Extracting Graphics Information for Better Video Compression

Kang Woon Hong, Won Ryu, Jun Kyun Choi, and Choong-Gyoo Lim

Cloud gaming services are heavily dependent on the efficiency of real-time video streaming technology owing to the limited bandwidths of wire or wireless networks through which consecutive frame images are delivered to gamers. Video compression algorithms typically take advantage of similarities among video frame images or in a single video frame image. This paper presents a method for computing and extracting both graphics information and an object's boundary from consecutive frame images of a game application. The method will allow video compression algorithms to determine the positions and sizes of similar image blocks, which in turn, will help achieve better video compression ratios. The proposed method can be easily implemented using function call interception, a programmable graphics pipeline, and off-screen rendering. It is implemented using the most widely used Direct3D API and applied to a well-known sample application to verify its feasibility and analyze its performance. The proposed method computes various kinds of graphics information with minimal overhead.

Keywords: Cloud gaming, video compression, graphics information, depth value, motion vector, occlusion map, call interception, programmable shader, off-screen rendering.

Manuscript received Jan. 31, 2014; revised May 13, 2015; accepted June 1, 2015.

This research was supported by 'The Cross-Ministry Giga KOREA Project' of The Ministry of Science, ICT and Future Planning, Rep. of Korea (GK13P0100, Development of Tele-Experience Service SW Platform based on Giga Media).

Kang Woon Hong (corresponding author, gwhong@etri.re.kr) and Won Ryu (wlyu@etri.re.kr) are with the Broadcasting & Telecommunications Media Research Laboratory, ETRI, Daejeon, Rep. of Korea.

Jun Kyun Choi (jkchoi59@kaist.ac.kr) is with the Department of Electrical Engineering, KAIST, Daejeon, Rep. of Korea.

Choong-Gyoo Lim (cglim@skhu.ac.kr) is with the Department of Computer Engineering, Sungkonghoe University, Seoul, Rep. of Korea.

I. Introduction

Cloud gaming has lately emerged as an interesting commercial service in the computer gaming industry. Some examples include OnLive and Gaikai [1]–[4]. Cloud gaming runs computer games on remote servers and transmits real-time images of consecutive game frames to gamers through wire or wireless networks. Some of its advantages can be summarized as follows [5]:

- instant play without any additional expensive hardware or accessories
- support of various kinds of user devices, and thus one-source multiple-use
- easy to monitor patterns of user game play and correct run-time errors
- instant installation of patches and upgrades
- no piracy possible

However, there are also some disadvantages of cloud gaming that cannot be alleviated easily. One of them is response latency, which is a result of network lag. A few frames of game play may be dropped owing to a poor Internet connection, thus leading to an unpleasant gaming experience. Another is the fact that gamers do not actually possess the game titles themselves. Even worse, the total service charge for online play that a gamer finally ends up paying is likely to far exceed the one-time purchase cost of a game title.

One of its key technologies is real-time video streaming, where consecutive frame images are compressed and delivered to users with minimal network delay [3]. Regardless of whether a network is wired or wireless, its bandwidth is limited to a certain extent, and an efficient and effective compression algorithm therefore needs to be developed. If there exists a variety of appropriate information on consecutive frame images, then a video compression algorithm, such as MPEG-4 or H.264, can be improved significantly, particularly with some clues on where similar image blocks are located between consecutive frames (see [6] for an example).

This paper presents a graphics information extraction method that extracts various kinds of graphics information from consecutive frame images of computer games. Because there is a high possibility in computer games of finding a similar collection of objects between consecutive frame images, then it is equally highly likely that similar or even identical image blocks exits between such frame images. The extraction method computes graphics information and then extracts it so that video compression algorithms can take advantage of this information by using it to determine the sizes and locations of similar image blocks. The graphics information provided by the extraction method includes pixel depth, pixel movement, and object boundaries. The aforementioned cloud-gaming services are known to use H.264 for their video compression [3], which is one of the well-established encoders in the industry. However, the codec details are rarely known, and whether they use any graphics information is not known either. This is the first paper to present a method on extracting graphics information from frame images of game applications.

In Section II, we briefly explain some related technologies needed for the implementation of the graphics information extraction method. In Section III, we then discuss in detail how to compute and extract various kinds of graphics information. In Section IV, we explain our method's current implementation and analyze its performance when applied to a well-known sample program. We conclude by discussing the pros and cons of the method and some future work.

II. Related Technologies

1. Programmable Shaders

Three-dimensional APIs have been widely used to develop 3D applications in the computer graphics and computer game industries. The most popular are OpenGL and Direct3D. OpenGL was originally developed by the computer graphics community to be used on many different platforms; namely, to make cross-platform applications. OpenGL performs its operations through its graphics pipeline, which consists of consecutive execution phases to construct 3D images on a device screen, as shown in Fig. 1 [7]. Owing to the very nature of its pipeline architecture, each phase is executed in parallel to the other phases to achieve a better performance. Recent advancements in computer graphics technologies have made it possible to customize the *vertex shader* and *fragment shader* *phases* for constructing appealing visual effects, such as a flame or splashing water. A programmable pipeline has been officially integrated into the traditional rendering pipeline with OpenGL 2.0 or later. At the fragment phase, the color value of each pixel is computed for a position determined through rasterization. The depth value of each pixel is also computed, which indicates how far each pixel is away from the camera. The pipeline saves the color and depth values into the color and depth buffers, respectively. The depth buffer can be accessed when one needs to perform additional tasks during the rendering process or for other tasks. Video compression algorithms can take advantage of depth values when necessary by accessing the buffer directly. No information on the pixel movement or object boundary is provided by the current graphics pipeline, however.

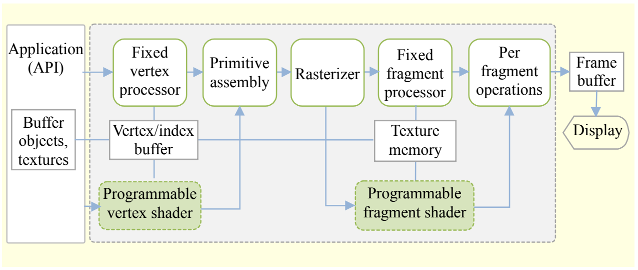

Fig. 1. OpenGL graphics pipeline (as of OpenGL 2.0).

Direct3D was developed by Microsoft to support the development of 3D games on its Windows platforms. Its pipeline is very similar to that of OpenGL. The fragment shader is called a *pixel shader* instead [8]. The programmability was first introduced through Direct3D 8.0 [9]–[10]. Unlike OpenGL, there was no direct access to the depth buffer until the release of Direct3D 10.0 [11]. There is no information available at all regarding pixel movements and object boundaries for either OpenGL or Direct3D.

2. Off-Screen Rendering

Rendering is the process of generating each frame image for 3D applications. Its output is directly displayed inside a window created by the window system for its application. The output can be saved into the system memory just like any other objects in the game, which is the off-screen rendering that most APIs support. An image of a size bigger than the actual device resolution may be constructed, and off-screen images may be used to describe other objects with textures linked to them. If a device supports the WGL_ARB_pbuffer extension, one of OpenGL's extensions, then off-screen rendering can be implemented using pbuffer [12]. Direct3D uses dynamic textures to conduct off-screen rendering [13].

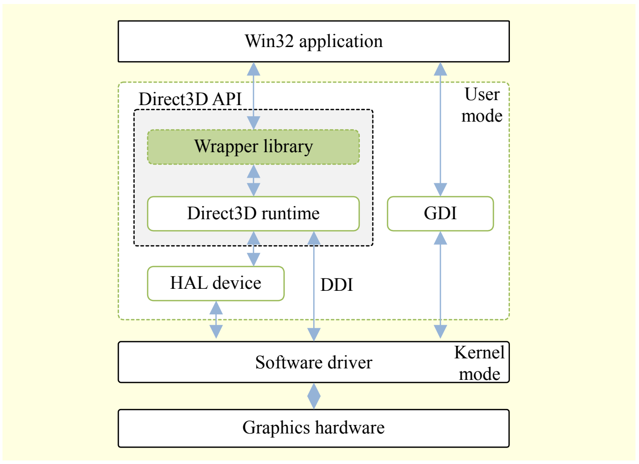

Fig. 2. Modification to Direct3D architecture.

3. Call Interception of API Functions

In a cloud-gaming service, game applications run on remote servers, which are likely dedicated systems for running the game applications. If an application's source code is modified, then various types of needed information can be directly obtained for better video compression algorithms. This is not desirable, however, because any change to the source code may incur additional development costs. Using a call interception of the API functions, one can avoid additional development costs.

There are two methods for a call interception of the API functions; namely, a physical modification method and a real-time modification method [14]–[15]. In a physical modification method, the execution code or library code is modified prior to running the application. There are three different ways to do this. One is to modify the start portion of the function code so that it loads another library and runs one of the library functions instead of the original function. Another way is to modify the *import table* of the execution file. When a function is called, it either loads another library or runs another part of the code. The final way is to use a *wrapper* library. In this case, each and every function that the application calls is rewritten, and a wrapper library is constructed by combining these functions. During run-time, the wrapper library is loaded and the rewritten code is called instead. A real-time modification method is used to insert an event hook during the execution of the application. The host process must have permission to do so. The wrapper library method is relatively easier to implement, because it rewrites library functions and replaces the original function with a newly constructed function, as shown in Fig. 2. This is one of the widely used methods in the literature, as described in [16]. Here, the original API functions may be called through the wrapping library. On the other hand, other applications are unable to call the original functions directly once the replacement has taken place.

III. Computation and Extraction of Graphics Information

The accessibility to a depth buffer differs among 3D APIs, as mentioned earlier. OpenGL allows direct access to a depth buffer, whereas until the release of version 10.0, Direct3D did not. This paper presents a method applicable to any version of 3D APIs. It computes not only a pixel's depth but also its movement and the object boundaries, and makes them available to other processes such as video compression algorithms.

To make it easy to implement while computing various kinds of graphics information, the method adopts a common set of steps that can be extended to compute other kinds of information. The method consists roughly of the following consecutive tasks:

1) At the start, the application loads DLL libraries including wrapper libraries. The method creates a dynamic texture for storing the graphics information.
2) In each frame, the application intercepts the function calls of a 3D API and stores them into the command queue. It issues the original function calls simultaneously to construct the frame image. After the first frame, the original render target has to be restored before the start of the frame because it will be set to a newly created render target at the next step.
3) To compute the graphics information, the application saves the current render target and switches to a new render target attached to the dynamic texture.
4) A customized vertex shader and pixel shader run to compute the graphics information while calling the original functions once more concurrently. The graphics information is stored into the dynamic texture while running the customized shaders.

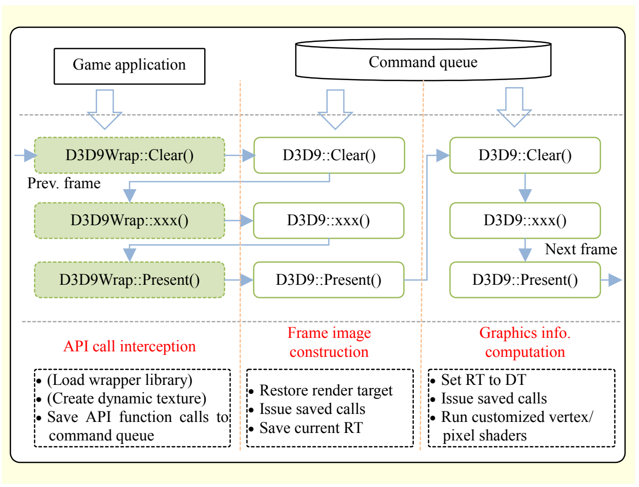

Fig. 3. Program flow of frame.

5) The application switches back to the original render target.

The whole procedure can be illustrated as shown in Fig. 3 using Direct3D. Step 1 is carried out once at the beginning of the application, and steps 2 through 5 are repeatedly performed at every frame.

1. Call Interception of 3D API Functions

The method intercepts the API function calls issued by the application and stores the call information including the function name, data types of the parameters, and parameter values. It runs the original function calls at the same time because it needs to construct the same frame image of the game application as it is supposed to do in the first place. The stored function calls are executed once more at the end of the frame to compute the graphics information. The whole procedure is performed by the frame rather than by the function call because a better performance is achieved by running a collection of consecutive function calls than by running each function call twice with alternating render targets.

OpenGL API consists of the core library (OpenGL), OpenGL utility library (GLU), and OpenGL toolkit library (GLUT). Each library is a collection of independent API functions. To implement a call interception, one needs to write wrapping codes for all functions of each library and combine them as a wrapping library that replaces the original library. If one of the original functions is called, then the wrapping code of the newly constructed library is called. On the other hand, Direct3D consists of the core library (Direct3D) and extension library (D3DX). Each library is a collection of various COM objects. One needs to define a wrapper class for every COM object. For each member function of a COM object, a wrapping code is written to become a member method for the wrapping class. The dynamic-link library (DLL) of the wrapper classes replaces the original DLL library. When the original method of a COM object is called, it calls the wrapping code of the wrapper class instead. In both OpenGL and Direct3D, original libraries are renamed and loaded upon execution so that the wrapper code can call the original functions.

2. Computation of Graphics Information

A. Pixel Depth

The graphics pipeline stores the pixel depth values into a depth buffer or Z-buffer, which is a part of the frame buffer, but these values are inaccessible for certain versions of 3D APIs. The method modifies the pipeline by running customized shaders whose codes are dedicated to computing the pixel depth values as follows:

$$(x', y', z', w') = (x, y, z, w) \cdot \mathbf{M}^{\mathrm{w}} \cdot \mathbf{M}^{\mathrm{c}} \cdot \mathbf{M}^{\mathrm{p}}, \qquad (1)$$

where (x, y, z, w) is a 4D vector of homogenous coordinates of a vertex position with (x', y', z', w') being the transformed vector, and $\mathbf{M}^{\mathrm{w}}$, $\mathbf{M}^{\mathrm{c}}$, and $\mathbf{M}^{\mathrm{p}}$ are the 4×4 matrices of the world transformation, camera transformation, and projection, respectively. The resulting depth value is z'/w' as computed using perspective division.

As explained earlier with a common set of steps, the method first creates a dynamic texture for off-screen rendering, which replaces the original render target temporarily. It then computes the pixel depth values by running customized shader programs while running the stored function calls at the same time, which were originally issued by the application. The following series of consecutive steps briefly represents the procedure of the vertex shader, particularly with Direct3D being the target API, where INPUT is a collection of input data and OUTPUT is a collection of output data:

1) Start.
2) Set the vertex position as INPUT.
3) Set the pixel position and depth as OUTPUT.
4) Compute the position of the current pixel by applying world, camera, and projection matrices of the current frame to the position vector of the input vertex (that is, vertex position).
5) Obtain the depth value from the z-value of the transformed position by conducting perspective division.
6) Store the pixel position as the OUTPUT position.
7) Store the pixel depth value as the OUTPUT depth value.
8) Finish.

The application feeds an array of vertices into the graphics pipeline, and thus into the vertex shader, which is a part of the pipeline. The final depth value is computed for each pixel through interpolation at the rasterization and is stored into the dynamic texture at the pixel shader, particularly into one of four color channels of the texture, for example.

B. Pixel Movement

Video compression algorithms such as MPEG and H.264 compute motion vectors to take advantage of pixel movement between frame images, and thus movement of a certain encoding block of the image. However, this typically consumes a large amount of system resources, such as the CPU time and memory, on a typical 2D image when no clues regarding a pixel's movement are available [6].

As with pixel depth, pixel movement can be easily computed by modifying the graphics pipeline, which is adjusted to run customized shaders whose codes are dedicated to computing the pixel movement as follows:

$$(x_{i-1}', y_{i-1}', z_{i-1}', w_{i-1}') = (x, y, z, w) \cdot \mathbf{M}_{i-1}{}^{\mathrm{w}} \cdot \mathbf{M}_{i-1}{}^{\mathrm{c}} \cdot \mathbf{M}_{i-1}{}^{\mathrm{p}}, \qquad (2)$$

$$(x_i', y_i', z_i', w_i') = (x, y, z, w) \cdot \mathbf{M}_i^{\mathrm{w}} \cdot \mathbf{M}_i^{\mathrm{c}} \cdot \mathbf{M}_i^{\mathrm{p}}, \qquad (3)$$

where (x, y, z, w) is a 4D vector of homogenous coordinates of a vertex position, and $(x_{i-1}', y_{i-1}', z_{i-1}', w_{i-1}')$ and (x_i', y_i', z_i', w_i') are the transformed vectors of the $(i-1)$th and ith frames, respectively, and $\mathbf{M}_{i-1}^{-}$ and $\mathbf{M}_{i}^{-}$ are the transformation matrices of the $(i-1)$th and ith frames, respectively. The resulting motion vector is $(x_i' - x_{i-1}', y_i' - y_{i-1}')$.

First, the method creates a dynamic texture for off-screen rendering, which replaces the original render target temporarily. It then computes the pixel movement by running the customized shader programs, while the application calls the stored functions again at the same time. The procedure of its vertex shader is very similar to that in computing the pixel depth value as follows, where INPUT is a collection of input data, and OUTPUT is a collection of output data. The procedure is also targeted for Direct3D API.

1) Start.
2) Set the vertex position as INPUT.
3) Set the pixel position and motion vector as OUTPUT.
4) Compute the previous position of the current pixel by applying world, camera, and projection matrices of the previous frame to the position vector of the input vertex (that is, vertex position).
5) Compute the current position of the current pixel by applying world, camera, and projection matrices of the current frame to the position vector of the input vertex (that is, vertex position).
6) Store the pixel position as the OUTPUT position.
7) Store the pixel motion vector as the OUTPUT motion vector.
8) Finish.

The application feeds an array of vertices and two sets of three world, camera, and projection matrices for both the current frame and the previous frame into the pipeline, and thus into the vertex shader. The final motion vector is computed for each pixel through interpolation at the rasterization and stored into the color buffer set to the dynamic texture at the pixel shader. The result is stored using two color channels of the texture because there are only two components of the motion vector on a 2D image.

C. Object Boundary

A scene in a computer game consists of many different objects, some of which are entirely represented on screen while others are partially displayed because some parts of the objects can be hidden by other objects from the viewpoint of the camera. If information is available on where the objects are displayed on screen and how big they are, then video compression algorithms can take this information into account to better determine the positions and sizes of the encoding blocks. One may also utilize the information in determining where similar image blocks are located between consecutive frames.

As in the pixel depth and movement, the proposed method can easily compute the boundaries of objects by modifying the graphics pipeline; namely, running the stored function calls and customized shaders. First, it creates a dynamic texture for off-screen rendering, which replaces the original rendering target temporarily. It computes the boundaries of objects by running the customized shaders while running the stored function calls at the same time. It then stores the result into the color buffer associated with the dynamic texture. The following series of consecutive steps briefly represents the whole procedure of the vertex shader, where INPUT is a collection of input data and OUTPUT is a collection of output data. The procedure is also targeted for Direct3D API.

1) Start.
2) Set the vertex position and object ID as INPUT.
3) Set the pixel position and object ID as OUTPUT.
4) Compute the position of the current pixel by applying world, camera, and projection matrices of the current frame to the position vector of the input vertex (that is, vertex position).
5) Store the pixel position as the OUTPUT pixel position.
6) Store the object ID as the OUTPUT object ID.
7) Finish.

The application feeds an array of vertices and a preassigned ID of the object into the pipeline, and thus into the vertex shader. The pixel shader reads the object ID and stores it into the pixel position of the color buffer, which is computed through interpolation during the rasterization phase of the pipeline. The object then has the same pixel value at the position where it is displayed on screen. Pixel values can be preassigned RGB color values instead.

IV. Implementation and Results

1. Implementation

To implement the proposed method, we used a system equipped with a 2.67 GHz Intel Core i5 M580 CPU and 4 GB of main memory. The method runs on MS Windows 7 (32 bit). To show that the method is feasible with a 3D API, where there is no direct access to its depth buffer, the implementation was targeted for Direct3D 9.0c (released in April, 2006).

The module of a call interception is implemented based on the wrapping library for easier implementation. The implementation is applied to an example application to show how well it works. For the example application, the *ShadowMap* sample provided together with DirectX SDK was modified. Applications must not create objects online and should use only a fixed pipeline.

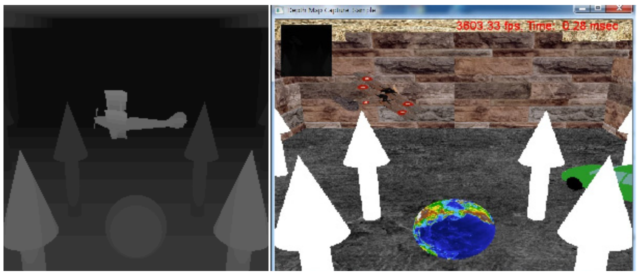

Fig. 4. Results of pixel depth computation (left, depth map image; right, screen image of sample run).

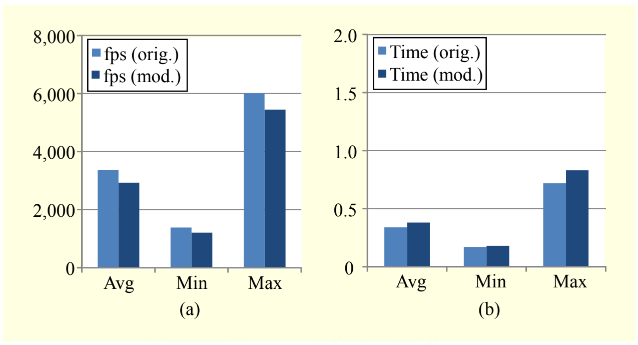

Fig. 5. Performance of pixel depth computation: (a) fps and (b) time per frame (ms).

2. Results

A. Pixel Depth

The current implementation computes the pixel depth, as shown in Fig. 4. It also shows an image of the current dynamic texture at the top-left corner of the screen at the same time.

It takes 0.042 ms longer, on average, to compute the depth values than without additional depth computation, as shown in Fig. 5. The method creates 440 fewer frames per second. The total time is increased by 12.4%, whereas the fps is decreased by 13.1%. Considering the extra amount of time to display the depth map texture on the screen simultaneously, the extra time to compute the pixel depth value is negligible. The overhead required to save into external storage is not taken into account.

B. Pixel Movement

The current implementation computes the pixel movement, as shown in Fig. 6. As with the pixel depth, the method additionally displays an image of the current dynamic texture on the screen.

It takes 0.040 ms longer, on average, to compute the motion vectors than without additional motion vector computation, as shown in Fig. 7. The method creates 585 fewer frames per second. The total time is increased by 14.5%, whereas the fps is reduced by 14.1%. As in computing the pixel depth value, the extra time required to compute the pixel motion vector is negligible.

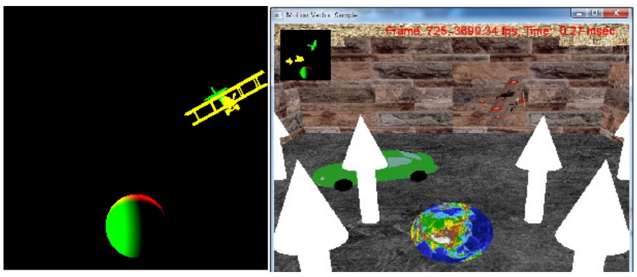

Fig. 6. Results of motion vector computation (left, motion vector image; right, screen image of sample run).

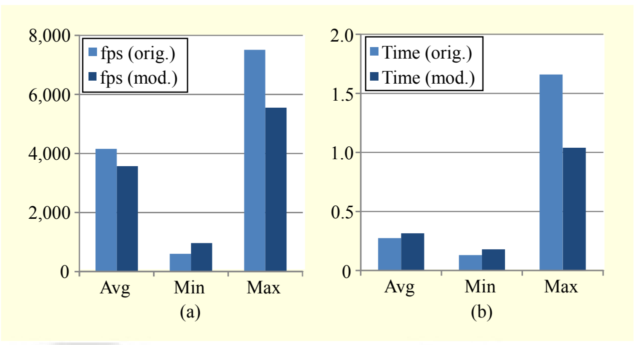

Fig. 7. Performance of motion vector computation: (a) fps and (b) time per frame (ms).

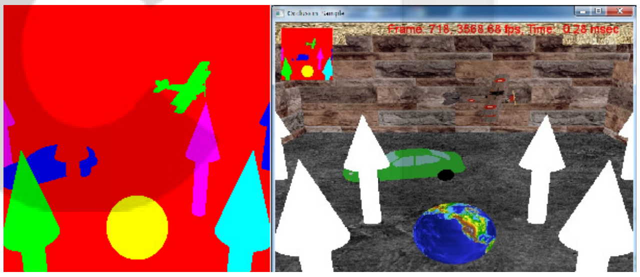

Fig. 8. Results of occlusion computation (left, occlusion map image; right, screen image of sample run).

C. Object Boundary

The implementation computes an object boundary, as shown in Fig. 8. As in the pixel depth and movement, the method additionally displays an image of the current dynamic texture on the screen as well.

It takes 0.058 ms longer, on average, to compute the occlusion than before the modification, as shown in Fig. 8. The method creates 562 fewer frames per second. The total time is increased by 18.3%, whereas the fps is decreased by 15.6%. The extra time to compute an object boundary is minimal, as in the pixel depth and movement.

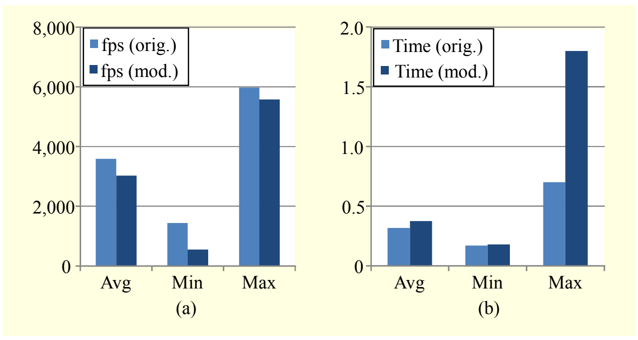

Fig. 9. Performance of occlusion computation: (a) fps and (b) time per frame (ms).

V. Conclusion

This paper presented a method to compute and extract various graphics information from frame images of computer games for video compression algorithms to take advantage of. According to its implementation and performance analysis, negligible overhead is incurred. Therefore, the proposed method can be used in implementing a real-time streaming service of game frame images and thus make cloud-gaming services more technologically feasible. The method currently provides pixel depth, pixel movement, and object boundaries; it can be easily extended to provide other various kinds of graphics information. One advantage of the method is the simplicity of its architecture. It simply runs API functions of each frame (a further time), a customized vertex shader, and a pixel shader of the rendering pipeline, while keeping the other parts of the graphics pipeline intact. Another advantage is that it does not modify the source code of the application by implementing the module of the API call interception, and thus incurs no additional development costs.

On the other hand, the proposed method runs saved function calls and customized shaders after running the original API functions for each frame. This means that it takes an extra amount of time to do so, though it incurs very little total overhead in the process, as discussed earlier. Another disadvantage is that even if the method finds an object boundary between consecutive image frames, there is likely a shading difference owing to the different lighting of consecutive frames. The proposed method may be limited to a certain extent in terms of compression ratios. The displayed images of an object are still similar enough in terms of pixel color values that an efficient compression can be achieved.

The current implementation was applied to an example with limited features, particularly with no objects created on-the-fly or programmable shaders. Support of both objects created on-the-fly and programmable shaders is an area of future work.

Appendix

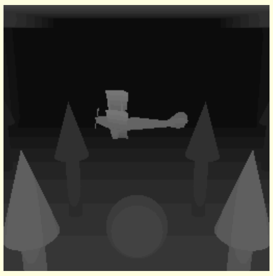

Fig. 10. Sample texture image of pixel depth.

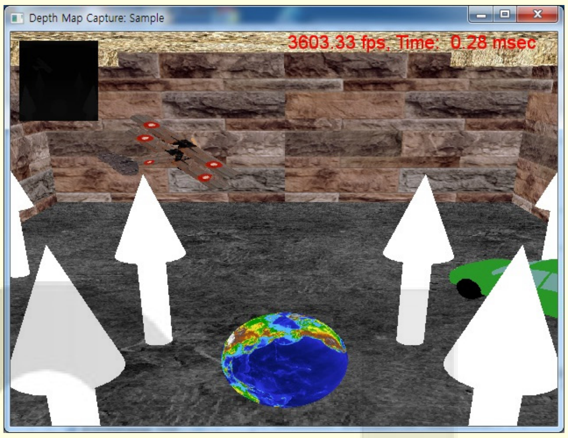

Fig. 11. Sample screen image of pixel depth computation.

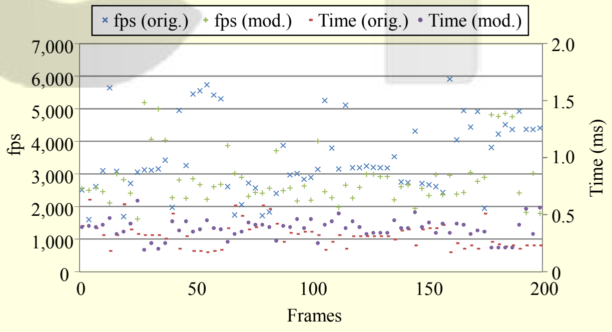

Fig. 12. Performance of pixel depth computation in detail (displayed every third frame for frames 200 to 400).

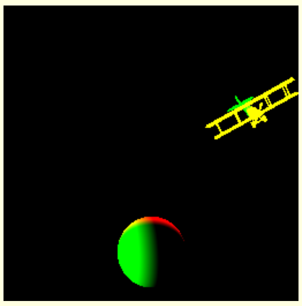

Fig. 13. Sample texture image of pixel movement.

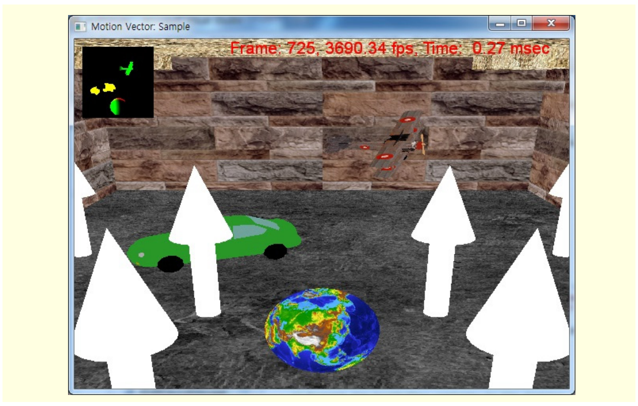

Fig. 14. Sample screen image of pixel movement computation.

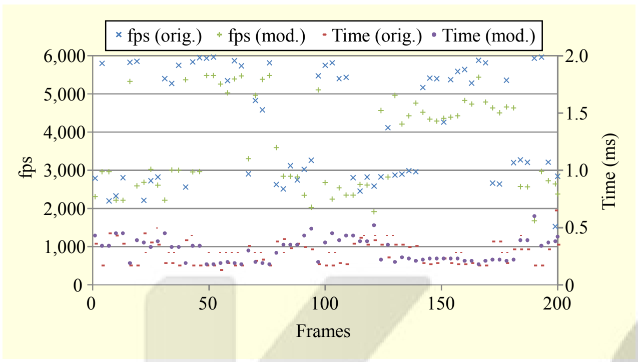

Fig. 15. Performance of pixel movement computation in detail (displayed every third frame for frames 200 to 400).

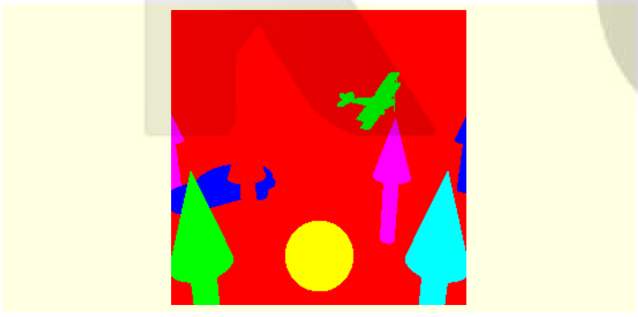

Fig. 16. Sample texture image of object boundary.

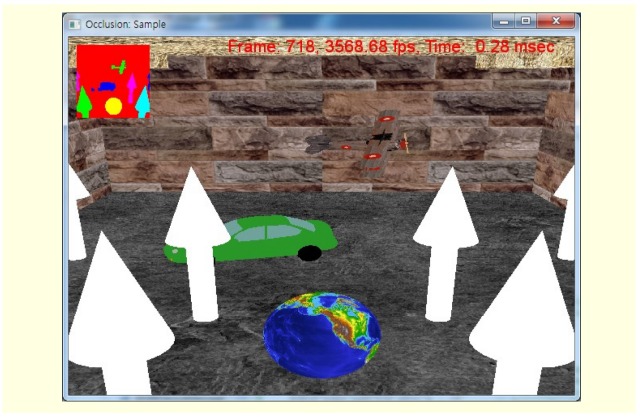

Fig. 17. Sample screen image of object boundary computation.

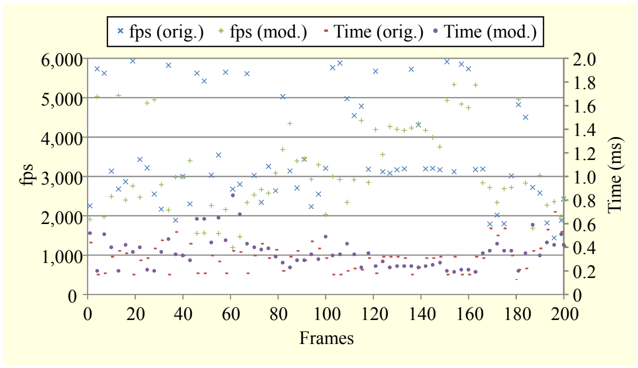

Fig. 18. Performance of occlusion computation in detail (displayed every third frame for frames 200 to 400).

References

[1] E. Sofge, *Games without Gizmos*, The Wall Street Journal, Nov. 12, 2012. Accessed Dec. 14, 2013. http://www.wsj.com/articles/SB10001424052970203707604578095413928945612

[2] K.I. Kim et al., "Cloud-Based Gaming Service Platform Supporting Multiple Devices," *ETRI J.*, vol. 35, no. 6, Dec. 2013, pp. 960–968.

[3] S. Perlman, *The Process of Invention: OnLive Video Game Service*, lecture, Botwinick Lab, The Fu Foundation School of Engineering and Applied Science, Columbia University, New York, NY, USA, Nov. 13, 2009. http://tv.seas.columbia.edu/videos/545/60/79

[4] D. Perry, *GDC 2012: Gaikai CEO David Perry Talks What Gaikai Offers*, YouTube video, posted by GamerLiveTV, uploaded Mar. 22, 2012. https://www.youtube.com/watch?v=6lN5E_pgBsU

[5] P. Bradley, *Cloud Gaming: The Pros and Cons*, *Masonic Gamer*, July 23, 2012. Accessed Dec. 14, 2013. http://masonicgamer.com/cloud-gaming-the-pros-and-cons/

[6] I. Richardson, "*H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia*," West Sussex, England: Wiley, 2003.

[7] D. Hearn, M. Baker, and W. Carithers, "*Computer Graphics with OpenGL*," 4th ed., NJ, USA: Prentice Hall, 2010.

[8] *Direct3D Architecture (Direct3D 9)*, Microsoft, Nov. 16, 2013. Accessed Dec. 14, 2013. http://msdn.microsoft.com/en-us/library/windows/desktop/bb219679(v=vs.85).aspx

[9] *Using Shaders in Direct3D 9*, Microsoft, Dec. 12, 2013. Accessed Dec. 14, 2013. http://msdn.microsoft.com/en-us/library/windows/desktop/bb509704(v=vs.85).aspx

[10] T. Jones, *DirectX 8 Graphics and Video: A Fresh Start*, Nov. 30, 2000. Accessed Dec. 14, 2013. http://archive.gamedev.net/archive/reference/articles/article1247.html

[11] I. Cantlay, "*High-Speed, Off-Screen Particles*, *GPU Gems 3*," H. Nguyen, Ed., NJ, USA: Addison-Wesley, 2008, pp. 513–528.

[12] C. Wynn, *Using P-Buffers for Off-Screen Rendering in OpenGL*, NVidia Technical Paper, Nvidia, 2002. Accessed Nov. 25, 2013. https://developer.nvidia.com/sites/default/files/akamai/gamedev/docs/PixelBuffers.pdf
[13] K. Harris, *Off-Screen Rendering*, Direct3D (DirectX 9.0) Code Samples, Feb. 11, 2005. Accessed Dec. 1, 2013. http://www.codesampler.com/dx9src/dx9src_7.htm
[14] *API Hooking*. Accessed Oct. 18, 2012. http://en.wikipedia.org/wiki/Hooking
[15] S.W. Kim, *Intercepting System API Calls*, Mar. 7, 2012. Accessed Oct. 18, 2012. http://software.intel.com/en-us/articles/intercepting-system-api-calls
[16] S.M. Jang, W.H. Choi, and W.Y. Kim, "Client Rendering Method for Desktop Virtualization Services," *ETRI J.*, vol. 35, no. 2, Apr. 2013, pp. 348–351.

Kang Woon Hong received his BS and MS degrees in computer science and engineering from Hanyang University, Ansan, Rep. of Korea, in 1996 and 1998, respectively. Since 2000, he has been with the Electronics and Telecommunications Research Institute (ETRI), Daejeon, Rep. of Korea. He is currently working as a principal researcher in the Intelligent Convergence Media Research Department, ETRI. His main research interests are distributed media processing, processing virtualization, and scalable system architectures.

Won Ryu received his BS degree in computer science and statistics from Pusan National University, Rep. of Korea, in 1983 and his MS degree in computer science and statistics from Seoul National University, Rep. of Korea, in 1988. He received his PhD degree in information engineering from Sungkyunkwan University, Suwon, Rep. of Korea, in 2000. He is currently working as the managing director of the Intelligent Convergence Media Research Department, Electronics and Telecommunications Research Institute, Daejeon, Rep. of Korea.

Jun Kyun Choi received his BS degree in electronics from Seoul National University, Rep. of Korea, in 1982 and his MS and PhD degrees in electrical and electronics engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Rep. of Korea, in 1985 and 1988, respectively. He worked for the Electronics and Telecommunications Research Institute, Daejeon, Rep. of Korea, from 1986 to 1997 and is now currently working as a professor at KAIST. Since 1993, he has contributed to consenting ITU Recommendations as a rapporteur or editor in ITU-T SG 13. His main research interests include web of objects, open IPTV platforms, energy saving networks, measurement platforms, and smart grids.

Choong-Gyoo Lim received his BS and MEd degrees in mathematics education from the Department of Mathematics Education, Seoul National University, Rep. of Korea, in 1998 and 1990, respectively, and his PhD degree in computer science from the Computer Science Department, Louisiana State University, Baton Rouge, USA, in 1998. From 1999 to 2011, he worked for the Electronics and Telecommunications Research Institute, Daejeon, Rep. of Korea. Since 2011, he has been with the Department of Computer Science and Engineering, Sungkonghoe University, Seoul, Rep. of Korea, where he is now an associate professor. His main research interests are geometric modeling, massive terrain rendering, and distributed rendering.