안드로이드 디바이스에서의 3 차원 모델 렌더링 속도 향상[1]

응 총 지에*, 강 대 기 (교신저자)**
*말레이지아 멀티미디어대학교 정보기술학과
**동서대학교 컴퓨터공학전공
e-mail : ncjlee78@gmail.com, dkkang@dongseo.ac.kr

Speeding up the 3D Model Rendering on Android Device

Cong Jie Ng*, Dae-Ki Kang (corresponding author)**
*Faculty of Information Technology, Multimedia University, Malaysia
**Division of Computer and Information Engineering, Dongseo University, South Korea

Abstract

Rendering complex 3D model on smart mobile device with limited processing power and memory is challenging. Without optimization, the complex 3D model cannot be rendered smoothly. Special techniques are required to take into account to speed up the processing. In this paper, we will discuss about some approaches to alleviate the problem.

1. Introduction

The emergence of smart mobile device including smart phone and tablet is becoming very popular. The usage of touch screen technology, gyroscope, accelerometer and etc on smart mobile device creates new way of interaction on the device. This also creates new way of implementing interactive game and high demand of games on smart mobile device. There are many games have been developed which utilize these technology. The game becomes more and more complex and requires intensive computational power especially games with high resolution 3D graphic with 3D animations.

In 3D graphic, the pre-animated animation is recorded in key-frames. To animate the animation, interpolation is required to calculate the intermediate vertex position or joint orientation to generate smooth transition between frames. Furthermore, if the lighting effect is enabled, the normal vector of each vertex has to be computed too. These operations are required on every frame. This is computationally expensive as the number of vertices of the 3D mesh is very high.

As smart mobile device is less powerful in term of its processing power, limited memory and constrained battery life, special techniques are required to alleviate the problem. In this paper, several approaches will be discussed to improve the performance.

2. The approaches

In this section, several approaches will be discussed including level of detail of 3D mesh, utilizing vertex buffer object, utilizing fixed point operation, usage of native code and Just-in-time compiler (JIT), and efficient coding style.

2.1 Level of Detail (LOD) of 3D mesh

The name LOD itself indicates the level of detail of a 3D mesh. This approach works by reducing the number of vertices of a 3D mesh to be processed. This approach is used when the model is distant from the viewport, because we are not able to see the model in detail since it is too small. By taking this advantage we can reduce the number of vertices of the 3D model without losing important data. In the meantime, operations like interpolation and normal vector calculation of the vertices in every frame can be further reduced. Hence, improve the performance. This approach is used by Cal3D a 3D Character Animation Library [1]. It provides real time LOD adjustment. It can be seen in Figure 1.

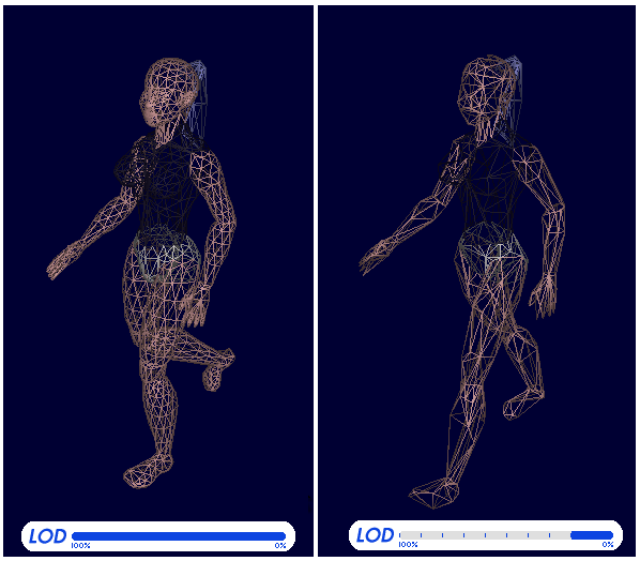

Figure 1 - **Cal3D sample program**. The 3D meshes with LOD 100% on the left and LOD 20% on the right.

[1] 본 연구는 지식경제부의 지원을 받는 동서대학교 유비쿼터스 지역혁신센터의 연구 결과로 수행되었습니다.

2.2 Vertex Buffer Object (VBO)

A vertex buffer object allows us to store certain data such as texture offset, vertices, normal vectors, and color pointer in high performance memory on graphic card [2]. By using VBO, we can reduce the overhead of memory transfer to graphic card. This is very useful for static 3D mesh which doesn't change as rendering data of the mesh can stay in graphic card buffer without any update. Moreover VBO is very useful for 3D objects that share the same data such as texture offset, element index data and vertices by binding the common buffer. Unfortunately, OpenGL ES 1.0 doesn't support Buffer Objects. However OpenGL ES 1.1 and onwards support Buffer Objects [3]. In addition, the newer Android device supports OpenGL ES 1.1 and later. Android 1.6 NDK Release 1 supports OpenGL ES 1.1 native library [4]. So, most of the Android devices today are able to utilize the VBO feature.

2.3 Fixed Point Arithmetic

Floating-point operation is about 2 times slower than integer operation on Android devices [5]. Fixed point number can be represented by an integer that has a fixed number of digits after the decimal point. In other words, the fixed-point number operation uses integer hardware operation and the decimal point is controlled by software implementation.

As the 3D model animation requires updating the vertex in every frame, the usage of fixed point arithmetic can improve the performance significantly.

An example of fixed-point operation to perform multiplication on 2 fixed point numbers with 16 bit of fractional part in C++ can be implemented in this way. R = ((long)num1 * (long)num2) >> 16; where R, num1 and num2 are integer [6].

2.4 Native Code and Just-In-Time compiler (JIT)

Interpreted language like Java requires an interpreter to execute the code and it is regarded to be very slow and bloated as compared to compiled language like C/C++. In Android, it supports native language, which is able to run compiled code, also known as native code to achieve high performance. But it is usually not advisable to be used, because it is machine dependent and hard to debug [7]. In the newer version of Android, 2.2 and onwards support Dalvik Just-In-Time compiler (JIT) [8].It is introduced to improve the performance of Android. It is able to perform 2x to 5x faster than without using JIT [8]. The performance of JIT is comparable with native code. As for native code, it supports since Android 1.5 [7].

As per experiment from the next section, the native code outperformed Dalvik JIT, however the difference is not significant. For Android device running version 2.2 and above, it is better to avoid using Native Code, unless it is really necessary.

2.5 Efficient coding style

Last but not least, coding style does affect the performance of execution too. One of the most important things that have to be avoided is the creation of unnecessary objects [5]. The memory allocation of object creation has a very high overhead, especially creating objects in a loop. When the object is no longer used, garbage collector will clean up the garbage by de-allocating the memory, which has very high overhead too. One way to avoid unnecessary object creation is by reusing the object.

Another thing to avoid is the internal getters/setters. To access an object field there are 2 ways. One way is by direct field access and another way is getters/setters. For example, to read a data from an object, we can either use obj.mCount or obj.getCount(). By using direct field access, it is 7 times faster than the trivial getter with JIT (Just-In-Time compilation). And without JIT, it is 3 times faster [5]. Of course there are other efficient coding styles to be taken into account too.

3. Experiments

In this section several experiments are carried out to evaluate the performance improvement as described in previous section. Throughout the experiment, Samsung Galaxy S is used and it is running Android 2.2. SGS comes with S5PC110 processor which is based on Cortex A8 Series and it comes with VFPv3 Floating Point Unit [9] [10] [11].

The conducted tests are PI value approximation by using numerical integration, Fibonacci Numbers computation by using Dynamic Programming and memory copying operation. Each of the tests is conducted in different environment.

3.1 PI value approximation (Floating Point)

The purpose of this experiment is to compare the performance of floating point operation among native code, Java code (with JIT) and Java code (without JIT).

Equation 1

$$f(x) = \int_0^1 \frac{4}{(1+x^2)}$$

Equation 2 – Integration with Trapezoidal rules

$$f(x) \approx \frac{3}{4n} + \sum_{i=1}^{n-2} \frac{4}{\left(1+\left(\frac{i}{n}\right)^2\right)}$$

The method used to approximate PI is by evaluating the Equation 1 by using numerical integration. The Equation 1 is transformed into numerical form by using Trapezoidal rule into Equation 2. It is used in the experiment to approximate the PI value. The n value is 10,000,000. The result is shown in Table 1

Environment	Average Time, seconds
Java (Without JIT)	3.393665267
Java (With JIT)	2.971758700
Native Code	2.599608233

Table 1 : Average time of 3 tests on PI value approximation

Based on Table 1, native code achieves the best performance. But the difference among each other is not very significant.

3.2 Fibonacci Numbers computation (Integer)

In previous test, it is used to test floating point operation. In this experiment it is used to test performance of the integer operation in native code, Java code (with JIT) and Java code (without JIT).

In this experiment, 10,000,000 of Fibonacci numbers are generated by using Dynamic Programming approach. Which uses this recurrence equation, F(n) = F(n-1) + F(n-2). The result is shown in Table 2.

Environment	Average Time, seconds
Java (Without JIT)	3.118516800
Java (With JIT)	0.975601687
Native Code	0.680855067

Table 2 : Average time of 3 tests of 10,000,000 Fibonacci Numbers computation

As shown in Table 2, again native code scores the shortest time execution. And the difference between Java without JIT and 2 other environments is very significant.

3.3 Memory copy result (native code, java code)

In this experiment, MD5 3D model by Id Software with 1656 vertices and 110 numbers of joints is used by rendering an animation. The difference between the 2 programs to be tested is 2 lines of code which is the memory copy operation in native code and in java code. The copy operation is applied to copy newly updated vertex position and normal for every frame into the vertex buffer to be rendered by OpenGL ES.

Figure 2 : Java code memory copy scores 32 FPS and Native code memory copy on the right scores 47 FPS.

To copy array into buffer in Java code:

```
vertexBuffer.put(vertexArray);
normalBuffer.put(normalArray);
```

To copy array into buffer in native code (C++):

```
memcpy( vertexArray, vertexBuffer , vertexLen <<  2);
memcpy( normalArray, normalBuffer , normalLen <<2);
```

Note that, the memory allocation length parameter in memcpy is applied left shift by 2 bits it is actually an optimization of multiplication of 4. We have to multiply the length by 4 because the length of float data type is 4 bytes. This optimization is retrieved from Baglogic Games's article [12].

Discussion

From the experiments, the performance of JIT is only slightly slower than native code, except the special case like memory copy. In this case we should consider using native code. In other occasion, we should avoid using native code. Because the compiled code is machine dependent and it is hard to debug. Besides that, as discussed in previous section, LOD can improve the performance significantly as the number of high-poly meshes increase.

References

[1] Cal3D. (n.d.). *3D Character Animation Library*. Retrieved March 24, 2011, from Cal3D: *http://home.gna.org/cal3d/*

[2] nVIDIA. (2003, October 16). White Paper Using Vertex Buffer Objects (VBOs).

[3] Khronos Group. (n.d.). *OpenGL ES 1.X for fixed function hardware.* Retrieved March 24, 2011, from http://www.khronos.org/opengles/1_X/

[4] Android Developer. (2009, September). *Download the Android NDK (Android NDK, Revision 2).* Retrieved March 24, 2011, from Android Developers: http://developer.android.com/sdk/ndk/index.html

[5] Android Developer. (n.d.). *Designing for performance*. Retrieved March 24, 2011, from Android Developer: *http://developer.android.com/guide/practices/design/performance.html*

[6] Lauha, J. (2006, September 13). The neglected art of Fixed Point Arithmetic.

[7] Android Developer. (n.d.). *What is NDK?* Retrieved March 25, 2011, from Android Developer: http://developer.android.com/sdk/ndk/overview.html

[8] Bray, T. (2010, May 25). *Dalvik JIT*. Retrieved March 25, 2011, from Android Developer Blogspot: http://android-developers.blogspot.com/2010/05/dalvik-jit.html

[9] ARM. (n.d.). *Cortex-A8 Processor*. Retrieved March 24, 2011, from ARM official website: http://www.arm.com/products/processors/cortex-a/cortex-a8.php

[10] Samsung. (n.d.). *Samsung S5PC110 ARM Cortex A8 based Mobile Application Processor*. Retrieved March 20, 2011, from Samsung: http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=834&partnum=S5PC110&xFmly_id=229

[11] Chan, J. (2010, Jun 1). *Samsung Galaxy S Scores in Benchmarks*. Retrieved from Cnet: http://asia.cnet.com/samsungs-galaxy-s-scores-in-benchmarks-62200389.htm

[12] Mario. (2010, September 11). *libgdx, md5 and direct buffer madness*. Retrieved March 23, 2011, from Badlogic Games: http://www.badlogicgames.com/wordpress/?p=904