A Hardware-like High-level Language Based Environment for 3D Graphics Architecture Exploration

Inho Lee, Joung-Youn Kim, Yeon-Ho Im, Yunseok Choi, Hyunchul Shin, Changyoung Han, Donghyun Kim, Hyoungjoon Park, Young-Il Seo, Kyusik Chung, Chang-Hyo Yu, Kanghyup Chun, and Lee-Sup Kim*

Multimedia VLSI Lab. Department of EECS, KAIST, 373-1
Guseong-dong, Yuseong-gu, Daejeon, 305-701, Republic of Korea
gate@mvlsi.kaist.ac.kr

ABSTRACT

The high complexity and the short lifetime of 3D graphics acceleration hardware increase the necessity of an environment for hardware development. For easy modification and fast testing of architecture, a high-level language based environment is desirable. Therefore, in this paper we propose a Graphics Architecture Testing Environment (GATE) that is based on Microsoft Visual C++. GATE models overall graphics hardware architecture through a modular approach, supports OpenGL, and offers easy modification and rapid testing of architecture. It also gathers computational statistics. A layered approach and Hardware Description Macro (HDM) support hardware modeling and architecture modification. Pre-defined types and operations provide statistical information. Several case studies of 3D graphics architecture on GATE show the capability of our environment.

1. INTRODUCTION

Recently, the capacity and the complexity of 3D graphics acceleration hardware have increased rapidly. On the other hand, the lifetimes of new accelerators have shortened greatly. Hence, fast modification and testing of complex architecture have become more important.

High-level language based design and a testing environment can be a solution. Unlike a low-level language such as HDL, a high-level language has the capability for a fast simulation and powerful debugging. However, since high-level language is not suitable for modeling hardware architecture, an intermediate layer is needed to help this modeling. Moreover, to test the new architecture on a target system, an API such as OpenGL [1] or DirectX [2] is also needed. In addition, statistical information that shows the advantages and feasibility of a modified architecture may be helpful for development.

In this paper, we propose an environment for developing 3D graphics acceleration hardware compatible with OpenGL. Based on C++ programming language, it models hardware, supports OpenGL, and offers easy architecture modification and computation statistics.

2. Previous Works

Ewins, et al [3] studied codesign of graphics hardware architecture. They emulated the rasterization engine part of the graphics pipeline with software functions named process elements and data input/output stacks among them for pipelining. Each graphics pipeline operation is modeled with a process element such as 'scan conversion', 'illumination', and so on. A function call is used for insertion and deletion of the process element. Using this, pipeline organization becomes very easy, but complicated architecture modification is difficult because the researchers assumed the architecture as a simple single pipeline. This assumption makes the

modification easy, but restricts its scope. Furthermore, since they modeled just the rasterization part, a part of the geometry engine aspects cannot be developed in their environment.

Mesa is an OpenGL compatible graphics library [4]. Mesa realized an OpenGL state machine and architecture on C language, and since the source code is open to the public, researchers or developers can modify and test the architecture for advanced algorithms. However, since Mesa is designed from a software point of view and the source code is optimized for software simulation, it is difficult to model and modify hardware architecture. For example, a developer must change several blocks to modify a block in Mesa because the function blocks are not modular.

3. Graphics Architecture Testing Environment (GATE)

GATE targets simple and rapid modification and testing of 3D graphics accelerator architecture based on an OpenGL compatible PC environment.

3.1 Features

GATE's essential features are as follows.

1. GATE is high-level language based.
GATE is based on C++ for the advantages of a high-level language, such as fast simulation, easy modification, and powerful debugging.
2. GATE is OpenGL-compatible.
Since GATE is compatible with OpenGL, it can substitute OpenGL library like Mesa. Not only external API but also internal architecture follows OpenGL architecture; thus GATE may quickly become very familiar to 3D graphics hardware developers.
3. GATE models the entire 3D graphics acceleration hardware architecture.
GATE models the 3D graphics pipeline from the geometry engine to the rasterization engine. To make hardware description on a high-level language easy, a C++ macro type intermediate layer is supplied.
4. GATE provides simple architecture modification and testing.
Since the control and datapath layers are separated from each other and each datapath block is independent of each other, block insertion, deletion, reordering, or modification can be performed locally.
5. GATE provides statistical information for complexity comparison.

Statistics about computations are very important information in selecting the intersecting point of quality and complexity. With this statistical information, a developer can decide whether to

* co-author

manufacture new architecture without actual hardware realization.

3.2 Layered Approach

For simple architecture modification, GATE has two separate layers, a control layer and a datapath layer, as shown in Figure 1. The control layer describes the operating sequence and dependency of datapath blocks. The control layer is composed of two layers. A pipeline layer represents block-operating sequences, and a link layer represents dependency between blocks. These two layers are just conceptually classified, and activate the datapath layer together.

Figure 1. Layered approach

The datapath layer is composed of hardware-modeling blocks. For easy insertion, deletion, replacement, reordering, and modification of blocks, these blocks are completely independent. Each datapath block represents a graphics pipeline operation similar to the block of the OpenGL state diagram [1].

Through this layered approach, a hardware developer can implement and test new architecture easily. For block insertion, a new block is first described, and then the pipeline and link layer of the control layer are modified. For reordering or deletion, it is only necessary to modify the pipeline and link layer of the control layer. For some cases, additional modification of the inputs and outputs of blocks is needed. For replacement, the new block is simply described with the same name as the block that is replaced.

3.3 Hardware Description Macro (HDM)

Generally, graphics architecture is developed by numerous researchers. Therefore, standard rules for describing a system are essential. For consistency, expansibility, and understandability, GATE restricts the blocks with their inputs and outputs with Hardware Description Macro (HDM). The exterior interconnections and the interior functions of blocks are completely separated from each other. By leaving the inside of each block as a black box, developers can freely describe the function of blocks with C++. This is similar to the approach of low-level languages such as HDL.

HDM is composed of C++ macros such as BLOCK, INPUT, OUTPUT, and so on. To reduce additional compilations, we adopt C++ macros that are basically based on C++ classes. HDM is suitable for describing a block diagram with inputs and outputs at algorithm level. Using HDM, it is easy to insert or delete a block in a system and easy to understand a system. The Flip-Flop example of HDM is shown below.

```
BLOCK( FlipFlop, PORT_IN( int TYPE in, int TYPE reset ),
      INPUT   ( int, in );
      INPUT   ( int, reset);
      OUTPUT  ( int, out);
      CTRL    ( clockFF );
      ) LINK_IN(in), LINK_IN(reset)
    {
      reset =0;
      out = 0;
    }
    CONTROL( FlipFlop, clockFF )
```

```
{
  if(reset)
    out = 0;
  else
    out = in;
}
```

A datapath block is represented with a macro BLOCK. A block can include other blocks as its sub-block. INPUT and OUTPUT macros are used for the block's inputs and outputs. The macro PORT_IN and LINK_IN describe the order of inputs. The main operation of the block is described in CONTROL. A block can have multiple CONTROLS. In CONTROL, an algorithm using standard C++ instructions can be freely used. In this Flip-Flop example, there is only one control and only one job for a control signal, *clockFF*. When the *clockFF* is called, according to the *reset* input, the output can be either a value of 0 or *in*.

4. Statistics

To show the efficiency of a new algorithm or hardware architecture, precise statistical results are essential. GATE enables users to gather statistical data traces.

4.1 Features

Arithmetic and logic operations of all primitive types in C/C++ and additional types supported in GATE are counted automatically, and the total number of block executions is also counted. For example, we can obtain the count of floating point multiplications that occur in the lighting block or the number of memory accesses from the texturing unit. This basic counting is performed in the background by GATE without the user's notification. Additional user-specified data can be obtained with simple modifications.

Since the place and the time of the events are the important factors, GATE supports a modular collection of data by block for place information and an assignment of the start and the end time of data gathering for time information.

The names of the block from which the statistical data are obtained are listed in an input script. GATE generates a default script for all blocks, and then users can modify this script easily for desired blocks. GATE then gathers the statistical data from the blocks listed in the script. GATE can have multiple scripts.

The start and the end time of the statistical data gathering are assigned by additional API functions in the rendering code. These APIs enable the users to obtain the data for some objects or certain pixels of interest. Figure 2 shows an input and an output diagram of the GATE statistics simulator.

4.2 Implementation

To obtain the statistical data of operations automatically, two kinds of information are needed, the type of operations executed, and the block in which the operations occur. The methods for obtaining this information are an operation overloading and a static variable activeObj in CBase class, as shown in Table 1.

Since no statistical information can be extracted from basic operations of original primitive data types in C/C++, new operators overriding basic operators are used. In addition, new classes replace all primitive types in order to inform operand types. The statistics of operators can be derived from the overridden operators, and the kind of operand can be derived from newly defined classes that replace primitive types.

All blocks are inherited from class *Cbase*, which defines the common block features, block hierarchies, and statistical supports. Class *Cbase* has a static variable named *activeObj* that points a

block object under execution. Since the type of variable *activeObj* is a pointer type of class *Cbase*, it can point all blocks inherited from class *Cbase* by polymorphism. Setting variable *activeObj* is processed automatically by HDMs, which define blocks and control functions. It is possible to obtain the information where an operation is executed by referring to the variable *activeObj*.

Figure 2. Statistics simulator

Which operations are executed.
1. Pre-defined class types instead of the primitive types.
2. Operator overloading for all operations.
In which block operations are executed.
1. Block inheritance from <i>CBase</i> class including <i>activeObj</i> (active object) variable.
2. Macros including the code to set <i>activeObj</i> .

Table 1. Methods for statistics

5. Case Studies

For examples of architecture modification, two case studies are provided. The first is culling, and the second is texture mapping.

5.1 Culling

In surface rendering of airtight objects, the back-faces of polygons cannot be seen. Culling reduces needless rendering efforts by eliminating these back-faced polygons. On a traditional OpenGL pipeline, the culling is located at the end of the geometry engine [1]. Therefore, just the rasterization parts of graphics pipeline benefit from culling. If we move this culling operation to the front of the pipeline, the geometrical operations after culling may profit from the reduced amount of triangles. To show the decreased computational effort, the culling block is moved to the front of a lighting block and tested on GATE. Since culling is performed on triangle, to move the culling block to the front, a primitive assembly block, which composes vertices into triangle, must be moved together with the culling block. The inputs and outputs of the blocks between the original position and this new position of the primitive assembly block must be modified because this block changes vertices to triangles. The original and modified diagram of the GATE structure is shown in Figure 3.

To test this reordering on GATE, the control layer must be modified for block reordering and linkage, and the datapath layer for the change of the inputs and outputs. Almost all geometrical processing blocks have advantages from the reduced triangles, and these blocks are represented as gray boxes in Figure 3. Since culling usually eliminates about fifty percent of triangles, the computational efforts of these blocks are reduced into nearly half.

The back-face test operation is performed on the 2D window coordinate as follows [1]

$$x_1 y_2 + x_2 y_3 + x_3 y_1 - x_2 y_1 - x_3 y_2 - x_1 y_3 \quad (1)$$

where x and y represent the window coordinates of the vertices of a triangle, and the subscripts indicate the vertex number. Since the new location of the culling block precedes the projection from 3D to 2D, the above expression is modified as follows.

$$x_1 y_2 z_3 + x_2 y_3 z_1 + x_3 y_1 z_2 - x_2 y_1 z_3 - x_3 y_2 z_1 - x_1 y_3 z_2 \quad (2)$$

As expressed in (2), six additional multiplications are needed per triangle for culling. But this increase is small compared to the overall reduction of operations.

The models for testing are shown in Figure 4, and the input and culled triangle counts are shown in Table 2. As mentioned, culling diminishes the number of triangles by about half. The statistical results of the migration of the culling block are shown in Figure 5 for the operations of addition, multiplication, and comparison. About a half of the operation count is reduced. The benefit of multiplication is small compared to that of addition or comparison due to the new culling expression, but the gain is still 47%. These statistical results are indications of the effectiveness of the relocation of the culling block.

Figure 3. Culling

Figure 4. Models for culling test

	Input	Culled	Gain (%)
Dinosaur	4587	2363	51.515
Horse	7257	3783	52.129
Chair	10062	5086	50.547
Turtle	13120	7223	55.053
Average			52.311

Table 2. Triangle counts

Figure 5. Operations counts for geometry engine

5.2 Texture mapping

Texture mapping is one of the most useful methods for real-time generation of a photorealistic image. It determines the color of a screen pixel by referencing a pre-generated texture map. The area of a screen pixel in the texture map space is called a footprint.

Since the footprint does not correspond to a texel, i.e. a pixel in a texture map, some filtering methods are needed. Since tri-linear filtering [1] is simple and effective, every recently announced 3D graphics accelerator basically adopts this method for texture filtering.

In traditional tri-linear filtering, the footprint is assumed as a square that contains the screen pixel area as shown in Figure 6(a). The difference between the pixel footprint and squared footprint results in a degradation of image quality. This difference is represented as a grayed region in Figure 6. Since the pixel footprint becomes long and narrow as the angle of polygon is more slanted, the quality also becomes poorer. Footprint assembly solves this problem by imitating the long footprint by multiple base squared footprints as shown in Figure 6(b). The area difference is greatly reduced in footprint assembly. The size and number of base squared footprints are determined with the major axis, minor axis, long side, and short side of the pixel footprint. For implementation of footprint assembly in hardware, since the number of base squared footprints is directly related to the texture memory referencing and filtering, a restriction of the anisotropy – the number of base squared footprints – is needed. NVIDIA GeForce limits this anisotropy up to 2:1 [6, 7] and ATI Radeon up to 16:1 [8]. The question then is how to determine this limitation on development. GATE can solve this problem. We tested tri-linear filtering and footprint assembly on GATE and compared their statistics. Through the change of anisotropy and its related test, the crossing point of quality and performance can be selected from their statistics.

The statistical results according to the variation of maximum anisotropies are shown in Figure 7. As the limitation of anisotropy increases, the count of squared footprints, memory access, and other arithmetic operations also increases. Since these increments are saturated as the limitation reaches 16:1, the reasonable anisotropy for best quality may be 16:1. If the limitation of memory access or other operations is lower than the count of 16:1, the developer must select the largest anisotropy within the limitation. GATE expedites this deciding process. In Figure 8, the change of the maximum anisotropy as the variation of oblique viewing angle is shown. This graph is similar to that in the paper of J. McCormack [9].

6. Conclusions

GATE is an environment for graphics hardware architecture development. GATE is designed with HDM to satisfy the need for modeling graphics hardware architecture in a high-level language with a hardware-like description. Since it is based on a high-level language, fast modeling and verification are possible, and computation statistics can be easily acquired

We adopted a layered modular structure for these features. The control layer describes pipeline order and linkage, and the datapath layer covers modularly subdivided graphics functions. The statistics of every supported data type and operation are collected block by block for a user-specified duration.

In addition, a convenient debugging method may be helpful for developing new architecture. A pixel-based back trace from a generated image to a graphics pipeline with the reverse order will simplify the debugging.

Figure 6. Tri-linear filtering and footprint assembly

Figure 7. Statistical data for footprint assembly

Figure 8. Maximum anisotropy as the viewing angles

7. References

- [1] M. Segal, K. Akeley, C. Frazer, and J. Leech. The OpenGL Graphics System: A Specification (Version 1.2.1). *Silicon Graphics Inc.*, 1999.
- [2] DirectX Homepage. www.microsoft.com.
- [3] J. P. Ewins, P. L. Watten, M. White, M. D. J. McNeill and P. F. Lister, Codesign of Graphics Hardware Accelerators. *Proceedings of the 1997 SIGGRAPH /Eurographics Workshop on Graphics Hardware*, pages 103-110.
- [4] B. Paul, Mesa 3d Graphics Library. www.mesa3d.org.
- [5] A. Schilling, G. Knittel, and W. Straßer. TEXRAM: A Smart Memory for Texturing. *IEEE Computer Graphics and Applications*, pages 32-41. May 1996: 16(3).
- [6] C. Everitt, Anisotropic Texture Filtering in OpenGL. *NVIDIA White Paper*. September 2000.
- [7] M. J. Kilgard. NVIDIA OpenGL Extension Specification, *NVIDIA Corporation*. May 2001.
- [8] ATI. Radeon Charisma Engine and Pixel Tapestry Architecture. *ATI White Paper*, 2000.
- [9] J. McCormack, R. Perry, K. I. Farkas, and N. P. Jouppi. Feline: Fast Elliptical Lines for Anisotropic Texture Mapping. *Computer Graphics (Proceedings of SIGGRAPH 99)*, pages 243-250.