A Thin-Client Approach for Porting OpenGL Applications to Pocket PC's

Zhe-Yu Lin Shyh-Haur Ger Yung-Feng Chiu Chun-Fa Chang
Department of Computer Science
National Tsing Hua University

Abstract

The display of a mobile device such as the pocket PC is usually small, which inspired us to explore image-based rendering as an alternative for 3D graphics on mobile devices. However, developing 3D graphics applications on mobile devices could be a very different experience from programming on personal computers, let alone using image-based rendering. Therefore we developed an application interface (API) for porting 3D graphics applications that are already written in OpenGL [6] to the Pocket PC platform. It is based on a client-server framework where the original program still runs on the PC server, and the Pocket PC acts as a thin client to interact with the end users. Image-based rendering is implemented in the client to improve the frame rates and to cope with the network delay.

1 Introduction

The user's demand for 3D graphics on mobile devices has been increasing. However, developing 3D graphics applications on mobile devices such as the Pocket PC's could be a very different experience from programming on personal computers, even with help of graphics libraries such as miniGL[5] or PocketGL[7]. Furthermore, insufficient processing power of mobile devices makes it impractical to render complex 3D models through the traditional graphics pipeline.

Let us look at the scenario of developing computer games with 3D graphics on Pocket PC's. Simply porting the traditional 3D graphics pipeline to Pocket PC's produces limited and non-scalable rendering performance because the rendering time increases almost linearly with the number of polygons to be rendered. More importantly, it fails to take advantage of the smaller display on a typical Pocket PC. In contrast, the run time of image-based rendering (IBR) method depends mainly on the display resolution, which makes it well suited for Pocket PC's.

In this work, we present an application interface (API) for porting 3D graphics applications written in OpenGL [6] to the Pocket PC platform. It is based on a client-server framework where the original program still runs on the PC server, and the Pocket PC acts as a thin client to interact with the end users.

The main ideas of our work are: we want to give the users the illusion that they are running the OpenGL programs on the Pocket PC when the users interact with the application program through our pre-compiled client-side program on the Pocket PC, even though the OpenGL program is actually running on the PC

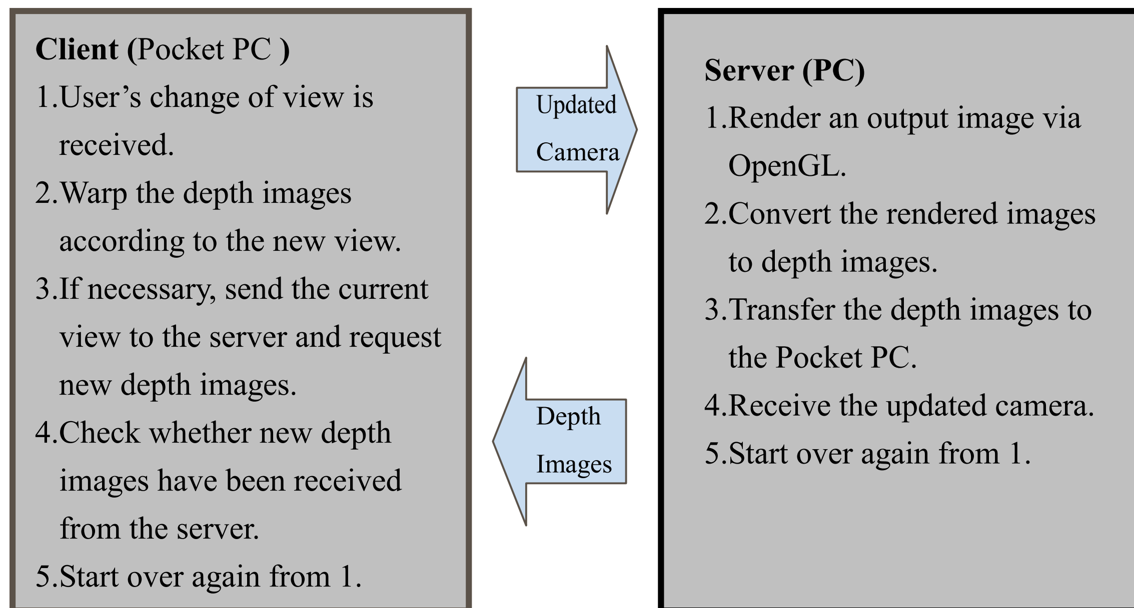


Figure 1: Overview of our client-server framework.

server. When the user's view changes, the updated view is sent back to the OpenGL application on the PC server. The server-side program receives the view change, renders the new view, and then sends the updated frame buffer to the client. The client program uses the received frame buffer as the input to the image-based rendering to display the final output. Figure 1 shows an overview of the client-server framework.

In a sense, our client-side program acts like a virtual frame buffer (or a video player) for the server program, especially if the network bandwidth is abundant. However, in practice, the network bandwidth is not sufficient to support such a kind of frame-by-frame playback. Thus, we have to rely on image-based rendering techniques to maintain interactive frame rates at the client side.

In the following sections, we describe the usage of our application interface, details of its implementation, and the results of porting several applications using our API.

2. Usage of the API Functions

We design our application interface such that a developer can convert an existing OpenGL application into a Pocket PC application with as few changes as possible. Our application interface communicates with an OpenGL application mainly through the following data structures:

1. The frame buffer: The OpenGL application still performs the rendering as usual. The rendered images in the frame buffer (including the color and depth buffers) are then converted to depth images and sent to the Pocket PC for 3D warping [3].
2. The camera setup: Since the user will be interacting with the client program on the Pocket PC, the OpenGL application needs to be modified to receive the camera setup from our API functions.

3. The UI-related commands: Similar to the cases in camera setup, the OpenGL application needs to invoke our API to receive the UI commands form the user.

The developer is expected to make the following changes to the original OpenGL application program to use our API functions:

1. Initialization and termination of the network.
2. Registration of callback functions.
3. Adding API function calls to receive the updated viewing setup and the other user interface commands such as those from pull-down menus.
4. Adding API function calls to send the rendered images at the frame buffer to the client.

The items 3 and 4 listed above are necessary because the user now interacts with the client program on the Pocket PC, not directly with the application on the server.

Currently, our system is also capable of automatically creating layered depth images (LDI) from a given viewpoint. We achieve this by shifting the camera to multiple nearby viewpoints and slightly rotating the viewing angles. However, the developer of an OpenGL application does not need to handle the details of creating an LDI from multiple rendering. All the developer has to do is to provide a display callback function, which is typically very similar to the GLUT display callback. Our application interface will then invoke the display callback function multiple times with the varied viewpoints to generate the LDI automatically.

To sum it up, a modified OpenGL application may look like the following (where the API-related modifications are shown in boldface):

```
main( ){
    [OpenGL setup]
    API_init_net();
    API_reg_display_callback(
         display_callback);
    API_reg_idle_callback(
         idle_callback);
    MainLoop();
    API_close_net();
} /* main() */

display_callback( ){
     [OpenGL rendering]
     glFlush();
} /* display_callback() */

idle_callback( ){
    switch(event){
    [cases for other events]
    case NET_RECEIVE_CAM:
       API_receive_cam();
       API_depth_img_gen();
       API_send_image();
    case NET_RECEIVE_UI_CMD:
      API_receive_UI_cmd();
       [actions for GUI command]
    }
} /* idle_callback() */
```

3. Implementation of the API

Our application interface may be divided into three categories: the server functions, the network functions, and the thin client. The server functions are responsible for converting

the OpenGL frame buffers into depth images. The network functions handle the communication between the server and the client. The thin client interacts with the users and displays the output images by image-based rendering. (Please see the appendix for a list of the API functions.)

3.1 The Server Functions

One of the server functions converts the OpenGL color and depth buffers into a depth image. Its task is relatively straightforward, except the part of converting the Z values in the depth buffer into the depth values in the depth image. The depth values stored in a depth image represent the distance to the camera and are proportional to the distance between the camera and the projection plane. Therefore, they are linear in the object space. Unfortunately, the Z values stored in the OpenGL depth buffer are more closely related to the distance to the near clipping plane, and are not linear in the object space. For example, if an object has the depth of Z along the z-axis in the eye space, the near plane distance is n and the far plane distance is f, then its depth value at the OpenGL depth buffer will be:

$$Z' = \frac{Z-n}{Z} \cdot \frac{f}{f-n}$$

To convert it to a linear depth value in the depth image, we use the following equation:

$$Z'' = \frac{f}{f - Z' \cdot (f-n)} \cdot$$

which eventually becomes Z/n.

3.2 The Network Functions

We implement the network functions between the server and the client using the Microsoft WinSock interface, via IEEE 802.11b based wireless network. The data that are sent on the wireless network are those that are already described in Section 2. As the user's view changes, the client sends the camera information back to the server. When the server receives the view change through a call to the network functions, the application program renders a new image, calls a server function (listed in Section 3.1) to convert the frames buffers into a depth image or Layered depth image, then calls another network function to send the depth image to the client.

3.3 The Thin Client

Our application interface includes a thin client program that actually interacts with the users and produces the displayed images. Therefore, the developer does not need to learn the details of programming in the Pocket PC environment. The provided thin client program handles the typical user input on the touch screen and the buttons. However, if a developer is familiar with the programming environment of the Pocket PC and would like to write a different client program, our thin client program provides a good example to start with.

The input to our client program is the depth image (or layered depth image) that it receives from the server. The client program then produces output images from the depth image using McMillan's 3D warping method [4]. To improve the performance, the floating-point arithmetic is implemented in fixed-point numbers, and the Game API [2] is used for pixel drawing. For more details about the implementation of the client program, please see our previous work in [1].

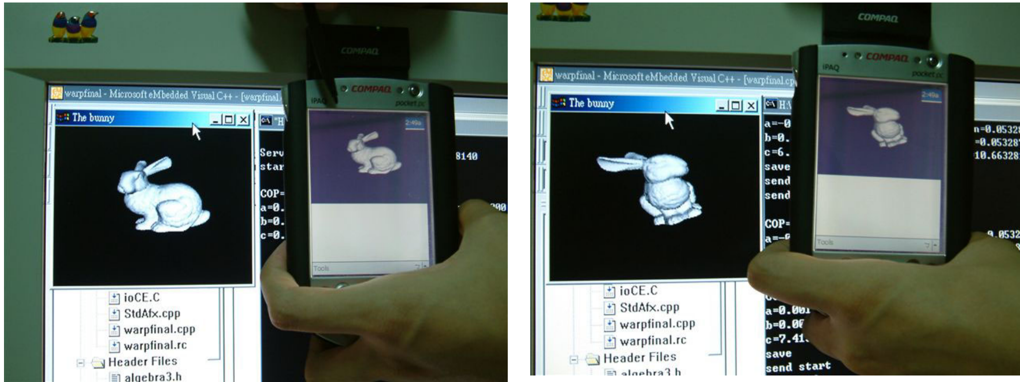
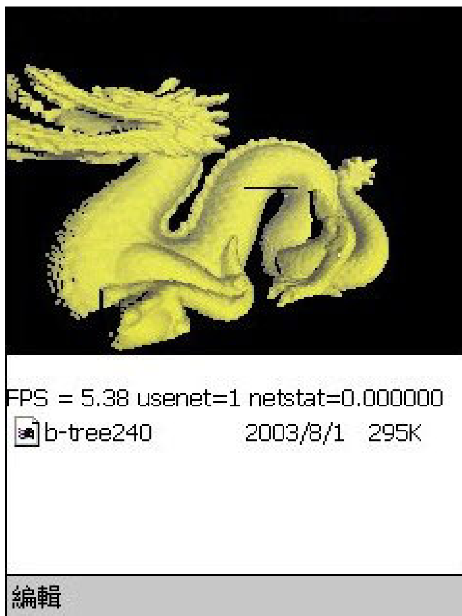


Figure 2: Our system at work. (Left) The user is changing his/her view on the Pocket PC. (Right) The server program has updated its view accordingly and the newly generated depth image is used on the client.



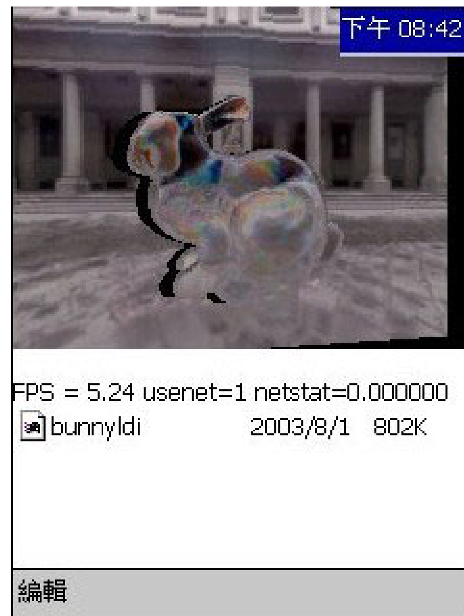


Figure 3: Two OpenGL programs running on the Pocket PC. (Left) The Dragon (Right) An NVIDIA shader demo.

4. Results

Figure 2 shows our system at work. The left image shows that the user is changing her or his view on the Pocket PC. The right image shows that the server program has updated its view accordingly and the newly generated depth image is used on the client.

Figure 3 show the results of porting two existing OpenGL applications using our application interface. As described in Section 2, porting them to the Pocket PC requires only minor changes to the original OpenGL program. The two applications are:

1. The Dragon: This is an application program that renders a dragon model, which contains 871,414 triangles.
2. An NVIDIA shader demo: This is a vertex shader demo program that demonstrates the refraction effect.

In our experiments, we obtain the

performance of about 5 frames per second for both applications on a 206MHz StrongArm processor based system (Compaq iPAQ H3870). The time to transmit a depth image from the server to the client ranges from half a second to two seconds. We use the IEEE 802.11b based wireless network equipments, all located within our laboratory.

5. Conclusions and Future Work

In this work, we demonstrate that image-based rendering provides an alternative approach to 3D graphics on a Pocket PC. Image-based rendering is appealing because its rendering cost is proportional to the resolution of the output image, which is usually low in mobile handheld devices. We also show that porting an existing OpenGL application to Pocket PC may be simplified by using our application interface.

A limitation of our method is that it does not work well with moving objects in the OpenGL application program. Although our client program can interactively display the new images when the view changes, the objects do not move in the object space until the next depth image from the server arrives. In the future, we plan to add a traditional polygon-based graphics pipeline to the client program in order to process the moving objects. As long as the scene contains only a small portion of moving objects, the benefit of using an image-based rendering approach may still be preserved.

Acknowledgement

This work is supported by NSC Grant 91-2213-E-007-032.

References

1. Chun-Fa Chang and Shyh-Haur Ger. "Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering". In Proceedings of 2002 IEEE Pacific-Rim Conference on Multimedia (PCM 2002).
2. The Game API website: http://www.pocketpcdn.com/sections/gapi.html
3. Leonard McMillan and Gary Bishop. "Plenoptic Modeling: An image-based rendering system". In *SIGGRAPH 95 Conference Proceedings*, pages 39–46, August 1995.
4. 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.
5. MiniGL by Digital Sandbox, Inc. The miniGL website: http://www.dsbox.com/minigl.html
6. OpenGL website: http://www.opengl.org
7. PocketGL website: http://www.sundialsoft.freeserve.co.uk/pgl.htm
8. Jonathan Shade, Steven Gortler, Li-wei He and Richard Szeliski. "Layered Depth images". In *SIGGRAPH 98 Conference Proceedings*, pages 231–242, July 1998.
9. 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.

Appendix: List of API Functions

API_init_net(socket, port);

/* initialize the wireless network and bind the socket to the selected port , return 0 to mean failure */

API_close_net(socket);

/* release the socket and close the wireless network */

API_reg_display_callback(display_callback());

/* register the display callback function */

API_reg_idle_callback(idle_callback());

/* register the idle callback function */

API_receive_cam();

/* change the camera parameters (eye position, look-at and up vectors) */

API_depth_img_gen();

/* generate the depth image or layered depth image from color and depth buffers according to the new camera settings */

API_receive_UI_cmd();

/* invoke actions (model selection , exit, etc,.) to the UI commands from the user */

API_send_image(socket,buffer, count);

/* send "count" bytes from the buffer to the socket */