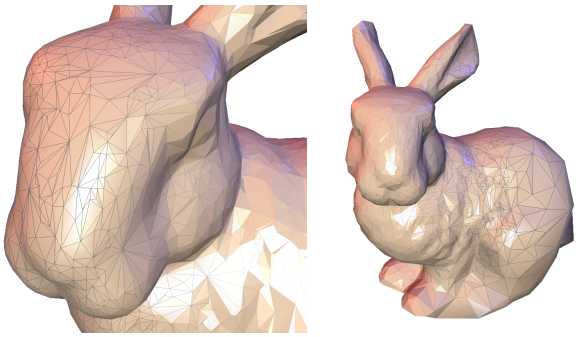
The authors would like to thank Igor Guskov for a 2-manifold 'Happy Budda' mesh model [6] and Christoph Vogel for help on multi-thread programming. The bunny model and the original 'Happy Budda' models are courtesy of the Stanford Computer Graphics Laboratory. This work was supported in part by the Korea Ministry of Education through the Brain Korea 21 program and the Game Animation Research Center.

References

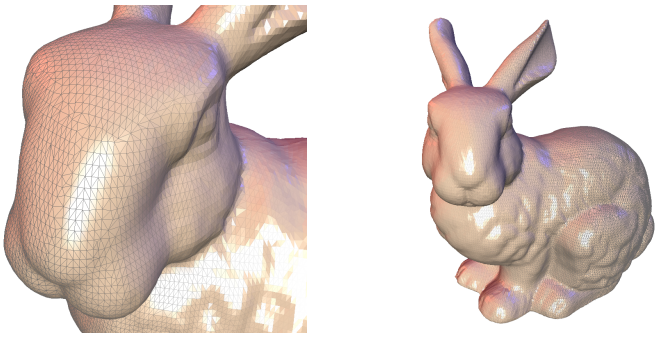
[1] S. Bischoff and L. Kobbelt. Towards robust broadcasting of geometry data. *Computers & Graphics*, 26(5):665–675, 2002.

[2] P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22(1):37–54, 1998.

[3] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. Sequential point trees. *ACM Computer Graphics (Proc. SIGGRAPH 2003)*, pages 657–662, 2003.



177,120 bytes (#faces: 4,534)
(a) our approach (left: screen, right: another view-point)



827,688 bytes (#faces: 68,884)
(b) view-independent streaming (left: screen, right: another view-point)

Figure 8. Comparison of view-dependent and view-independent streaming with a fixed visual quality: (a) Our view-dependent streaming satisfies the visual quality after downloading 177,120 bytes. (b) In contrast, view-independent streaming satisfies the same visual quality after downloading much more data, 827,688 bytes.

[4] T. K. Dey, H. Edelsbrunner, S. Guha, and D. V. Nekhayev. Topology preserving edge contraction. Technical Report rgi-tech-98-018, Raindrop Geomagic, 1998.

[5] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum (Proc. Eurographics'99)*, 18(3):83–94, 1999.

[6] I. Guskov and Z. J. Wood. Topological noise removal. In *Proc. Graphics Interface 2001*, pages 19–26, 2001.

[7] P. S. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. *SIGGRAPH '97 Course Notes # 25*, 1997.

[8] H. Hoppe. Progressive meshes. *ACM Computer Graphics (Proc. SIGGRAPH '96)*, pages 99–108, 1996.

[9] H. Hoppe. View-dependent refinement of progressive meshes. *ACM Computer Graphics (Proc. SIGGRAPH '97)*, pages 189–198, 1997.

[10] H. Hoppe, T. DeRose, T. Dunchamp, J. McDonald, and W. Stuetzle. Mesh optimization. Technical Report TR 93-01-01, University of Washington, 1993.

[11] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. In *Proc. IEEE Visualization 2003*, pages 465–472, 2003.

[12] A. Khodakovsky, P. Schröder, and W. Sweldens. Progressive geometry compression. *ACM Computer Graphics (Proc. SIGGRAPH 2000)*, pages 271–278, 2000.

[13] J. Kim and S. Lee. Truly selective refinement of progressive meshes. In *Proc. Graphics Interface 2001*, pages 101–110, 2001.

[14] J. Kim and S. Lee. Transitive mesh space of a progressive mesh. *IEEE Trans. Visualization and Computer Graphics*, 9(4):463–480, 2003.

[15] U. Labsik, L. Kobbelt, R. Schneider, and H.-P. Seidel. Progressive transmission of subdivision surfaces. *Computational Geometry Journal: Theory and Applications*, 15(1-3):25–39, 2000.

[16] D. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, 21(3):24–35, 2001.

[17] R. Pajarola. Fastmesh: Efficient view-depenent meshing. In *Proc. Pacific Graphics 2001*, pages 22–30. IEEE Computer Society Press, 2001.

[18] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Trans. Visualization and Computer Graphics*, 6(1):79–93, 2000.

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

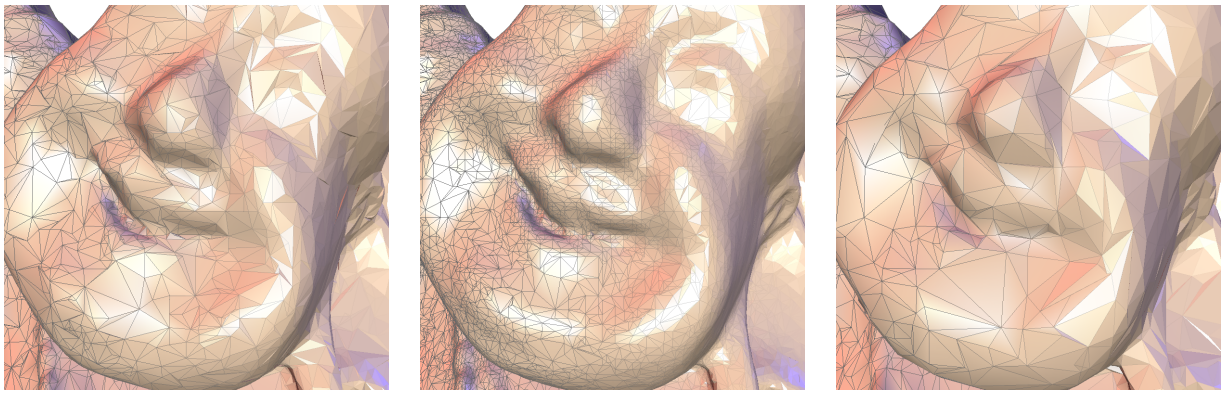
[20] R. Southern, S. Perkins, B. Steyn, A. Muller, P. Marais, and E. H. Blake. A stateless client for progressive view-dependent transmission. In *Proc. 2001 Web3D Symposium*, pages 43–49, 2001.

[21] D. S. P. To, R. W. H. Lau, and M. Green. A method for progressive and selective transmission of multi-resolution models. In *Proc. ACM Symposium on Virtual Reality Software and Technology*, pages 88–95, 1999.

[22] J. Wu and L. Kobbelt. A stream algorithm for the decimation of massive meshes. In *Proc. Graphics Interface 2003*, pages 185–192, 2003.

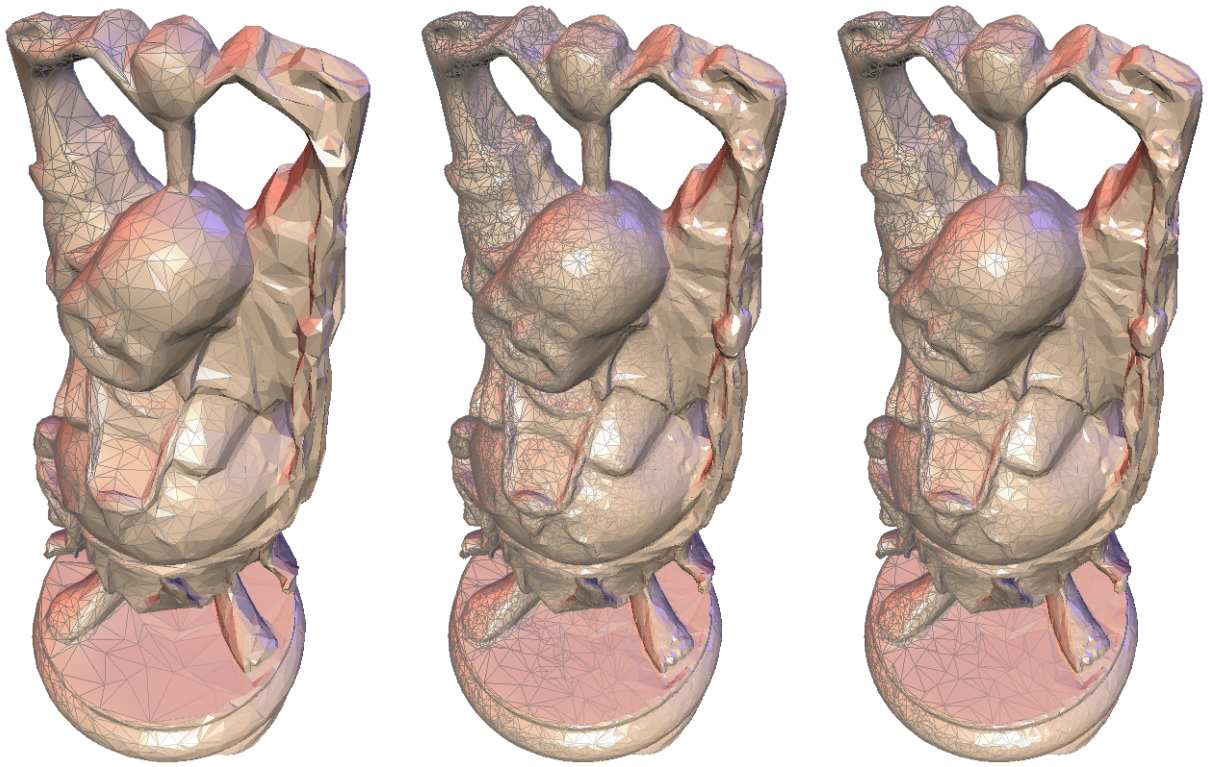
[23] J. C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *Proc. IEEE Visualization '96*, pages 327–334, 1996.

[24] S. Yang, C.-S. Kim, and C.-C. J. Kuo. A progressive view-dependent technique for interactive 3d mesh transmission. *IEEE Trans. Circuits and Systems for Video Technology*, page accepted for publication.



#faces: 12,654 #faces: 41,824 #faces: 41,824

(a) close-up of the Buddha face



#faces: 12,654 #faces: 41,824 #faces: 41,824

(b) look-down of the Buddha model

Figure 9. Comparison of view-dependent and view-independent streaming with a fixed communication data size (0.5MB): The images in the left column are generated by our view-dependent streaming technique. The center images are generated by a variation of our technique, where only the cut vertices are transmitted, instead of the fundamental cut vertices with the information for view-dependent refinement, as discussed in Section 7. The right images are generated by view-independent streaming. In (a), although the mesh in the left column has a smaller number of faces than that in the right column, the left column shows a better image quality than the right one because the triangles that really help the visual quality have been transmitted. When the viewpoint is far from the model as in (b), the left column shows a worse image quality than the right one because the amount of additional information for view-dependent refinement overwhelms the amount of data for triangles which does not contribute the screen-space image quality. However, in both case (a) and (b), the numbers of transmitted triangles in the middle and right columns are the same and the middle column shows a much better image quality because most of the transmitted triangles really help to improve the visual quality.