Collaborative Graphic Rendering for Improving Visual Experience

Xiaoxin Wu and Guodong Pei

Intel China Research Center Ltd, Beijing, China
{xiaoxin.wu,guodong.pei}@intel.com

Abstract. Handheld devices such as UMPC, though convenient, bear weakness of size constraint for display. To mitigate such a problem and enhance user experience for owners of small devices, in this paper we design a collaborative rendering platform. When running game graphic applications at a handheld, the generated OpenGL graphic commands are intercepted and then delivered to a device with a larger display. The graphics are rendered and displayed at that device. The performance of the collaborative rendering platform is determined by graphic computing resources and network bandwidth. Analysis and simulation prove that other than providing a better display, the collaborative system can improve game experience also by increasing frame rates. In particular, at a low computing cost, a further collaboration between GPUs of collaborators can improve frame rate by eliminating the negative impact from network delay on applications that require GPU feedback.

1 Introduction

Collaborative computing accomplishes tasks that may be difficult or even impossible for individual device to handle, by jointly utilizing resources such as computing components [1,2], storage [3,4], and services [5,6] provided by collaborative parties. As a typical collaborative computing case, Grid Computing [7] has been widely investigated and implemented for different applications. Grids are built up by a group of network-connected servers, providing computing services and data services. To support user mobility and more dynamic grid membership, wireless grid computing [8] has been proposed, where wireless networks are used for connection among grid members for convenience.

Handheld computing devices such as UMPC are becoming popular. Compared to larger devices, e.g., laptops, UMPC has comparable CPU computing capability and same comunication interfaces including wireless communication interfaces for WiFi and Bluetooth. Yet it is much easier to carry a UMPC because it is only $1/5$ of the size of a laptop and weights much lighter. A shortcoming of handheld devices, however, is that the small-sized body may result in a worse user experience. In particular, as UMPC has a much smaller display, for applications such as game and video that require good visual effect, it can not bring users as much joy as a laptop does. In addition, UMPC generally does not have strong display card or GPU. It then may not process complicated graphic calculations that are required for, e.g., $3D$ graphic applications at all.

E. Bertino and J.B.D. Joshi (Eds.): CollaborateCom 2008, LNICST 10, pp. 609–622, 2009.

We apply the concept of collaborative computing and build small-sized grids to enhance visual experience of handheld users. Closely located collaborative parties, e.g., a UMPC and a laptop, are connected by networks so that graphic or video results generated at UMPC can be shown at the laptop's display. The network can be either wired (e.g., cable or USB) so that a high bandwidth, reliable communication is applicable, or wireless (e.g., WLAN, UWB, etc.), which supports mobility and convenient usage models. Since OpenGL [9] is a strong and well established graphic tool for generating pictures with sophisticated visual effect (For example, it creates $3D$ graphics in computer games and virtual life), we investigate how to display OpenGL games through collaborative systems and design a platform that supports sharing of displays and GPUs among collaborative parties.

Such a collaborative display can be implemented at different layers. It can be implemented at a very high layer, where the display side does almost everything including game execution. The UMPC then works only as a user interface for game operation inputs. Although the approach can maximumly free UMPC and fully utilize the computing resources including both CPU and GPU from its collaborator, it requires that the collaborative parties have exactly the same game environment. This means the display side has to be very powerful and load all the games that a UMPC user may play with. Therefore, the solution lacks flexibility and scalability, and is less practical. On the other hand, the collaboration can also be done at a very low layer. The UMPC executes the games, renders the graphic, and generates low-layer graphic elements (e.g., pixels, color units). It then sends the elements to its collaborator where graphics can be reconstructed. The display side only needs the computing capability for graphic generation (probably with a graphic card installed) and networking interface. The drawback for this approach is that the bandwidth requirement for delivering graphic results from UMPC to the display side is too high. This limits the use of wireless collaboration because it may jam the wireless link, which itself is a scarce resource that has to be shared with a lot of other users. Another drawback of the approach is regarding to power. Because the UMPC has to go through the entire graphic display process and transmit a large amount of data, the load is too heavy and it may soon run out of battery.

To address the problems in both of the above approaches, we propose collaboration at a relatively middle layer. In particular, we "outsource" the display role to a stronger device by first intercepting OpenGL calls for graphic rendering and then sending these calls to that device, where OpenGL is supported so that graphics can be rendered and displayed. Other than obtaining, generally, a much larger display, such a collaboration may also help to improve game performance in terms of frame rate if good sets of computing resources and network is applied.

Related work of rendering graphics by intercepting graphic commands can be found in [10,11,12,13]. Other than building a real game remote rendering system, in this work we design mechanisms for performance improvement and develop analysis model that can be used as platform design guide. In summary, our major contributions are summarized as follows:

- We apply the concept of collaborative computing to enhance visual experience for users of small devices. We design and implement a collaborative rendering platform that intercepts and transmits OpenGL calls, so that any OpenGL based graphic

applications executed at a device can be rendered and displayed at other destined devices. The platform is flexible and supports usage models such as multi-view games and large-scale display wall.
- We develop mathematical models for analyzing system performance of different settings, to theoretically evaluate the gain of collaborative system. We conduct extensive experiments and present detailed data from real case study as well.
- Based on analysis and simulation study we identify the potential performance bottlenecks for the collaborative rendering platform, and determine the collaborative settings that may improve game performance. To eliminate the negative impact of slow network on games with GPU feedback requirement, we design a mechanism that utilizes extra GPU resources to reach a desired frame rate at a balanced communication and computation cost.

The rest of the paper is organized as follows. In Section 2 we present platform architecture and a few performance improvement mechanisms. In Section 3 we build mathematical model to analyze game improvement through the collaborative systems. In Section 4 we show experimental results and data analysis. We conclude in Section 5.

2 Collaborative Graphic Rendering Design

2.1 Architecture

A collaborative rendering system is built up by client and server(s), plus the network connecting them. Generally, the client has a small display and probably, a weaker GPU. A server, on the other hand, should have a larger display and a stronger GPU. When executing OpenGL graphic applications such as games over the collaborative rendering system, the client executes the games by itself and generates the OpenGL commands for game graphics. It then sends the OpenGL commands to the server. Installed with OpenGL environment, the server can render the graphics by executing the received OpenGL commands and display. Since transmitting OpenGL commands requires a much less bandwidth than transmitting graphic elements, the requirement for network is less stringent. Either high-speed cable network or low-bandwidth WLAN (802.11g) can be used for system connectivity. In this platform TCP is used as the transport layer for communication reliability.

The collaborative rendering architecture is shown in Fig. 1. The key function block is the OpenGL Command Intercepter block at the client. It gets all the graphic rendering commands and sends them to the server. When graphics have to be rendered at multiple displays, this function block also determines which of the OpenGL command streams will be sent, and to which servers the different streams should be sent. For example, for a multi-view game, the OpenGL Command Intercepter block finds the OpenGL commands for a particular view and then sends the commands regarding to that view to a destined server. By doing so, different views will be shown on different displays for an impressive game experience.

2.2 Potential Performance Bottlenecks

The most important criteria for game performance is frame rate, or the number of frames per second (FPS). In the collaborative rendering platform, frame rate depends on the

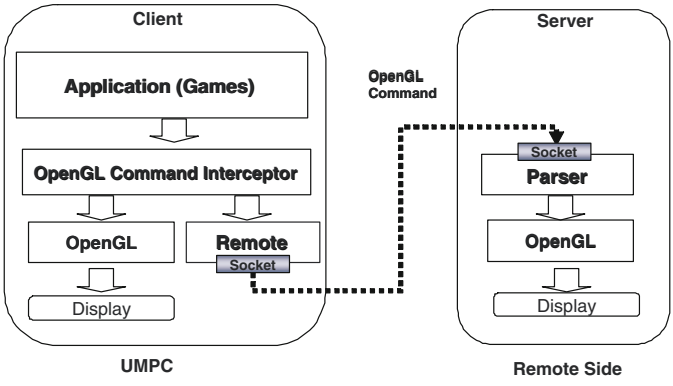


Fig. 1. Collaborative rendering system architecture

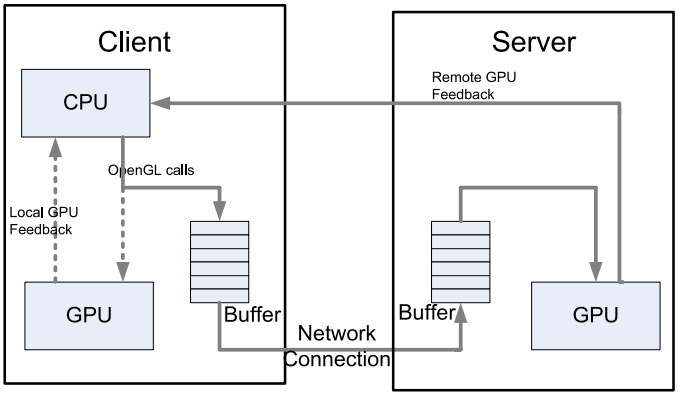


Fig. 2. Remote rendering flow

performance of a few components. As shown in Fig. 2, the remote rendering may involve CPU at client, GPU at server, and network connecting them. It may also involve GPU at client side, depending on how to generate GPU feedback that is required for game computing. The dash lines in the figure illustrates a scenario that the local GPU generates some of GPU results and sends the results to the local CPU. The reason for doing this will be explained in 2.3.

Here we summarize in more details on how the capability of above mentioned components may slow down the game as follows.

- A slow CPU at the client will slow down the game. This is obvious because it will take longer time to execute the game and generate OpenGL commands.
- A slow GPU at the server will slow down the game. This is because a slow GUP will cause the buffer at the server to be filled very soon. Consequently, the buffer at the client will be full as well. In our design, when client buffer is full, an interrupt has to be sent to the CPU so that no new commands will be generated.

– A slow network will slow down the game. A slow network will result a filled buffer at the client side. No more frames are generated until some of generated OpenGL calls are delivered to the server.

To speed up games running over the collaborative rendering platform, a strong CPU at the client, a high-performance GPU at the server, and a good network connection are needed. In case when system settings and network can not be changed, other techniques can be developed to most efficiently utilize the available resources for obtaining the best game experience.

2.3 Improve Game Performance

Game FPS can be improved by reducing the bandwidth requirement for delivering OpenGL calls, and therefore reducing the communication delay over the network. FPS can also be improved by program optimization, and by selectively using the available computing resources at client and server to minimize the overall frame generation time that consists of communication and computing delay.

Reducing Bandwidth Requirement. To reduce the impact of communication delay, a straightforward solution is to reduce the data amount required to be transmitted for every game frame. The techniques adopted in the collaborative rendering platform are as follows:

– *Object Caching.* To reduce the data amount transmitted from client to server during a game, objects (most time an object is a texture) that will be used in the game will be transmitted from the client to the server at the game setup stage. The server will cache the objects, and sync with the client through agreed object index. Later during the game when a specific object is called by an OpenGL command, the object itself will not be delivered to the server. Instead, an object index number is transmitted. The server then can render the graphics by calling the correct object based on the received number. As in our platform, objects are delivered and cached at the server at game set-up stage, during the game there is no object delivery. The bandwidth for collaborative rendering during the game then can be significantly reduced.
– *Command Labelling.* Index is also used for indicating commands (including parameters) that have been transmitted in the previous frame(s). The client checks whether any command has been transmitted previously. If so, it will transmit an index number for that command. Because there are not many types of OpenGL commands, the index requires only a few bits. Transmitting command index can result in a much smaller load for the network. To further reduce the bandwidth requirement, sophisticated data compression technique such as Huffman code can be used for index compression. The compression gain, however, will be traded off by the increased computing load and the corresponding larger delay.

Program Optimization. We use multi-threaded game computing in the collaborative OpenGL rendering platform to reduce the impact of communication overhead. The OPenGL commands are generated at the client and then sent to the server. If a single thread is used and OpenGL command interception and delivery are executed in a

series manner, before calculating further game operations, the game engine will wait until the delivery for previous command finishes. This will decrease the frame rate. In the multi-thread platform, we add a thread for command delivery, so that the game engine will continue calculation while the previous results are delivered to the server. This program optimization speeds up the overall process, and increases the frame rate for a better game view.

GPU Collaboration for Computing and Communication Resource Optimization. In most games, CPU has to take some of the previous GPU results for future frame generation. In the collaborative rendering platform, GPU data required for game calculation has to be generated at server GPU and sent back from the server to the client. The corresponding delay on feedback transmission will cause significant performance degradation in particular when a large amount of feedback is required, because the computation gain obtained at GPU has been offset by the communication loss at the network. The longer the network delay is, the worse the performance is.

To solve this problem, we propose partially utilizing GPU on the client side, to locally generate the data required for client CPU. The results generated at the server GPU then do not have to be sent back to the client side. This solution is also called GPU collaboration, as server GPU and client GPU work together to resolve the problem.

In the GPU collaboration, as the first step, we identify the OpenGL calls that require feedback from GPU, and test how frequent these calls are generated in the game. In the typical car racing game Torcs, we identify five OpenGL calls that have to use previous GPU results. We further identify that three of these commands are only needed at the beginning of the game, and will not have much impact on game performance. For the rest of two, one command is especially used for error correction which seldom happens and can be neglected. Then for only one of these commands the client GPU has to execute the related commands to generate the required feedback data. Compared to the entire game, such a calculation load is small. In the GPU collaboration the client GPU will not be a bottleneck for game performance. As the negative impact from feedback delay has been removed, the frame rate can be significantly increased.

3 Frame Rate Analysis: Mathematical Models

For analysis tractability, we assume that each game frame consists of n OpenGL calls, and for each of these calls, the CPU computing time is the same, which is denoted as T_{cpu}. We further assume that the GPU rendering time for each call at either client side or server side is the same. In particular, for each of the calls, the GPU rendering time at the client is denoted as T_{gpu1}, and the GPU rendering time at the server is denoted as T_{gpu2}. In general, as server has a stronger GPU, $T_{gpu2} < T_{gpu1}$.

We assume that the processing time for CPU to fetch data (e.g., the feedback from GPU) from the GPU located at the same device can be neglected. The CPU generates a new OpenGL commands when it receives the required feedback information. In the case that no feedback is required, the CPU generates a new OpenGL command if the previous generated command has been executed, i.e., transmitted from client to server. For analysis simplicity, we assume that the buffer depth at both of the client and server

is zero. As when rendering graphics, CPU and GPU work in a pipeline manner, the component that takes a longer time to accomplish its task is the performance bottleneck.

Denote T_{round} as the average round trip delay between the client and the server. Let t_1 be the average frame interval when game is executed at client itself, t_2 be the average frame interval when game is rendered through the collaborative rendering platform, and t_3 be the average frame interval when game is rendered through the collaborative rendering platform where GPU collaboration is applied (i.e., local GPU is partially on for generating feedback data for client CPU). The corresponding frame rates for these three settings then are $1/t_1$, $1/t_2$, and $1/t_3$. Denote p as the probability that in a game when generating the next OpenGL command, the CPU needs GPU feedback.

3.1 Basic Collaborative Rendering Platform

We calculate when client and server have different settings (i.e., CPU and GPU), the average frame interval in the local graphic rendering system and the collaborative rendering platform.

We first neglect the network factor (e.g., we assume a high speed network is in use). When client CPU is the bottleneck for game frame generation, i.e., when $T_{cpu} > T_{gpu1}$ and $T_{cpu} > T_{gpu2}$,

$$\begin{aligned} t_1 &= ((1-p)T_{cpu} + p(T_{cpu} + T_{gpu1})) \times n, \\ t_2 &= ((1-p)T_{cpu} + p(T_{cpu} + T_{round} + T_{gpu2})) \times n. \end{aligned} \tag{1}$$

For this case, the collaborative rendering platform brings computation gain in frame generation when $t_1 > t_2$, i.e., when $T_{gpu1} > T_{round} + T_{gpu2}$.

When client GPU is the performance bottleneck for rendering graphics locally at the client, i.e., $T_{cpu} < T_{gpu1}$, and client CPU is the performance bottleneck for rendering graphics through the collaborative rendering platform, i.e., $T_{cpu} > T_{gpu2}$,

$$\begin{aligned} t_1 &= ((1-p)T_{gpu1} + p(T_{cpu} + T_{gpu1})) \times n, \\ t_2 &= ((1-p)T_{cpu} + p(T_{cpu} + T_{round} + T_{gpu2})) \times n. \end{aligned} \tag{2}$$

For this case, the collaborative rendering platform brings performance gain when $T_{gpu1} > (1-p)T_{cpu} + p(T_{round} + T_{gpu2})$.

When client GPU is the performance bottleneck for rendering graphics locally, i.e., $T_{cpu} < T_{gpu1}$, and server GPU is the performance bottleneck for rendering over the collaborative rendering platform, i.e., $T_{cpu} < T_{gpu1}$,

$$\begin{aligned} t_1 &= ((1-p)T_{gpu1} + p(T_{cpu} + T_{gpu1})) \times n, \\ t_2 &= ((1-p)T_{gpu2} + p(T_{cpu} + T_{round} + T_{gpu2})) \times n. \end{aligned} \tag{3}$$

For this case, remote rendering platform brings computation gain only when $T_{gpu1} > T_{gpu2} + pT_{round}$.

For all of the above cases, if we consider a slow network and communication round trip time is the bottleneck, i.e., if $T_{round} > max\{T_{cpu}, T_{gpu2}\}$,

$$t_2 = ((1-p)T_{round} + p(T_{cpu} + T_{round} + T_{gpu2})) \times n. \tag{4}$$

3.2 Collaboration with Local GPU Partially on

When GPU feedback is required for CPU to generate OpenGL commands, GPU at the client can be partially on and collaborate with the GPU at the server to improve the game performance. If CPU is the performance bottleneck, the average frame interval for collaborative system, t_3, can be calculated as

$$t_3 = ((1-p)T_{cpu} + p(T_{cpu} + T_{gpu1})) \times n. \tag{5}$$

If the server GPU is the bottleneck,

$$\begin{aligned} t_3 &= ((1-p)T_{gpu2} + max\{p(T_{cpu} + T_{gpu1}), pT_{gpu2}\}) \times n \\ &= ((1-p)T_{gpu2} + p(T_{cpu} + T_{gpu1})) \times n. \end{aligned} \tag{6}$$

If the communication delay is the bottleneck,

$$t_3 = ((1-p)T_{round} + max\{p(T_{cpu} + T_{gpu1}), pT_{round}\}) \times n. \tag{7}$$

4 Performance Evaluation

4.1 Numerical Results

Based on the above analysis, we evaluate the frame rate per second (FPS) increase in the collaborative rendering system. We use the frame increase ratio, i.e., the ratio between the FPS obtained in a collaborative system and the FPS obtained when client executes the game individually for performance gain evaluation. We set the average CPU computing time for generating an OpenGL call as the basic unit, i.e., $T_{cpu} = 1$. We assume the GPU at server is stronger than that at client, and $T_{gpu1} = 5T_{gpu2}$. For round trip delay, we use $T_{round} = 10T_{cpu}$ for a shorter round trip delay, and $T_{round} = 20T_{cpu}$ for a longer one. We assume the average number of OpenGL commands for generating a frame is 10^4, and the probability that an OpenGL command generation depends on GPU feedback, p, is 1.0×10^{-5}.

Figure 3 shows the performance gain in the collaborative rendering system. The x-axis indicates the computing speed of client CPU referred to client GPU. The collaboration brings computation gain only when the y-axis, i.e., the frame increase ratio has a value greater than 1. Basically, the collaborative rendering system helps to improve FPS gain when the client GPU becomes computation bottleneck, i.e, when $T_{gpu1} > T_{cpu}$. In the system where the client GPU is off, the collaborative system brings performance gain when compared to the client GPU, the client CPU is fast enough. The gain from a system with a longer round trip delay is less than that with a shorter one. When applying GPU collaboration and have client GPU partially on, the FPS in the collaborative system is at least the same as that obtained on a single device. GPU collaboration helps to achieve a much better gain especially when CPU gets stronger. However, when CPU is strong enough and server GPU becomes the bottle neck (i.e, $T_{gpu2} > T_{cpu}$), speeding CPU no longer results in much more gain.

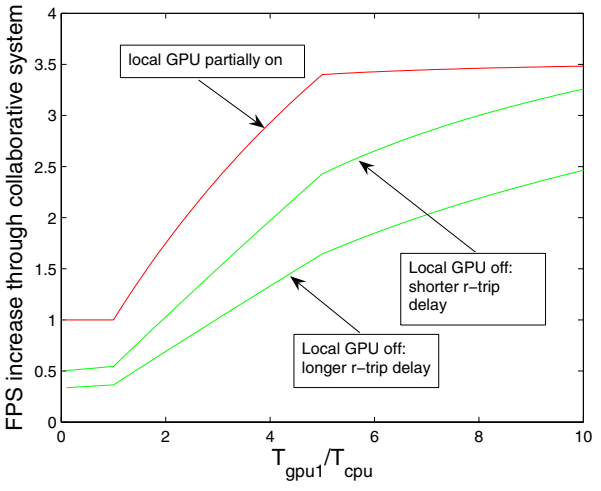


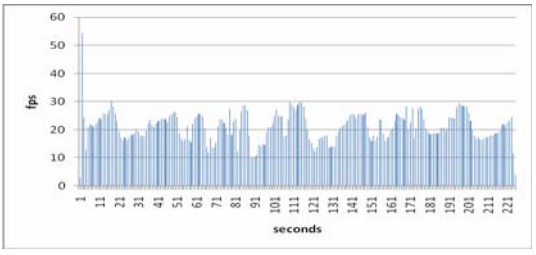
Fig. 3. FPS improvement through collaborative system

4.2 Measurement: Tocrs over Collaborative Rendering Platform

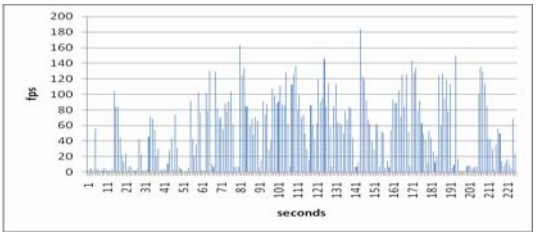
Unless otherwise specified, we use Sony UMPC as the client and $T61p$ laptop as the server in the collaborative rendering system. On UMPC, CPU is Intel core (TM)solo $u1500@1.33GHZ$. Graphic card is Intel 945 card. The OpenGL version is mesa library 6.5. $T61p$ has an Intel core 2 Due $L7700$ CPU, and a Graphic card NVIDIA Quadro FX $570M$. The OPenGL version on $T61$ is 7.5. A car racing game Torcs is loaded and executed at UMPC. The game pictures are rendered at $T61$ and shown on its larger display. In the experiment we only measure the game that a single view has been displayed at the server. We use 100Mb/s wired network for connection between client and server.

Figure 4 shows FPS statistics under different system scenarios. Their correspondent average FPS over a time period is shown in Figure 5.

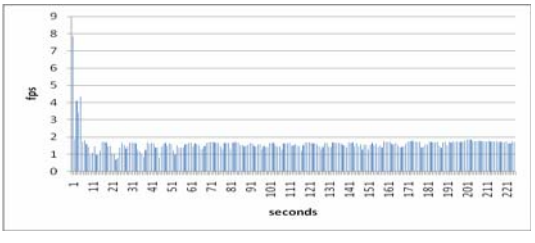
Figure 4 (a) shows the FPS when game runs locally on UMPC. This reference performance is used to find out whether collaborative rendering can help to improve game performance. When exporting OpenGL commands to a stronger GPU for rendering the graphics, as shown in Figure 4 (b), if p, the probability that generating an OpenGL command requires feedback from GPU, is a small value, the performance for the collaborative rendering platform is better. Compared to the reference case, collaborative rendering can help to improve the game performance because the gain of high-speed rendering at the server (thanks to its stronger GPU) overwhelms the lost on the delay caused by GPU feedback. However, the obtained FPS is bursty. When p increases, as shown in Figure 4 (c), the collaborative rendering platform results in a very poor performance, because the round trip delay at each of feedback from the server GPU to the client CPU will significantly delay the frame generation.



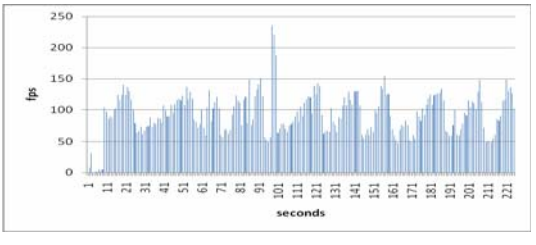
(a) Local Rendering



(b) Remote rendering ($p = 0.0002$)



(c) Remote rendering ($p = 0.0016$)



(d) Remote rendering with GPU collaboration

Fig. 4. Frame rate per second under cases of a)game rendered locally, b)game rendered remotely with a low feedback probability, c)game rendered remotely with a high feedback probability, and d)game rendered remotely with GPU collaboration

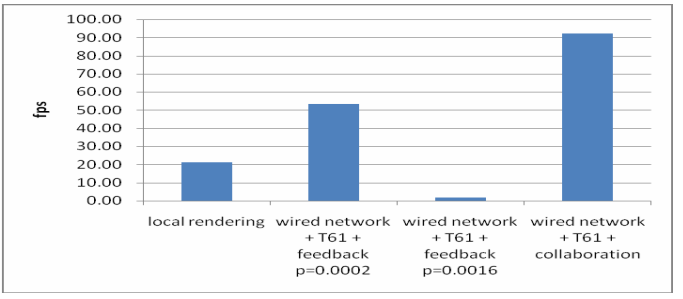


Fig. 5. Average frame rate for different cases

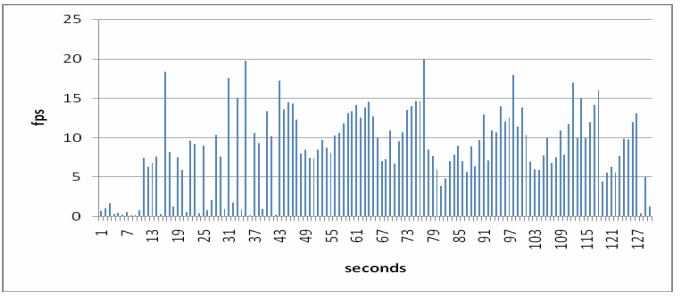


Fig. 6. FPS when having a slower network

To deal with the problem observed from Figure 4 (c), we use our proposed GPU collaboration scheme and turn on the GPU at the client as well, to calculate the graphic results that are required by OpenGL command generations. As shown in Figure 4 (d), through GPU collaboration the average frame rate is much higher than the reference case. The reason is that no feedback from the server is needed. Therefore, the negative impact from the feedback delay has been removed. The load on client GPU, on the other hand, is small so that it will not take much time for it to accomplish its assignment. Actually in this setting the CPU at the client is the bottleneck. Another observation is that as there is no feedback from remote side, the FPS statistics are smooth and not bursty at all.

Figure 6 shows when wireless network (802.11g) has been used instead of wired network, the FPS will be low even when client GPU is partially on. Compared to the system with a wired network, the average FPS has decreased to 8.57, a significant decrease compared to the average FPS 92.36 in the wired system. The results indicate that though wireless network does have advantages of convenience, its limited bandwidth slows down the frame generation.

Figure 7 shows when server has a relatively weaker GPU, how the collaborative rendering performance suffers. When using $T43$ laptop that has a slower GPU than $T61$, the average FPS decreases to 55.5.

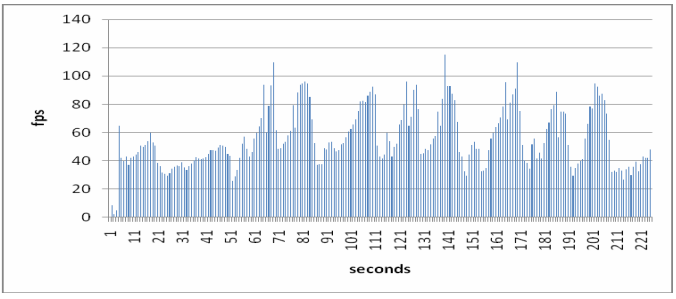


Fig. 7. FPS when having a weaker server

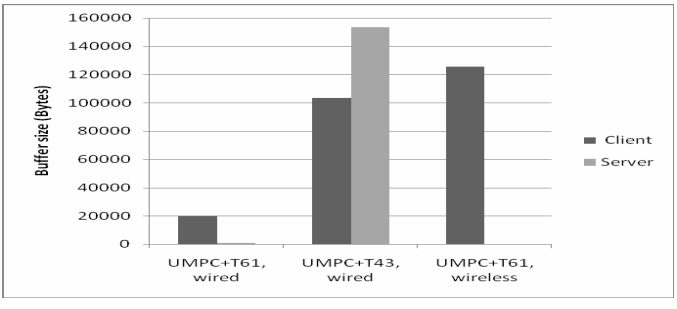


Fig. 8. Buffer size for different settings

Finally, we measure the buffered data at both client and server to further analyze the results we have obtained, thus to figure out the performance bottlenecks. As shown in Figure 8, in the setting when a UMPC collaborates with a T61, the buffered data at both client and server sides is not much. This indicates that neither network nor server is the performance bottleneck. When the computing capability at the client increases, a FPS improvement in the system should be expected. Not shown in the paper, this has been proved in our test when using T43 as a client, under which the FPS increases. The figure also shows when server has a weaker GPU (a T43 instead of a T61), there will be more buffered data at the server side. The server then is the performance bottleneck as the relatively long GPU processing time on OpenGL calls delivered to the server will make the server buffer full, which results in a full buffer at the client as well because the generated OpenGL calls from the client CPU cannot be delivered to the server. The full buffer at the client consequently impedes the frame generation. In this case, using a stronger sever can improve the game performance. When using wireless network, there is a large amount of data buffered at the client yet little data buffered at the server. This means the network should be the performance bottle neck, as for each second, only a small number of OpenGL commands can be delivered to the server. In this case, using a faster network can improve the game performance.

5 Conclusion

In this paper we design and evaluate a collaborative rendering platform, under which OpenGL graphics generated at a device can be rendered at a network-connected device with a larger display and a better GPU. Such a collaborative system can help to overcome the size limit of small devices (e.g., UMPC). Based on analysis and real system experimental results, we prove that the collaborative rendering system can improve game performance in frame rate, and identify the factors that may affect the platform performance, which may be a slow CPU at client, a slow GPU at server, and a slow network. We also find that in a game if a large amount of GPU feedback is required for game calculation, the performance will significantly degrade. To deal with this problem, we propose to partially turn on the GPU at the client to collaborate with the GPU at the server for speeding up the games. Experiment results show that the GPU collaboration can greatly improve frame rate by avoiding the impact of feedback delay. The GPU collaboration can smooth the game FPS as well.

Acknowledgement

We thank Lakshman Krisnurmathy for his guide on architecture design of remote rendering platform. We also thanks Khanh Nygeun and Zan Ding for their contribution to code development.

References

1. Abramson, D., Giddy, J., Kotler, L.: High Performance Parametric Modeling with nimrod/g: Killer Application for the Global Grid? In: Proceedings of the 14th International Parallel and Distributed Processing Symposium, IPDPS 2000 (2000)
2. Buyya, R., Abramson, D., Nimrod/G, G.J.: An architecture for a resource management and scheduling system in a global computational Grid. In: Proceedings of the International Conference on High Performance Computing in Asia–Pacific Region (2000)
3. Chervenek, A., Forster, I., Kesselman, C., Salisbury, C., Tuecke, S.: The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. Journal of Network Computer Applications 23(3), 187–200 (2000)
4. Vazhkudai, S., Ma, X., Freeh, V., Strickland, J., Tammineedi, N., Simon, T.A., Scott, S.L.: Constructing Collaborative Desktop Storage Caches for Large Scientific Datasets. ACM Transaction on Storage, TOS (2006)
5. Georgakopoulos, D., Schuster, H., Cuchoicki, A., Baker, D.: Managing Process and Service Fusion in Virtual Enterprises. Information System, Special Issues on Information System Support for electronic Commerce 24(6), 429–456 (1999)
6. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality Driven Web Services Composition. In: Proceedings of WWW 2003 (2003)
7. Krauter1, K., Buyya, R., Maheswaran, M.: A taxonomy and survey of grid resource management systems for distributed computing. Software–Practice and Experience 32, 135–164 (2002)
8. Ahuja, S.P., Myers, J.R.: A Survey on Wireless Grid Computing. The Journal of Supercomputing 37, 3–21 (2006)

9. OpenGL - The Industry Standard for High Performance Graphics, http://www.opengl.org/
10. Lamberti, F., Zunino, C., Sanna, A., Fiume, A., Maniezzo, M.: An accelerated remote graphics architecture for PDAS. In: Proceedings of the Eighth international Conference on 3D Web Technology (2003)
11. Yang, S.J., Nieh, J., Selsky, M., Tiwari, N.: The Performance of Remote Display Mechanisms for Thin-Client Computing. In: Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (2002)
12. Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P.D., Klosowski, J.T.: Chromium: a stream-processing framework for interactive rendering on clusters. In: Proceedings of the 29th Annual Conference on Computer Graphics and interactive Techniques (2002)
13. Stegmaier, S., Magallón, M., Ertl, T.: A generic solution for hardware-accelerated remote visualization. In: Proceedings of the Symposium on Data Visualisation (2002)