COMPUTATION-EFFECTIVE 3-D GRAPHICS RENDERING ARCHITECTURE FOR EMBEDDED MULTIMEDIA SYSTEM

Bor-Sung Liang and Chein-Wei Jen
Department of Electronics Engineering
National Chiao Tung University, Taiwan, R.O.C.
E-mail: {bsliang,cwjen}@twins.ee.nctu.edu.tw

ABSTRACT

A new architecture is proposed to realize 3-D graphics rendering for embedded multimedia system. Because only 20% to 83% triangles in original 3-D object models are visible by simulation, our architecture is designed to eliminate the redundant operations on invisible triangles without image quality loss. It bases on our index rendering and enhanced deferred lighting approaches, and its feature is dual pipeline rendering architecture. The simulation and analysis results show that this architecture can save up to 63.4% CPU operations compared with traditional architectures.

1. INTRODUCTION

3-D graphics emerges rapidly in consumer electronics. Because of vivid visual effect, 3-D graphics plays important roles in multimedia, entertainment, virtual reality and user interface. Although lots of approaches are proposed in PC-based or entertainment platform, 3-D graphics rendering still seldom appears in embedded systems, such as PDA, mobile phone, car navigation system, etc.

One of the major reasons is computing power. Many embedded systems equip low-tier CPUs. Especially in portable devices, low-power low-cost requirement limits the employ of high-performance CPU. Hence 3-D graphics rendering by pure software suffers from low speed and poor image quality. Previous research tried to improve this by modified API [1], and 10k polygon/s was reported without lighting, shading and texture mapping. The speed and image quality is hard to support fantasy 3-D graphics applications.

On the other hand, the approach of 3-D processor [2][3] costs too much to be realized in embedded system. Because 3-D graphics rendering is computation-intensive, and high image quality requirement of 3-D graphics applications, commercial 3-D processors are designed to achieve high performance. The performance-driven architecture desires high computation power, large memory size and huge bandwidth. Those factors are bottlenecks to realize 3-D graphics rendering in embedded system.

Hence, the 3-D graphics rendering approach for embedded system is desired, and it can be utilized in lots of consumer electronics devices, such as set-top box, car navigation system, PDA, and mobile phone. In our previous researches, we proposed index rendering [4] and deferred lighting [5] approaches. These approaches can reduce redundant operations on hidden pixels and lighting operations on invisible triangles. These approaches can be applied in embedded system. Moreover, we further extend deferred lighting approach to eliminate the transformations on invisible triangles in this paper. Because transformations are huge burden in geometry subsystem, the enhanced version of deferred lighting can save more operations. Because of these design issues, the architecture of traditional rendering pipeline is divided into two pipelines, and this new architecture can reduce lots of unnecessary operations without image quality loss.

The organization of this paper is as following: In Section 2, we first review 3-D graphics pipeline, and show the strategies to reduce operations. Then, we introduce our new architecture in embedded system in Section 3. Because of index rendering and enhanced deferred lighting, this architecture has the feature of two separated pipeline. In Section 4 we present simulation and analysis of this architectures. Finally, we conclude this paper in Section 5.

2. 3-D GRAPHICS RENDERING PIPELINE

3-D graphics rendering pipeline generally divided into two parts: geometric subsystem and raster subsystem. The geometry subsystem transforms vertices, and performs lighting and perspective transformation. Raster subsystem receives output of geometry subsystem, and renders transformed polygons for display. Those two subsystems are pipelined for high throughput in general. Fig.1 is an example of traditional 3-D graphics rendering pipeline.

This work was supported by National Science Council, Taiwan, R.O.C. under Grant NSC-89-2215-E009-052

Original manuscript received June 19, 2000

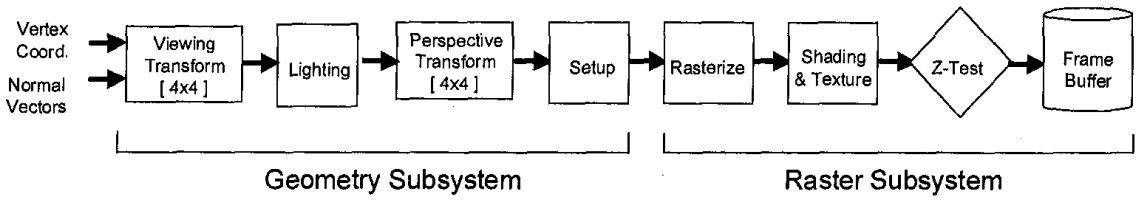


Fig.1 Traditional 3-D Graphics Rendering Pipeline

2.1 Viewing and Perspective Transformation

Transformations are major operations in geometric subsystem. It can be handled by 4x4 matrix operations. The general form is:

$$[x \;\; y \;\; z \;\; w] \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} = [x' \;\; y' \;\; z' \;\; w'] \qquad (1)$$

Generally speaking, the CPU in embedded system handles matrix transformations. Because the CPU often has no extra hardware for accumulation operations of matrix transform, more zero terms in 4x4 matrix means less multiplication calculations in CPU operations. In 3-D graphics rendering, the essential transformations are viewing and perspective. The viewing transformation transforms objects from the world space to the view space. The perspective transformations from the view space to the projection space, and then maps to screen coordinates.

In viewing transformation, the CPU needs 12 *multiplication* and 9 *addition* instructions perform viewing transformation. The equation is as following:

$$[x \;\; y \;\; z \;\; w] \begin{bmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ j & k & l & 1 \end{bmatrix} = [x' \;\; y' \;\; z' \;\; w'] \qquad (2)$$

On the other hand, in perspective transformation, the equation of perspective transformation is assumed as following [6]. In this transformation, the CPU only needs 5 *multiplication* and 1 *addition* instructions.

$$[x \;\; y \;\; z \;\; w] \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & e \\ 0 & 0 & d & 0 \end{bmatrix} = [x' \;\; y' \;\; z' \;\; w'] \qquad (3)$$

2.2 Lighting

Lighting is an essential procedure to calculate illumination on assigned position by lighting model. Nowadays 3-D graphics rendering often uses Phong lighting model [7], as shown in following equation. Lighting calculation is complex and related to exponentiation. Besides, each vector needs to be normalized before applying in this equation, and normalization requires calculation of reciprocal square root. Hence lighting operation is very computation-intensive.

$$I = I_a \times K_a + [I_i \times K_d (\boldsymbol{L} \cdot \boldsymbol{N}) + I_i \times K_s (\boldsymbol{H} \cdot \boldsymbol{N})^n] \qquad (4)$$

Where

I_a : intensity of the ambient light
I_i : intensity of the light source
$\boldsymbol{L}$: the unit vector from pixel to the light source
$\boldsymbol{N}$: the normal vector of the pixel
$\boldsymbol{H}$ = the unit vector of $(\boldsymbol{L}+\boldsymbol{V})/2$
$\boldsymbol{V}$: the vector to the viewer
n : gloss to the model high light
K_a, K_d, K_s : coefficients to model the characteristic of the material

In order to simplification, we can limit the gloss factor n into 2's exponentiation: 1,2,4,8,and 16. Hence the exponentiation can be realized by 4 *multiplication* instructions. $I_a \times K_a$, $I_i \times K_d$ and $I_i \times K_s$ can be pre-computed to avoid redundant operations in the runtime. However, vector normalization can not be pre-computed. For each lighting, vector $\boldsymbol{L}$, $\boldsymbol{N}$ and $\boldsymbol{H}$ should be normalized before be applied into this formula. To normalize a vector, 1 *reciprocal radical*, 6 *multiplication* and 2 *addition* instructions are necessary. Therefore, the CPU needs to perform 3 *reciprocal radical*, 34 *multiplication* and 19 *addition* instructions for each lighting operation.

2.3 Data Setup

Setup is the operation to prepare the necessary information for further rasterization. In the Setup operation, two kinds of data are generated for further rasterization. The first is the data related to shape information, while the second is related to color information.

2.3.1 Setup for shape information

Because the triangles are described by vertices in geometry subsystem, the setup of shape information is to help scan-converting triangles into a group of pixels. The

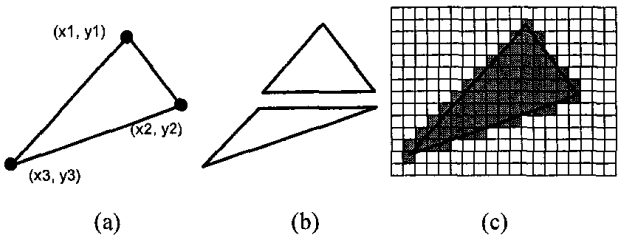


Fig.2 Triangle Shape Generation

scan-convert method affects the work of setup for shape information. In traditional method, the triangles are decomposed into two scanline-aligned parts, and each part can be described by left-side edge, right-side edge, maximal and minimal y-coordinates boundaries, as shown in Fig. 2(a) and (b). Then, the Rasterize operation uses the information to decomposed two scanline-aligned parts into a group of pixels. As shown in Fig. 2(c), the grid means the pixel array in screen coordinates, and the gray-colored grids mean the pixels covered by the triangle in Fig 2(a). Because this method of setup can simplify the data transfer between geometry subsystem and raster subsystem, lots 3-D graphics rendering hardware [8] utilize this method for shape information setup.

This method works will in PC-based platform, but the description of left-side and right-side edge is a problem in embedded system. The edges are usually described by their edge slopes, and division operations are needed to generate the slopes. Because most embedded systems employ low-tier CPUs, division operations for edge setup are large burden. On the other hand, this method decomposes a triangle into two scanline-aligned ones, and hence other triangle information is duplicated for data transmission. It may cause bandwidth problems.

In embedded system, Pineda's algorithm [9] is more suitable, because its algorithm is all by integer and not needed to decompose triangle. This algorithm represents each edge of a triangle by a linear edge function. The edge function can divide a plan into two parts. As shown in Fig.3(a), two vertices (x_1,y_1) (x_2,y_2) can define a linear equation $E_{12}(x,y)$. To detect which part does a pixel (x,y) locates, we can simply apply the (x,y) coordinates into $E_{12}(x,y)$, and see the result value greater or less then zero. For three edges in a triangle, the pixels are inside triangles only when all edge functions are positive or negative, as shown in Fig. 3(b) and Fig. 3(c). Because the all-positive or all-negative result depends on the direction of three edge vectors, this algorithm can also perform back-face culling in runtime.

Although the setup needs to setup three edge functions in Pineda's algorithm, we do not need to find out the real parameters in E(x,y). Because two end-point vertices are on the edge, the edge function must be zero on the vertices. For example, for an edge function $E_{12}(x,y)$ defined by vertices (x_1,y_1) and (x_2,y_2), $E_{12}(x_1,y_1)$ and $E_{12}(x_2,y_2)$ must be zero. For other pixel (x_k,y_k), the edge function becomes:

$$E_{12}(x_k,y_k) = (x_k-x_1)dy_{12}-(y_k-y_1)dx_{12} \quad (5)$$
$$dx_{12}= x_2 - x_1$$
$$dy_{12}= y_2 - y_1$$

Therefore, the setup and rasterizing triangle shape can be all integer operations. Although Pineda's algorithm is designed for parallel rendering, it is also suitable in 3-D graphics rendering in embedded system.

2.3.2 Setup for color information

For rasterizing color information, Gouraud shading [10] is a widespread method to generate acceptable image quality without huge computation. It applies lighting only on vertices of each triangle, and then shades each pixel inside triangle by interpolating color on vertices. Hence it can render polygons with smooth color gradation. In traditional method, the color interpolation is handled along the edge[8]. This way is related to traditional triangle shape generation, but makes color interpolation complex. Generating triangle shape by Pineda's algorithm, we can treat color interpolation by plain equation. This method is developed for parallel rendering in pixel-planes [11]. Because of linear interpolation, the intensity of each color can be generated by the linear expression, $Ax + By + C$. We can treat the color intensity (R, G or B) as the third

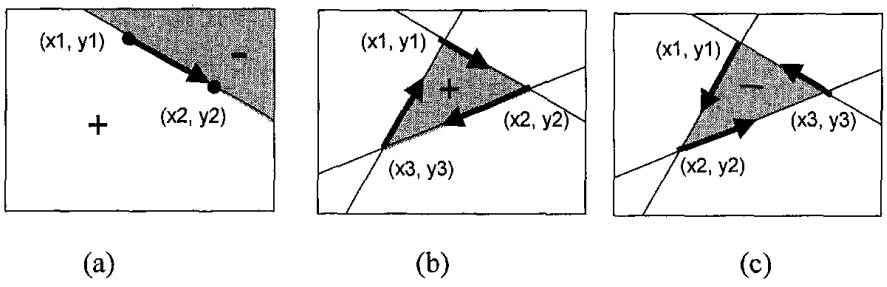


Fig.3 Triangle Shape Generation by Pineda's Algorithm

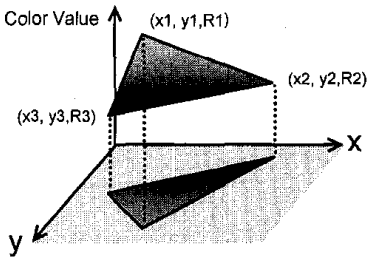


Fig.4 Color Value Plane

dimension associated by screen coordinates. As shown in Fig. 4, we take R value as example. The lighting operation gives the color intensity on three vertices, and hence a plain is defined in this space:

$$dR/dx = Ka * [(R_2-R_1) * dy_{23} - (R_3-R_2) * dy_{12}] \quad (6)$$
$$dR/dy = Ka * [(R_3-R_2) * dx_{12} - (R_2-R_1) * dx_{23}]$$
$$Ka = 1 / (dx_{12} * dy_{23} - dx_{23} * dy_{12})$$
$$dx_{12}=x_2-x_1 \qquad dx_{23}=x_3-x_2$$
$$dy_{12}=y_2-y_1 \qquad dy_{23}=y_3-y_2$$

The intensity of R in (x_1,y_1) is known as R_1, therefore the R value in (x_k,y_k) is:

$$R(x_k,y_k) = R_1 + (x_k-x_1)dR/dx + (y_k-y_1)dR/dy \quad (7)$$

The value in Eq. 6 is calculated in setup stage. Because the *Ka* term is only related to the vector of vertices, and it is the same in different color component R, G, B. Hence, only one division is necessary for each triangle and other color information can be generated by *multiplication* and *addition* operations.

2.4 Raster Subsystem

In conventional rendering pipeline, raster subsystem handles rasterization. Rasterization consists of three subtasks: scan conversion, visibility comparison and shading [12]. Scan conversion decouples polygon into a group of pixels. It is handled in the rasterize block in Fig.1. Hence, the operations before Rasterize are triangle-level operations, while after Rasterize are pixel-level operations. Shading operation colors each pixel for display, and texture mapping is also applied here. Visibility comparison determines the visibility of each pixel, and Z-test algorithm is the most common one.

In order to eliminate redundant operations on invisible triangles and invisible pixels, we utilize index rendering and deferred lighting to realize raster subsystem. We will discuss more in following section.

3. The Proposed Architecture

In this paper, we propose a new architecture for 3-D graphics rendering in embedded system. As shown in Fig.5, the chipset, Rasterizer Controller (RC) and Color Shader (CS), realizes the raster subsystem and setup, while the CPU handles the geometry subsystem. Two blocks of memory are utilized for data storage. One is for original object models, and can be realized by ROM or RAM, which depends on the applications. This database is named GTdb (Global Triangle Database). The another memory block is for temporal storage in 3-D graphics rendering, therefore it should be realized by RAM. The hardware architecture is based on our index rendering [4] and enhanced version of deferred lighting[5] approaches.

3.1 Index Rendering

Index rendering is an approach that can avoid redundant operations on invisible pixels. It is also the essential architecture to realize deferred lighting. The major concepts of index rendering are separating triangle/pixel data to explore parallelism, and rearranging operations for optimal data flow.

Traditional rendering architecture is a long pipeline, and therefore triangles and pixels carry their whole data to pass all pipeline stages. In fact, most of pipeline stages only relates to some parts of data. The other parts of data are only stored-and-forwarded. On the other hand, the nature of pipeline is fixed data flow, and hence limits the

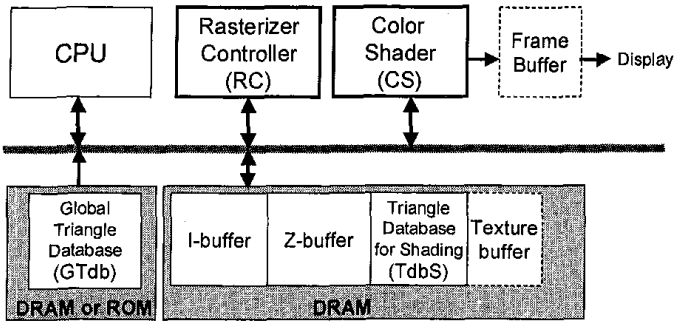


Fig. 5 3-D Graphics Rendering System for Embedded System

optimization of operation rendering.

The relationship of Z-test and shading is an example to show limitation of traditional rendering pipeline. Most shading operations relate to triangle information, and triangles must be scan-converted into groups of pixels before Z-test. Hence, shading operation applies on all pixels in traditional rendering pipeline, even though the pixels are invisible (fail in Z-test). In previous researches, *Deferred Shading* [13][14][15][16] can improve this problem. It defers the shading operations after Z-test, but each pixel needs to carry a copy of shading-related information. It leads to bandwidth problem, and hence not suitable for embedded system.

In our approach, index rendering, we utilize *index* to separate triangle/pixel data to explore parallelism. The index is a serial number of each polygon. We use this index to denote the information and pixels from this parent polygon. In the rendering pipeline, the information is stored in database, and each pixel only carries its index number to pass the long rendering pipeline. If one part of information is necessary in a pipeline operation, we can fetch the database on demand.

In order to eliminate redundant shading operations on invisible pixels, the approach of index rendering stores shading information in TdbS (Triangle Database for Shading), as shown in Fig.5. After Z-test, the index numbers of visible pixels are stored into a screen-size buffer, named I-buffer, as shown in Fig. 6. Then, we can calculate color values of each pixel from the Eq. 7 to generate the final image.

In the Fig.5, CS handles the shading operations, while RC handles the other operations in raster subsystem and setup. The job of RC is to generate the index pattern in I-buffer and data in TdbS, and the CS utilizes the data in I-buffer and TdbS to generate the final result. Because the I-buffer and TdbS keep enough information to generate the final result, hence frame buffer can be optional if the CS can generate pixels in screen scan-out rate.

3.2 Enhanced Deferred Lighting

More than eliminating redundant operations on invisible pixels, our deferred lighting approach can avoid redundant operations on invisible triangles. This approach defers lighting calculation after Z-test. If all pixels of a triangle fail in Z-test, it implies that this triangle is invisible, hence we can eliminate lighting calculation on invisible polygon. This idea is straightforward but hardly to be realized in traditional rendering pipeline. With the approach of index rendering, this idea can be realized in 3-D graphics rendering pipeline.

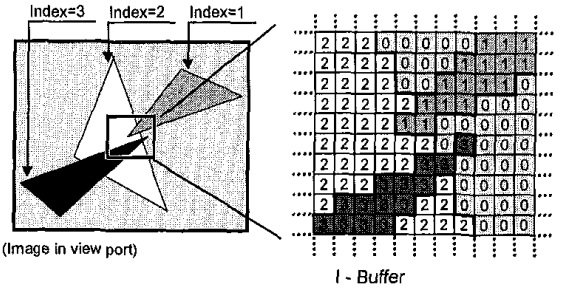


Fig.6 Index pattern in I-buffer

This approach was proposed in [5]. In this paper, we further extend deferred lighting approach to eliminate the transformations on invisible triangles. In order to do this, the triangle information must be separated before operations. The triangle information related to geometry, such as the (x,y,z) coordinates, goes first to define the shape of this triangle, and then to be scan-converted into a group of pixels. After all pixel are Z-tested, we can know whether this triangle is hidden. If any pixel of this triangle passes the Z-test, the triangle information related to shading enters the rendering pipeline. After setup operation, the shading information is stored in TdbS. Finally, the result image is generated by I-buffer and TdbS.

3.3 Dual Pipeline Rendering Architecture

According to index rendering and enhanced deferred lighting approaches, the 3-D graphics rendering hardware becomes dual pipeline architecture. Fig. 7 shows the rendering pipeline. The Fig.7(a) shows traditional one, while Fig. 7(b) our dual pipeline architecture. The major difference is that we divide the 3-D rendering pipeline into two parallel pipelines. The ASIC chip handles the operations with gray shaded area, and the CPU of embedded system handles the other area.

To render a triangle in the dual pipeline architecture, the upper pipeline goes first. The upper pipeline needs the input of triangle information related to geometry, which are the coordinates of vertices. After all pixels of this triangle are Z-tested, a signal is sent to the CPU to denote whether this triangle is visible or not. If this triangle is invisible, the other part of triangle information is discarded. If this triangle is visible, the other part of triangle information enters the second pipeline. Because the Phong lighting model is applied, hence the triangle information related to shading is the normal vectors on the vertices of this triangle.

In the dual pipeline architecture, we can find that the setup is divided into two parts. The setup in the upper pipeline handles the shape generation, and setup in the second pipeline helps the color generation. The shared terms of

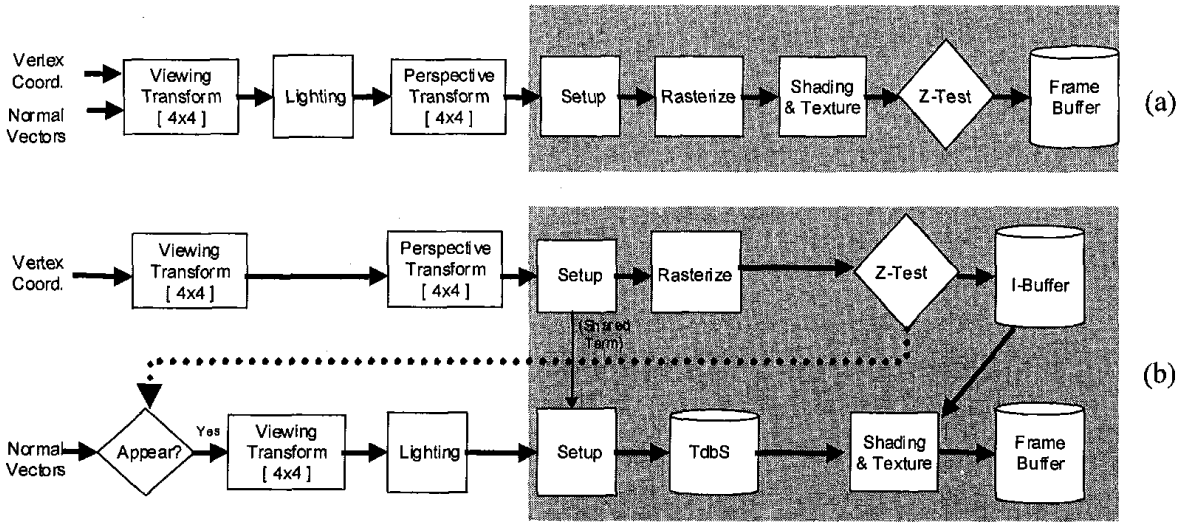


Fig.7 3-D Graphics Rendering Pipeline (a) Traditional (b) Proposed
(The blocks in gray area are handled by rendering hardware)

two setup blocks are the vectors of vertices: dx_{12}, dy_{12}, dx_{23} and dy_{23} . Because the vectors are appeared in both Eq.5 and Eq.6, the vectors can be reused to reduce operations.

4. Simulation and Analysis

The performances of index rendering and deferred lighting have been analyzed and simulated [4][5]. Hence, we will demonstrate the performance of the enhanced version of deferred lighting. Compared with our previous deferred lighting approach, enhanced deferred lighting can further eliminates the redundant transformations on invisible triangles. Because the CPU handles this part in embedded system, we focus on the reduction in the CPU's operations.

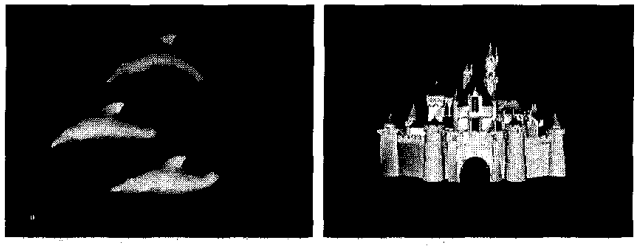
The Java [17] and Mesa [18] were utilized to develop our simulation environment and two 3-D object models, *Dolphins* and *Castle*, are applied, as shown in Fig.8 [19]. Their original triangle numbers are listed in Table 1. Because the Pineda's algorithm is applied, the triangles are not need to be divided into two scanline-aligned ones.

In this experiment, we measure the original triangle number and visible triangle number that passed the Z-test. For fair comparison, we also measure the triangle numbers after back-face culling, because some traditional rendering architecture performs this way before lighting. Hence, we refer the rendering pipeline that performs back-face culling before lighting as traditional architecture TYPE I, which is a better way to realize traditional architecture. The rendering pipeline performs back-face culling after lighting before triangle setup as traditional architecture TYPE II, which is a straightforward method to realize traditional architecture but wastes computation power. After simulation in resolution 320x200, we find the triangle numbers that can pass the back-face culling and be visible in final image. We denote the original triangle number as (a), the triangle number pass the back-face culling as (b), and visible triangle number as (c). The visible ratio equals the result of that visible triangle number (c) divides the triangle number that should be

Table 1. Triangle Number in Each Model by Simulation

Model Name	Triangle number			Visible Ratio	
	Original	After Back-faced Culled	Visible in Final Image	*1	*2
	(a)	(b)	(c)	(c) / (b)	(c) / (a)
Dolphins	3000	1451	1206	83%	40%
Castle	14122	6432	2828	44%	20%

*1 Compared with traditional archi. TYPE I (Back-faced culling before lighting)
*2 Compared with traditional archi. TYPE II (Back-faced culling before triangle setup)



(a) (b)

Fig. 8 Simulation Models (a) Dolphins (b) Castle [19]

Table 2. Numbers of Operations in Different Architectures

	Trad. Archi. Type I		Trad. Archi. Type II		Proposed Archi.	
	Vertex Coord.	Normal Vectors	Vertex Coord.	Normal Vectors	Vertex Coord.	Normal Vectors
Viewing Transform	(a) x 3	(a) x 3	(a) x 3	(a) x 3	(a) x 3	(c) x 3
Perspective Transform	(a) x 3	-	(a) x 3	-	(a) x 3	-
Lighting	-	(b) x 3	-	(a) x 3	-	(c) x 3

lighted in traditional architecture, i.e. (c)/(b) in TYPE I, and (c)/(a) in TYPE II.

Compared with original models, the results show that only 20% triangles in *Castle* and 40% triangles in *Dolphins* are visible. The results show that both traditional architecture TYPE I and TYPE II waste lots of unnecessary lighting and transformation operations on invisible triangles. The proportions are up to 80% in *Castle* model and 60% in *Dolphins* model. Although back-face culling technique is utilized in TYPE I, there are still lots of invisible triangles are lighted and transformed. We can find that the number in column (b) still larger than visible triangle numbers in column (c). The ratios of column (c) and (b) are 44% in *Castle* model and 83% in *Dolphins* model. It means that only 44% and 83% lighting and transform operations are needed in TYPE I, and others are unnecessary. The reduction is more significant in the *Castle* model. Because the *Dolphins* model illustrates the shape of animal, and its model is simple and convex. Hence there are fewer hidden triangle, especially after back-face culled. On the other hand, the *Castle* model is combined by lots of walls, and its walls cover each other. Therefore many triangles cover each other, and visible triangle numbers are relatively low.

In order to analyze total CPU operations, the instruction numbers of each operation are evaluated according to Direct3D transform pipeline [6], and the numbers of desired CPU instructions are illustrated in Fig.9. From the basis of Fig.9, we can analyze the CPU costs for transformations and lighting. On the three types of architecture: traditional architecture Type I, Type II, and proposed architecture. Because of no specified CPU and platform, we reasonably assume the costs of each CPU instructions as a basis to measure performance. The cost of *addition* instruction is 1, multiplication instruction is 2, and *reciprocal radical* instruction is 16. Because the lighting and transform operations are applied on vertices of triangle, the number of operations equals to three times of related triangle number. Triangle strip and fan are not discussed here for fair comparison.

Table 2 shows the numbers of each operation in different architectures. The (a), (b) and (c) denote the related triangle numbers, those refer to Table 1. The vertex coordinates and normal vectors are handled separately for analysis. The vertex coordinates are the information to generate triangle shape. Before applying setup for shape information, which we described in section 2.3.1, the viewing and perspective transformations are necessary in order to generate correct shape on screen. The transforms on vertex coordinates are essential and not reducible. Hence, in Table 2, the operation numbers on vertex coordinates all equal to three times of original triangle numbers, (a) x 3, no matter which architecture is applied. On the other hand, the operation numbers on normal vectors very depend on architecture. In order to perform lighting operations on each vertex, the normal vectors are necessary information. In traditional architecture without back-face culling (TYPE II), the number of lighting equals three times of original triangle numbers, (a) x 3. In traditional architecture with back-face culling (TYPE I), the number equals (b) x 3. In our proposed architecture, the number becomes (c) x 3. Due to the data in Table 1, we can find the improvement on reducing lighting operation. Besides, due to enhanced deferred lighting, the operation number of viewing transform also reduced into (c) x 3 on normal vectors.

Then, the CPU costs are analyzed to generate the 3-D graphics. Table 3 lists the analysis results. For example, if we need to know the *multiplication* operation count (MUL) to generate *Dolphins* model in traditional architecture TYPE I, we can calculate the number from data in Table1, 2 and Fig. 9. From Table 2, in order to generate 3-D model in traditional architecture TYPE I, the CPU need to handle (a) x 3 viewing transforms, (a) x 3 perspective transforms for vertex coordinates, and (a) x 3 viewing transforms, (b) x 3 lighting operations for normal vectors. Referred to Table 1, we know that in *Dolphins* model, (a) equals 3000 triangles, while (b) equals 1451. Referred to Fig. 9, we know there are 12 *multiplication* operations in viewing

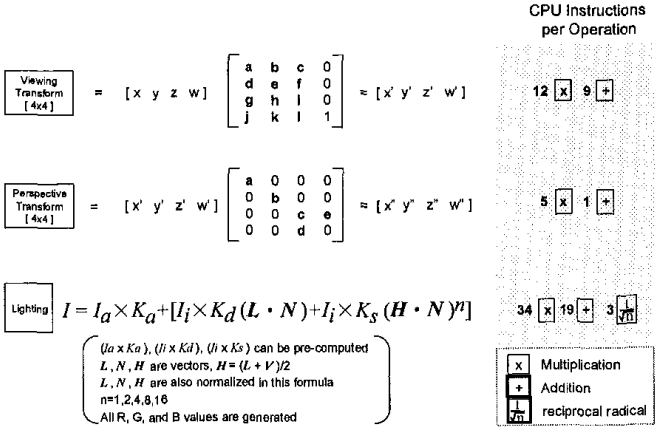


Fig. 9 CPU Instructions for Each Operations

transform, 5 in perspective transform and 34 in lighting. Therefore, the operation count of MUL is:

(3000x3x12+ 3000x3x5+ 3000x3x12+ 1451x3x 34) = 409002 (8)

Viewing Transforms for Vertex Coord.	Perspective Transforms For Vertex Coord.	Viewing Transforms for Normal Vectors	Lighting for Normal Vectors	Total Opts of MUL

Hence, we know the number of MUL is 409K for *Dolphins* model in traditional architecture TYPE I. By the same way, we can also know the *addition* number (ADD) is 254K, and the *reciprocal radical* operation number (Rec.Rad.) is 13K. After multiplied assumed instruction cost, we can calculate the equivalent CPU cost is (409K x 2 + 254K x 1 + 13K x 16) = 1281K.

After all equivalent CPU costs are calculated, we can compare the cost of three architectures in different models. Compared with traditional architecture TYPE I, we can find that the proposed architecture only needs 78.4% equivalent CPU cost in *Dolphins* model, and 56.1% in *Castle* model to generate the same results. In comparison with traditional architecture TYPE II, the ratio becomes more significant. The proposed architecture only needs 52.6% equivalent CPU cost in *Dolphins* model, and 36.6% in *Castle* model. Hence, in traditional architecture TYPE II, up to 100%-36.6% = 63.4% CPU operations in *Castle* model are redundant.

5. CONCLUSION

A new architecture is proposed in this paper for computation-effective 3-D graphics rendering in embedded multimedia system. It bases on our index rendering and enhanced version of deferred lighting approaches. Comparing with traditional architecture, its feature is dual pipeline rendering architecture. This architecture is computation-effective because it can render 3-D graphics image by fewer operations without image quality loss. We achieve this goal by eliminating the redundant operations on hidden pixels and invisible triangles.

By simulation and analysis in resolution 320x200, the result shows our dual pipeline architecture can reduce equivalent CPU cost into 56.1% ~ 78.4% compared with traditional architecture TYPE I, and into 36.6% ~ 52.6% compared with traditional architecture TYPE II. Hence, by analysis our dual pipeline design can save up to 63.4% CPU cost, and is very suitable for low cost embedded multimedia system.

REFERENCE

[1] K.Yoshida, T. Sakamoto and T.Hase, "A 3D Graphics Library for 32-bit Microprocessors for Embedded systems," *IEEE Tran. Consumer Electronics*, vol. 44, No.3, pp. 1107-1114, Aug. 1998

[2] J. Torborg and J.T. Kajiya, "Talisman: commodity realtime 3D graphics for the PC", SIGGRAPH '96, pp. 353-363, 1996.

[3] J. McCormack, R. McNamara, C.Gianos, L.Seiler, N.Jouppi and K.Correll, "Neon: a single-chip 3D workstation graphics accelerator," *Proc.Workshop on Graphics Hardware*, pp. 123-132, 1998.

[4] B.-S.Liang, Y.-C. Lee, W.-C. Yeh, and C.-W. Jen, "Index rendering: A hardware-efficient architecture for 3-D graphics," *Proc. VLSI/CAD Symposium*, Taiwan, pp.137-140, 1999.

[5] B.-S. Liang, W.-C. Yeh, Y.-C. Lee, and C.-W. Jen, "Deferred Lighting: A Computation-Efficient Approach for Real-time 3-D Graphics, " *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, p.p. IV.657-IV.660, Geneva, Switzerland, May 2000.

[6] Microsoft, "The Direct3D Transformation Pipeline", http://msdn.microsoft.com/library/backgrnd/html/d3dxfrm6.htm, Apr. 1998

[7] B.T.Phong, "Illumination for computer generated pictures,"

Table 3. CPU Cost for Geometry Transform and Lighting

Model Name		Trad. Archi. TYPE I			Trad. Archi. TYPE II			Proposed Archi.			Ratio of Equiv. CPU Cost *1	Ratio of Equiv. CPU Cost *2
		MUL	ADD	Rec. Rad.	MUL	ADD	Rec. Rad.	MUL	ADD	Rec. Rad.	Compared w/ TYPE I	Compared w/ TYPE II
Dolphins	Opts	409K	254K	13K	567K	342K	27K	319K	191K	11K	**78.4%**	**52.6%**
	Cost	2	1	16	2	1	16	2	1	16		
	Equiv. Cost	1281K			1908K			1004K				
Castle	Opts	1885K	1172K	58K	2669K	1610K	127K	1110K	661K	25K	**56.1%**	**36.6%**
	Cost	2	1	16	2	1	16	2	1	16		
	Equiv. Cost	5867K			8982K			3289K				

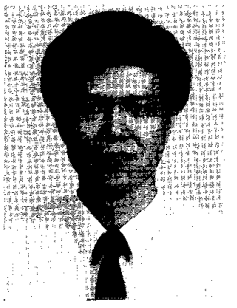
*1 Ratio of CPU cost = (Equal cost in Proposed Architecture) / (Equal cost in Traditional Architecture TYPE I)

*2 Ratio of CPU cost = (Equal cost in Proposed Architecture) / (Equal cost in Traditional Architecture TYPE II)

*3 Assume the cost of CPU instruction: Add = 1, Mul =2 , Reciprocal radical = 16

Comm. of the ACM, 18(6), pp. 311-317, Jun. 1975.
[8] 3Dlabs,*GLINT 300SX Programmer's Reference Manual*, p.p.42, Oct., 1994.
[9] J. Pineda ,"A parallel algorithm for polygon rasterization," *SIGGRAPH'88* , p.p. 17-20, 1988
[10] H. Gouraud, "Continuous shading of curved surfaces," IEEE Transactions on Computer, Vol.C-20, No.6, pp. 623-629, Jun. 1975.
[11] H.Fuchs, *et al.*, "Fast spheres, shadows, textures, transparencies, and image enhancements in Pixel-Planes," *SIGGRAPH '85*, Vol. 19, No. 3, pp. 111-120, Jul. 1985.
[12] J. Foley, A. van Dam, S.Feiner and J.Hughes , *Computer Graphics: Principles and Practice*, 2nd Ed., Addison-Wesley, 1990.
[13] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics", *SIGGRAPH '88* , p.p. 21-30, 1988.
[14] B.-O. Schneider and U. Claussen, "PROOF: An Architecture for Rendering in Object Space", Advances in Computer Graphics Hardware III, p.p.121-135, Springer-Verlag, 1991.
[15] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: high-speed rendering using image composition", p.p.231-240, *SIGGRAPH*, 1992.
[16] J. Eyles, S. Molnar, J. Poulton,T.Greer, A.Lastra, N.England and L.Westover, "PixelFlow: the Realization", *Proceedings of the 1997 EUROGRAPHICS/ SIGGRAPH workshop on Graphics hardware*, Pages 57- 68 , 1997.
[17] Sun Microsystems, "JAVA™ development kit version 1.1 software", http://java.sun.com/ products/jdk/1.1/, 1999.
[18] B. Paul, "Mesa - The free OpenGL work-alike library", http://www.mesa3d.org/Mesa/ Mesa.html, Feb. 1999.
[19] Silicon Graphics Incorporated, "OpenGL - more samples: Wavefront .OBJ file format model reader/writer manipulator", *OpenGL Developer Tools*, http://trant.sgi.com/ opengl/examples/more_samples/more_samples.html, 1997.

Bor-Sung Liang was born in Kaohsiung, Taiwan, R.O.C. in 1972. He received the B.S. degree in 1994 and M.S. degree in 1996 from National Chiao Tung University, Taiwan. He is a Ph.D candidate now in Department of Electronics Engineering, National Chiao Tung University, Taiwan, R.O.C.
His major research interests include VLSI design, 3D graphics rendering architecture and Internet architecture.



Chein-Wei Jen was born in Shanghai, China, in 1948. He received the B.S. degree from National Chiao Tung University in 1970, the M.S. degree from Stanford Uni-versity in 1977, and the Ph.D. degree from National Chiao Tung University in 1983.
He is a professor in the Department of Electronics Engineering and the Institute of Electronics, National Chiao Tung University, Hsinchu, Taiwan. From 1985 to 1986 he was a Visiting Researcher at the University of Southern California, USA. His current research interests include VLSI signal processing, VLSI architecture design, design automation. He is a member of IEEE and Phi Tau Phi.