M3G Overview

Tomi Aarnio
Nokia Research Center
Objectives

• Get an idea of the API structure and feature set
• Learn practical tricks not found in the spec
Prerequisites

- Fundamentals of 3D graphics
- Some knowledge of OpenGL ES
- Some knowledge of scene graphs
Mobile 3D Graphics APIs

Native C/C++ Applications

Java Applications

M3G (JSR-184)

OpenGL ES

Graphics Hardware
Why Should You Use Java?

• It has the largest and fastest-growing installed base
  – 580M Java phones sold by Feb 2005 (source: Sun Microsystems)
  – Nokia alone shipped 125M Java-enabled phones in 2004
  – Less than 12M also supported native Symbian applications

• It increases productivity compared to C/C++
  – Memory protection, type safety ➔ fewer bugs
  – Fewer bugs, object orientation ➔ better productivity
Java Will Remain Slower

Benchmarked on an ARM926EJ-S processor with hand-optimized Java and assembly code.
So why is it that not even hardware acceleration can make Java run as fast as native code? Some reasons are listed on this slide.

First we have things like array bounds checking, dynamic type checking, garbage collection. These are built-in features of Java that you can’t avoid. Then we have expensive native calls. I should mention that you can’t even have native code in your own app, but the built-in libraries often need to call native functions.

One important thing is that you get no access to SIMD instructions and other special CPU features. When you’re working in native code, you can get a big performance boost by writing some of your critical routines in assembly and using the ARM equivalents of Intel’s MMX and SSE.

Then finally, there’s the problem Java bytecode has a stack-based execution model, whereas all CPUs are using registers. It’s hard for the VM to compile stack-based code into fast register-based code, and that’s probably one of the reasons why the HotSpot VM performs so badly. But there are other reasons, too.

So the bottom line is that Java will remain slower than native code, and we just have to live with that fact. The performance gap will become smaller, but it will not go away.
M3G Overview

Design principles
Getting started
Low-level features
The scene graph
Deforming meshes
Keyframe animation
Summary & demos
M3G Design Principles

#1 No Java code along critical paths

• Move all graphics processing to native code
  – Not only rasterization and transformations
  – Also morphing, skinning, and keyframe animation
  – Keep all data on the native side to avoid Java-native traffic
#2 Cater for both software and hardware

- Do not add features that are too heavy for software engines
  - Such as per-pixel mipmaping or floating-point vertices

- Do not add features that break the OpenGL 1.x pipeline
  - Such as hardcoded transparency shaders
M3G Design Principles

#3 Maximize developer productivity

- Address content creation and tool chain issues
  - Export art assets into a compressed file (.m3g)
  - Load and manipulate the content at run time
  - Need scene graph and animation support for that
- Minimize the amount of “boilerplate code”
Here are some more design issues that we had to keep in mind.

Number four, minimize engine complexity. This meant that a commercial implementation should be doable in 150k, including the rasterizer.

Number five, minimize fragmentation. This means that we wanted to have a tight spec, so that you don’t have to query the availability of each and every feature. There are no optional parts or extensions in the API, although some quality hints were left optional. For instance, perspective correction.

And finally, we wanted to have a compact API that can be deployed right away, but so that adding more features in the future won’t cause ugly legacy.
Why a New Standard?

- OpenGL ES is too low-level
  - Lots of Java code, function calls needed for simple things
  - No support for animation and scene management
  - Fails on Design Principles 1 (performance) and 3 (productivity)
  - …but becomes more practical as Java performance increases

- Java 3D is too bloated
  - A hundred times larger (!) than M3G
  - Still lacks a file format, skinning, etc.
  - Fails on Design Principles 1, 3, and 4 (code size)

Okay, so why did we have to define yet another API, why not just pick an existing one?

OpenGL ES would be the obvious choice, but it didn’t fit the Java space very well, because you’d need a lot of that slow Java code to get anything on the screen. Also, you’d have to do animation yourself, and keep all your scene data on the Java side. Basically you’d spend more time writing your code, and yet the code would run slower in the end. That might change in the future, when Java VMs become faster, but don’t hold your breath.

The other choice that we had was Java 3D. At first it seemed to match our requirements, and we gave it a serious try. But then it turned out that the structure of Java 3D was simply too bloated, and we just couldn’t simplify it enough to fit out target devices. Besides, even though the Java 3D is something like a hundred times larger than M3G, it still lacks crucial things like a file format and skinning. It’s also too damn difficult to use.
M3G Overview

Design principles

Getting started
Low-level features
The scene graph
Deforming meshes
Keyframe animation
Summary & demos
The Programming Model

• Not an “extensible scene graph”
  – Rather a black box – much like OpenGL
  – No interfaces, events, or render callbacks
  – No threads; all methods return only when done

• Scene update is decoupled from rendering
  – render ➔ Draws an object or scene, no side-effects
  – animate ➔ Updates an object or scene to the given time
  – align ➔ Aligns scene graph nodes to others
Key Classes

- **Graphics3D**: 3D graphics context
  Performs all rendering

- **Loader**: Loads individual objects and entire scene graphs (.m3g and .png files)

- **World**: Scene graph root node
Rendering State

- Graphics3D contains global state
  - Frame buffer, depth buffer
  - Viewport, depth range
  - Rendering quality hints

- Most rendering state is in the scene graph
  - Vertex buffers, textures, matrices, materials, ...
  - Packaged into Java objects, referenced by meshes
  - Minimizes Java-native data traffic, enables caching
Graphics3D: How To Use

• Bind a target to it, render, release the target

```java
void paint(Graphics g) {
    myGraphics3D.bindTarget(g);
    myGraphics3D.render(world);
    myGraphics3D.releaseTarget();
}
```

• Tip: Do not mix 2D and 3D rendering
M3G Overview

Design principles
Getting started
Low-level features
The scene graph
Deforming meshes
Keyframe animation
Summary & demos
Renderable Objects

- **Sprite3D**: 2D image placed in 3D space, Always facing the camera
- **Mesh**: Made of triangles, Base class for meshes
Sprite3D

- 2D image with a position in 3D space
- Scaled mode for billboards, trees, etc.
- Unscaled mode for text labels, icons, etc.
Mesh

- A common VertexBuffer, referencing VertexArrays
- IndexBuffers (submeshes) and Appearances match 1:1
### VertexBuffer Types

<table>
<thead>
<tr>
<th></th>
<th>8-bit</th>
<th>16-bit</th>
<th>32-bit</th>
<th>Float</th>
<th>2D</th>
<th>3D</th>
<th>4D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Texcoords</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Normals</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Colors</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Relative to OpenGL ES 1.1

Floating point vertex arrays were excluded for performance and code size reasons. To compensate, there are floating point scale and bias terms for vertex and texcoord arrays. They cause no overhead, since they can be implemented with the modelview or texture matrix.

Homogeneous 4D coordinates were dropped to get rid of nasty special cases in the scene graph, and to speed up skinning, morphing, lighting and vertex transformations in general.
IndexBuffer Types

<table>
<thead>
<tr>
<th></th>
<th>8-bit</th>
<th>16-bit</th>
<th>implicit</th>
<th>Strip</th>
<th>Fan</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Lines</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Points</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Point sprites</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td>✗</td>
</tr>
</tbody>
</table>

Relative to OpenGL ES 1.1 + point sprite extension

The set of rendering primitives was reduced to a minimum: triangle strips with 16-bit indices (equivalent to glDrawElements) or implicit indices (glDrawArrays).

Point sprites are missing for a good reason: The M3G spec had been publicly available for almost a year until point sprites were added into OpenGL ES, and even then, only as an extension.
Buffer Objects

- Vertices and indices are stored on server side
  - Very similar to OpenGL Buffer Objects
  - Allows caching and preprocessing (e.g., bounding volumes)

- Tradeoff – Dynamic updates have some overhead
  - At the minimum, just copying in the Java array contents
So how should you implement a particle system, given that points and point sprites are not supported?

The first idea that comes to mind is to use Sprite3D. However, that would make every particle an independent object, each with its own modelview matrix, texture, and other rendering state. This implies a separate OpenGL draw call and lots of overhead for each particle.

It is more efficient to represent particles as textured quads, all glued into one big triangle strip that can be drawn in a single call. To make the particles face the viewer, set up automatic node alignment for the Mesh that encloses the particle system.

At run time, just update the particles’ x, y, z coordinates and colors in their respective VertexArrays.
Functionally related blocks of rendering state are grouped together. Appearances as well as individual Appearance components can be shared by arbitrary number of meshes.

This saves memory space, reduces garbage collection, and allows implementations to quickly sort objects based on their rendering state.
The Fragment Pipeline
Rendering Tips

• Most OpenGL ES performance tips apply
  – Use mipmapping to save in memory bandwidth
  – Use multitexturing to save in T&L and triangle setup
  – SW: Minimize per-pixel operations
  – HW: Minimize shading state changes

• Some of the tips are used by M3G engines
  – Rendering state sorting
  – View frustum culling

Most OpenGL ES performance tips given by Ville in the previous presentation apply also for M3G applications. A few of the most important and universally applicable tips are repeated here.

M3G engines generally perform shader state sorting and view frustum culling in retained mode. However, any culling done by the engine is very conservative. The engine does not know which polygon mesh is a wall that’s going to stay where it is, for instance. If you have a scene that could be efficiently represented as a BSP tree, you can’t expect the engine to figure that out. You need to construct the tree yourself, and keep it in the application side.
Rendering Tips

• Use layers to impose rendering order
  – Appearance contains a layer index (integer)
  – Defines a global ordering for submeshes & sprites
  – Useful for multipass rendering, background geometry, etc.

• Use the perspective correction hint – but wisely
  – Usually much faster than increasing triangle count
  – Nokia: 2% fixed overhead, 20% in the worst case
  – Use the hint where necessary, and nowhere else
M3G Overview

Design principles
Getting started
Low-level features
The scene graph
Deforming meshes
Keyframe animation
Summary & demos
Scene graph nodes can’t have more than one parent, so the scene graph is actually just a tree.

Even though nodes can’t be instanced, their component objects can. Textures, vertices, and all other substantial data is in the components, and only referenced by the nodes.
Node Transformations

- From this node to the parent node
- Composed of four parts
  - Translation T
  - Orientation R
  - Non-uniform scale S
  - Generic 3x4 matrix M
- Composite: \( C = T R S M \)
Node Transformations

**Tip:** Keep the transformations simple
- Favor the T R S components over M
- Avoid non-uniform scales in S

**Tip:** Rotating about an arbitrary point (pivot)
- No direct support for pivot translation: $C = T P^{-1} R P S M$
- Method 1: Combine $T' = T P^{-1}$ and $M' = P S M \Rightarrow C = T' R M'$
  - Drawback: Does not allow S to be animated
- Method 2: Use extra Group nodes
Terrain Rendering

**Tip:** Easy terrain rendering
- Split the terrain into tiles (Meshes)
- Put the meshes into a scene graph
- The engine will do view frustum culling

**Tip:** Terrain rendering with LOD
- Preprocess the terrain into a quadtree
- Quadtree leaf node == Mesh object
- Quadtree inner node == Group object
- Enable nodes yourself, based on the view frustum

Since the modelview matrix of each tile will be unique, small rounding errors in the vertex pipeline may cause cracks between tiles. A simple solution is to make the tiles overlap each other a bit.
The File Format

• Characteristics
  – Individual objects, entire scene graphs, anything in between
  – Object types match 1:1 with those in the API
  – Optional ZLIB compression of selected sections
  – Can be decoded in one pass – no forward references
  – Can reference external files or URIs (e.g. textures)
  – Strong error checking
M3G Overview

Design principles
Getting started
Low-level features
The scene graph
**Deforming meshes**
Keyframe animation
Summary & demos
Deforming Meshes

- MorphingMesh
  - Vertex morphing mesh

- SkinnedMesh
  - Skeletally animated mesh
MorphingMesh

- Traditional vertex morphing animation
  - Can morph any vertex attribute(s)
  - A base mesh $\mathbf{B}$ and any number of morph targets $\mathbf{T}_i$
  - Result = weighted sum of morph deltas

$$ R = \mathbf{B} + \sum_i w_i (\mathbf{T}_i - \mathbf{B}) $$

- Change the weights $w_i$ to animate
MorphingMesh

Base

Target 1
eyes closed

Target 2
mouth closed

Animate eyes
and mouth
independently
SkinnedMesh

• Articulated characters without cracks at joints
• Stretch a mesh over a hierarchic “skeleton”
  – The skeleton consists of scene graph nodes
  – Each node (“bone”) defines a transformation
  – Each vertex is linked to one or more bones

\[ v' = \sum_i w_i M_i B_i v \]

– \( M_i \) are the node transforms – \( v \), \( w \), \( B \) are constant

In the equation,
• \( v \) is the vertex position in the SkinnedMesh node’s coordinates
• \( B_i \) is the fixed at-rest transformation from SkinnedMesh to bone \( N_i \)
• \( M_i \) is the dynamic transformation from bone \( N_i \) to SkinnedMesh
• \( w_i \) is the weight of bone \( N_i \) (the weights are normalized)
• \( 0 \leq i \leq N \), where \( N \) is the number of bones associated with \( v \)
• \( v' \) is the final position in the SkinnedMesh coordinate system
SkinnedMesh

Neutral pose, bones at rest

shared vertex, weights = (0.5, 0.5)

"skin"

non-shared vertex

Bone A

Bone B
The empty dots show where the vertex would end up if it were associated with just one of the bones, respectively.

As the vertex is weighted equally by bones A and B, the final interpolated vertex lies in between the empty dots.
SkinnedMesh

No skinning

Smooth skinning
two bones per vertex
M3G Overview

Design principles
Getting started
Low-level features
The scene graph
Deforming meshes
Keyframe animation
Summary & demos
Animation Classes

- **KeyframeSequence**: Storage for keyframes. Defines interpolation mode.
- **AnimationController**: Controls the playback of one or more sequences.
- **AnimationTrack**: A link between sequence, controller and target.
- **Object3D**: Base class for all objects that can be animated.
Animation Classes

Object3D

AnimationTrack

AnimationController

KeyframeSequence

Identifies animated property on this object
KeyframeSequence

Keyframe is a time and the value of a property at that time
Can store any number of keyframes
Several keyframe interpolation modes
Can be open or closed (looping)

Diagram courtesy of Sean Ellis, Superscape
KeyframeSequence

Keyframe is a time and the value of a property at that time
Can store any number of keyframes
Several keyframe interpolation modes
Can be open or closed (looping)

Diagram courtesy of Sean Ellis, Superscape
KeyframeSequence

Keyframe is a time and the value of a property at that time
Can store any number of keyframes
Several keyframe interpolation modes
Can be open or closed (looping)

Diagram courtesy of Sean Ellis, Superscape
AnimationController

Can control several animation sequences together
Defines a linear mapping from world time to sequence time
Multiple controllers can target the same property

Diagram courtesy of Sean Ellis, Superscape
Animation

1. Call animate(worldTime)
2. Calculate sequence time from world time
3. Look up value at this sequence time
4. Apply value to animated property

Object3D → AnimationTrack → AnimationController → KeyframeSequence

Diagram courtesy of Sean Ellis, Superscape
Tip: You can read back the animated values
- Much faster than Java if you need floating point interpolation
- Target N-dimensional tracks (N > 4) to a dummy MorphingMesh

Tip: Interpolate quaternions as ordinary 4-vectors
- SLERP and SQUAD are slower, but need less keyframes
- Quaternions are automatically normalized before use

Almost any property in the API can be targeted by keyframe animation. Once you have called animate(), you can read back the updated values with the usual get() methods.

If you need to do interpolation with floating point quantities elsewhere in your application, you can use the M3G animation engine for that. Set up your keyframes and a dummy object as the animation target, animate it, and read out the result.

Then what if you have a large array with N elements that you need to interpolate? There is only one animation target that can take keyframes with more than four components, and that’s the MorphingMesh weights array. So, you need to set up a dummy MorphingMesh with N morph targets, and target your animation to that.

Although that’s inconvenient and somewhat ridiculous, it doesn’t really cost you that much in terms of memory. The size of the dummy mesh will be a few dozen bytes per morph target. You will probably run out of CPU cycles – doing floating point interpolation – before you run out of memory.
M3G Overview

Design principles
Getting started
Low-level features
The scene graph
Deforming meshes
Keyframe animation

Summary & demos
Predictions

• Resolutions will grow rapidly from 128x128 to VGA
  – Drives graphics hardware into all high-resolution devices
  – Software rasterizers can’t compete above 128x128

• Bottlenecks will shift to Physics and AI
  – Bottlenecks today: Rasterization and any Java code
  – Graphics hardware will take care of geometry and rasterization
  – Java hardware will increase performance to within 50% of C/C++

• Java will reinforce its position as the dominant platform
Summary

• M3G enables real-time 3D on mobile Java
  – By minimizing the amount of Java code along critical paths
  – Designed for both software and hardware implementations

• Flexible design leaves the developer in control
  – Subset of OpenGL ES features at the foundation
  – Animation & scene graph features layered on top

Installed base growing by the millions each month
Demos
Q&A

Thanks: Sean Ellis, Kimmo Roimela, Nokia M3G team, JSR-184 Expert Group