A study on Optimizing Mobile 3D Game Engine using JSR-184

JongKeun Cho
Dept. of Computer Science, Soongsil University
Dongjak-ku, Seoul, Korea

ShinJun Lee
Dept. of Computer Science, Yonsei University
Shinchon-dong, Seoul, Korea

MoonWon Choo
Division of Multimedia, Sungkyul University,
Anyang-city, Kyunggi-Do, Korea

ABSTRACT

This study focuses on modeling mobile 3D game engine and suggesting modified skinned-mesh schema based on JSR-184 in order to improve the performance in terms of memory consumption and time complexity. Most of the 3D games have used OpenGL-ES low-level APIs, which may limit portability and fast developing time. Hence, the 3D mobile game engine providing high-level APIs which works on GSM (Global System for Mobile Communication) phones on J2ME, is proposed here in order to optimize the performance for Java environment abiding JSR-184 standard. To prove performance enhancement, skinned-mesh schema on JSR-184 engine is modified and tested. The experimental results are shown.

Keywords: *mobile application, mobile game, 3D game engine, mesh skinning*

1. INTRODUCTION

The JSR(Java Specification Request)-184 has recently become very popular in mobile 3D industry[1,2]. It is a standard 3D graphic API optimized for the Java environment. The issue is that if low-level OpenGL-ES is used in Java 2 Platform Micro Edition (J2ME) environment for the 3D graphics, the code will be lengthy and large in volume, resulting in unfitness for MIDP(Mobile Information Device Profile)[3][4][5]. MIDP are combined with the Connected Limited Device Configuration (CLDC), which defines the base set of application programming interfaces and a virtual machine for resource-constrained mobile devices, in order to provide a standard Java runtime environment for today's most popular mobile information devices and define a platform for dynamically and securely deploying optimized, graphical, networked applications.

J2ME technology is delivered in API bundles called configurations, profiles, and optional packages. A configuration provides the most basic set of libraries and virtual-machine features that must be present in each implementation of a J2ME environment. When coupled with one or more profiles, the CLDC gives developers a solid Java platform for creating applications for consumer and embedded devices. On the other hand, a profile is a set of standard APIs that support a narrower category of devices within the framework of a chosen configuration. A specific profile is combined with a configuration like CLDC to provide a complete Java application environment for the target device class. MIDP, which is a profile supported by CLDC, provides a rich run-time environment. An optional package is a set of technology-specific APIs that extend the functionality of a Java application environment. The CLDC supports a number of optional packages that allow product designers to balance the functionality needs of a design against its resource constraints. The CLDC-based optional packages include several API collections related to wireless messaging and mobile media, supporting J2ME applications targeted at cell phones and other devices that can send and receive wireless messages.

The goal of the CLDC specification is to standardize a highly portable, minimum-footprint Java application development platform for resource-constrained, network-connected devices[4]. The CLDC and MIDP provide the core application functionality required by mobile applications, in the form of a standardized Java runtime environment and a rich set of Java APIs. Developers using MIDP can write applications

Corresponding author. E-mail : mchoo@sungkyul.edu
Manuscript received Nov.20, 2007 ; accepted Dec.7, 2007

and deploy them quickly to a wide variety of mobile information devices. MIDP has been widely adopted as the platform of choice for mobile applications. It is deployed globally on millions of phones and PDAs, and is supported by leading integrated development environments (IDEs). Companies around the world have already taken advantage of del is proposed and is compared with that of Nokia's JSR-184 engine. The rest of paper consists of as follows. Section 2 explains the system design and implementation. Section 3 and section 4 discuss the skinned mesh algorithm. Section 5 and section 6 is about experimentation and short conclusion remarks.

2. SYSTEM DESIGN

Nokia's JSR-184 mobile 3D engine is organized as shown in the Fig. 1. [6]

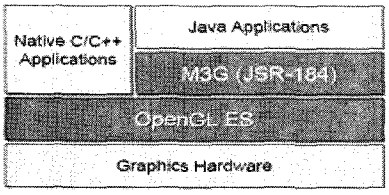


Fig. 1. Diagram of Nokia's JSR-184 3D engine

In this engine, Java Virtual Machine is integrated into OpenGL-ES and Java GUI into M3G(Mobile 3D Graphics API, JSR-184). The M3G is a J2ME Optional Package that allows three-dimensional (3D) graphics to be rendered at interactive frame rates on mobile, resource constrained devices. It also includes facilities for 3D scene management and animation, as well as a file format for efficient over-the-air deployment of 3D content. This architecture is not quite different from the system implemented in this paper. Fig. 2 shows the conceptual diagram of the proposed architecture.

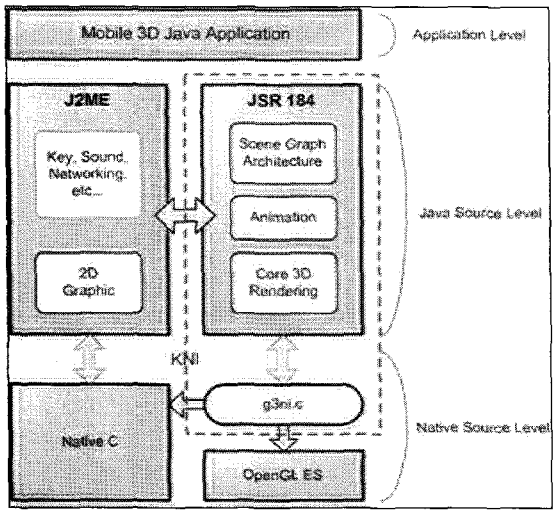


Fig. 2. The proposed JSR-184 3D Engine

MIDP to write a broad range of consumer and enterprise mobile applications[6].

In this paper, mobile 3D engine based on J2ME environment using JSR-184 that accommodates a various kinds of 3D multimedia contents is proposed. Also, faster and more efficient way to process skinned-mesh mo

For Java source level, 30 classes and 250 methods of JSR-184 are implemented and written in Java[7]. The specifications of these classes are categorized into several functional levels as follows.

3D Graphics

The classes in this level are essentially related to 3D graphics such as Modeling, Transform, Camera, Texture mapping, Material, and Lightings[8]. These are made up of Java wrappers which allow full use of existing OpenGL-ES engine[9][10]. The followings illustrate some of them.

Appearance: define mesh and sprite 3D rendering property
Background: decide whether to clear viewport or not and how to clear
Camera: the position of the viewer and how 3D will be projected in 2D
CompositingMode: composite property of each pixel
Fog: fog effect
Graphics3D: definition of graphic context that will be drawn on rendering target
Image2D: definition of 2D image that will be texture, background or sprite image.
IndexBuffer: how geometrical object and vertex will be connected
Light: define the type of light
Material: define the material for the light calculation
Mesh: definition of polygon surface of the 3D object
MorphingMesh: define morphing effect of polygon mesh
PolygonMode: definition of polygon property
SkinnnedMesh: definition of polygon mesh expession for bone animation
Sprite3D: billboard effect using 2D images
Texture2D: definition of texture mapping for the mesh
Transform: definition of 4x4 matrix for transformation
Transformable: class for transformation of node and texture
TriangleStripArray: array declaration for storing triangle strip
VertexArray: vector array for the vertices position, normal, color and texture coordinate

Animation

The classes in this level are written in C and built into Java wrappers. The OpenGL-ES doesn't have animation features, but it includes Keyframe animation, Bone animation, and Motion morphing.

AnimationController: define position, speed, and rate of the Animation sequence
AnimationTrack: animation controller that have values that can be animation and definition of KeyframeSequence
KeyframeSequence: animation data for each sequence

Scene Graph

Written in Java, it includes Hierarchy, Object traverse, Picking and Alignment. Some of the classes in this level are as follows:

Group: require information when storing nodes structurally
Node: common property related to SceneGraph nodes
Object3D: common properties related to 3D objects
RayIntersection: definition of features for the picking
World: top level container for SceneGraph

Additionally, the class **Loader** is specified to define specific data format for describing the SceneGraph structure when downloading SceneGraph nodes and node components.

As in J2ME, JSR-184's core is mostly written in C. Thus, when passing the data from Java to C, a proper structure for data passing is required, which is characterized by KNI in this case. KNI(K-Native Interface) is used for mapping JSR to OpenGL-ES functions, which is based on customized OpenGL-ES engine function[11]. KNI is a native function interface that provides high performance and low memory overhead without the pitfalls of low-level interfaces in order to facilitate the integration of native functionality across a wide variety of CLDC target devices. The whole JSR-184 engine was designed by following steps shown in Fig. 3. For more details, you may refer to the related paper[7].

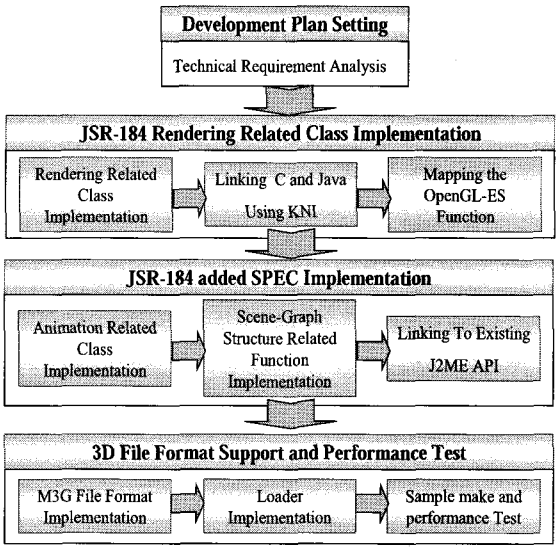


Fig. 3. 3D Engine Implementation using JSR-184

3. OVERVIEW OF SKINNED MESH ALGORITM

The use of skinning when rendering meshes is nowadays widely used. The technique works as follows: for a model, a skeleton is defined by using a hierarchy of bones and the joints of the skeleton are attached to the vertices of the mesh (the "skin"). To animate the mesh only the skeleton have to be animated because the vertices of the mesh are deformed based on the joints they are attached to. The vertices of the mesh can be attached to a number of joints. A vertex is needed to attach to at least one joint and for current game animation of human-like characters, animators need at most four joint influences per vertex. If exactly one joint per vertex is used, this is called simple skinning. If multiple joints is needed to influence a vertex , which is called smooth skinning, which amount every joint influences the vertex with a total influence(or weight) that sums to one[12] should be specified. The final transformed vertex position is a weighted average of the initial position transformed by each of the attached joints. For example, the vertices in a character's knee could be partially weighted to both the hip joint (controlling the upper thigh) and knee joint (controlling the calf). Many vertices will only need to attach to one or two joints and rarely is it necessary to attach a vertex to more than four. In this paper, smooth skinning method is considered to improve the rendering performance.

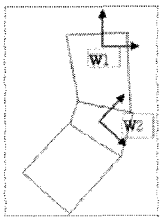


Fig. 4. Smooth skinning on a cylinder[12]

Using smooth skin, a vertex can be attached to more than one joint with and receive a weighted average of the transformations. Let us say that a particular vertex is attached to N different joints. Each attachment is assigned a weight W_i which represents how much weight the joint will have on it. To ensure that no undesired scaling will occur, we enforce the constraint that all of the weights for a vertex must add up to exactly 1. To compute the world space position $\mathbf{v}'$ of the vertex, it is transformed by each joint that it is attached to. Then it needs to be computed a weighted sum of the results as follows:

$$v' = \sum w_i v \cdot B_{[i]}^{-1} \cdot W_{[i]},$$

where v is the untransformed vertex in skin local space, in which the skin mesh was originally modeled. The matrix $W_{[i]}$ is the world matrix of the joint for attachment i. The indexing notation $[i]$ is used to indicate that we don't want the matrix of the i^{th} joint in the skeleton (which would be written W_i), but instead we want the world matrix of attachment i's joint.

The matrix $B_{[i]}$ is called the binding matrix for joint $[i]$. This matrix is a transformation from joint local space to skin local space, and so the inverse of this matrix, $B^{-1}{}_{[i]}$, represents the opposite transformation from skin local space to joint local space. As the number of joints is likely to be small compared to

the total number of vertices that need to be skinned, it is more efficient to define and compute $M_{[i]}$ for each joint before looping through all of the vertices as follows:

$$M_{[i]} = B_{[i]}^{-1} \cdot W_{[i]} .$$

The skinning equation that must be computed for each vertex then simplifies to

$$v' = \sum w_i v \cdot M_{[i]} .$$

In addition to transforming the vertex positions into world space, the skinning algorithm must also transform normals that the renderer will need to perform lighting calculations. We will assume that in the initial untransformed mesh, a normal n is specified for every vertex v. This normal is usually specified offline through an interactive modeling tool. To compute the world space normals, we use exactly the same skinning weights as the vertices, and so the normals are treated in very much the same way:

$$n^* = \sum w_i n \cdot M_{[i]} .$$

Even if the untransformed normal n is unit length, the weighted averaging in the equation could cause the length of the intermediate blended normal n^* to vary. To accommodate for this change in length, the final blended normal n' will most likely need to be normalized after the skinning is applied, in order for lighting calculations to work properly:

$$n' = n^* / |n^*| .$$

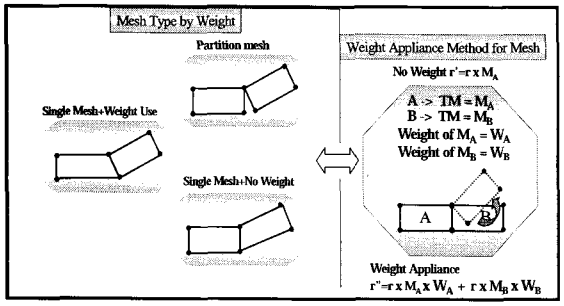


Fig. 5. Comparison of skinned animation values

For illustration, Fig. 4 shows the comparison of skinned animation values between different mesh types. When a skin A and a skin B change in animation, partitioned mesh type can't do much about it. However, in case of single mesh plus weight, ***Ma***, matrix of animation ***A*** and ***Mb***, matrix of animation B will combine weight ***Wa*** and ***Wb*** and apply them to compute the final value. Since the connecting vertex has two coordinates, each result will be relevantly mixed and can be expressed as follows:

$$\begin{bmatrix} Rx \\ Ry \\ Rz \end{bmatrix} = WM \begin{bmatrix} Ax \\ Ay \\ Az \end{bmatrix} + (1-W)M \begin{bmatrix} Bx \\ By \\ Bz \end{bmatrix} .$$

If W=0.5 is applied for the vertex belongs to each coordinates, the averaged result will be obtained. If the other part of bone will have the weight of 0 or 1, the vertex belonging to only one coordinate is influenced. However, the loop will visit each vertex and calculate 4 blend weight. The skin matrix will be calculated from matrix palette based on weighted bone and index using the following formula:

$$v_{world} = v_{local} * M_{[0]} * W_0 + v_{local} * M_{[1]} * W_1 + v_{local} * M_{[2]} * W_2 + v_{local} * M_{[3]} * W_3$$

The details on skinned mesh can be found at many references [12][13].

4. THE PROPOSED SKINNED-MESH ALGORITHM

JSR-184 decide specific vertex's position and weight from the **addTransform(Node bone, int weight, int firstVertex, int numVertice)** function[9]. It associates a weighted transformation, or bone, with a range of vertices. An integer weight is supplied as a parameter for each added transformation. Prior to solving the transformation equation, the weights are automatically normalized on a per-vertex basis such that the individual weights are between [0,1] and their sum is 1.0. This is done by dividing each weight pertaining to a vertex by the sum of all weights pertaining to that vertex. For example, if two bones with any equal weights overlap on a vertex, each bone will get a final weight of 0.5.

Automatic normalization of weights is convenient because it significantly reduces the number of times that this method must be called (and hence the amount of data that must be stored and transmitted) in cases where more than one bone is typically associated with each vertex. The same Node may appear multiple times among the bones. This is to allow multiple disjoint sets of vertices to be attached to the same bone.

The number of bones that can be associated with a single vertex is unlimited, except for the amount of available memory. However, there is an implementation defined limit (N) to the number of bones that can actually have an effect on any single vertex. If more than N bones are active on a vertex, the implementation is required to select the N bones with highest weights. In case of a tie (multiple bones with equal weights competing for the last slot), the selection method is undefined but must be deterministic. Parameters have the following meanings:

Bone: a node in the skeleton group to transform the vertices with

weight: weight of bone; any positive integer is accepted

firstVertex: index of the first vertex to be affected by bone

numVertices: number of consecutive vertices to attach to the bone node

4.1 Nokia's Skinned-Mesh Applied Algorithm

Nokia uses inner class utilizing WT(Weighted Transform) to implement Skinned-Mesh. Table. 1 shows the variables used in

Nokia's Skinned-Mesh Applied Algorithm.

Table 1. Variables used in Nokia's study

variable	description
WT	Weighted Transform with inner class for skinned mesh implementation
n	The number of vertices
i	Index for vertices
v_i	Vertex from v_1 to v_n
RT	Relevant Weighted Transform
L	Weighted Transform List from WT_1 to WT_n
PT	Position Transform
NT	Normal Transform
VT	Vertex Transform

When **addTransform()** occurs, new ***WT*** object is created and the bone will have first index and last index(v_1, v_2, ... , v_n) influenced by transform and weight in rest state, which will be added in the list, ***L***. It can be expressed as follows:

$$\text{if } v_i \in \boldsymbol{L}, \text{ find } \boldsymbol{WT} \text{ (which includes } v_i)$$

When rendering, the Weighted Transform List, ***L*** for each vertex is checked. If the current vertex is included, then the ***WT*** is searched.

$$\sum \boldsymbol{WT_i / n}, (i = 1 \ldots \boldsymbol{n})$$

RT(Relevant Weighted Transform) weight will be summed and divide the weight by the sum, which can be improved in terms of performance.

$$\sum (\boldsymbol{PT}+\boldsymbol{NT}) \cdot \boldsymbol{WT_i} \cdot \boldsymbol{v_i}, (i = 1 \ldots \boldsymbol{n})$$

Then Position Vertex value and Normal vertex value are calculated and accumulated using surplus number of ***PT***(Position Transform), ***NT***(Normal Transform) and inner class in rendering time. In other word, ***PT*** and ***NT*** have to be accounted for current state of ***VT***(Vertex Transform), whose calculation should be done in each rendering time.

Nokia's algorithm, in some cases, may be inefficient. As **addTransform()** function call increases, memory usages will increase proportionally, resulting in more rendering time.

4.2 The proposed algorithm

The Nokia's algorithm can be modified to reduce space and time complexities. Table 2 shows the variables used in the proposed skinned-mesh algorithm.

Table 2. Variables used in the proposed study

variable	description
$\boldsymbol{WL_{ij}}$	Weight List(i: vertex index, j: bone index)
n	The number of vertices
b	The number of bones
i	The index of vertex
j	The index of bone

The $\boldsymbol{WL_{ij}}$ is the new variable for skinned-mesh rendering. 2D array (WeightList[Vertex Number][Bone Number]) will store the weight and how each vertex will be influenced from which bone. From this variable, the weighted sum of all bones for each vertex is calculated as follows;

$$\boldsymbol{WS_i} = \sum \boldsymbol{WL_{ij}}, (\boldsymbol{j = 1 \ldots b})$$

Arrays will be initialized as 0 and when the function call **addTransform()** occurs, the weight each vertex receives will be accumulated. For the vertex where update occurred, WeightSum will be updated so that sum of weight will be calculated when addTransform() is called. In this case at the rendering time, only ***PT*** and ***NT*** for number of bones are calculated. Since sum of weight for each vertex is already calculated, this process can be omitted in rendering time. If the vertex is not influenced by the bone (the weight for a bone is 0), the calculation can be totally omitted. The core part of proposed algorithm is as follows:

```
/* Stores Deform Information */
int weightList[Number of vertex][Number of bone]
int weight Sum[Number of vertex]
Transform toBone[Number of Bone]
Transform positionTransform[Number of Bone]
Transform normalTransform[Number of Bone]
Vector boneList

  Accumulate weight in weightList at specified position.
  Recalculate accumulated weightSum
  for(int i=0; i<Number of Bone; i++)
  {
        Calculate positionTransform
        Calculate normalTransform
  }
  for(int i=0; i<Number of Vertex; i++)
  {
    for(int j=0; j<Number of Bone; j++)
    {
      Calculate vertex position and normal deform
                using weightSum and transform.
    }
  }
```

Basically, skinned-mesh object processing should be calculated real-time as shown in Fig.3 for the accurate movement and natural link between the meshes. In this algorithm, the weights of vertices associated with bones are not computed at every rendering time, rather once at loading time. Hence, it can save memory usage and longer rendering time due to skipping the normalization processes which should be done in Nokia's solution.

5. EXPERIMENTS

Sample file made by 3D MAX 7.0 is exported using M3G exporter. Using this file, the proposed JSR-184 engine and Nokia solution are tested and compared in terms of time

complexity. Fig. 6 shows the workspace for creating sample polygons using 3D MAX 7.0 for the engine performance test. The created file is exported as JSR-184 standard, M3G file. The actual file used in this experiment contains 3D objects using skinned-mesh, texturization and camera applications.

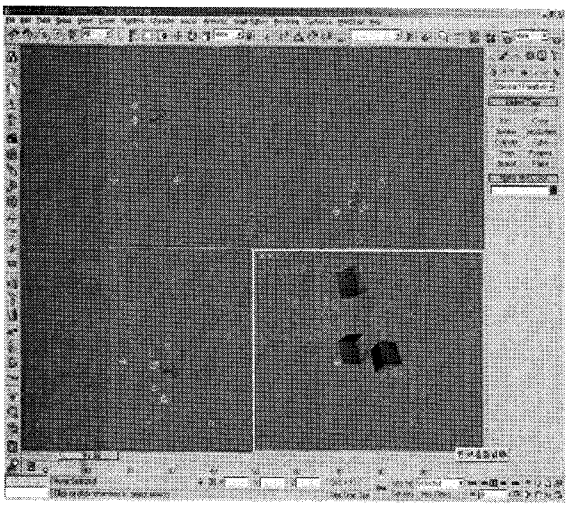


Fig. 6. Creating 3D Object in 3D MAX

Table 3 shows 3D sample objects in this test. Fig. 7 shows the three screen shots of simple skateboarding boy running JSR engine using M3G file format. Fig.8 shows the result of comparison between Nokia and proposed JSR-184 engine. It shows that the resultant performance is specially better at 2-3 fps(Frames Per Second). When games run on small LCD size of the mobile phones with 240x176 or 320x240, the performance issue is critical. The algorithm proposed here will be a definite benefit.

Table 3. 3D sample objects used in experiment.

JSR-184 Type	Objects	File format
Number of Polygon1	100 Polygon/Sec	Arm.M3G
Number of Polygon2	350 Polygon/Sec	Boy.M3G
Number of Polygon3	500 Polygon/Sec	dragonfly.M3G
Number of Polygon4	1200 Polygon/Sec	Monster.M3G

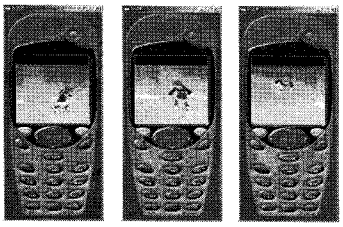


Fig. 7. Screen shots after running with JSR-184 engine

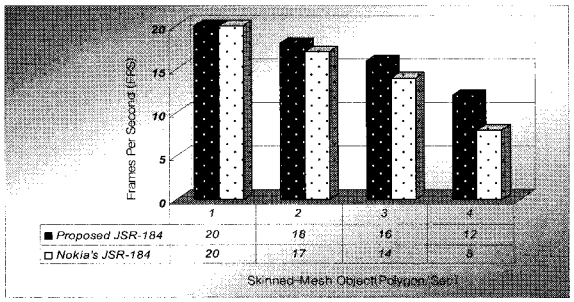


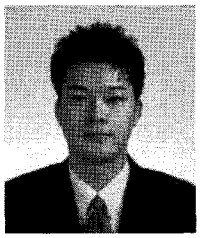
Fig. 8. Nokia and Proposed JSR-184 Engine Performance

. CONCLUSION

This paper has discussed the improvement of mobile 3D graphics engine based on JSR-184 for the GSM phone market which takes up 70% of global phone market. The experiment result shows that the proposed algorithm using Optimized JSR gives better performance compared with Nokia's engine.

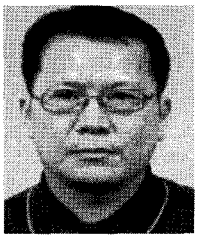
. REFERENCES

[1] JSR-184 Expert Group, **Mobile 3D Graphics API for J2ME**, version 1.0, Nov. 19, 2003.
[2] Y.J. Kim, **3D Game Programming**, Hanvit Media, 2004.
[3] J.W. Muchow, **Core J2ME Technology MIDP**, Prentice Hall, 2001.
[4] http://java.sun.com/.
[5] Khronos Group, http://www.khronos.org.
[6] Nokia Forum, http://www.forum.nokia.com/java /jsr184.
[7] J. K. Cho, Y. H. Park, and J. M. Kim, "Design and Implementation of Mobile 3D Engine using JSR-184 on J2ME," **Korea Information Science Society**, Vol.32(2), pp. 673-675, Oct. 2005.
[8] Enrico. Gobbetti and Fabio. Marton, "Far Voxels:A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Modle on Commodity Graphics Platforms," **ACM Trans on Graphics**, Vol.24(3), pp.878-885, July 2005.
[9] J.K.Cho, **Mobile 3D Programming using G3SDK**, Gomid 2005.
[10] Gomid Corp, http://www.gomid.com.
[11] J. K. Cho and J. M. Kim, "Design and Implementation of Mobile 3D Bluetooth Engine based on OpenGL-ES," **Journal of Korea Game Society**, Vol.6(1),pp. 23-28, March 2006.
[12] Skinned Mesh Export: Optimization http://www.gamasutra.com/features/.
[13] James. Dong. L and Twigg. Christopher. D, "Skinning Mesh Animation," **ACM Trans**, Vol.24(3), pp. 399-407, July 2005.



JongKeun Cho received the M.S. degree and the Ph.D. degree in computer science from Soongsil University, Seoul, Korea, in 2001 and 2004. His main research interests include mobile 2d/3D system, 3D computer graphics, and multimedia application.

ShinJun Lee received the MS degree in Computer Science from Yonsei University, Seoul, Korea, in 1999. He is currently a PhD student in the Department of Computer Science at Yonsei University. His research interests include 3D Graphics, Real-time Rendering, 3D Navigation, 3D Data Compression.



MoonWon Choo received the Ph.D. degrees in computer science from Steven Institute of Technology, Hoboken, NJ, in 1996. He is currently Associate Professor in the Division of Multimedia at Sungkyul University since 1997. His main research interests include adaptive vision system and multimedia application.