

# ***Mobile 3D Graphics Programming***

*June 24, 2005*



***WHITE PAPER***

---

## *Mobile 3D Graphics Programming*

By

Motocoder staff

---

Recently, more and more phones are starting to support JSR 184 specification (Mobile 3D Graphics API). For many Motorola J2ME devices, such as C975, C980, E1000/E1000R, V975, V980, A780 and E680, the 3D function extends the device's multi-media ability and brings to the customer a new visual experience. It is expected that 3D games or applications based on Mobile 3D Graphics will be significantly increasing with the growing of 3D functions in mobile devices.

---

### ***Mobile 3D Graphics API introduction***

The Mobile 3D Graphics API is an optional package comprising about 250 methods in about 30 classes. There are several import classes in this package as follows:

- Object3D class – the most important class because it is the base class of almost all the classes in the package. All the extended classes from Object3D class can be rendered and loaded from m3g file.
- World class – the root of the scene graph structure. All 3D objects should be added into the world object. When loading an M3G file, the root 3D object usually is the world object. While rendering a scene, the world object should be passed to Graphics3D object as a rendered object.
- Graphics3D class – is a singleton 3D graphics context that can be bound to a rendering target. All rendering is done here.
- Loader class - a synchronous loader (deserializer) for entire scene graphs, individual branches, and attribute objects. This class can be used to load an M3G file which contains all the 3D objects.

---

### ***Basic 3D Application Framework***

A 3D image should be rendered on a Canvas or GameCanvas object, so we need to define a class which extends from Canvas or GameCanvas. Besides this, the application usually also needs a timer or thread to show animation or control the movement of the object. Below is a basic 3D canvas implementation.

```
class M3GCanvas extends GameCanvas implements Runnable{  
    public M3GCanvas(){  
        super(false);
```

```

        setFullScreenMode(true);

        // create/load world and other objects

        Thread t = new Thread(this);
        t.start();
    }

    public void run() {
        Graphics g = getGraphics();

        while(true){
            // rotate, move or animate object
            try{
                // Binds the given Graphics or mutable Image2D
                // as the rendering target of this Graphics3D
                g3d.bindTarget(g);

                g3d.render(world); // Render the world
            }finally{
                g3d.releaseTarget();
            }
            flushGraphics();

            try {
                Thread.sleep(100);
            } catch (Exception e) {
            }
        }
    }
    ...
}

```

Here, it is recommended to add a thread sleep statement as below, because it will give other threads the chance to run and avoid taking up too much CPU resource.

```

        try {
            Thread.sleep(100);
        } catch (Exception e) {
        }
    }

```

The following code is a 3D MIDlet implementation. It very simply just creates a 3D canvas in startApp() method.

```

public class TetrahedronDemo extends MIDlet implements CommandListener{
    private Command cmdExit;
    private Display d;
    private M3GCanvas m3gCanvas;

    public TetrahedronDemo(){
        d = Display.getDisplay(this);

        cmdExit = new Command("Exit", Command.EXIT, 0);

        m3gCanvas = new M3GCanvas();
        m3gCanvas.addCommand(cmdExit);
        m3gCanvas.setCommandListener(this);
    }

    public void startApp() {
        d.setCurrent(m3gCanvas);
    }

    public void pauseApp() {

    }

    public void destroyApp(boolean unconditional) {

    }

    public void commandAction(Command c, Displayable d){
        notifyDestroyed();
    }
}

```

---

## Loading 3D Object

The above code shows a basic 3D application framework. However, I did not give how to create 3D object. There are two ways to acquire a 3D object: from data array or from an M3G file. M3G file format is defined in JSR 184 specification and is provided as a compact and standardised way of populating a scene graph. The code below shows how to get a world object from an M3G file.

```
try {
    // Load a m3g file, returns all root object3d object.
    Object3D[] roots = Loader.load("mytest.m3g");

    // Usually, the world is the first root node loaded.
    myWorld = (World)roots[0];
} catch(Exception e) {
    e.printStackTrace();
}
```

Loading 3D content from an M3G file is an easy way to program and get 3D content. It can be used to load a complex scene. However, you have to use a third-party tool to obtain an M3G file. For some simple applications, it is not necessary to purchase such a tool before programming. Instead of loading 3D content from an M3G file, object data also can be stored into arrays. Using this method, the 3D object can be created manually inside the program. The method below creates a colored tetrahedron.

```
private void createTetrahedron() {

    // The vertices used by the tetrahedron. x, y, z
    short []POINTS = new short[] {0, 2, 0, //point 0, top
        0, 0, 1, // point 1
        -1, 0, -1, // point 2
        1, 0, -1 // point 3
    };

    // The points sequence.
    int []INDICES = new int[] {3, 0, 1,
        0, 1, 2,
        1, 2, 3,
        2, 3, 0
    };

    //color for each point
    byte []COLORS = new byte[] { 127, 127, 0,
        127, 0, 0, //R
        0, 127, 0, //G
```

```

        0, 0, 127    //B
    };

    // The length of each sequence in the indices array.
    // the tetrahedron is built by four triangles
    int []LENGTH = new int[] {3, 3, 3, 3};

    VertexArray POSITION_ARRAY, COLOR_ARRAY;
    IndexBuffer INDEX_BUFFER;

    // Create a VertexArray to be used by the VertexBuffer
    POSITION_ARRAY = new VertexArray(POINTS.length / 3, 3, 2);
    POSITION_ARRAY.set(0, POINTS.length / 3, POINTS);
    COLOR_ARRAY = new VertexArray(COLORS.length / 3, 3, 1);
    COLOR_ARRAY.set(0, COLORS.length / 3, COLORS);
    INDEX_BUFFER = new TriangleStripArray(INDICES, LENGTH);

    // VertexBuffer holds references to VertexArrays that contain the
    // positions, colors, normals, and texture coordinates
    // for a set of vertices
    VertexBuffer vertexBuffer = new VertexBuffer();
    vertexBuffer.setPositions(POSITION_ARRAY, 1.0f, null);
    vertexBuffer.setColors(COLOR_ARRAY);

    // Create the 3D object defined as a polygonal surface
    tetrahedron = new Mesh(vertexBuffer, INDEX_BUFFER, null);

    //Set appearance for mesh object
    Appearance appearance = new Appearance();
    PolygonMode polygonMode = new PolygonMode();
    polygonMode.setPerspectiveCorrectionEnable(true);
    // Specifying that both faces of a polygon are to be drawn
    polygonMode.setCulling(PolygonMode.CULL_NONE);
    // Specifying that smooth shading is to be used
    polygonMode.setShading(PolygonMode.SHADE_SMOOTH);
    polygonMode.setTwoSidedLightingEnable(true);
    appearance.setPolygonMode(polygonMode);

```

```

// Set the appearance to the 3D object
tetrahedron.setAppearance(0, appearance);
//mesh.setAppearance(0, new Appearance());

// move the tetrahedron into the screen.
tetrahedron.setTranslation(0.0f, -1.0f, -3.0f);
world.addChild(tetrahedron);
}

```

The code looks quite complex, but in fact many statements can be re-used. If you want to create another 3D object, you just need to change vertices, indices, colours ( if you need colour) and norms ( if necessary) arrays and a little code. The code above creates a tetrahedron object shown as figure 1.



Figure 1

Loading from an array method not only can be used to create a simple object but also can be used to create a complex scene. If you want to convert a complex scene into data array, you may need a

converter tool. Unlike converting M3G file format, there are many free tools available to download. I usually convert a 3D scene file into obj file format first and then convert the obj file into a data array. Java 3D API supports obj file format by providing ObjectFile class which can load obj file into a scene object.

---

## ***Notes of using Mobile 3D Graphics API***

Currently emulator A.3 does not support the JSR 184 feature. We will implement this feature on the SDK for A.3 in a future release. To work around this, the developer may use M.3 emulator to develop and debug JSR 184 based application and then re-compile on A.3 emulator. In most cases, the application works well on A.3 Motorola devices.

Except JSR 184 API, some Motorola phones support Motorola 3D API. The two sets of API are not the same, so please pay close attention to device capabilities when choosing a device. Please refer to the API Matrix of our handsets in the SDK documentation for details on which handsets support JSR 184 API or Motorola 3D API.

---

## ***Conclusion***

Mobile 3D Graphics API provides an easy way to develop 3D based application for J2ME mobile devices. This article introduces two possible ways to prepare 3D content for applications. In the actual development, you can choose the one which best suits your applications requirements.

---

## ***References***

1. JSR 184, Mobile 3D Graphics, <http://jcp.org/en/jsr/detail?id=184>
2. Getting Started With the Mobile 3D Graphics API for J2ME, <http://developers.sun.com/techtopics/mobility/apis/articles/3dgraphics/>
3. Killer Game Programming in Java, <http://fivedots.coe.psu.ac.th/%7Ead/jg/index.html>