An Optimized Soft 3D Mobile Graphics Library Based on JIT Backend Compiler

Bailin Yang[1,2], Zhigeng Pan[1], Qizhi Yu[1], and Guilin Xu[1]

[1] College of Computer Science, Zhejiang University
310027 Hangzhou, China
{ybl,zgpan,yqz,xuguilin}@cad.zju.edu.cn
http://www.cad.zju.edu.cn/vrmm/index.htm

[2] Zhejiang Gongshang University, Hangzhou, 310035, P.R. China

Abstract. Mobile device is one of the most widespread devices with rendering capabilities now. With the improved performance of the mobile device, displaying 3D scene becomes reality. This paper implements an optimized soft 3D mobile graphics library based on JIT backend compiler, which is suitable for the features of the mobile device. To deeply exploring the advantages of JIT technology, this paper improves the traditional rasterization model based on JIT technology and proposes a hybrid rasterization model which integrates the advantages of both the per-scanline and per-pixel rasterization models. As we know, the backend compiler is the critical factor in running 3D application programme. In this paper, we implement a backend compiler for certain CPUs and propose some optimization techniques accordingly. The experimental results indicate that our 3D graphics library has achieved fine performance.

1 Introduction

Mobile graphics is the 3D and 2D graphics which are used in the embedded devices such as the PDA, mobile phone etc. For the 3D graphics, there are the following standards: OpenGL ES constituted by Khronos, JSR 184 by Nokia and Direct3Dm by Microsoft. OpenGL ES, current specification version is 1.13, is a well-defined subset of desktop OpenGL and supported by many mobile manufacturers, companies and research institutes. The specification is targeted primarily at embedded devices that have the drawbacks including lacking of float point, small amounts of memory, little bandwidth, and limited power consumption. Therefore, it deletes many unusually used functions and modifies some APIs according to the embedded devices' features [1].

There are two kinds of implementation based on the OpenGL ES specification: hardware implementation and soft implementation. Now, many manufactures such as ATI, NVIDIA and PowerVR have developed different embedded video cards. In contrast to soft implementation, hardware can greatly improve the performance of 3D rendering, but it also has some disadvantages such as the high cost and long research period. For the soft implementation, the greatest advantage is its cheap price and satisfactory effect if we achieve excellent design and implementation. There also have some excellent implementations in the market including Hybrid's OpenGL? ES API Framework[2] and HI's Micro3D[3]. In our research group, we also have implemented our soft implementation M3D. Now, Hybrid has good corporations with Nokia, Philips and other mobile devices manufacturers, which indicates that soft implementation has a bright future.

In our research group, we have implemented a soft mobile graphics library, M3D, conformed to the OpenGL ES Specification 1.0 and worked out many 3D games based on M3D. The most critical disadvantage of soft implementation is the low performance, which leads to the slow running speed and screen flicker. To solve these problems, many techniques, algorithms and architecture have been put forward. At present, there exist mainly two ways to improve the performance: pipeline architecture and system level optimization.

(1) 3D pipeline optimization. There are many different algorithms and pipeline architecture suitable for the mobile graphics such as Tile-based rendering [4], Early Culling and Deferred Shading[5,6], Vertex caching technology and its optimization[7], inexpensive multisampling scheme and texture compressing[8], etc.

(2) System level optimization. This method is very important for soft implementation. In practice, we often adopt the following techniques. 1) High programming language optimization for certain embedded device CPU. 2) Using the assembly language to rewrite source code instead of the high-level language which consumes more CPU resources. 3) Adopting JIT backend compiler to improve the performance [2,9,10].

JIT compiler, which is often used in Java language, dynamically translates the high-level language code into the machine code during runtime. JIT compiler produces different machine codes for different types of CPU chips dynamically to meet the portability requirement. In 3D graphics pipeline, the JIT technology is often used due to the following characteristic. From the most abstract point of view, a renderer is a state machine. In the whole rendering, testing the states can actually take longer than the real rendering. But these states don't change all that often in fact. Thousands of pixels are all rendered using the same state. Using the JIT compiler, we can successfully eliminate the redundant state checking, and only execute the functions that perform the render operation corresponding to current state.

This paper focuses on adopting JIT technology, rasterization model and backend compiler to improve the M3D performance. It is organized as follows. First, some traditional rasterization models are introduced. Second, a hybrid rasterization model which is suitable for the JIT technology is presented. Then the JIT backend compiler we implemented is introduced and some techniques and methods are proposed to improve its performance. After that, the experimental result is illustrated and discussed. Finally, a conclusion and future work are offered.

2 Improved Rasterization model based on JIT

2.1 Traditional Rasterization Model

The rendering process consists of two stages: geometry processing, and rasterization. In the geometry processing stage, triangle vertices are transformed from object space into screen space. In the rasterization stage, a triangle is converted into pixels, which are depth buffered into the frame memory. Because rasterization stage consumes more CPU resources [11], how to improve its performance is the most important problem for 3D soft implementation. In Section 1, some common techniques are introduced to solve this problem. Then, we will further analyze it from both the rasterization pipeline and JIT compiler in the following sections.

Basically there are two models for the rasterization, which are per-scanline and per-pixel. In the per-scanline case, there is an early test for whether each operation (texturing, shading, etc.) will contribute to the pixels in question and later these textured pixels are combined with the resultant pixels of other operations, such as shading, fog, etc. In the early test, Z-test is to be done at the very end after all other operations have already been executed. This means that although a complete scanline might not be visible we still have to texture, shade, fog every pixel. The disadvantage of this solution is that it is quite slow. It is easy to see that this buffer filling and combining takes a lot of memory bandwidth, which is a serious bottleneck on mobile platforms [12]. Per-pixel based rasterization is a popular model in the hardware 3D rasterization pipeline.

In this model, we would have to do some different tests for each pixel such as z-test, texturing-test, shading-test, fogging-test, blending-test, etc. Generally, this model is slower than the per-scanline model for soft implementation. However, we can adopt some ways to improve it for example bringing forward the Z-test operation. To this model, the most amazing benefit is that the advantages of JIT technology can be fully explored for soft implementation. JIT backend compiler can execute all the pixel related operations, which are the most time-consuming operations during the whole rasterzation stage.

2.2 Hybrid Rasterization Model Based on JIT Backend Compiler

As introduced in Section 1,, JIT backend compiler has been used in the 3D graphics pipeline because of its particularity. The per-pixel model employs the JIT backend compiler to dynamically compile and translate the intermediate code into machine code. First, it rewrites the fragment functions with intermediate language, which are the most time-consuming function. Second, the different fragment functions are compiled when they are needed. Finally, the machine code corresponding to the fragment function will be executed by the application.

This technique improves the program running speed and achieves better result. Unfortunately, this method has the following shortcomings. 1) The machine code for different fragments functions seems too lengthy. Function cache is used to store the machine code. Comparing with PC device's cache, the capacity of mobile devices' cache is limited even less of 1M. 2) Lots of repeated codes exist in the fragment function, which is not beneficial to the cache performance. 3) The function schedule algorithm is not the best. 4) The frequently-used state functions have not been made the best use.

To solve the above problems, we propose a hybrid model that is based on both the per-scanline and per-pixel model. This method employs the JIT technology just like the per-pixel model, which dynamically compiles the fragment function. Furthermore, we adopt the fixed function, which is often used by 3D soft implementation, to advance the performance [13].

During the running of general 3D scene, for instance, running the real-time 3D game, some pipeline state combinations are often invoked. These state combinations have a fixed set of state setting, such as only the shading, shading and texturing combined. There exists a certain frequently-used function corresponding to each state combination in the programme,. To make for improving the programme performance greatly, we put these frequently-used functions into the function cache and store them permanently.

Before the geometry stage, the pre-compiler stage is introduced into our hybrid model, in which the JIT backend compiler is used to compile the frequently-used functions. These functions are directly written into the function cache like the per-scanline model rather than per-pixel model. By this way, we can make best use of these frequently-used functions, which are executed fast because they are stored in the function cache all the time. What's more, this method saves the code space efficiently, since the frequently-used functions' source code is smaller than the fragment functions'.

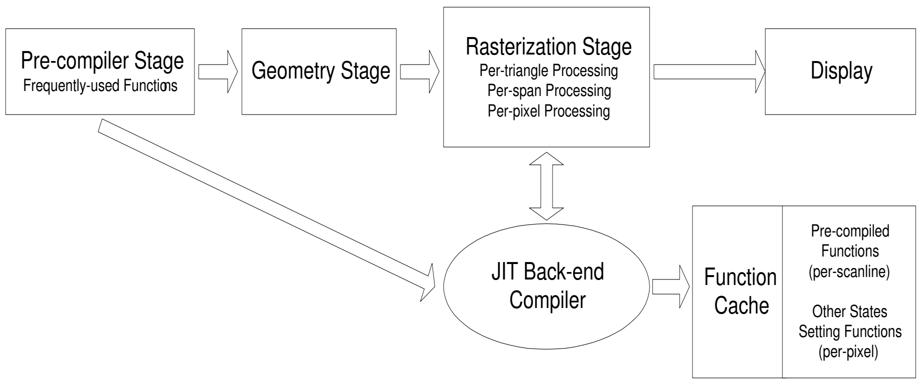

Fig. 1. Framework of hybrid rasterization model

The hybrid rasterization model consists of three stages including pre-compiler stage, geometry processing stage and rasterization stage. The framework of our model is illustrated in Fig. 1. A typical 3D application will go through the process as follows.

- **Step I**: Pre-compiler stage. When the 3D application begins to run, we utilize the JIT back-end compiler to translate the intermediate code of frequently-used functions into machine code and store it in the cache.
- **Step II**: Geometry stage. This stage is independent of the JIT backend compiler. We just operate it as usual.
- **Step III**: Rasterization Stage. In this stage, we first perform the prepare operation, that is to obtain certain function's address from the cache if current state setting is the same as this function's state setting. Otherwise we will compile the function corresponding to current state. We can use the function cache schedule algorithm to decide how to do it. Second, we perform the following steps: per-triangle processing, per-span processing, and per-pixel processing.
- **Step IV**: Display the result.

Function Cache Schedule Algorithm can be described as follows.

1)When the combined state setting appears, it should be compared with the state settings for the frequently-used functions. If the result is equal, then the function in the cache will be executed later. Otherwise the current state setting will be compared with the other state settings of the functions again. If there exists the same state setting, we will also execute this function corresponding to the current state setting later.

2) If we can't find any same state as the current's after two comparisons, the JIT back-end compiler will compile this function and store the machine code in the function cache.

3) As we know, the function cache's capacity is limited. There are so many different combined state settings for the whole 3D pipeline that we can not store all the machine code of the function into the function cache. Therefore, we should design a high performance schedule algorithm to replace and adjust the functions in the cache. In our implementation, we use an advanced LRU (Least-Recently Used) algorithm to replace the function which has not been used or frequently used recently when the cache is full. The difference between our algorithm and the common LRU algorithm is that we classify all functions in the cache with different priorities. The pre-compiled frequently-used functions are assigned the top priority and the others are assigned different priority according to their executive numbers. When the RUN algorithm is performed, we will replace certain function with current compiled function according to the priority. In this way, the pre-compiled frequently-used functions usually will not be replaced.

3 JIT Backend Compiler

3.1 Efficiency Analysis

JIT backend compiler is one of the dynamically compile technology which compile the source code into the machine code. We define the total execution time for certain function in the function cache, which consists of two main parts.

$$T(total) = Tc + n * Te \tag{1}$$

Tc is the time for compiling this function. Te is the time for executing the function's binary code in the cache. In order to get the minimum T(total), we must decrease both the Tc and Te . However, there seems to be a relationship between Tc and Te. Generally, when compiler time is short, the execute time will be long and vice versa [14]. To optimize the programme fully, we should find a balance between both sides. Considering the features of 3D application's execution and our hybrid model, decreasing the Te is the key factor for optimizing our application, because of the following reasons: 1 As indicated in Section 2, the functions in the cache will be called many times and then the number of n will be up to 100 or more, especially for the pre-compiled function. When n is large enough, the total time T(total) will be equal to n* Te.2 The Tc for pre-compiled function does not occupy the application execution time. This case is just like the static compile function, whose compile time is ignored by the user.

3.2 Framework of our JIT Backend Compiler

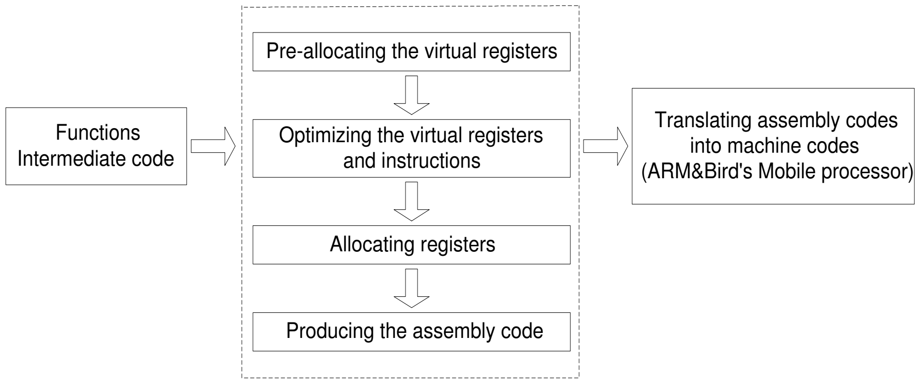
Functions Intermediate code
Pre-allocating the virtual registers
Optimizing the virtual registers and instructions
Allocating registers
Producing the assembly code
Translating assembly codes into machine codes (ARM&Bird's Mobile processor)

Fig. 2. Framework of JIT backend Compiler

Fig. 2 is the framework of our JIT backend compiler (BJIT). As a backend compiler, there is no need for the BJIT compiler to do the lexical parsing, grammar parsing and semantic parsing, just only to convert the intermediate code into target machine code. In BJIT compiler, we redefine intermediate code which is similar to assembly language and related to the target machine's CPU instruction set. For the purpose of implementing our BJIT on both the ARM processor and Bird Company's special processor, we should define two sets of intermediate code. To produce the target machine code, we first rewrite the origin function with the intermediate language.

Secondly, these functions will be compiled into the corresponding assembly language for different processor by our BJIT. This process consists of the following steps. 1) Pre-allocate the virtual register. 2) Optimize the virtual register and pseudo instruction.3) Registers allocate using the Graph Coloring algorithm. 4) Producing the assembly code suitable for the target machine processor. This stage is the key factor for improving the performance of BJIT. We will perform most of our optimizations in this stage. After getting true assembly codes, we translate them into machine codes suitable for ARM or Bird Company's processor and store the generated codes in the function cache.

3.3 Optimization of BJIT Compiler

Because there are frequent memory write and read operations in our programme, we could do the optimizations from several aspects including Register Allocate algorithm, Copy propagation and spill elimination. We will adopt the linear-scan algorithm [13] instead of the Graph Coloring algorithm, which is not so well for the register allocating in our programme. Copy propagation and spill elimination are classical optimizations in compiler technology, which are used to delete the redundant copy and store operations. The optimizations play an important role for our programme, because a large numbers of copy and store operations exist. Up to now, we have finished this BJIT compiler successfully. The optimizations for the BJIT are the future work.

4 Experiments and Results

Our work in this paper is based on the M3D, which is conformed to the OpenGL ES 1.0 speciation and runs on Bird E868 and HP iPaq Pocket PC. M3D is implemented with standard C language rather than C++ language for portability because some mobile's compilers do not support C++ language. The motivation for us to explore the M3D is to provide the 3D library for developing 3D mobile game. To evaluate our hybrid rasterization model's performance, we use two games, which are the maze and the magic cube, and different number of points, lines and triangles as our benchmarks. The hardware for our experiment is on HP iPaq equipped with ARM processor, which can reach 400MHz frequency at most.

Table 1. The Performance Comparison of our hybrid model library and M3D library

	Hybrid Rasterization model	M3D library without JIT
2 Triangles	22.546667	61.375000
960 Lines	45.000000	156.060608
1920 Points	160.409088	189.615387

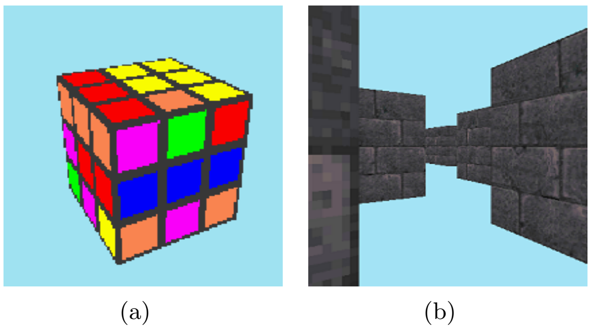

Fig. 3. Screen shots from our 3D mobile game Maze and Magic.

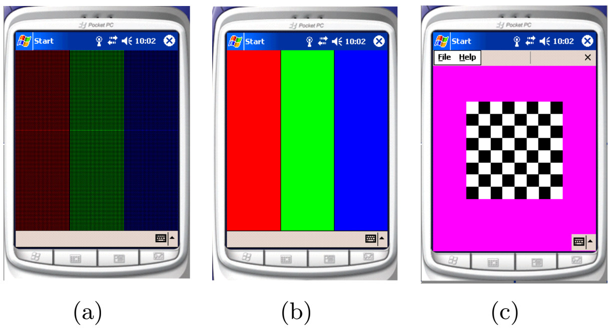

Fig. 4. Screen shots from our experiments. (a) contains 1960 points. (b) contains 960 lines. (c) contains 2 triangles with texture.

Table 1 shows the performance comparison for the above benchmark. The second column is the performance data for our hybrid rasterization model of M3D library with our BJIT, while the third column is for M3D library without JIT. The performance data in the table is the running time for each frame in ms (micro second).

Comparing the two columns, we see that the performance has been improved greatly by means of our hybrid rasterization model based on JIT backend compiler. For rendering two triangles with texture, the efficiency increases almost 2.72 times. For lines, it achieves a more significant improvement to 3.47 times. However, there is no remarkable effect for points. As a result of our experiments, triangle rendering, which is the primary operation in 3D application, achieves promising efficiency with our hybrid model based on BJIT. Due to this improvement, the maze game, developed on our M3D library, runs more smoothly than before.

5 Conclusion and Future work

We have implemented an optimized soft 3D mobile graphics library based on JIT backend compiler. In this novel 3D library, a hybrid rasterization model, which fully makes use of the JIT backend compiler and integrates the advantages of both the per-scanline and per-pixel rasterization models, is proposed for improving its performance. Furthermore, we have implemented our own BJIT backend compiler, which is targeted at the ARM and Bird Company's mobile processor.

However, there is still much work to do in the future for improving M3D's performance. The better management of function cache is important especially the function schedule algorithm during the execution of global programme. Furthermore, the optimization for our BJIT backend compiler is the urgent task for us in the next phase.

Acknowledgment

This project is co-supported by 973 project (grant no: 2002CB312100), Excellent Youth Teacher Program of MOE in China, and the Bird Company. We would like to thank Mr. Shijie Wang and Mrs. Yuanjun Liu from Bird Company for their kind help in this project. They have given us some good suggestions and methods on improving the 3D library. Also, I would like to thank Mrs. Haihong Wu for her kind help in writing this paper.

References

1. van Leeuwen, J. (ed.): Computer Science Today. Recent Trends and Developments. Lecture Notes in Computer Science, Vol. 1000. Springer-Verlag, Berlin Heidelberg New York (1995)
2. OpenGL ES Overview. http://www.khronos.org/embeddedapi/.
3. OpenGL ES API Framework, http://www.hybrid.fi/main/esframework/index.php.
4. Mascot Capsule Engine Micro3D Edition,http://www.hicorp.co.jp/.
5. D. Crisu, S. D. Cotofana, S. Vassiliadis, P. Liuha, Efficient Hardware for Tile-Based Rasterization, Proceedings of 15th Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC 2004), pp. 352-357, Veldhoven, The Netherlands, November 2004.
6. S. Kumar, D. Manocha, B. Garrett, and M. Lin. Hierarchical back-face culling. In 7th Eurogmphics Workshop on Rendering, pages 231-240, Porto, Portugal, 1996.
7. S. Kumar, D. Manocha, William F. Garrett, Ming C. Lin: Back-Face Computation of Polygon Clusters. Symposium on Computational Geometry 1997: 487-488.
8. Alexander Bogomjakov, Craig Gotsman: Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes. Computer Graph Forum 21 (2): 137-148 (2002)
9. T. Akenine-Moller and J. Strom, "Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones", ACM Trans. on Graph.,vol 22, nr 3,2003, pp. 801-808.

10. swShader, http://sw-shader.sourceforge.net/technology.html.
11. A 3-D Rendering Library for Mobile Devices, http://ogl-es.sourceforge.net/architecture.htm.
12. D. Crisu, S.D. Cotofana, S. Vassiliadis, and P. Liuha,"GRAAL- A Development Framework for Embedded Graphics Accelerators", Proc. Design, Automation and Test in Europe (DATE 04), Paris, France, February 2004.
13. Klimt software architecture overview, http://studierstube.org/klimt/documentation.php .
14. I.H. Kazi, H. H. Chen, B. Stanly, D.J. Lilja,"Technique for Obtaining High Performance in Java Programs,"ACM Computing Surveys,2000, Vol.32, No. 3, September,pp.213-240.
15. M. Polwrro, V. Sarkar, Line scan register allocation, ACM transactions on Programming Languages and System, vol.21, no.5, Sep.1999, pp.895-913.