Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/092L22

A Distributed Stylization Mechanism for Line Extraction Process of a 2D Image

Mingyu Lim[1], *HyungSeok Kim*[1] *and Yunjin Lee*[2,*]

[1] Department of Internet & Multimedia Engineering, Konkuk University, Seoul, Korea
[2] Division of Digital Media, Ajou University, Suwon, Korea

Received: 25 May 2014, Revised: 25 Jul. 2014, Accepted: 26 Jul. 2014
Published online: 1 Apr. 2015

Abstract: In this paper, we propose a distributed rendering mechanism for image data using multiple servers. Since line extraction of large images requires high computation overhead to render an image, a client divides it into several image pieces. Each piece is sent to a different server, which then performs the rendering. A client then merges the rendered pieces into one output image. The proposed method enables large image data to be rendered by a collaboration of multiple servers with reasonable processing and communication cost.

Keywords: Remote rendering, image stylization, line extraction, distributed processing, multi-server communication

1 Introduction

With the advancement of computing devices and Internet technologies, we can easily create and obtain varying multimedia content. Due to the proliferation of mobile devices with embedded cameras, users have even more chance to manipulate 2D image data with a variety of applications. As recent high-end devices provide sufficient computing and rendering resources, users want to process, transform, and visualize higher quality 2D image data.

However, normal desktop PCs and mobile devices still lack sufficient rendering power due to hardware limitations. To address this issue, various studies have been performed using remote rendering, especially for 3D content [1,2,3,4,5,6,7]. The remote rendering method enables low-end devices to display images that have been rendered by a remote server, but it has an intrinsic problem. Although a single server has the role of surrogate renderer of a client, it can become overloaded in terms of rendering time if the server performs rendering for a large amount of content. As the content size increases, it takes a longer time to process it. To overcome the limitation of a single server, some existing approaches use parallel processing on a multi-core graphics card [8]. However, this approach requires expensive hardware and sufficient knowledge of parallel programming [9]. In addition, it is difficult to add more graphics cards if a single card is not sufficient to complete the processing of an input image with an acceptable delay.

In this paper, we aim to distribute the rendering overhead of a single server to multiple servers in order to reduce the overall delay of image stylization. In particular, we focus on the line extraction process, which represents one of the image stylization methods. To this end, a client is in charge of dividing an input image into several small pieces, if the image size is greater than a threshold value. Since the stylization of a pixel is affected by its surrounding pixels in our stylization method, an input image splits such that the boundary pixels of the pieces overlap. Otherwise, the stylization result at the boundary pixels of two image pieces cannot be merged seamlessly. Each image piece is then sent to a different server so that multiple servers handle only the separated image piece instead of the whole input image. Multiple servers conduct the line extraction from the received input image piece independently and in parallel. The servers do not interact with each other, and just send their resulting image pieces back to the requesting client. While receiving result image pieces from servers, a client assembles them into one stylized image, which is then displayed on the client screen. The proposed distributed stylization approach addresses the bottleneck problem of

* Corresponding author e-mail: yunjin@ajou.ac.kr

a single server rendering with modest changes on the client side, making it possible to stylize even a large input image with reasonable delay.

The remainder of this paper is organized as follows. Sec. 2 discusses the literature survey of the existing remote rendering and image stylization schemes. In Sec. 3, we describe the main problem and the fundamental idea of our proposed approach. In Secs. 4 and 5, we introduce a previous local stylization technique and its extension to the distributed manner of stylization. Communication aspects of the proposed distribution model benefit from communication middleware (CM), which is described in Secs. 6 and 7. In Sec. 8, we analyze our method to measure the performance in terms of the processing and communication overhead. Finally, we conclude our paper in Sec. 9.

2 Related Work

In this section, we discuss existing research on 2D image stylization techniques and remote rendering approaches.

For 2D image stylization, which is a target of this paper, there have been various mechanisms that require high rendering computation. For real-time processing, many image stylization methods have been designed to be highly parallel and have been implemented on the Graphics Processing Unit (GPU) [10,11,12,13]. Pixel operations are performed on the neighborhood around a pixel, and the operations for each pixel are independent of the other pixels. The locality and independence of the computations are suitable for our distributed rendering framework.

Shi et al. [1] proposed a real-time remote rendering system for mobile devices. In their approach, a server sends the selected depth images to the mobile client, which then runs 3D image warping on the received depth images to synthesize an updated image at the current viewpoint. They reduce the interaction latency by separating the rendering process between a client and a server. Doellner et al. [2] devised a server-based rendering scheme for large 3D scenes using G-buffer cube maps. Their approach also splits the rendering process between a server and a client. A server is responsible for the rendering of virtual panoramas which are represented by G-buffer cube maps, and a client uses these maps to reconstruct the 3D scene. Diepstraten et al. [3] developed a remote rendering method for 2D image generation tasks from a 3D model in mobile devices. A server extracts 2D line primitives from arbitrary 3D scenes, and a client receives and renders only 2D line primitives. There are also other similar remote 3D visualization approaches [4, 5], as well as a multitude of real-time rendering [6], and image-based rendering schemes [7]. Yoo et al. [8] presented a parallel design of the proxy server in the remote rendering framework. Using Compute Unified Device Architecture (CUDA) [9], they developed parallel rendering on GPUs in the proxy server.

In summary, most existing approaches focus on remote rendering for 3D content using a single server. Although some of this research lightens the burden on the server, a complicated parallel programming model is required for high-performance graphics hardware.

3 Design Consideration

Remote rendering is used to address the limitations of a device which does not have sufficient computing resources for processing input content. The remote rendering method was developed for displaying 3D content on a low-end device that has no 3D graphics accelerating capability. Even though a device has a 3D accelerator, it can delegate rendering tasks to a high-performance machine in order to reduce the processing delay. The same thing happens in 2D image stylization because it also requires a great deal of rendering computation on a GPU.

While the remote rendering method lessens the processing overhead, it causes another overhead: communication delay between a client and a server. Communication delay occurs when a client sends source content to a server, and when a server sends transformation results back to a requester. The delay is affected by the available network bandwidth and content size. Connection speeds of several Mbps surely cause a longer delay than Gbps networks with the same content size, and it takes a longer time to send larger content. Therefore, the remote rendering method has a performance benefit only if the communication and server process overhead is lower than the overall cost in the case of only local rendering.

In addition to the communication delay, a server may suffer from high processing overhead if the input content size is greater than the acceptable size due to the limitations of the graphics hardware. In this case, a graphics engine needs multiple rendering iterations because the buffer cannot read all of the bytes of the content at a time. Server performance can be improved by using a multi-core graphics card and parallel programming model. However, this requires high-performance hardware with parallel GPU architecture and knowledge of a parallel computing platform such as CUDA. Furthermore, a single GPU may have the same problem if the quality or the size of the input image goes beyond the expected capability, and it is difficult to extend the performance by adding more graphics cards. An alternative, more cost-effective approach is distributed rendering, in which parallel processing is conducted by multiple servers, as shown in Fig. 1. A client divides an image into several image pieces, and sends each piece to a different server. A server takes on the same role as that of remote rendering, but it lessens the burden of the rendering cost as it handles only a separate part of the whole image. Rendered image pieces are sent back to the client, which then collects the results and generates a rendered image. This approach can be easily extended by adding more servers, as the image size increases.

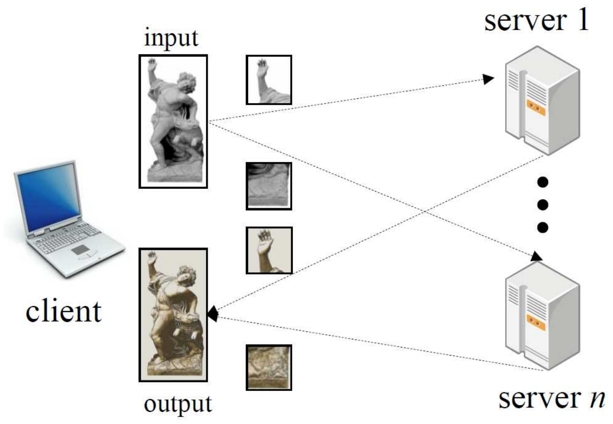
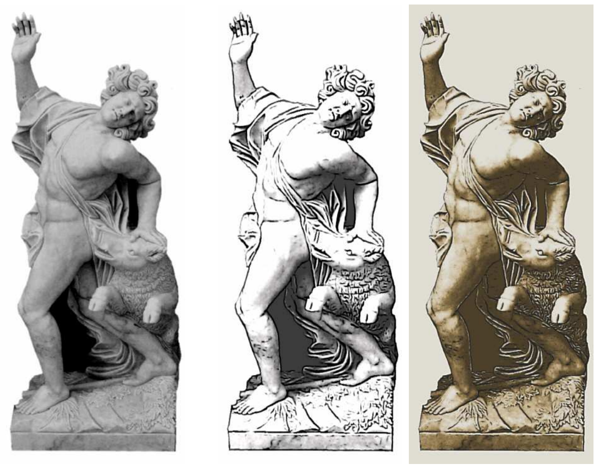


Fig. 1: Distributed rendering

In order to realize distributed rendering, a client must be able to partition an image into independent pieces, and rendered pieces must be reassembled such that the boundary pixels of the pieces are seamlessly connected. In other words, the recreated image must have the same quality as an input image stylized by one server. Another aspect to consider is the additional overhead incurred by distributed rendering. The overall overhead of distributed rendering consists of the partitioning and assembling process, as well as communication delay. Distributed rendering could be a reasonable scheme if the additional processing overhead is marginal and does not significantly affect the overall performance.

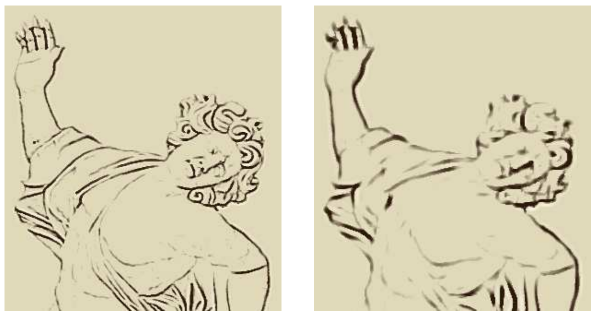
4 Stylization of Image Data

To stylize 2D image data in our system, we use a GPU-based line drawing method proposed by Lee et al. [12]. Although this line drawing method renders a 3D mesh as a line drawing, it can be applied to any scene representation such as a point set, implicit surface, or image-based representation because the method extracts lines from a shaded rendering of a scene. Therefore, it also works quite well using a 2D image as an input, as in Fig. 2. In Fig. 2(b), dark lines are drawn along thin dark areas of tone and along boundaries between dark and light regions, and highlight lines are drawn along thin bright areas of tone. In addition, the method can capture the tone variations in a broad region by combining toon shading with the lines. We can draw lines with various line widths as shown in Fig. 3, and the line widths are controlled according to the desired level of detail.



(a) input image (b) line drawing and shading results

Fig. 2: 2D Image stylization



(a) detailed lines (b) abstracted lines

Fig. 3: Control of line width

Viewing the tone image as a height field, highlight lines and dark lines correspond to ridges and valleys, respectively, as shown in Fig. 4. To extract lines along thin areas, we apply a ridge detection method using a polynomial fitting. At each pixel, we fit a 2nd-degree polynomial $f(x,y) = a_0x^2 + 2a_1xy + a_2y^2 + a_3x + a_4y + a_5$ to the tone values near the pixel using a least squares approximation. In practice, we use 9 sample points arranged in a 3×3 grid around the pixel location, with spacing set to half the desired line width. With pixel positions (x_i, y_i) and tone values t_i of sample points p_i, we can obtain the unknown coefficients of the polynomial by solving the equation like Eq. (1).

$$\begin{bmatrix} x_0^2 & x_0y_0 & y_0^2 & x_0 & y_0 & 1 \\ & & \vdots & & & \\ x_i^2 & x_iy_i & y_i^2 & x_i & y_i & 1 \\ & & \vdots & & & \\ x_n^2 & x_ny_n & y_n^2 & x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} t_0 \\ \vdots \\ t_i \\ \vdots \\ t_n \end{bmatrix} \quad (1)$$

We can determine if a pixel is on a ridge or valley using geometric properties computed from a fitted polynomial. Denoting the point (x,y) by x and a constant term by C, polynomial $f(x)$ can be expressed as $f(x) = (x-c)^T M(x-c) + C$, where

$$M = \begin{bmatrix} a_0 & a_1 \\ a_1 & a_2 \end{bmatrix}, \text{ and } c = -\frac{1}{2}M^{-1}(a_3 a_4)^T. \qquad (2)$$

The principal curvatures and directions of the polynomial at c are the eigenvalues and eigenvectors of M. Here, the line through c in the low-curvature direction is the ridge or valley in $f(x)$.

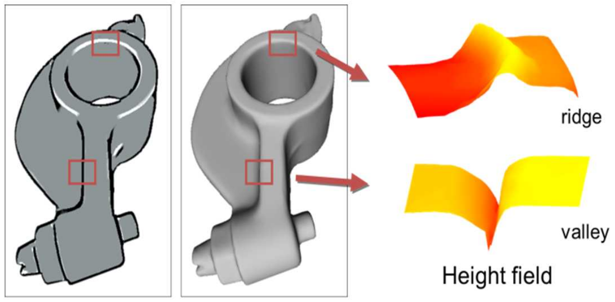


Fig. 4: Ridges and valleys in the height field

However, since we use a 2nd-degree polynomial, we cannot distinguish two cases: a pixel near a ridge or valley (Fig. 5(a)) and a pixel on an edge (Fig. 5(b)). To distinguish such cases, we use an iterative search method, which moves the point sampling toward the detected ridge or valley line, fits a polynomial with new samples, and measures the curvature and the distance to the new ridge or ridge line. In our implementation, the iterations are repeated at most five times. In case (b), the new computed curvature falls below a threshold or the new location moves outside the fitted region. Details of the approach are described in [12].

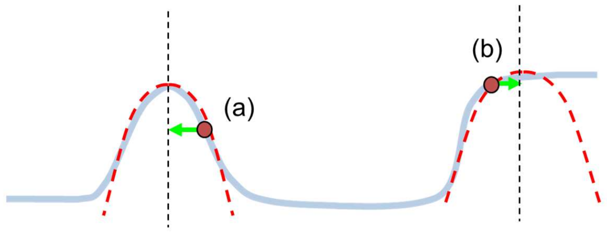


Fig. 5: Ridge searching

The line extraction process consists of two passes: the first is used to generate a tone image, which is a shaded rendering of a scene, and the second is used to detect ridges and valleys in the tone image. In the case of a 2D image as an input, the input image itself can be used as a tone image and only a blurring step is performed to reduce noise in the image or sampling artifacts in the second pass. After the line extraction process, we can augment lines with toon shading to stylize tone variations in broad regions. The whole process can be performed on a GPU using a fragment shader, which uses pixels only in a local region around each pixel over all passes. However, a GPU is not available in small mobile devices, which increases the processing time dramatically with a single CPU. In addition, a single GPU also has the limitations of parallel processing, as the quality and size of an input image increases. Algorithms 1 and 2 show the pseudo codes of the first and second passes in the line extraction process performed by a fragment shader.

Algorithm 1 1st pass of the line extraction: blurring step

```
// 1st pass: Blurring step for pixel $x$
$t_{sum} \leftarrow 0$, $w_{sum} \leftarrow 0$
**for** each pixel $p$ in a Gaussian kernel of size $k$ **do**
    $t_{sum} \leftarrow t_{sum} + G(p) \cdot t_p$
    // $G(p)$ is a Gaussian weight at pixel $p$
    // $t_p$ is the tone of pixel $p$
    $w_{sum} \leftarrow w_{sum} + G(p)$
**end for**
$blurred\_tone \leftarrow t_{sum}/w_{sum}$
```

Algorithm 2 2nd pass of the line extraction: ridge searching step

```
// 2nd pass: Ridge searching step for pixel $x$
**for** $i$:=1 to 5 **do**
    $a = compute\_coefficent(x)$ // solve Eq. (1)
    // $a$ is a vector of coefficents
    $[c,d] = compute\_curvature\_and\_distance(a)$
    // compute by Eq. (2)
    // $c$ is a principal curvature
    // $d$ is a distance to a ridge or valley line
    **if** $c < thr$ or $x + s \cdot d$ is outside the fitted region **then**
        break
    **end if**
    // $thr$ is a threshold
    // $s$ is a constant value between 0 and 1
    $x \leftarrow x + s \cdot d$ // move the pixel position
**end for**
```

5 File Partition for Distributed Rendering

For our distributed content stylization, we separate the required processes between a client and a server. A client is responsible for dividing an input image, merging the resulting sub-images into one output image, and visualizing an output image. The server takes on the role of the main processes for stylizing an image piece received from a client. At the client side for the distributed rendering, we have to consider how to partition an input image. In this step, we need to allow the boundary regions between the sub-images to be shared in order to provide the same inputs to a pixel shader for the pixels around the boundaries as the inputs in the local content stylization. For each pixel, a fragment shader uses 9 pixels sampled around the pixel location to fit a polynomial in the second pass of the stylization. The sampling center can move inside the initial sampling region in the iterative search process, and we sample 9 pixels around a new sampling center for each iteration. As shown in Fig. 6, if the spacing between samples is w and the initial sampling center is at (x,y), the maximum horizontal or vertical distance from (x,y) to the pixels used by the fragment shader in the second pass is $2w$. As the pixel values are blurred by a Gaussian kernel of size k in the first pass of the stylization, we need a $(2(2w+k)+1)(2(2w+k)+1)$ region centered at each pixel from an input image to determine if a pixel is rendered as a line or not.

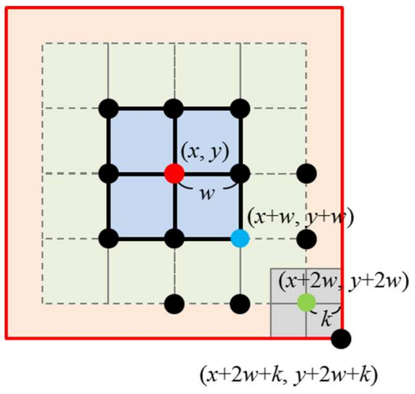


Fig. 6: A local region required for the processing

In the image division step, we add a region of width $2w+k$ or height $2w+k$ along the boundary of each sub-image. Fig. 7 shows an example where an input image is divided into two sub-images horizontally. The dashed regions are added to each sub-image in order to compute pixel values correctly around the boundary between the two sub-images. Each sub-image is then sent to a different server and processed independently. The solid-colored regions of the sub-images (I_0 and I_1) from different servers are sent back to a client, and are combined into the final line drawing result of the input image.

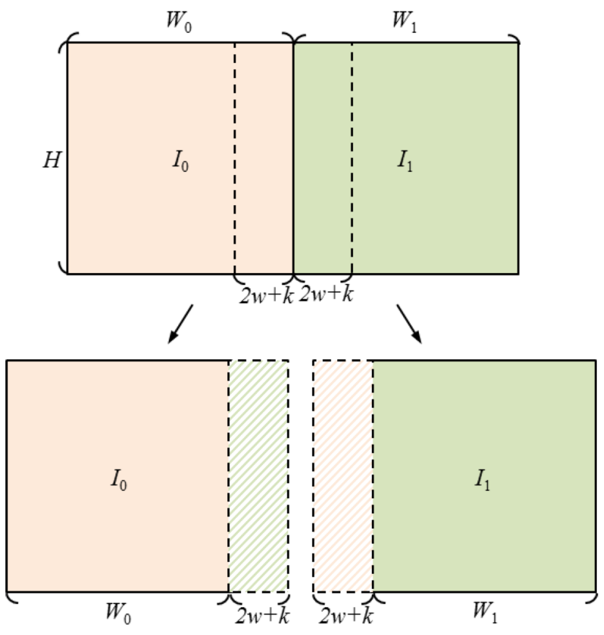


Fig. 7: Image partitioning

6 Communication Middleware

The proposed distributed stylization system is developed by separating the visualization process and rendering process between a client and servers, and the communication between participating nodes is realized by our communication middleware (CM) [14]. In this section, we introduce the supporting functionalities and modification of the CM for the development of the distributed stylization system.

The CM originally aims to provide an easy and efficient way of developing distributed applications. It supports various functionalities with options for different developer requirements. The middleware deals with communication methods which have to be implemented by developers if they have only fundamental networking support. Our system plays the role of a bridge between an application and the underlying network infrastructure. A fundamental element of this is to deliver messages and content between these two entities, by which communicating nodes can interact with one another. With Application Programming Interfaces (APIs) provided by the CM, application developers can create, send, receive, and process a remote event. In addition to dealing with events, it supports other operations which detect a specialized event and conduct a dedicated service according to the event type. To support them, we follow the layered approach to design the middleware. Fig. 8 shows the overall architecture of the CM. From the application's point of view, the CM consists of four main modules in the following order: CM stub, collaboration manager, event manager, and communication manager.

We describe the details of each module in the following sub-sections. To support the proposed distributed processing of the image data, the CM is extended to enable a multi-server-based client-server architecture. Additional processing servers can be dynamically added or removed to/from the current server network according to the developer requirements.

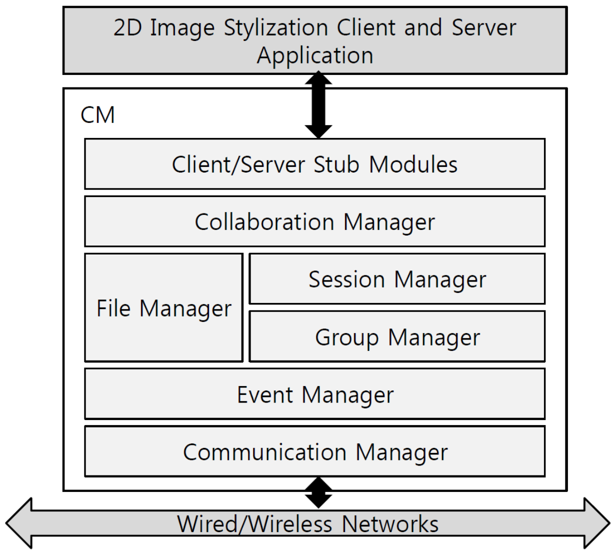


Fig. 8: CM architecture

6.1 CM stub

The CM stub is a core module which provides communication interfaces to an application. A developer can access most of the supporting functionalities of the middleware using this module. In general, it provides APIs to start and stop the CM, register and deregister an event to be used among CM nodes, send an event, and assign an event handling callback function which is called by the CM whenever it receives an event from a remote node.

In addition to these fundamental operations, depending on the application type (a client or a server) and the communication architecture (client/server, peer-to-peer, or hybrid), which are assigned in a configuration file by a developer, the CM stub provides appropriate useful functions with various options. For example, if the application is a client type in the client/server architecture, the CM stub offers a function to connect to a server. What a developer has to do is to set the server information like the IP address and port number in the configuration file and call the simple connection function.

6.2 Collaboration manager

The collaboration manager manages a different level of user interaction area according to the application requirement. It defines the number and relation of sessions and groups in a hierarchical structure. As illustrated in Fig. 8, the collaboration manager contains session managers, each of which handles a session, and a session manager can have more than one group manager, which manages a client (or a user) group. The developer can organize the structure of sessions and groups using the configuration file or APIs provided by the manager modules. In the CM architecture, we need at least one session and group to allow users to interact with each other. This means that the minimal unit of interaction is a group. A user always has to belong to a group. For the proposed distributed stylization system, we use the default session and group which is the minimum requirement, because there is only one client for visualizing stylized images.

An event passing through the CM stub module from the application layer stops by the collaboration manager which then checks the destination users in sessions or groups, and the event is delivered to the event manager before being sent. All inbound events from the network are also delivered to the collaboration manager. It checks the event header and conducts internal processes if it is required. If the target of an event is a session manager or a group manager, the collaboration manager delivers it to an appropriate manager which processes the event.

6.3 Event manager

The event manager is in charge of an event in the system. An *event* is a high-level form of a message between application nodes. As it includes semantics to be exchanged so that it can be understandable at the application layer, high-level manager modules (the collaboration manager, session manager, and group manager) and the application uses an event as a way to exchange information. One of the key roles of the event manager is to change an event to a low-level message (sequence of byte array) which is delivered to the underlying communication manager, and vice versa. The event manager provides several send functions which treat an event as a parameter and transform it into a message so that it can be sent via the communication manager. To the contrary, an incoming message given by the communication manager is converted to an event in the event manager which then delivers it to the collaboration manager for internal operation. The event manager also forwards the converted event to the CM stub module, which eventually delivers it to an application for the event process at that layer.

6.4 Communication manager

The communication manager controls messages and provides APIs to manage the communication channels. The main role is to send and receive messages to/from the underlying network. To support this, the communication manager runs a separate thread which waits for every incoming message. Whenever it receives a message from the network, the communication manager forwards it to the event manager. It also uses its own send functions embedded in each channel when a message to be sent is delivered from the event manager. If an application creates a channel, it is maintained in the communication manager as a channel list. Every event channel in the list is checked by the receiving thread, and the event manager is notified of any received message. The communication manager provides different dedicated communication sockets which wrap the native socket APIs and make it easily possible to open and close a channel depending on the requirement. Supported socket types are a server socket, a stream socket (TCP), datagram socket (UDP), and a multicast socket. For the proposed distributed stylization system, we use the server and stream socket between a server and a client application, because our system requires reliable communication of the input and output image pieces.

6.5 File manager

The main role of the file manager is to transfer files between communicating nodes such as a client and a server. Using the APIs of the file manager, an application can set a directory where a received file or a source file is located. A client or a server can then send/receive a file to/from a target node with one of two transmission modes, using the pull and push mode. The two modes are different in terms of the node that triggers the file transfer. In the pull mode, a requester node asks for a file from a source node, and the file transfer starts. On the other hand, in the push mode, a source node notifies a target node of its intention to send a file, and then the file transfer starts.

As the file transmission functionality is considered to be one of the most crucial features of the CM for the proposed distributed rendering mechanism, a detailed procedure of the file transfer in each mode is described as in the following. Fig. 9 shows the flow of the pull mode. When an application of a requester node (a receiver) calls the *requestFile()* function of the CM with two parameters (a file name and a name of source node), the CM sends a control event (*REQUEST_FILE_TRANSFER*) to the source node (a sender) in order to indicate its intention to receive a file. If the requested file is available, the source node adds the request information (requester name, file name, and file size), and replies to the request event (*REPLY_FILE_TRANSFER*). The source node then sends a control event (*START_FILE_TRANSFER*), which asks the requester if it is ready to receive the file or not. As a reply, the receiver sends a control event (*START_FILE_TRANSFER_ACK*) to the sender. The sender then starts the file transfer by creating a new thread which is dedicated to opening the target file and sending file blocks to the receiver (*CONTINUE_FILE_TRANSFER*). While the receiver receives file blocks, it writes them to a new local file. After it sends the last file block, the sender thread closes the source file, stops the dedicated thread, and sends a control event (*END_FILE_TRANSFER*) to notify the receiver of the completion of the file transfer. When the receiver receives the end event, it closes the written file and sends back a reply event (*END_FILE_TRANSFER_ACK*) to the sender, which then finalizes the file transfer.

In push mode, as shown in Fig. 10, most steps are the same as that of the pull mode except that the first request and its reply events are omitted. In this mode, a sender first asks a receiver if it is ready for receiving a file, because the sender application starts the file transfer by calling the *pushFile()* function of the CM with two parameters (a file name and a name of the target node).

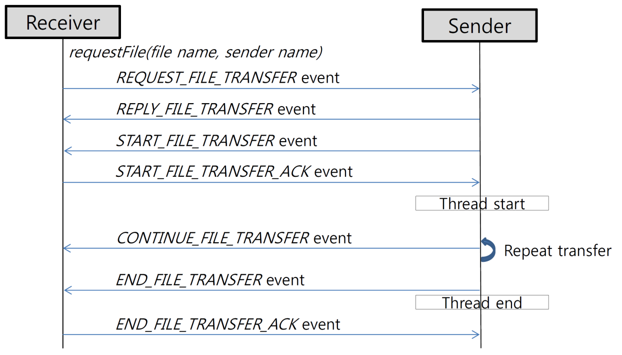

Receiver
Sender
requestFile(file name, sender name)
REQUEST_FILE_TRANSFER event
REPLY_FILE_TRANSFER event
START_FILE_TRANSFER event
START_FILE_TRANSFER_ACK event
Thread start
CONTINUE_FILE_TRANSFER event
Repeat transfer
END_FILE_TRANSFER event
Thread end
END_FILE_TRANSFER_ACK event

Fig. 9: Pull mode transmission

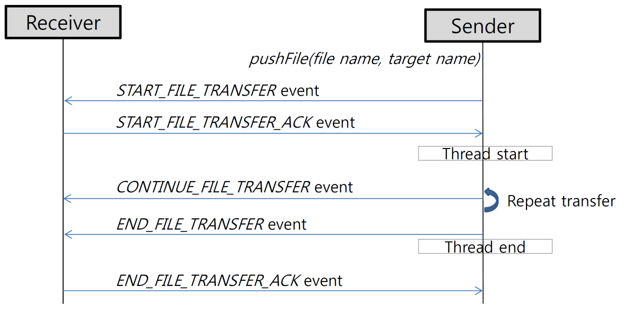

Receiver
Sender
pushFile(file name, target name)
START_FILE_TRANSFER event
START_FILE_TRANSFER_ACK event
Thread start
CONTINUE_FILE_TRANSFER event
Repeat transfer
END_FILE_TRANSFER event
Thread end
END_FILE_TRANSFER_ACK event

Fig. 10: Push mode transmission

7 Multi-Server Support of the CM

A distributed application can benefit from the CM by simply organizing communication nodes and calling communication-related APIs in order to interact with other nodes. However, one limitation of the previous CM was that it supported only one server in the client/server model. To realize the distributed rendering mechanism, we extend the CM to let a client application connect to multiple servers.

The organization process of multiple servers is shown in Fig. 11. As there can be more than one server in a system, we classify them with a default server and an additional server. To this end, the configuration file of a server CM has fields for the address of a default server and its own address. The server CM can identify its application whether it is a default server or not by comparing the two addresses in the configuration file. If they are equal, a server is a default server. Otherwise, it is regarded as an additional server. By default, a client establishes a connection with a default server.

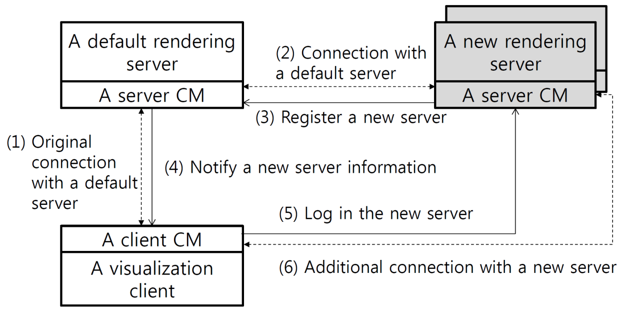

A default rendering server
A server CM
(2) Connection with a default server
A new rendering server
A server CM
(3) Register a new server
(1) Original connection with a default server
(4) Notify a new server information
(5) Log in the new server
A client CM
A visualization client
(6) Additional connection with a new server

Fig. 11: Multi-sever organization process

In the previous CM, the server was connected only with clients. However, now an additional server can also make a connection with a default server. Thus, the initialization phase of the server CM has changed such that it makes a connection with a default server if it is not the default server. If a system administrator wants to add an additional server to the currently running system, a server application can request its registration to a default server by calling the connection function and registration function of the server CM. In the registration request, an additional server can designate its name which will be used by other servers and clients as a target name. This name and the address information such as the IP address and the port number are transferred to the CM of a default server. When a default server receives a registration request from another server, it stores the delivered information on the requesting server in the list of other servers, and notifies its clients of the new server information. A default server maintains the list of other servers so that it can reply when a client asks about currently available servers. Therefore, any client can know the list of available servers either by explicitly requesting it from a default server, or by receiving update notifications using a default server. A default server now keeps a channel list which includes both its clients and other servers.

If an additional server leaves the system, it conducts the opposite procedure to registration by requesting deregistration and disconnection from the default server. When the default server receives the deregistration event, it deletes the requesting server information, and notifies clients of the server departure. This process is also triggered when a server application is unexpectedly terminated. In this situation, a default server detects that it is disconnected from a server, and implements the deregistration procedure as well.

The server side of the CM does not change much of its architecture from the previous development except the additional connection to a default server at the initialization phase and the management of the registration and deregistration of additional servers. The client CM, however, should manage multiple connections with different servers. It implies that the same interaction with a default server could happen with other servers as well. Therefore, the client CM should maintain a different interaction context with multiple servers. To do that, we use the server list, which is maintained in the collaboration manager of the server CM. The client CM also uses this data structure since it has almost the same architecture as the server CM, but it adds more information, which is required for a client to interact with additional servers. When the client CM receives the information on additional servers from a default server, it stores the information in its server list. This information is used when a client logs in to a new server in order to identify the target server. During the login process, the client CM also stores the server-specific information to the corresponding server element of the list, including various server policies, session information, names of the current session and group, and so on.

Another use of the client CM for a distributed rendering system is that APIs are provided to communicate with multiple servers. The previous version of the client CM provides APIs for communication only with the default server. Using the updated client CM, a client can interact with multiple servers by specifying a target server name in the APIs for connection (or disconnection) and login (or logout). The operation of such functions is divided into two cases. If a target server name designates a default server, the client CM calls a corresponding previous version of the function, which uses only a default server. Otherwise, the client CM finds a channel which maps to the server name, and uses it to send a message.

With the integration of the image partition and the multi-server support of the CM, the overall procedure of distributed rendering is shown in Fig. 12, which is an example of two rendering servers. The example starts with a client and a default server (*server1*). In order to use the distributed rendering functionality, the client should connect to and log in to the default server (1). When a new server (*server2*) is available, it registers itself to the default server (2). The default server then notifies the client of the new server information (3). The client makes another connection with *server2* and logs it in as well (4). Here, more servers could surely join the distributed rendering process, and the client could establish more connections with them. When the client receives an input image, it partitions the file into image pieces (5). The number of image pieces is determined to be the same number of currently connected servers. In this example, the image file is divided into two pieces (*piece1* and *piece2*). The client then sends each file piece to a different server in order (6 and 7). *Server1* and *server2* receive *piece1* and *piece2*, respectively, and they independently convert each input piece to the line-extracted file (8 and 9). After rendering the image pieces, the servers send them back to the client (10 and 11), and the client finally merges them into a resulting image file.

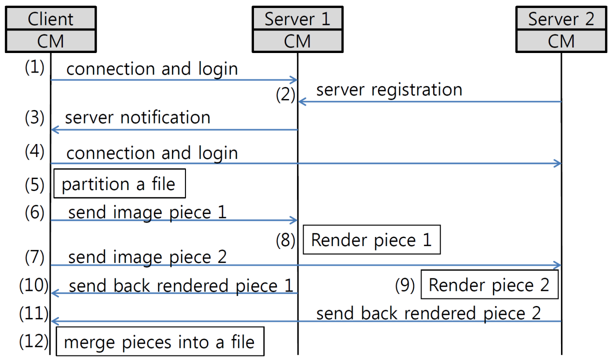


Fig. 12: Overall procedure with two server

8 Performance Evaluation

In this section, we discuss the analysis of the performance of the proposed distributed rendering scheme. The performance consists of the local processing cost and communication delay. In the proposed method, the input image content, C, is divided into n pieces, C_i $(1 \leq i \leq n)$. R_i is the rendered image piece of C_i, and R is a merged image of all R_i. S is a set of m participating servers, each of which is represented as S_i $(1 \leq i \leq m)$. The overall cost of distributed rendering for content C, $CostC$, is defined as follows:

$$CostC = Proccl + Comm(cl,S) + ProcS + Comm(S,cl), \tag{3}$$

where $Proccl$ and $ProcS$ are the processing delay at a client, cl, and servers, S, respectively. $Comm(cl,S)$ and $Comm(S,cl)$ are the communication delay of the content from a client, cl, to servers, S, and vice versa.

$Comm(cl,S)$ is the delay summation of transmitting all C_i to S_i, and $Comm(S,cl)$ is the summation of the transmission delay from all S_i to cl as described in Eqs. (4) and (5), respectively. If the existing single-server-based rendering approach uses a server S, its communication delays, $Comm(cl,S)$ and $Comm(S,cl)$, are almost the same as that of the proposed scheme, because all of the image pieces must be sent to one server. If multiple transmissions are conducted in parallel using separate channels, it can reduce the delay.

$$Comm^{(cl,S)} = \sum_{i=1}^{n} Comm_{C_i}^{(cl,S_i)} \tag{4}$$

$$Comm^{(S,cl)} = \sum_{i=1}^{n} Comm_{C_i}^{(S_i,cl)} \tag{5}$$

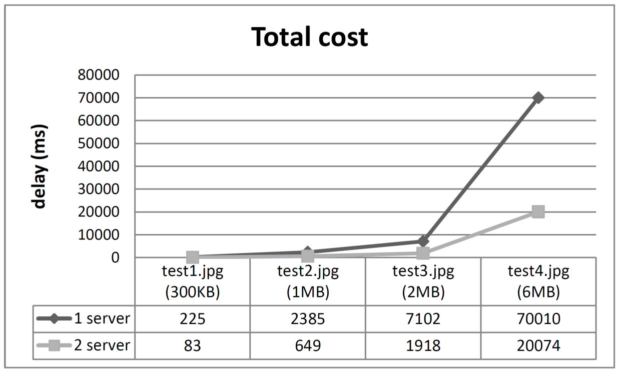
The processing delay of a server set, $ProcS$, occurs when participating servers process and stylize the input image pieces. compared to our approach, the processing overhead of the single server method causes much higher overhead, because only one server must be responsible for rendering all of the content. While the proposed scheme reduces its server processing and communication overhead, it pays a higher cost for client processing overhead. $Proccl$ is defined as Eq. (6) where $Dpart$ is the local processing cost for partitioning an input image, $Dmerge$ is the cost for merging the resulting image pieces, and $Ddisp$ is the cost for displaying an output image.

$$Proccl = Dpart + Dmerge + Ddisp \tag{6}$$

For the proposed scheme, $Dpart$ and $Dmerge$ are the additional processing cost compared to the single server approach, since the existing scheme does not need to partition and merge tasks.

We conducted an experiment to quantitatively measure the overall cost of the distributed processing as we changed the number of servers and the size of the input images. For the experiment, we implemented a test client/server application using Visual Studio 2010 on Windows 7. The test machines were connected through a 100 Mbps LAN. When the client splits the image into pieces, the number of pieces is automatically decided by the number of connected servers. If the client connects to one server, it does not split the image.

The overall cost is measured as the total elapsed time from the moment when the client splits an image to the moment when it finishes merging the stylized image pieces. Fig. 13 shows the experimental results. Compared to the case of a single server, the two-server-case surely reduces the overall cost. Furthermore, the larger the image size, the greater the cost reduction.



	test1.jpg (300KB)	test2.jpg (1MB)	test3.jpg (2MB)	test4.jpg (6MB)
1 server	225	2385	7102	70010
2 server	83	649	1918	20074

Fig. 13: Overall cost of distributed processing

9 Conclusion

In this paper, we proposed a distributed rendering scheme with multiple servers. In our approach, a client splits an input image into pieces and then sends them to different servers. Each server receives an image piece and runs a processing routine for the stylization of the piece. When processed image pieces are sent back to the client, it merges them into one image. Using multiple servers which stylize only parts of a whole image, the proposed system overcomes the limitation of a single rendering server, and high quality images are processed with marginal processing cost.

Currently, our research is still in progress, and we are planning to conduct more quantitative performance evaluations according to different image sizes and the number of servers in an extensive manner. We also have a plan to extend our approach to adaptive distributed rendering techniques. In this approach, the number of participating servers is dynamically chosen according to the size of the input image in order to make the system more scalable than the current fixed environment.

Acknowledgements

This paper was written as part of Konkuk University's research support program for its faculty on sabbatical leave in 2014, and this research was also supported by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (NIPA-2014-H0301-14-1001) supervised by the NIPA(National IT Industry Promotion Agency).

References

[1] S. Shi and K. Nahrstedt and R. Campbell, A real-time remote rendering system for interactive mobile graphics, ACM Transactions on Multimedia Computing, Communications, and Applications, **8**, 46:1-46:20 (2012)

[2] J. Doellner, B. Hagedorn, J. Klimke, Server-based rendering of large 3D scenes for mobile devices using G-buffer cube maps, Proceedings of the 17th International Conference on 3D Web Technology, 97-100 (2012)

[3] J. Diepstraten, M. Gorke, T. Ertl, Remote line rendering for mobile devices, Proceedings of Computer Graphics International, 454-461 (2004)

[4] F. Lamberti and A. Sanna, A streaming-based solution for remote visualization of 3D graphics on mobile devices, IEEE Transactions on Visualization and Computer Graphics, **13**, 247-260 (2007)

[5] G. Paravati, A. Sanna, F. Lamberti, L. Ciminiera, An open and scalable architecture for delivering 3D shared visualization services to heterogeneous devices, Concurrency and Computation: Practice & Experience, **23**, 1179-1195 (2011)

[6] E. Gobbetti, D. Kasik, S. Yoon, Technical strategies for massive model visualization, Proceedings of the 2008 ACM Symposium on Solid and Physical Modeling, 405-415 (2008)

[7] C. Chang and S. Ger, Enhancing 3D graphics on mobile devices by image-based rendering, Proceedings of the third IEEE Pacific Rim Conference on Multimedia: Advances in Multimedia Information Processing, 1105-1111 (2002)

[8] W. Yoo, S. Shi, W. Jeon, K. Nahrstedt, R. Campbell, Real-time parallel remote rendering for mobile devices using graphics processing units, Proceedings of IEEE International Conference on Multimedia and Expo, 902-907 (2010)

[9] J. Nickolls, I. Buck, M. Garland, K. SSkadron, Scalable parallel programming with CUDA, Magazine Queue GPU Computing, **6**, 40-53 (2008)

[10] J. Kyprianidis, Image and video abstraction by multi-scale anisotropic Kuwahara filtering, Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering (NPAR '11), 55-64 (2011)

[11] H. Winnemöller, S. Olsen, B. Gooch, Real-time video abstraction, ACM Transactions on Graphics, **25**, 1221–1226, (2006)

[12] Y. Lee, L. Markosian, S. Lee, J. Hughes, Line drawings via abstracted shading, ACM Transactions on Graphics, **26**, 18:1-18:5 (2007)

[13] J. Lopez-Moreno, J. Jimenez, S. Hadap, E. Reinhard, K. Anjyo, D. Gutierrez, Stylized depiction of images based on depth perception, Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering (NPAR '10), 109-118 (2010)

[14] M. Lim, B. Kevelham, N. Nijdam, N. Magnenat-Thalman, Rapid development of distributed applications using high-level communication support, Journal of Network and Computer Applications, **34**, 172-182 (2011)

Mingyu Lim is an Associate Professor at the Department of Internet and Multimedia Engineering, Konkuk University, Korea from the year 2009. Before joining Konkuk University, he was a senior researcher at MIRALab, University of Geneva, being involved in various research activities on networked virtual environments and ubiquitous computing systems. He received his PhD in Computer Science in February 2006 at ICU, Korea. His major research field is supporting scalability in networked virtual environments. His current research activities are focused on efficient communication middleware, event transmissions, and content distribution in networked ubiquitous computing systems.

HyungSeok Kim received a PhD degree in Computer Science in February 2003 at VR Lab, KAIST. He is an Associate Professor at the Department of Internet and Multimedia Engineering, Konkuk University, Korea. Before joining Konkuk University, he was a senior researcher at MIRA Lab, University of Geneva, being involved in research activities on virtual reality. His major research field is real-time interaction in virtual environments and multimodal interaction mechanisms. His current research activities are focused on topics of shape modeling for real-time rendering and evoking believable experiences in virtual environments.

Yunjin Lee is an Associate professor of in the Division of Digital Media at Ajou University. She received her BS degree in 1999 and her PhD degree in 2005, all in Computer Science and Engineering from POSTECH in Korea. Her research interests include nonphotorealistic rendering, 3D mesh processing, and data compression.