Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering

Chun-Fa Chang and Shyh-Haur Ger

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan, R.O.C.
chunfa@cs.nthu.edu.tw

Abstract. Compared to a personal computer, mobile devices typically have weaker processing power, less memory capacity, and lower resolution of display. While the former two factors are clearly disadvantages for 3D graphics applications running on mobile devices, the display factor could be turned into an advantage instead. However the traditional 3D graphics pipeline cannot take advantage of the smaller display because its run time depends mostly on the number of polygons to be rendered. In contrast, the run time of image-based rendering methods depends mainly on the display resolution. Therefore it is well suited for mobile devices. Furthermore, we may use the network connection to build a client-server framework, which allows us to integrate with non-image-based rendering programs. We present our system framework and the experiment results on PocketPC® based devices in this work.

1. Introduction

With the recent advances in processing power and memory capacity, small portable or handheld devices have emerged as a popular computing platform. Nowadays, typical handheld devices are capable of supporting graphical user interface, audio and video playback, and wireless communication. These new capabilities also open up new areas of applications for handheld devices.

However, rendering three-dimensional (3D) graphics on handheld devices is still considered a formidable task. Because of the vast computational power that is required by 3D graphics applications, even a desktop personal computer or workstation often replies on dedicated hardware and architecture design (such as Intel AGP interface) for 3D graphics to achieve real-time performance. Currently those dedicated hardware supports are still lacking in handheld or mobile devices.

There are actually several implementations of the traditional polygon-based 3D graphics pipeline on mobile devices today. Two examples are the miniGL [6] on Palm OS platform and Pocket GL [8] on Microsoft PocketPC platform. They are both subsets of the popular OpenGL API [7]. Currently their performances are still limited. The performance of Pocket GL is considerably faster than miniGL, mostly due to the fact that the PocketPC devices have more processing power than the Palm

devices. Even so, the polygon counts of 3D models that Pocket GL can display at interactive rates are still limited.

This reveals a fundamental issue of the polygon-based 3D graphics pipeline: its rendering time increases linearly with the number of polygons that enter the pipeline. Although we may expect future generations of mobile devices to be equipped with more processing powers, there will also be more complex models with higher polygon counts to be rendered.

In this paper we explore an alternative approach, image-based rendering, to achieve the 3D graphics capability on mobile devices. Unlike the polygon-based 3D graphics pipeline, the rendering time of image-based rendering depends on the screen resolution of the output images rather than the complexity of the input models. This offers a potential advantage for mobile devices that typically have small display areas.

We also present a client-server framework for mobile devices that we equipped with networking capability, e.g., via the IEEE 802.11b based wireless network. Using our framework, a 3D graphics programs (which do not need to use image-based rendering) running on a desktop computer may be integrated with our system to interact with users on mobile devices. This can simplify the process of developing 3D graphics software on mobile devices and offer a way to offload part of the 3D rendering task to the server.

2. The 3D Warping Algorithm

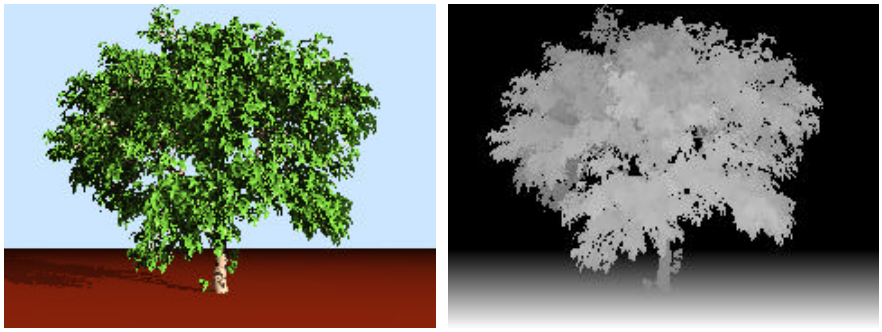

Fig. 1. An example of input depth images. **Left**: the color components. **Right**: the depth components.

The image-based rendering technique that we use in our work is McMillan's 3D warping method [4][5]. The inputs to 3D warping are depth images, which are 2D color images containing depth information at each pixel. Each depth image also contains a viewing matrix (3×4 as described in [5]) that describes the camera or viewing setup. Figure 1 shows an example where the image on the left shows the

color components of the depth image and the image on the right shows the depth components in grayscales.

Compared to the traditional 3D graphics pipeline, 3D warping demands much less computing power. The core of the 3D warping algorithm is the following warping equation:

$$(u_2, v_2) = (\frac{u_1 a + v_1 b + c + d_1 d}{u_1 i + v_1 j + k + d_1 l}, \frac{u_1 e + v_1 f + g + d_1 h}{u_1 i + v_1 j + k + d_1 l}) \quad \textbf{(1)}$$

The warping equation calculates the coordinates (u_2, v_2) on output image for each input pixel at (u_1, v_1). The variable δ_1 is the depth information (or the disparity) of the input pixel. The variables *a* through *l* are controlled by the viewing matrices of the input and output images. They are recomputed only when the view of either the input or the output image changes. Therefore they remain constant across pixels of the same output image.

Because the warping equation is computed once for each pixel, the time complexity of 3D warping is $O(n^2)$ where *n* represents the image resolution in horizontal or vertical direction. It is independent of the scene complexity that is usually measured by the number of polygons in the scene. Although the image resolution refers to the input image here, it is actually more closely related to the output image as demonstrated at [1]. This is good news for the small screen sizes of typical mobile devices. Furthermore, the warping equation is easy to compute as it involves only 20 arithmetic operations[1].

When there is only a single input image, the output image is likely to exhibit the occlusion (or exposure) artifact, which is caused by revealing parts of the 3D scene that are occluded in the input image. To avoid such a problem, the input data format may be extended in a fashion that is similar to the Layered Depth Image [11] or Layered Depth Cube [3]. We implement the Layered Depth Image in our system. However, in order to simplify the discussion, we describe our work as if regular single-layered depth images were used (except when we present the results).

3. System Framework

First, we describe the stand-alone (non-networked) version of our system, which consists of two parts: a model constructor and an interactive warper. Their roles are described in Sections 3.1 and 3.2. Then we describe in Section 3.3 how it is extended to a client-server framework when network connection is available.

[1] Note that the two denominators in the warping equation are the same.

3.1. Model Constructor

Usually the 3D models to be displayed are initially provided by the users as a set of polygons. The job of the model constructor is to convert those 3D models into depth images that are amicable to 3D warping. The model constructor can be considered as a preprocessing step. Therefore it may run on desktop computers rather than on mobile devices. There are many ways to construct the depth images from 3D polygons. In this paper, we modify the POV-RAY ray-tracing program [9] to build the depth images. An alternative is to render the 3D models in OpenGL [7], then combine the resulting frame buffer and depth buffer into a depth image.

The file format of our depth images is simply a concatenation of the image size, the viewing matrix, the color components, and the depth components. No data compression is currently used.

3.2. Interactive Warper

The actual 3D warper runs on mobile devices to accept user input and display the new views interactively. It is an implementation of the 3D warping algorithm that were described in Section 2.

If we traverse the pixels of an input depth image in a particular order, then we can guarantee that the pixels are warped to the output image in back-to-front order. This technique is called the occlusion compatible order by McMillan in [5], and is implemented in our system. Its implementation also means that we do not need the Z-Buffer for hidden surface removal.

The warping equation involves floating-point arithmetic. However most mobile devices do not have floating-point units in their processors. Therefore we use fixed-point number representations in our warping equation, which results in about 350% speedup. (The frame rate improves from about 1.7 frame/second to about 6.0 frame/second in one of our tests.)

When an input pixel is warped to the output image, we simply copy its color to the new output pixel. This could produce gaps between neighboring pixels such as those shown in Figure 2. We can avoid those artifacts by drawing each pixel as a circle that is slightly larger than a pixel, or by using the splatting techniques described in [13] or [10]. However splatting is not currently implemented in our system. We plan to support it in the future using a look-up table method similar to [11].

3.3. Extension to a Client-Server Framework

In Sections 3.1 and 3.2, we have described the stand-alone (non-networked) version of our system. Once the input depth images are constructed, they are loaded to the mobile devices and become static. However this is no longer the case if the networking capability is available on the mobile devices.

When the mobile devices are equipped with networking capability, we can build a client-server framework, where the client is the interactive 3D warper running on

mobile devices and the server is a dynamic model constructor running on a more powerful computer such as a desktop workstation. In this framework, the user's interactions with the client are periodically sent to the server via the network. Then the server updates the depth image based on user's current view and transmits the new depth image to the client. The features of this client-server framework are:

1. The client can hide the network latency by performing 3D warping to update the display at interactive rates. Even when the network is down and the server fails to update the input depth image, the client can still work in stand-alone mode.
2. The server may take advantage of the specialized 3D graphics hardware on the desktop workstations.
3. Most importantly, the client-server framework makes it possible to modify an existing 3D graphics program (on desktop computers) to display its results and interact with users on mobile devices.

4. Results

For the stand-alone version, we modify the POV-RAY program to produce depth images for our 3D warper. The depth images are generated on desktop computers and then downloaded to mobile devices where the 3D warper resides. We build and test our 3D warper on Microsoft PocketPC®-based mobile devices, such as the Compaq iPaq H3800 series Pocket PC.

The output images may be displayed via either the GDI functions or the Game API [2] of Windows CE. We opt for the Game API because we found that the GDI functions incur too much operating system overhead.

Figure 2 shows results of the 3D warper using the input model that is shown in Figure 1. The original 3D model contains more than 37,000 primitives, which would be too complicated to be rendered interactively on current mobile devices using the traditional 3D graphics pipeline. However our system is able to render it at the speed of 5.9 to 6.2 frames per second on a 206MHz StrongArm processor based system.

We also implement the layered depth images in our work. Figure 3 shows how the layered depth images reduce the occlusion artifacts.

For the networked (client-server) version, we modify an OpenGL program to continuously generate depth images from the frame buffer (including the depth buffer). The OpenGL program acts as our server and communicates with the client program on an iPaq Pocket PC via IEEE 802.11b based wireless network. Whenever a depth image is ready, the server sends it to the client and queries the client for the current user's view, which is used to generate the next depth image. The client simply uses the most recently received depth image as the input data and performs 3D warping to update the display at interactive rates, regardless how fast the input depth image can be updated. Figure 4 shows the results. The image on the left shows the user changes his/her view on the Pocket PC. The image on the right shows that the server program has updated its view accordingly and the newly generated depth image is now used on the client.

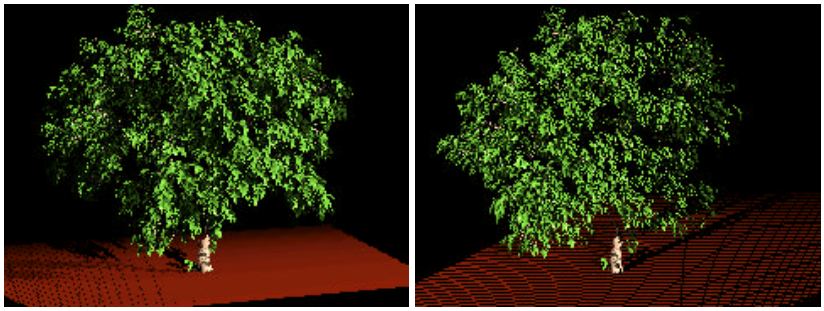

Fig. 2. Output images (in 240×180 resolution) of the 3D warper for two different user's views. The input model is described in Figure 1.

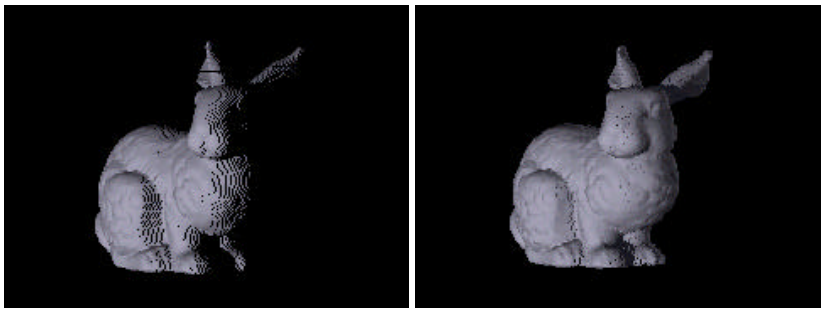

Fig. 3. Using layered depth images can reduce the occlusion artifacts. The image on the left is produced with a regular single-layered depth image. The image on the right is produced with a layered depth image that combines images from four different views.

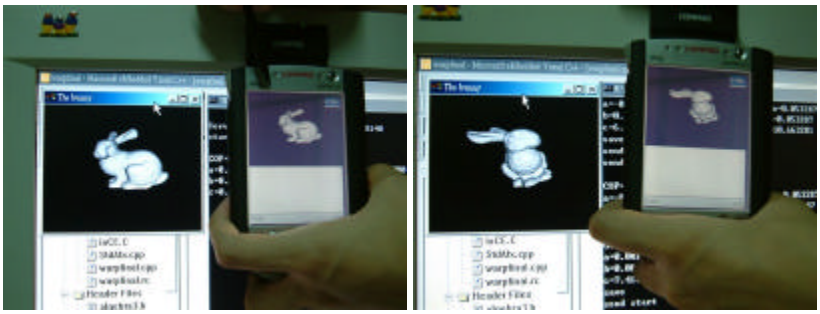

Fig. 4. The networked version of our system at work. The image on the left shows that the user is changing his/her view on the Pocket PC. The image on the right shows that the server program has updated its view accordingly and the newly generated depth image is used on the client.

5. Conclusions and Future Work

We have presented an alternative approach to accomplish 3D rendering on mobile devices. It takes advantage of the smaller display areas of mobile devices and is capable of rendering complex 3D models because its performance does not degrade for 3D models with large polygon counts.

This work also represents the first step in our ongoing effort to build a client-server 3D rendering framework for mobile devices in networked environment. In the future, we hope to release a library that will requires only minimal effort to port any existing 3D rendering program such as those written in OpenGL or DirectX to interact with users on mobile devices, without the users noticing that most of the rendering is actually done on a remote server.

6. Acknowledgement

We would like to thank Professor Shi-Nine Yang for pointing out some interesting 3D graphics applications on mobile devices. Thanks also to Zhe-Yu Lin and Yi-Kai Chuang for various help in programming. This work is supported by R.O.C. DOE Grant 89-E-FA04-1-4 (Program for Promoting Academic Excellence of Universities) and NSC Grant 91-2213-E-007-032.

References

1. Chun-Fa Chang, Gary Bishop and Anselmo Lastra. "LDI Tree: A Hierarchical Representation for Image-Based Rendering". In *SIGGRAPH 1999 Conference Proceedings*, pages 291–298, August 1999.
2. The Game API website: http://www.pocketpcdn.com/sections/gapi.html
3. Dani Lischinski and Ari Rappoport. "Image-Based Rendering for Non-Diffuse Synthetic Scenes". *Rendering Techniques '98 (Proc. 9th Eurographics Workshop on Rendering)*, June 29–July 1, 1998.
4. Leonard McMillan and Gary Bishop. "Plenoptic Modeling: An image-based rendering system". In *SIGGRAPH 95 Conference Proceedings*, pages 39–46, August 1995.
5. Leonard McMillan. *An Image-Based Approach to Three-Dimensional Computer Graphics*. Ph.D. Dissertation. Technical Report 97-013, University of North Carolina at Chapel Hill, Department of Computer Science, 1997.
6. MiniGL by Digital Sandbox, Inc. The miniGL website: http://www.dsbox.com/minigl.html
7. OpenGL website: http://www.opengl.org
8. PocketGL website: http://www.sundialsoft.freeserve.co.uk/pgl.htm
9. POV-RAY website: http://www.povray.org
10. Szymon Rusinkiewicz and Marc Levoy. "QSplat: A Multiresoluton Point Rendering System for Large Meshes". In *SIGGRAPH 2000 Conference Proceedings*, pages 343–352, July 2000.
11. Jonathan Shade, Steven Gortler, Li-wei He and Richard Szeliski. "Layered Depth images". In *SIGGRAPH 98 Conference Proceedings*, pages 231–242, July 1998.

12. S. Teller and C. Sequin. "Visibility Preprocessing for Interactive Walkthroughs" In *SIGGRAPH 91 Conference Proceedings*, pages 61–70, July 1991.
13. Lee Westover. *SPLATTING: A Parallel, Feed-Forward Volume Rendering Algorithm*. Ph.D. Dissertation. Technical Report 91-029, University of North Carolina at Chapel Hill. 1991.