Providing Direct3D Features over the Desktop OpenGL

Nakhoon Baek[1] and Kwan-Hee Yoo[2,*]

[1] *Kyungpook National University, Daegu 702-701, Korea*
oceancru@gmail.com

[2] *Chungbuk National University, Cheongju Chungbuk 361-763, Korea*
** corresponding author: khyoo@chungbuk.ac.kr*

Abstract

In this paper, we aimed to provide Direct3D graphics features on Linux-based systems, which are actively used for various portable game platforms and mobile phone devices. Direct3D is used as one of the most important middle-wares for game and graphics applications developed on Microsoft Windows operating systems. However, this graphics library is not commonly available for other operating systems. We present a prototype library to provide Direct3D functionalities on Linux-based systems, using the OpenGL graphics library. In typical Linux-based systems, only the X window system and OpenGL graphics library are available. There are lots of needs to port Direct3D-based applications on these systems, and our Direct3D-on-OpenGL library would be a good starting point. Selecting a set of widely-used Direct3D data structures and functions, we implemented selected Direct3D functionalities and finally acquired a prototype implementation. Our implementation currently covers 3D transformations, light and material processing, texture mapping, simple animation features and more. We showed its feasibility through successfully executing a set of Direct3D demonstration programs on our implementation.

Keywords: *DirectX, OpenGL, Implementation, black-box testing.*

1. Introduction

In this paper, we present a prototype implementation of *Direct3D* graphics functionalities on *Linux*-based systems. Notice that the Linux-based systems are now used for various portable game platforms and mobile phone devices[1,2,3]. In contrast, Currently, Direct3D is used as one of the most important library for graphics output, mainly for applications developed on *Microsoft Windows* operating systems[4]. In contrast, this graphics library is not commonly available for other operating systems. Thus, we are hard to use it on other operating systems, at least at this time.

As the first step to provide an easy way of porting Direct3D-based game applications to other operating systems, we designed and implemented a graphics library which provides Direct3D graphics API (application program interface) functions on Linux-based systems. Since this library can be used to directly port the graphics and game applications originally developed for PC desktops in a straight-forward manner, we expect it to be a cost-effective way of porting these programs.

As shown in Figure 1, our final goal is to get the same graphics output for both of the desktop Direct3D application programs and the new implementation of our *Direct3D-on-OpenGL* architecture, which corresponds to the right side of the figure.

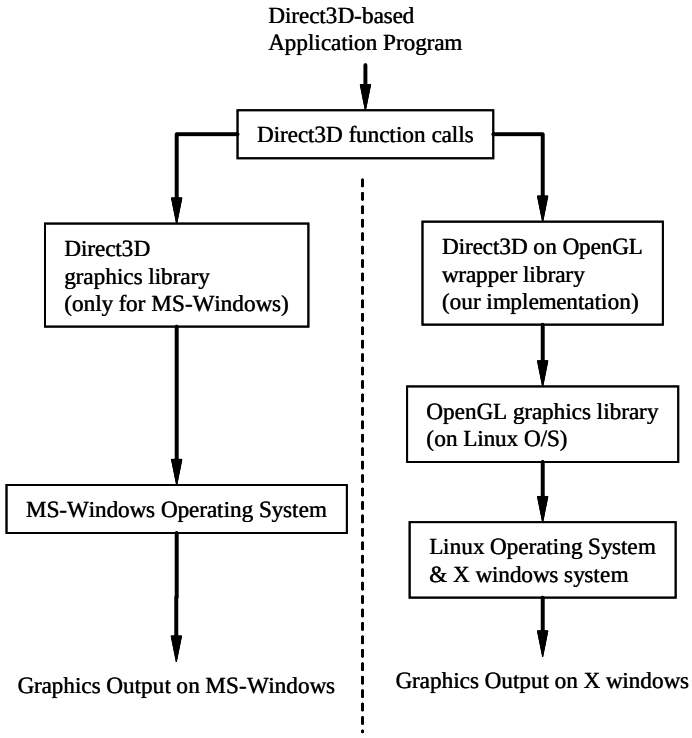


Figure 1. The Design Concept of Our System.

2. Related Works

Typical Linux-based systems, or especially for embedded Linux-based systems, they usually provide *OpenGL* (or its equivalent) for 3D graphics output. OpenGL is one of the most widely-used 3D graphics libraries, and continuously improved to reflect the current state of the art[5].

As an example, *OpenGL ES* is newly released for handheld devices including mobile phones[6]. OpenGL ES shows a good example of re-constructing a general purpose desktop 3D graphics library for small embedded systems[7]. We also take a similar way, to get a selected set of original Direct3D functions.

In the case of Microsoft Windows operating systems, they provide *DirectDraw* or Direct3D graphics libraries[4]. Recently, they are integrated into *DirectX* system, and widely used in Microsoft Windows systems.

As the developing environment for Windows systems, the Visual Studio system consistently provides supports for embedded systems, and currently Visual Studio for Embedded System is also available. However, these all facilities are only available for Microsoft Windows systems and thus, we need another way of providing Direct3D features on other operating systems such as Linux.

3. Design and Implementation

At the design stage of our *Direct3D-on-OpenGL* library, we need to select the set of supported functions among the original Direct3D features. Considering its technical and marketing aspects, we selected Direct3D 9.0 as the major target. From the technical point of

view, the *vertex shaders* and *pixel shaders*, which are based on the hardware GPU's, are excluded from the first prototype implementation. They will be added in the next implementations. In this way, we selected the *FVF(flexible vertex flags)*-based graphics programs as our first target.

Thus, our implementation naturally supports the following Direct3D classes:

- IDirect3D9 – The starting point of Direct3D programming. It generates IDirect3DDevice9 objects on demand.
- IDirect3DDevice9 – It handles core graphics primitives in Direct3D.
- IDirect3DTexture9 – Added to support texture mapping facilities.
- IDirect3DVertexBuffer9 – Providing coordinates and their related information for graphics primitives.
- D3DXMATRIX – Added for matrix processing.

During Implementation of those classes, the following classes are additionally needed:

- _D3DCOLORVALUE – Providing color information as (R, G, B, A)-quadraples.
- _D3DVECTOR – Providing 3D vectors with (x, y, z).
- _D3DRECT, _RGNDATA, _RGNDATAHEADER – Specifying regions as specific areas on the screen.
- _D3DLIGHT9 – Defining light sources for the light-and-material processing.
- _D3DMATERIAL9 – Defining material information for the light-and-material processing.
- _D3DPRESENT_PARAMETERS – Providing some features related to the overall screen updates.

To show the implementation details of these classes, the supported member functions of the IDirect3D9 class is listed as follows:

- IDirect3D9 (void)
- ~IDirect3D9 (void)
- ULONG Release (void)
- HRESULT CreateDevice (UINT adaptor, D3DDEVTYPE deviceType, HWND hFocusWindow, DWORD behaviorFlags, D3DPRESENT_PARAMETERS *pPresentationParameters, IDirect3DDevice9 **ppReturnedDeviceInterface)

As the next one, our IDirect3DDevice9 class has the following member functions:

- IDirect3DDevice9 (DWORD behaviorFlags, D3DPRESENT_PARAMETERS *pPresentationParameters)
- ~IDirect3DDevice9 (void)
- ULONG Release (void)
- HRESULT CreateVertexBuffer (UINT length, DWORD usage, DWORD fvf, D3DPOOL pool, IDirect3DVertexBuffer9 **ppVertexBuffer, HANDLE *pShareHandle)

- HRESULT BeginScene (void)
- HRESULT EndScene (void)
- HRESULT Clear (DWORD count, CONST D3DRECT *pRects, DWORD flags, D3DCOLOR color, float z, DWORD stencil)
- HRESULT SetStreamSource (UINT streamNumber, IDirect3DVertexBuffer9 *pStreamData, UINT offsetInBytes, UINT stride)
- HRESULT SetFVF (DWORD fvf)
- HRESULT DrawPrimitive (D3DPRIMITIVETYPE primitiveType, UINT startVertex, UINT primitiveCount)
- HRESULT Present (CONST RECT *pSourceRect, CONST RECT *pDestRect, HWND hDestWindowOverride, CONST RGNDATA *pDirtyRegion)
- HRESULT SetTransform (D3DTRANSFORMSTATETYPE state, CONST D3DMATRIX *pMatrix)
- HRESULT SetRenderState (D3DRENDERSTATETYPE state, DWORD value)
- HRESULT SetTexture (DWORD sampler, IDirect3DTexture9 *pTexture)
- HRESULT SetLight (DWORD index, CONST D3DLIGHT9 *pLight)
- HRESULT LightEnable (DWORD lightIndex, BOOL bEnable)
- HRESULT SetMaterial (CONST D3DMATERIAL9 *pMaterial)

In the case of the IDirect3DTexture9 class, it mainly provides the texture mapping features, and has the following member functions:

- IDirect3DTexture9 (void)
- ~IDirect3DTexture9 (void)
- void FromRGB (UINT width, UINT height, void *data)
- void FromBGR (UINT width, UINT height, void *data)
- void FromBGRA (UINT width, UINT height, void *data)

Some graphics primitives require the IDirect3DVertexBuffer9 class. We implemented some selected member functions from that class, as follows:

- IDirect3DVertexBuffer9 (UINT length, DWORD usage)
- ~IDirect3DVertexBuffer9 (void)
- ULONG Release (void)
- HRESULT Lock (UINT offsetToLock, UINT sizeToLock, void **ppbData, DWORD flags)
- HRESULT Unlock (void)

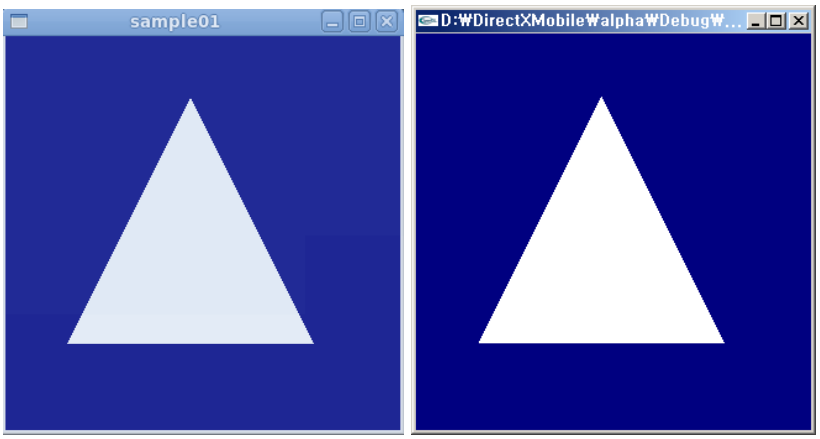


Figure 2. Testing a simple triangle on the different window systems. (left: our implementation on the Linux platform, right: original Direct3D output on the Windows platform)

After carefully selecting the member functions to be supported, we finally implemented all the selected classes and their selected member functions.

4. Test and Results

To show the feasibility of our implementation, we used a lot of demonstration programs and compare their results with those from desktop Direct3D applications. Our test was basically a kind of black-box testing, since we did not reflect the internal structures of our implementations. We will show each of our selected test programs and the corresponding technical problems.

Figure 2 is the first test program to show that the original Windows-based Direct3D programs can work on the Linux platforms. The sample program itself is so simple to output a white triangle on the blue background.

From the technical point of view, these simple test demonstrates our consistent window-system independent management technique, which works on both of the Windows and Linux systems. Due to their distinguished management techniques, it was not an easy work to implement our window-system independent management technique. In this paper, we used a similar technique to the *GLUT* library[8], which also supports both of Microsoft Windows and Linux *X-window* systems. Finally, our management technique proved its feasibility through successfully executing the original Windows-based source codes, without any source-level modifications.

Figure 3 is the sample program used for testing light-and-material processing. As shown in Figure 2, there are no particular differences between the original Direct3D output (left side) and that of our Direct3D-on-OpenGL implementation (right side).

In this sample program, we have solved the following technical problems:

- support for the 3DFVF_XYZRHW flag: This feature is not at all supported by the OpenGL core. In the case of Direct3D, the coordinate values (x, y, z, 1/w) can be assigned to the device coordinates. In our implementation, all the device coordinates are transformed to (x/w, y/w, z/w), and then, we

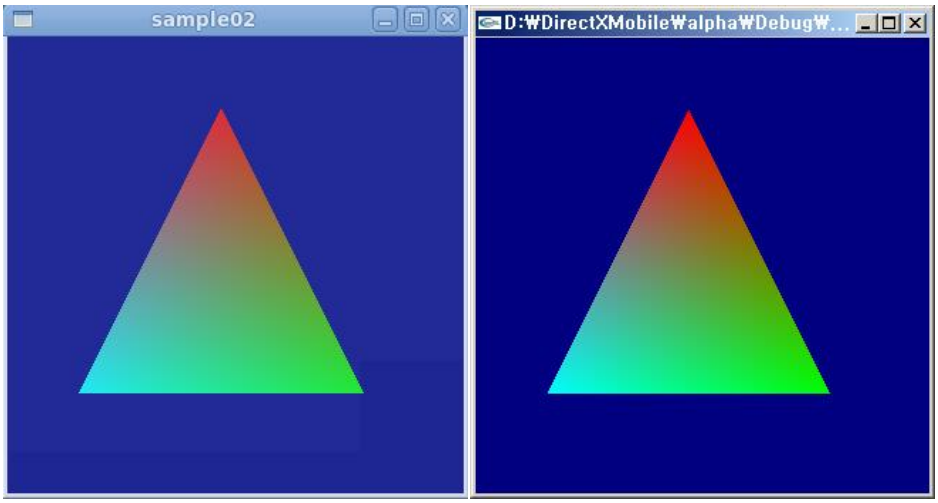


Figure 3. Color tests with various RGBA colors. (left: our implementation on the Linux platform, right: original Direct3D output on the Windows platform)

modify the transformation matrix through using the glOrtho( ) function, to perform the exactly same operations.

- color interpretations: In the case of Direct3D, D3DCOLOR and D3DXCOLOR structures store the color values in the (B, G, R, A)-order. In contrast, OpenGL requires (R, G, B, A)-order. Thus, we stored all the color values in both of the (B, G, R, A) and (R, G, B, A)-order, for more efficiency.

Figure 4 shows the multi-primitive test, which checks any problems with multiple output of difference graphics primitives. As shown in Figure 4, we put a triangle with various colors on the left side, while another single colored rectangle is located on the right side, to finally test whether various output primitives are working on the same frame. In our implementation, this program requires the complete and correct processing on the transformation matrix handling.

Figure 5 shows our first program for animation output, while our previous tests only check the static images. For this purpose, as shown in Figure 5, a quadrangular pyramid is constructed and rotated in real time. Though we show a static image, the test program performs a dynamic animation of rotating the quadrangular pyramid.

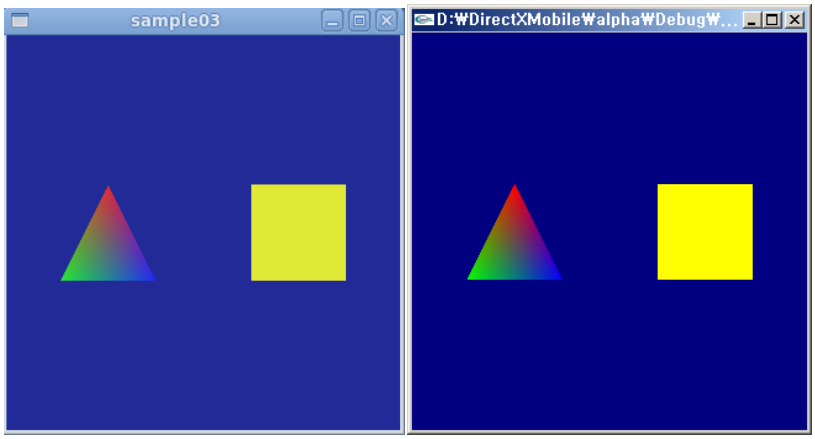


Figure 4. Various primitive tests. (left: our implementation on the Linux platform, right: original Direct3D output on the Windows platform)

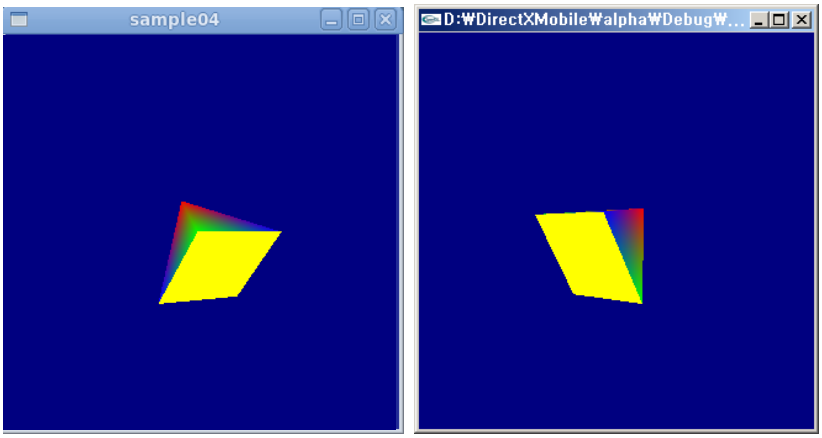


Figure 5. Animation test. (left: our implementation on the Linux platform, right: original Direct3D output on the Windows platform)

For the full-scale animations, we needed to solve some technical difficulties on the matrix handling and rendering pipelines, as follows:

- different coordinate frames between Direct3D and OpenGL: OpenGL is based on the right-handed coordinate system, while Direct3D uses the left-handed coordinate system. Thus, we predicted there would be some problems in the conversion of their coordinate systems. In real implementation, we uses transpose matrices for the OpenGL pipeline, and negative rotation angles to correct mismatches, and finally got the same transformation results.
- need to implement SetRenderState( ) function for the Direct3D: In the case of Direct3D, most state variables are set by the SetRenderState( ) function, while OpenGL uses a set of independent functions for controlling each state variable. A considerable amount of efforts were required to support these features.

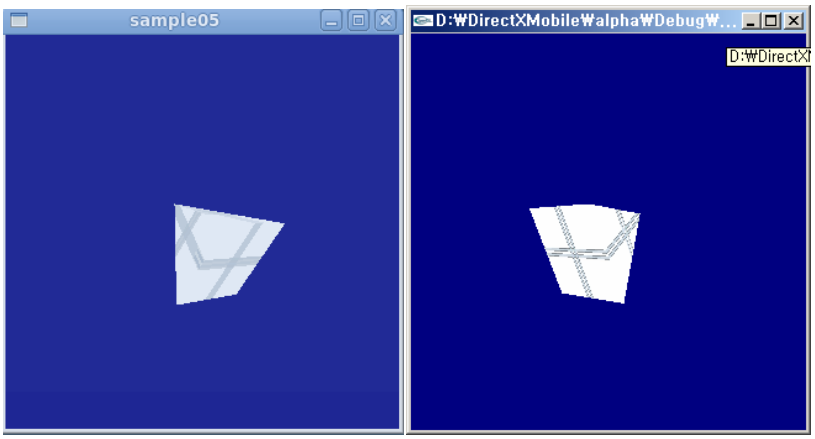


Figure 6. Animation with texture mapping. (left: our implementation on the Linux platform, right: original Direct3D output on the Windows platform)

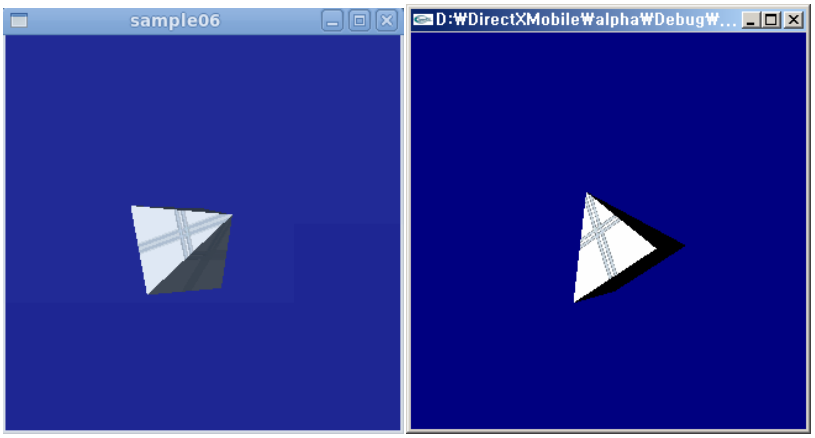
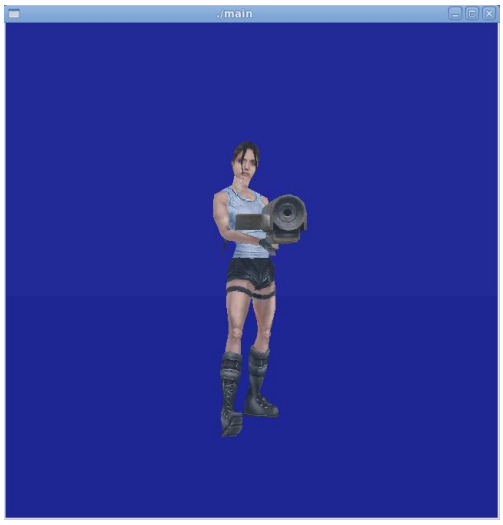


Figure 7. Animation with both of lighting and texture processing. (left: our implementation on the Linux platform, right: original Direct3D output on the Windows platform)

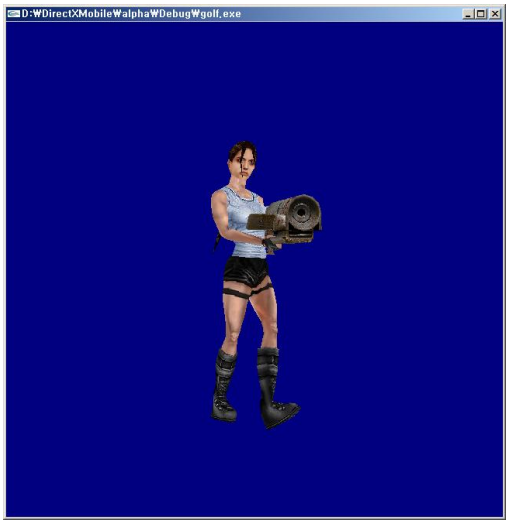
Figure 6 shows the texture mapping techniques, which are essentially required for advanced graphics programs. Texture mapping is a graphics technique to overlay a graphical image on the object surface, rather than a single color, as shown in Figure 6. This technique requires a large amount of modifications in the graphics rendering pipeline. The final results are successful, as shown in the figure.

For this demonstration program, we should overcome various technical difficulties, since the texture mapping is not a simple technique. Especially, we should notice that the texture mapping techniques in the Direct3D and OpenGL are quite different.

In the case of Direct3D, the IDirect3DTexture9 class perform the core features of the texture mapping. Thus, CreateTextureFromFile( ) function actually performs most of required works, and SetTexture( ) function only performs the final binding of the textures. In contrast, OpenGL's glTexImage( ) function requires many computing time, and thus, we should restrict



(a) our implementation on the Linux platform



(b) (b) original Direct3D output on the Windows platform

Figure 8. A game

the number of its execution. Thus, in our final implementation, we pre-construct the required textures in CreateTexture( ) function, using gluBuild2DMipmaps( ) function. Then, SetTexture( ) function uses glBindTexture( ) function to bind the already constructed texture, to minimize the required resources.

In Figure 7, we demonstrated the lighting and material features, which are always required for advanced graphics techniques. The lighting features are essential to implement the real-world light sources, while the material features are also required to show the real-world reflections on the object surfaces. Our problem was that the light and material features

in Direct3D and OpenGL are so different, while we finally achieved the same graphics output on both platforms.

Using all of these demonstration programs in a step-by-step manner, we finally showed that the original Direct3D programs implemented on Windows platforms can be executed on the Linux platforms, without any source-level modifications. In other words, the Direct3D features are available on the Linux systems. Since our implementation supports light-and-material processing, perspective transformation, animation loop and texture mapping, we can integrate all these features into a single program. Figure 8 is an example of those kinds of programs and shows an animation sequence of a game character. All these programs show that our implementation works well at least with these sample programs.

We applied much more test cases. Furthermore, these cases are applied in a step-by-step manner. After implementing a set of functions, then a set of test-cases are applied to test the feasibility and correctness of our partial implementation. We repeated these steps to get the currently finalized prototype implementation.

5. Conclusion

In this paper, we aimed at implementing a Direct3D-on-OpenGL library to acquire the same graphics output on the typical Linux-based systems with OpenGL library, with respect to the desktop Direct3D library. Based on our design strategies, we implemented a Direct3D-on-OpenGL emulation library and we showed that a set of demonstration programs produce the same result with respect to the original Direct3D-based outputs. A sample program even shows an animation sequence of a game character, to finally show the feasibility of our implementation.

Actually, all these sample programs are used to perform a kind of black-box testing on our implementation. We perform these tests in a step-by-step manner, according to the implementation schedules.

In near future, we plan to release a prototype system with more improved functionalities. We are also implementing a set of related libraries including OpenGL ES and EGL from Khronos group, which are already commercially available. Our Direct3D-on-OpenGL implementation will be also available in near future.

Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (Grants 2011-0014886).

This research was also supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (Grants 2011-0025849).

References

[1] K. Pulli, T. Aarnio, K. Roimela, and J. Vaarala, "Designing graphics programming interfaces for mobile devices," IEEE CG&A, vol.25, no.6, pp.66-75, (2005)

[2] http://code.google.com/android/what-is-android.html

[3] http://developer.apple.com/iphone/

[4] Yahya H. Mirza and Henry da Costa, "Introducing the New Managed Direct3D Graphics API in the .NET Framework," MSDN magazine, (Jul 2003)

[5] Mark Segal and Kurt Akeley, The OpenGL Graphics System: A Specification, version 3.1, (2009).

[6] A. Munshi and J. Leech, OpenGL ES Common/Common-Lite Profile Specification, version 1.1.12 (Full Specification), Khronos Group, (2008).
[7] K. Pulli, T. Aarnio, V. Miettinen, K. Roimela, and J. Vaarala, Mobile 3D Graphics: with OpenGL ES and M3G, Morgan Kaufman,(2007).
[8] M. J. Kilgard, The OpenGL Utility Toolkit (GLUT) Programming Interface, version 3, Silicon Graphics Inc., (1996).

Authors

Nakhoon Baek is currently an associate professor in the School of Computer Science and Engineering at Kyungpook National University(KNU), Korea. He received his BA, MS, and Ph.D degree in Computer Science from the Korea Advanced Institute of Science and Technology(KAIST) in 1990, 1992, and 1997, respectively. His research interests include real-time and mobile graphics systems.

Kwan-Hee Yoo is a professor in the Department of Computer Education and in the Department of Information and Industrial Engineering at Chungbuk National University, Korea. He received his BA in Computer Science from Chonbuk National University in 1985, and his MS and PhD in Computer Science from KAIST (Korea Advanced Institute of Science and Technology) in 1988 and 1995, respectively. His research interests include computer graphics, 3D characters animation, and dental/medical applications.