The Design and Implementation of Bare PC Graphics

Alexander Peter
Computer Science
Towson University
Towson, MD 21252
apeter9@students.towson.edu

Ramesh K. Karne
Computer Science
Towson University
Towson, MD 21252
rkarne@towson.edu

Alexander L. Wijesinha
Computer Science
Towson University
Towson, MD 21252
awijesinha@towson.edu

Patrick Appiah-kubi
Computer Science
Towson University
Towson, MD 21252
appiahkubi@towson.edu

Abstract—**Most multimedia applications today run with the support of an operating system, a graphics driver and related libraries. We present a lean graphics architecture for a bare PC that has no operating system or kernel running in the machine. The architecture enables a multimedia application to be independent of any computing environment and avoids dependencies on other software. To maintain simplicity, the graphics implementation uses the basic primitives to display a pixel, line, circle and a bitmap image. It can be used to implement complex graphics in spite of its simplicity. The bare PC graphics implementation is small in size, extensible and easy to maintain. This design allows graphics programmers to achieve higher performance by eliminating operating system overhead and using direct interfaces to the hardware.**

Keywords-Bare Machine Computing (BMC); Multimedia Graphics; Bare PC Graphics; Graphics Design; Application Graphics Object (AGO).

I. MOTIVATION

The bare machine computing (BMC) paradigm has been demonstrated using applications such as Web servers, Web mail servers, Email servers and clients, SIP servers [1], secure applications [13], and VoIP soft-phones [2, 14]. These applications used text-only interfaces for user interactions. The availability of a bare graphics interface will enrich these applications and make them more convenient for users. BMC applications are self-contained and independent of any operating system (OS), kernel, or execution environment. The work presented is a first step towards building bare PC graphics interfaces and lays a foundation for future multimedia applications that can run on bare devices.

II. INTRODUCTION

Current multimedia and graphics applications are built on graphics, video and audio drivers that are accessible through some platform such as Windows or Linux. In handheld devices, multimedia software is embedded in the devices to allow graphics capabilities. These embedded systems rely on some lean OS or kernel. In most cases, multimedia or graphics applications are dependent on the device platform. Modern multimedia applications use a graphics processing unit (GPU) along with video memory to provide parallel processing power before rendering to screen or storage. The video card's processing power and technological advancements in hardware pave the way for new software architectures that exploit the capabilities of modern systems. For example, since video cards provide gigabytes of low cost memory, paging and virtual memory are unnecessary, and multiple address spaces can be avoided by using a single monolithic executable code with real memory [15]. Today's high definition video cards can stream well over 60 frames per second, and are even over-clocked to higher speeds for 3D simulation using the Wiggle effect [4].

However, the above technological trends and techniques are platform-dependent and are not easily ported from one environment to another. The paper considers the design and implementation of a bare graphics architecture that is self-contained and does not require any operating system, kernel or environment to run. It is written in C/C++ and accesses video memory directly from its application program. This paper describes our approach in detail with some preliminary data.

The remainder of the paper is organized as follows. Section III presents related work; Section IV describes the architecture of bare PC graphics; Section V discusses its design and implementation; Section VI presents the results; and Section VII contains the conclusion.

III. RELATED WORK

The BMC concept also known as dispersed operating system computing (DOSC) [10] enables computer applications to run on a bare machine or a bare PC. Eliminating operating system abstractions [6] has been studied by many authors and the benefits include significant performance improvements as shown in Exokernel [7], Micro-kernel [9], lean kernel [18] and OS-Kit [19]. The BMC approach in contrast completely avoids any centralized OS or kernel. This results in the BMC paradigm wherein an application programmer has sole control of the application and its execution environment. Multimedia applications can also be built using the BMC paradigm by extending the Application Object (AO) [12] concept to develop an application graphics object (AGO) model comprising graphics, voice and video. The AGO run on a bare PC without the support of any OS or kernel. The AGO provides direct hardware communication interfaces to AO programmers thus eliminating all the abstraction layers introduced by OSs and their environments. Direct BMC hardware interfaces for C/C++ applications are described in [11]. These interfaces enable program load, screen display, mouse and keyboard access, process management, and network and audio card control. This paper describes new BMC interfaces constituting a hardware API for graphics applications.

There has been considerable research and significant advances in the areas of graphics and multimedia. In [5], an OpenGL-based scalable parallel rending framework that provides a graphics API was discussed. In [20], two user interfaces for interactive control of dynamically-simulated character using embedded system platforms were demonstrated. A lean mapping graphics interfaces that uses a method for real-time filtering of specular highlights in bump and normal maps was described in [16]. All such approaches require conventional OS-based platform support. A comprehensive low-level graphics design and implementation was described in [17]. However, these graphics interfaces use DOS (Microsoft Disk Operating System) primitives and

 ISBN: 978-1-61208-202-8

interrupt 21h, which require DOS environment. In bare PC applications, only required interrupts are used and included with the application. At present, there appears to be no direct hardware API for multimedia applications that can run on a bare PC with no OS support.

IV. ARCHITECTURE

A. Architectural Decription

The BMC Graphics architecture differs in many aspects from that of conventional or embedded graphics systems as the interfaces are directly accessible by the AO programmer. A given interface executes without any interruption as a single thread of execution. The AO programmer can control activation, suspension and resumption of this thread at program time. Figure 1 illustrates our graphics architecture that is suitable for any IBM PC based system. Currently, this architecture has been implemented using only Intel IA32 processor-based PCs.

An application programmer writes a graphics application (e.g., animation, visualization) application object (AO) in C++ or C using the direct hardware graphics interfaces (API). These interfaces are provided by the application graphics object (AGO). The AGO implements high level application logic if needed and sets up parameter passing for shared memory. The AGO invokes "C" language interfaces (using extern "C" {}) to invoke C calls from C++. The C calls in turn will invoke assembly calls for a given graphics interface. The assembly call then calls a graphics API software interrupt (int 0xfa). The AO, AGO, C, assembly calls have full access through a memory interface to read or write data in shared memory. This is accomplished by using a MEMDataSel selector that allows access to shared memory in real and protected modes using zero base select value. All of this code is executed in protected mode.

The software interrupt above is an interrupt gate that takes the call to real mode. The graphics interfaces are implemented in assembly code that run in real mode. These interfaces in real mode have access to video memory as shown in Figure 1. PC BIOS interrupts are also used to control video memory and graphics modes. Interrupt descriptor table (IDT), global descriptor table (GDT), local descriptor table (LDT), task state segment (TSS), boot, loader, interrupt service routines are all part of an AO. The AO is a self-contained, self-managed and self-executable module. The AO programmer has sole control of the facilities that are needed to run a given application.

The graphics architecture views the screen as just a multi-dimensional array of pixels that can be represented using vector-based mathematical models. This differs from conventional approaches that deal with each pixel every time the graphic changes.

The BMC graphics interfaces reduce complexity by providing a pointer to video memory that dynamically binds the interfaces at the hardware level to the video memory buffer. In this approach, there is no need to synchronize the GPU with the CPU functions [4] as done in a conventional system.

The AGO architectural design is classified into eight broader categories:

- *Application Program (AO) Protected Mode:* actual user program main() executes here.
- *Application Graphics Object (AGO):* software and hardware API used by AO.
- *C-Programming Interface:* used as a gateway between C++ and ASM language abstracts.
- *PC Assembly Interface:* used to interact with real-mode shared memory.
- *Software Interrupt*: Interrupt to bridge between real and protected mode dynamically.
- *Interrupt Gate to Real Mode:* used in real-mode to interact directly with the Hardware Video Memory.
- *Graphics Operations Real Mode:* used to invoke low-level graphics primitives including screen access, screen framework, font/symbol, and other attributes as outlined in section V.

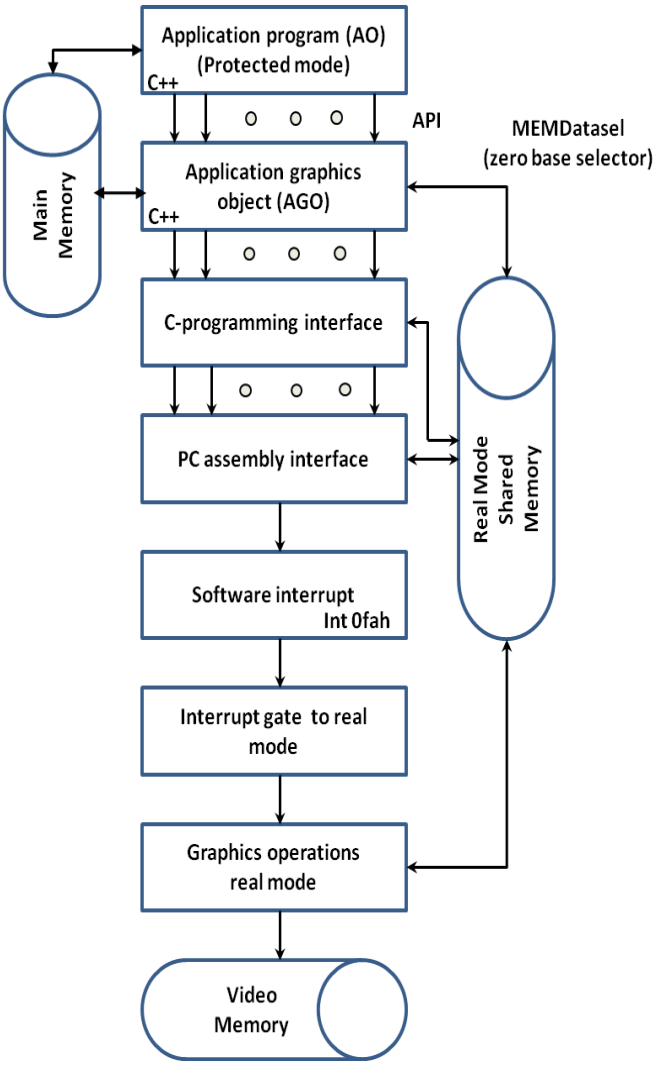

Figure 1. BMC Graphics Architecture

B. Architecural Novelties

The lean graphics architecture has many novel characteristics. The interfaces developed here can be directly invoked from standard C or C++ programs and are fully controlled by the programmer. The architecture is very generic and can be implemented on any pervasive device. As this

approach uses only video memory instead of graphics card interfaces, a given graphics application can be ported easily to many types of devices. The current AGO does not use any graphics accelerators and hardware support, bare interfaces to audio, video and graphics cards can be written if necessary to resolve any performance issues. When more sophisticated bare graphics and multimedia becomes feasible, users can carry their own USB with a Web client and browse the Web on any bare device without carrying any hardware and with no reliance on any OS, kernel or environment. In addition to being convenient for users, this graphics architecture eliminates overhead and may be easier to secure due to its simplicity.

V. DESIGN AND IMPLEMENTATION

A Bare PC graphics system is designed to perform graphics functions, such as drawing geometric figures with fraction (fractal) primitives, displaying text characters, and performing other attributes such as color, pixel, line, circle and displaying a bitmap image.

The following graphics functions are implemented using standard C and Intel assembly language. They are based on the design and implementation principles in [17] and modified to work with a bare PC system as described below:

- *Screen access:* clear screen, set the entire screen to a color or attribute, save the screen image in memory, and restore a saved screen image.
- *Screen framework:* set a shape screen area to a given color or attribute, save and retrieve a screen area in video memory.
- *Font/Symbols:* based on Vector / ASCII
- *Images/video:* based on pixel and compression
- *Shapes:* based on Vector/Pixel Mathematical Algorithms.
- *Attributes:* set the current drawing color, set the current fill color, set the current shading attribute, set the current text color, set the current text font, set the current line type (continuous, dotted, dashed, etc.), and set the current drawing thickness.
- *Image transformation:* scale, rotate, translate, and clip image.
- *Bit operations for performance:* BIT Shifters; XOR, OR, NOT and AND bitwise operations.

In order to illustrate our implementation, we describe five basic direct graphics APIs:

(1) **draw_pixel():** This interface takes x and y coordinates of a given pixel and computes its video memory location. The video memory address, location of a pixel, color of a pixel and its "opcode" are stored in shared memory. As illustrated in Figure 1, this interface goes from protected mode to real mode to the graphics operations code. It then obtains the pixel parameters from shared memory and places the pixel in the video memory. After displaying the pixel on the screen, it will return to its AO. The entire API process is executed as a single thread of execution. More optimal approaches to drawing a pixel on the screen will be considered later.

(2) **draw_line():** This is simply drawing many pixels to draw a line using Bresenham's algorithm [3].

(3) **draw_box():** This API uses draw_line() API repeatedly to plot a box.

(4) **draw_circle():** The circle is implemented without using the sine and/or cosine functions. It uses the algorithm described in [8] and uses draw_pixel() API.

(5) **draw_bitmap():** First, the bitmap file is read from the removable device (USB) during the program load and stored in main memory. Second, the bitmap file header is parsed for size, color and image data offset location parameters as shown in Figure 3. Third, the display mode is setup to match minimum color palette requirements for the image as shown on Figure 2. Fourth, the video memory address, pointer to the bitmap image data and its opcodes are stored in shared memory as illustrated in Figure 1. Finally, the AGO will copy the image data from shared memory to video memory for display as shown in Figure 3. After placing the bitmap on the screen, it will return to its AO.

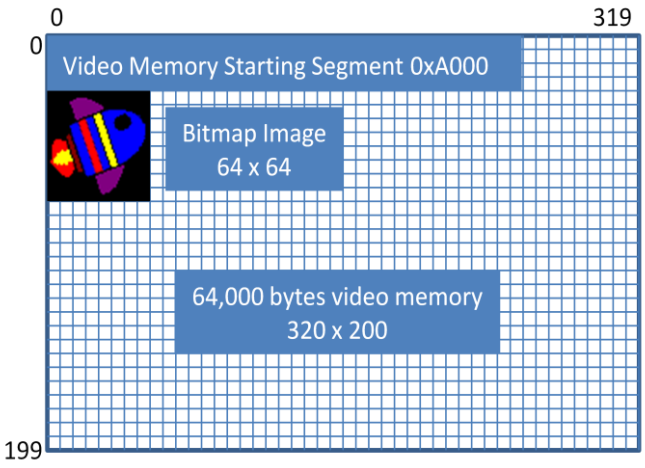

Figure 2. Video Memory Layout

Video memory is a contiguous linear addressing model, which differs from the x and y coordinates of the computer screen. To plot a pixel, the offset is calculated from the beginning of the video memory as follows: y coordinate multiplied by the total width of the screen and the x coordinate added to it.

In the example shown in Figure 2, we use the VGA Mode 0x13 with screen dimension of 320 pixels in width and 200 pixels in height. This translates to 0 to 319 on the x axis (width) and 0 to 199 on the y axis (height). The top left corner starts at coordinate (0, 0). Each pixel represents 8 bits (1 byte). Thus, the memory needed to store images of this size (320x200) is 64,000 bytes.

Figure 3 shows the Bitmap Image structure. It consists of several components that are described below.

The File Header in Figure 3 contains the FileType which starts with 4D42h ("BM"). FileSize is the Size of the image file in bytes. Reserved fields are used for future enhancements, with default values set to 0. The BitmapOffset stores the starting position of image data in bytes. The total size of the File Header is 14-bytes.

"Size" is the size of this header in bytes, and Width and Height are the Image width and height in pixels. A plane is the number of color planes and BitsPerPixel is the number of bits per pixel. The total size of the Bitmap Header is 40-bytes.

 ISBN: 978-1-61208-202-8

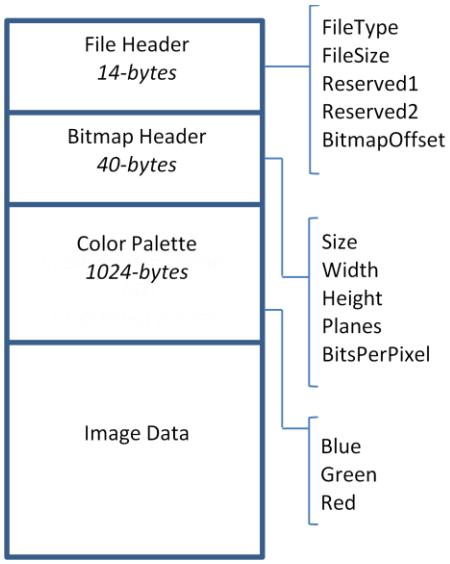

Figure 3. Bitmap Image Format

The BMC Color Palette specifies the red, green, and blue values of each pixel in the bitmap data by storing a single value used as an index into the color palette. In the newer versions of the BMP standard, the Color Palette and Image Data are merged together. In our example, we are using the Image Data directly, since we have pre-defined our palette to 256-colors. The total size of the Color Palette is 1024-bytes.

The pseudo-code in Figure 4 is used to display a 64x64 bitmap to video memory; since video memory is linear, a simple computation based on the x and y values can be used.

```
void draw_bitmap(AO, AGO Object Reference)
Initialize Memory Locations;
Initialize Variables;

Loop While screen_height < 200
{
 For (y=0; y < image_width; y++) {
   For (x=0; x < image_height; x++){
    //AGO Implementation, copy to Video Memory

        Video_Memory_Pointer [x+y*320]=
            Shared_Memory_Pointer[x+y*64] } }
}
```

Figure 4. Bitmap to Video Memory Pseudo-Code

VI. TESTING

The testing was conducted on a standard VGA graphics card and VESA enabled BIOS on VGA Mode 13, with 320-by-200 pixel resolution in 256-Colors. The graphics was tested on a Dell Optiplex GX260 PC with 512 MB memory.

The preliminary response time in Table I was conducted using the UNIX system time; it is an end-to-end measurement, which includes other components such as the VGA hardware device and other display intermediaries.

Below, we illustrate five basic primitives as described in Section V. Each API is a direct hardware interface available to the AO programmer that can be invoked directly from C or C++ code.

TABLE I. BMC GRAPHICS – PRELIMINARY RESPONSE TIME

AGO Object	Response Time (Microseconds)
draw_pixel() graph Figure 5	2.25
draw_line() graph Figure 6	83.25
draw_ circle() graph Figure 7	250
draw_bitmap() graph Figure 8	5

A. Pixels

Using the draw_pixel() AGO API, 5000 pixels were plotted as shown in Figure 5. The x, y coordinates and color were chosen randomly. Preliminary performance tests shows a response time of 2.25 microseconds.

Figure 5. BMC Graphics - 5,000 Random Pixels and Colors

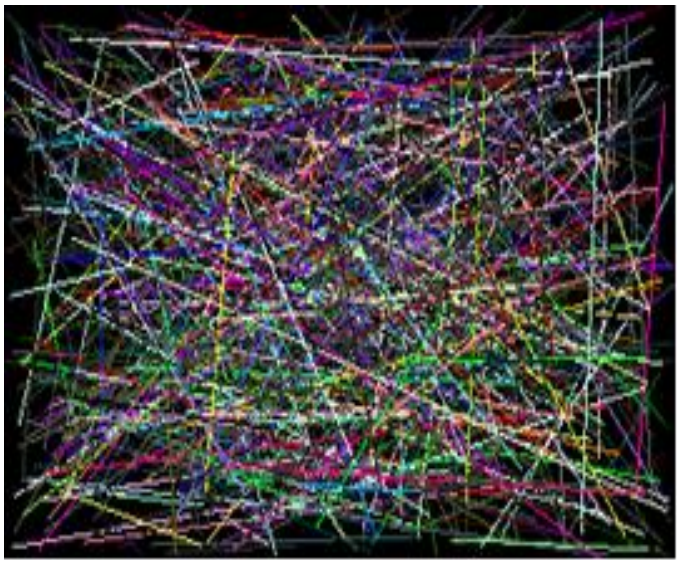

Figure 6. BMC Graphics - 5,000 Random Lines and Colors

 ISBN: 978-1-61208-202-8

B.Line

Using the draw_line() AGO API, 5000 lines were rendered as shown in Figure 6. The x1, x2, y1, y2 coordinates and color where chosen randomly. The draw_circle API is a direct hardware interface inherited from the draw_pixel() AGO. Preliminary performance tests shows a response time of 83.25 microseconds.

C.Circle

Using the draw_circle() AGO API, 5,000 circles were rendered as shown in Figure 7. The x, y coordinates, radius size and color were chosen randomly. The draw_circle API is a direct hardware interface inherited from the draw_pixel() AGO. Preliminary performance tests shows a response time of 250 microseconds.

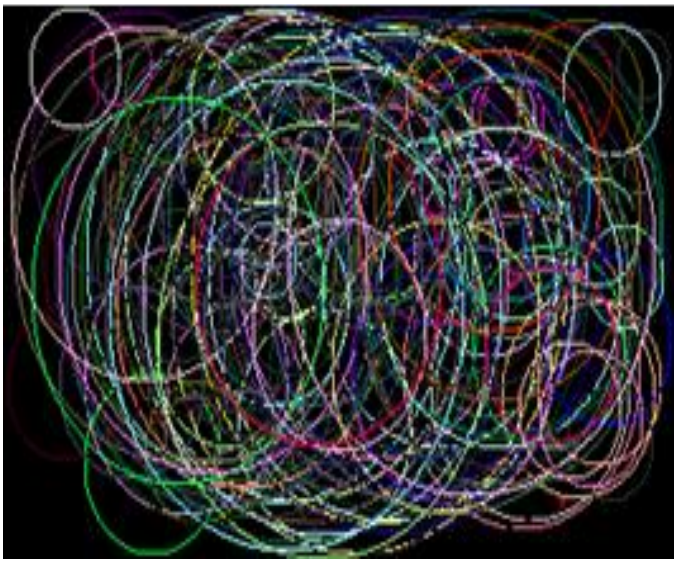

Figure 7. BMC Graphics - 5,000 Circle with Random Size and Color

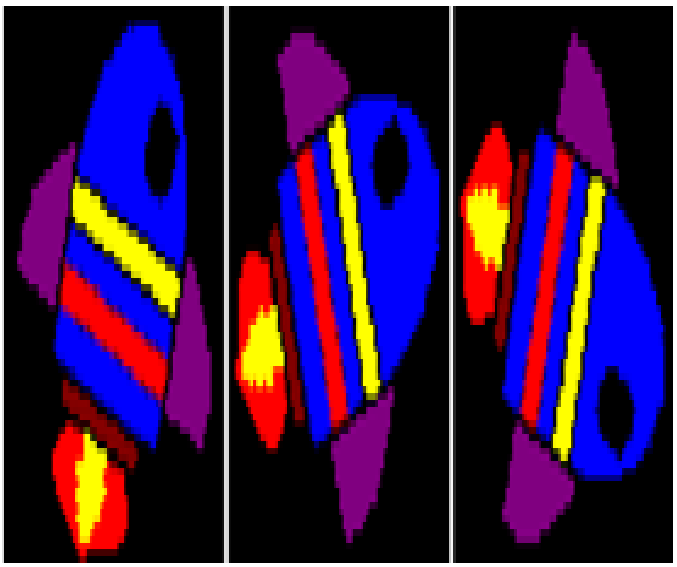

Figure 8. BMC Graphics – Bitmap

D. Bitmap

The bitmap used in this example is a 64 by 64 256-color bitmap with 8 bits per pixel, the file format is Windows RGB-encoded BMP format uncompressed. For a 256-color bitmap, there is a 54-byte header and a 1024-byte palette table in addition to the actual bitmap data.

Using the draw_bitmap() AGO API, the bitmap image was loaded into video memory directly for display. The bitmap was loaded three times with different orientation to show rotation and animation. This can be achieved by changing the pixel loading order while copying to memory as shown on Figure 8. Preliminary performance tests shows a response time of 5 microseconds.

VII. CONCLUSION

We presented the architectural design for building graphics applications that can run on bare devices with no operating system, kernel or environment support. We also gave details of its implementation model using C/C++. The AGO API illustrates some of the fundamental graphics elements and their functionality. The preliminary performance data indicates applicability of bare machine graphics for complex graphics applications. The direct hardware graphics API can be used in a variety of pervasive devices to achieve common graphics operations. We have also presented the benefits of running graphic applications on a bare PC including simplicity, elimination of abstraction layers, and self-containment. With bare PC graphics, the programmer has direct access to the video graphics device and complete control of all hardware resources enabling autonomy with performance advantages due to elimination of system overhead.

ACKNOWLEDGMENT

We sincerely thank NSF and in particular the late Dr. Frank Anger, who initially supported this work by funding through SGER grant CCR-0120155. Without his encouragement, bare machine computing concept and explorations could not have been possible.

REFERENCES

[1] A. Alexander, A. L. Wijesinha, and R. Karne, "A Study of Bare PC SIP Server Performance," The Fifth International Conference on Systems and Networks Communications, ICSNC, Nice, France, pp. 392 – 397, August 2010.

[2] A. Alexander, A. L. Wijesinha, and R. Karne, "Implementing a VOIP Server and a User Agent on a Bare PC," The Second International Conference on Future Computational Technologies and Applications, Future Computing, Portugal, Lisbon, pp. 8 – 13, November 2010.

[3] D. Brackeen, Developing Games in Java. Berkeley, CA: New Riders Games, pp. 63-70, 2003.

[4] D. Salomon, The Computer Graphics Manual, Ithaca, NY: Springer-Verlag Publisher, pp. 200 – 240, 2011.

[5] S. Eilemann, M. Makhinya and R. Pajarola. "Equalizer: A Scalable parallel rending framework," IEEE Transactions on Visualization and Computer Graphics, pp. 436 – 452, June 2008.

[6] R. Engler and M.F. Kaashoek, "Exterminate all operating system abstractions," In Fifth Workshop on Hot Topics in Operating Systems, pp. 78, May 1995.

[7] D. Engler, "The Exokernel Operating System Architecture," Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Ph.D. Thesis, 1998.

[8] J. E. Frith. "Fast Circle Algorithm," http://www.tutego.de/aufgaben/j/insel/additives/base/fcircle.txt, Copyright (c) 1996 James E. Frith, Email: jfrith@compumedia.com [Retrieved: March, 2011].

 ISBN: 978-1-61208-202-8

[9] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullman, "Interface and execution models in the Fluke Kernel," Proceedings of the Third Symposium on Operating Systems Design and Implementation, USENIX Technical Program, New Orleans, LA, pp. 101-115, February 1999.

[10] R. K. Karne, K.V. Jaganathan, T. Ahmed, and N. Rosa, "DOSC: Dispersed Operating System Computing," OOPSLA, 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Onward Track, Sandiego, CA, pp. 55-61, October 2005.

[11] R. K. Karne, K. Venkatasamy and T. Ahmed, "How to run C++ applications on a bare PC," In proceeding of 6th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel / Distributed Computing (SNPD), pp. 50 – 55, May 2005.

[12] R. K. Karne, "Application-oriented Object Architecture: A Revolutionary Approach," In 6th International Conference, HPC Asia, Poster presentation, December 2002.

[13] N. Kazemi, A. L. Wijesinha, and R. Karne, "Evaluation of IPsec Overhead for VoIP using a Bare PC," 2nd International Conference on Computer Engineering and Technology (TCCET), vol. 2, pp. 586 – 589, April 2010.

[14] G. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, "A Peer-to-Peer Bare PC VoIP Application," IEEE Consumer Communications and Networking Conference, Seamless Consumer Connectivity, CCNC, Las Vegas, Nevada, pp. 803 – 807, January 2007.

[15] P. Kovach and J. Richter, Inside Direct3D, Redmond, WA: Microsoft Press, 2000.

[16] M. Olana, and D. Baker, "Lean Mapping," Proceedings of 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, pp. 181-188, February 2010.

[17] J. Sanchez and C. Maria, Computer Animation Programming Methods & Techniques, McGraw-Hill, 1995.

[18] "Tiny OS," Tiny OS Open Technology Alliance, University of California, Berkeley, CA, 2004, http://www.tinyos.net/ [Retrieved: June, 2007].

[19] "The OS Kit Project," School of Computing, University of Utah, Salt Lake City, UT, June 2002, http://www.cs.utah.edu/flux/oskit [Retrieved: May, 2009].

[20] P. Zhao and M. Van de Panne, "User interfaces for interactive control of physics-based 3D characters," Proceeding of the 2005 symposium on Interactive 3D graphics and games, pp. 87 – 94, April 2005.

 ISBN: 978-1-61208-202-8