Developing Mobile 3D Games Using MIDP 2.0 Game API and JSR 184 Mobile 3D Graphics (M3G) API

Yu Han
School of Computer Engineering
Nanyang Technological University

#45-6-861 Hall of Residence 9
24 Nanyang Avenue, NTU
Singapore 639811
Telephone number 65-93363651
yuha0002@ntu.edu.sg

Abstract
This paper discusses in details how to apply the JSR 184 M3G API and the MIDP 2.0 Game API in the development of mobile 3D immersive games based on two games developed by the author. The use of third party content creation software, the problems of importing .m3g format resource files and the solutions proposed by the author will also be addressed.

As the processing power of mobile phones is still not up to the requirement of 3D immersive games, it is essential that the games are properly optimized to offer a satisfying gaming experience with reasonably fast frame rate. The optimization techniques which were employed to solve specific problems arising during the development of the sample games will be illustrated in detail.

Keywords
MIDP 2.0 Game API, Mobile Game Programming, Mobile 3D gaming, JSR 184 M3G API.

1 Introduction

The Java™ Mobile 3D Graphics (M3G) API has the power to turn a good game into an outstanding mobile gaming experience and profoundly change the way of presenting information on mobile devices. As the standard implementation of JSR 184 matures and the Java 3D enabled devices become widely available, mobile 3D games are starting become common commercialized applications instead of just being research topics in the laboratories.

Although proprietary 3D graphics software does exist in the current market, the trend among the mobile industry is to implement and support a common 3D graphics API that will facilitate portability of the applications and ease the process of development. For Java, the JSR 184 M3G API is considered the most suitable candidate. This paper discusses in details how this API combined with MIDP 2.0 Game API can be used not only to develop 3D immersive mobile games but also to improve the user interface of the game, while in some cases increasing the memory footprint of the application.

Moreover, the limitations of memory and processing power still exist for mobile devices and must be properly addressed in order for computationally expensive applications such as 3D games to run. This paper will also discuss several optimization techniques which can be employed to improve the performance of the games.

2 Exposition

2.1 Building Lightweight GUIs with MIDP 2.0 Game API

Previously, the GUI of the mobile games for the control and game settings were written simply based on J2ME built-in container classes such as Form or List. However, this has become increasingly inadequate given the high demand from consumers for more visually appealing GUIs. Using proprietary software to create GUIs requires the developer to learn to program in their specific ways and may sometimes involve licensing problems.

One way to solve this problem is to use the J2ME MIDP 2.0 Game API to write GUIs tailored to the specific game. The Game API offers excellent support in manipulating graphics and animations. If implemented as a Runnable thread, it enables the use of polling technique to simulate key events.

Even though the APIs allow porting applications to different devices, even across vendors, the size of the display varies from device to device. Therefore the UI elements should not be positioned using absolute coordinates, but in relative coordinates based on the screen size and the size of the content. For example,

X =((screen width) – (string width))/2

Figure 1: User menu implemented with Game API

The required information can be extracted from Font and GameCanvas classes. With each up or down key press, the indicator is moved by a predefined amount along the vertical axis to correspond to different choices. Internally, an index is updated to log the information of which option is currently highlighted. With the press of the select key, the current thread is terminated and the respective function handling the event is called.

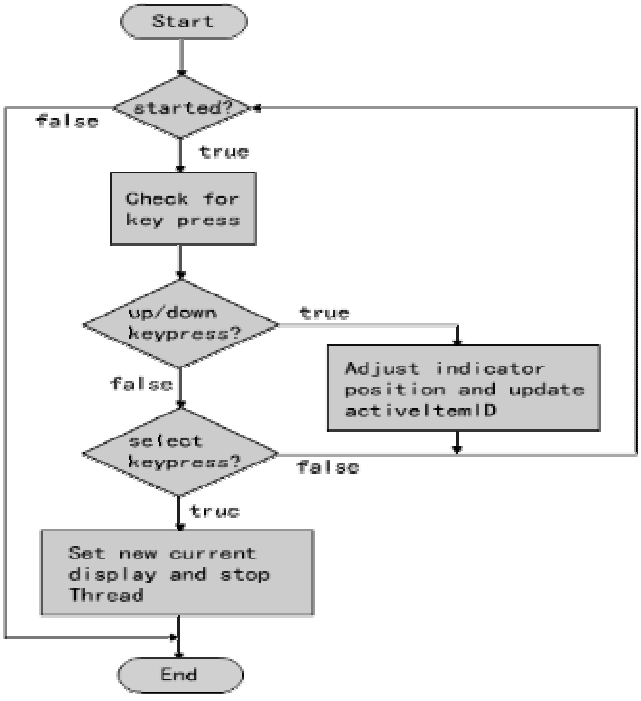

Figure 2: Flowchart of simulating key events using polling

When large numbers of resource files are being loaded, the game may appear not responsive for a long time. Users may suspect their mobile devices are crashed by the program and undesirable actions such as shutting down the device may be taken. To add in more interactivity, a progress bar can be implemented using J2ME MIDP 2.0 Game API to show the progress of loading the game. In the loading function, set different points to update a global integer value which represents the percentage of the game loaded. Then in the second thread, keep reading this value and redraw the progress bar. In this case, a separate thread is mandatory for the loading and drawing of the progress bar to be executed concurrently.

Figure 3: Progress bar for loading the game

2.2 Loading M3G Files

M3G files are a specially defined group of files which caters specifically to the JSR 184 M3G API. Objects or even entire scenes created with content creation tools such as 3D Studio Max or Maya can be exported as .m3g files and used by mobile games. This enables the games to have fairly complex objects or scenes which, in turn, enhance the gaming experience of the players.

However, loading a .m3g file may not be so straight forward as it seems to be. There is a discrepancy of implementation between the scene graph hierarchy of the .m3g file format and the JSR 184 M3G API: in the API, the root of the scene graph is an object of class World and all other 3D objects are children of this World object. But in the .m3g exporter, the root of the scene is an instance of the class AnimationController. Because of this difference, the developer must do a manipulation in the program to extract the World object out.

As each .m3g may have different indices for different objects, there is no way to be certain which index corresponds to the World object. In order to extract the desired object, the World object must be found first. Therefore, we use the following piece of code to locate the World object:

```
Object3D[] roots = Loader.load( filename );
World world;
for( int i = 0; i < roots.length; i++ ){
        if( roots[i] instanceof World ){
                world = (World)roots[i];
                break;
        }
}
```

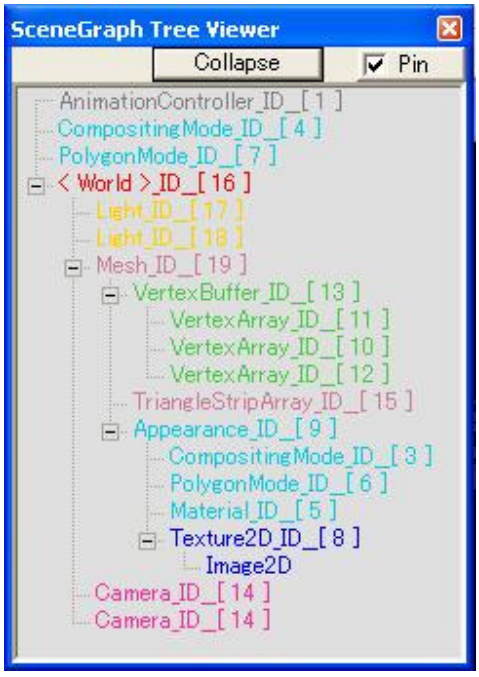

Figure 4: Scene graph of a .m3g file

Loop through the objects and find the only one which is of World type. As only one World object is allowed in each .m3g file, we can be certain that the World object is found when we first encounter it and exit the loop.

Another problem associated with .m3g file arises from the fact that each 3D object can only be child of one World. When multiple instances of the same object need to be present in the same World, the resource file needs to be reloaded and the object re-extracted which takes considerable amount of time. This problem will be addressed in detail in the later optimization section.

2.3 Perspective Distortion

When using graphics primitives such as triangleStrip to construct planar surface, perspective distortion may occur during texture mapping

Each planar face of the wall in Figure 5 consists of 2 triangles. When drawing the pixels of the texture, the positions of the ones near the upper edge are calculated with respect to the upper edge and the positions of the ones near to the lower edge are calculated with respect to the lower edge. Since these two edges appear to be not parallel from this perspective view, distortion occurs. The line separating the two triangles can be seen clearly in the figure below.

Figure 5: Perspective distortion

To alleviate this problem, more triangles can be used to represent one planar surface. As more and more triangles are added in, the effect of perspective distortion appears to be less and less noticeable. However, this method not only consumes more memory to store the extra triangles, it is also tedious for programmers to implement. Instead, the M3G API provides a function called setPerspectiveCorrectionEnable(boolean enable) in the PolygonMode class which eliminates the perspective distortion. However, the perspective correction flag is only a hint, so some implementations may not respect it.

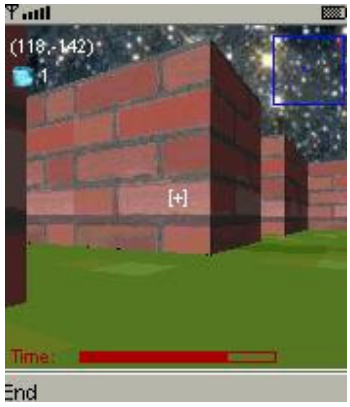

Figure 6: Perspective correction

2.4 Collision Detection

The JSR 184 M3G API provides a pick() function under the Group class for collision detection by ray intersection. An imaginary ray is cast from the center of the camera to infinity and the first mesh surface intersecting it at a predefined distance is considered causing a collision. This method is considered sufficient when navigating in a complex scene setting and there are too many objects to test for potential collisions by implementing bounding box or bounding sphere. However, the ray cast is only along the same direction as the camera. Therefore, when trying to detect collision during backward motions, the camera has to be temporarily reversed, test for collision, then reversed back.

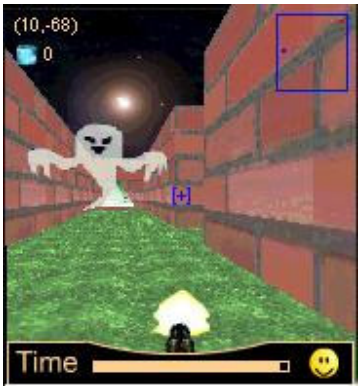

Figure 7: Collision detection by ray casting for firing of ammunitions

However, the pick() function may be implemented differently by some mobile phone manufacturers (I have tried using the pick() function with SonyEricsson phones and the collision detection did not work). Moreover, it is not suitable to use when the ammunitions are to be seen flying away (e.g. firing a missile). Therefore, in this case, bounding box or bounding sphere should be used.

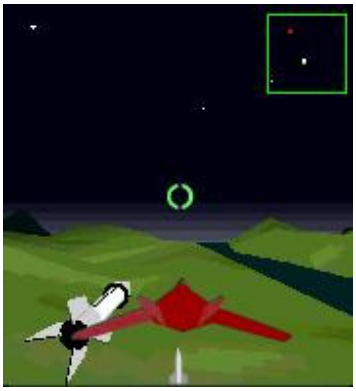

Figure 8: Bounding box collision detection

For mobile devices with limited processing power, bounding box is a better choice since it requires less calculation to detect a collision:

$$(X1-X2)^2 + (Z1-Z2)^2 + (Y1-Y2)^2 < distance^2$$

In the case when all the objects appear on roughly the same level, a bounding cylinder can further simplify the calculation and thereby improve the performance of the game:

$$(X1-X2)^2 + (Z1-Z2)^2 < distance^2$$

In this case, a cylinder along y-axis with infinite height is used for collision detection.

2.5 Optimization

Due to the limitation of memory and processing power of mobile devices, the games should always be optimized in order to make efficient use of the resource and achieve better performance. This is especially the case for 3D games which generally require more expensive computations than 2D games when rendering the screen.

First of all, not all the components have to be 3D. The graphics shown on the screen are basically a way of presenting the information and logic of the game, so whenever possible, developers should resort to 2D graphics which generally render faster. For example, in Figure 8, there is a jet fighter image attached to the camera. It banks to the left or right when the player presses the left or right key respectively as shown in Figure 10. To use a 3D model to accomplish this requires a lot of computations when rotating the plane and rotating the camera about the plane. Instead, we use a 2D image with the various positions of the plane pre-captured and place the corresponding image on the screen when necessary.

Figure 9: 2D images of the plane

Figure 10: Plane banking to the right

Thread objects should not be used unless absolutely necessary. By creating too many processes running concurrently, the overhead of context switching can make a 3D mobile game run unbearably slowly. Therefore, sometimes it is a good idea to manage all the moving objects in several Vector objects and move all of them in the same Thread running the game loop.

As mentioned in earlier section, the M3G API disallows one Object3D instance to be child of multiple Worlds. By default, there is a World object in every .m3g resource file and even if one only wants to export a 3D object, it is automatically attached to this default World.

The most intuitive way to solve this problem is to load the resource file each time one new instance of the object is needed, extract the object, remove it from the old World and attach it to the World in which the game is set. However, loading external resource files takes a long time and this can cause a significant delay during the game play, resulting in disruptive and unpleasant gaming experiences.

To eliminate the delay, a concept similar to double buffering has been employed in one of my games. For each 3D object for which multiple instances might be needed, one copy of it is stored in the memory throughout the playing time of the game. Each time a new instance is required, the original copy is duplicated and the new copy is attached to the World. To do the duplication, the duplicate() function of Object3D class is used. It not only creates an exact replica of the original object but also set the parent to null if the object is a Node. Then this new copy can be a child of the game World without causing any error.

Although this method consumes more memory by saving one copy of each object regardless of whether it is being used, it eliminates the need to reload external resource files and thereby remove the long pauses during reloading. Overall, the performance of the game is improved.

Alternatively, the components of the object like IndexBuffers and VertexBuffers of a mesh can be replicated and the mesh reconstructed later. However, to locate these pieces of information in the .m3g exporter file requires the knowledge of the indices of them. And more often then not, the sheer amount of IndexBuffers and VertexBuffers involved in complex models would make implementing this method a daunting task.

3. Observation

Although there have been significant improvements in the processing power, memory capacity and floating point support in the recently launched mobile phone models, the 3D graphics performance does not offer a very satisfying game play experience especially in the case of a first-person-view game.

However, with the proper application of the optimization techniques mentioned above, an approximately 30% increase in frame rate has been achieved.

Phone Model	**Frame rate before optimization /FPS**	**Frame rate after optimization /FPS**
SonyEricsson K300	6.5	11
SonyEricsson K500	7	10.7
SonyEricsson K700	8.3	12.5
SonyEricsson F500i	7.6	11.5

Table 1: Performance of the same game before and after optimization on various phone models

The values shown above are average values for one complete game session on actual devices. With further refinement in optimization techniques and continued improvement in mobile phone hardware, the 3D immersive games developed with JSR 184 M3G API will surely achieve a satisfying frame rate of 20~25 FPS and offer a new outlook to the mobile gaming industry.

References:

SonyEricsson general article, *Java 3D - a new opportunity in mobile gaming*, March 9, 2005

Qusay H. Mahmoud, *Getting Started With the Mobile 3D Graphics API for J2ME*, September 21, 2004

Sony Ericsson Developers Network, *Mobile 3D Graphics and Java Applications Development for Sony Ericsson Phones*, November 2004.

Sun Microsystem Inc, *JSR 184 Mobile 3D Graphics (M3G) API.* http://java.sun.com/j2me/docs/index.html

Tomi Aarnio, Kari Pulli, Nokia Research Centre,

Advanced Game Development with the Mobile 3D Graphics API

Alexei Sourin, *Computer Graphics – From a Small Formula to Virtual Worlds*, published in 2005 by Prentice Hall, ISDN 981-244-743-1