M3G Chapter 4.5. Using M3G File Models

It's easy to load a model from an M3G file: call Loader.load(), and that's it, or is it?

What if you're not sure what the M3G file contains? Perhaps you only need a single 3D shape, embedded deep inside the scene graph? What if you need a reference to the shape's AnimationTrack or its Material? You need to examine the loaded scene graph to find those nodes.

Even if you manage to get the model into your application, it may be incorrectly scaled, or positioned off in the heavens. You need an interactive way of adjusting the shape's size and location, and those adjustments should be used whenever the model is subsequently loaded.

The ViewM3G application loads an M3G file into a scene familiar from previous chapters: a floor showing a large green XZ grid, a blue background, with lighting. The camera is mobile, so the model can be viewed from different angles. More importantly, the model can be scaled and translated. Figure 1 shows a massive tube loaded into ViewM3G, but in Figure 2 we've cut it down to size, and repositioned it. The scaling and translation values are shown in the top-right corner of the screen.

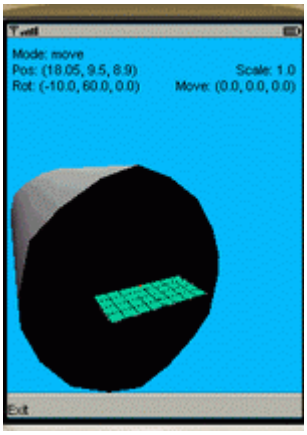

Figure 1. The Tube when First Loaded.

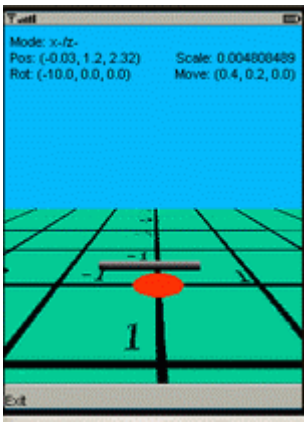

Figure 2. The Tube after Rescaling and Translation.

The scaling and translation information displayed by ViewM3G can be utilized when loading the model into other applications, as we'll see later.

The other main component of ViewM3G is its examination of the loaded scene graph, printed to standard output. Figure 3 shows the report when tube.m3g is loaded.

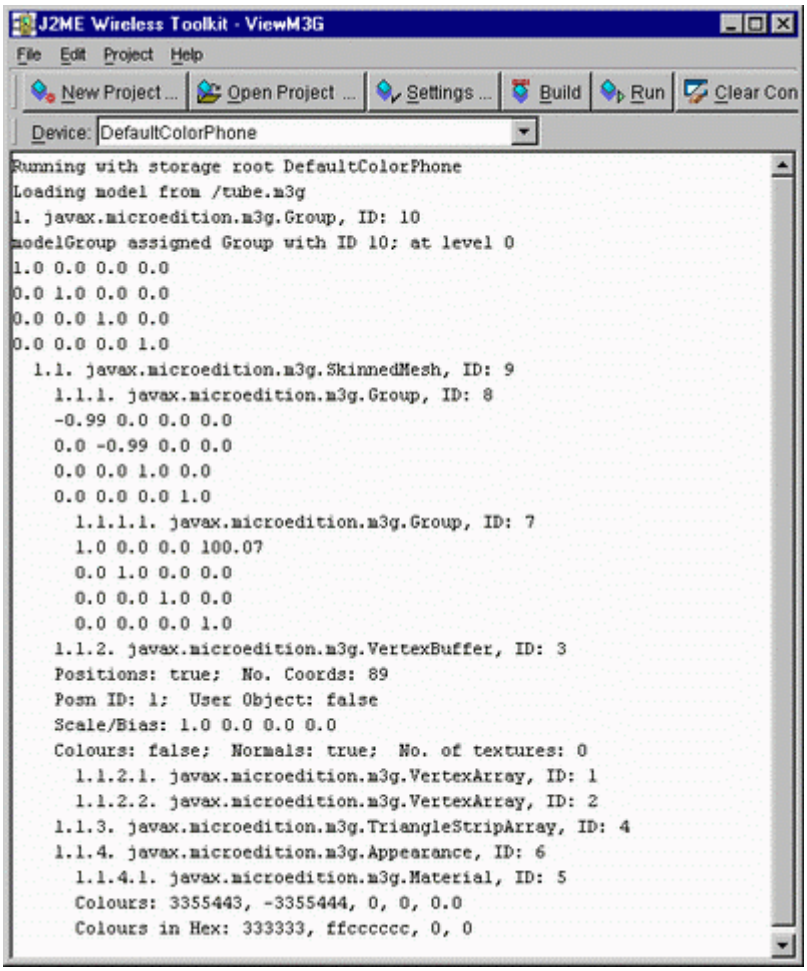

Figure 3. Analysis of the Tube's Scene Graph.

This cascade of class names and numbers will be explained later. It shows, for example, that the tube is a skinned mesh with two Group nodes as joints.

1. The M3G File Format

The M3G file format allows an entire scene graph to be stored in a file, with a World node at the root, and lights, a background, cameras, and shapes, below it. Most of the currently available M3G files contain complete scenes of this kind. However, the format also allows scene branches and individual nodes to be stored. More details can be found in the Loader class description, and the "File Format for Mobile 3D Graphics API" article which comes with the M3G documentation.

A M3G file with a World node at its root has a format like the one shown in Figure 4.

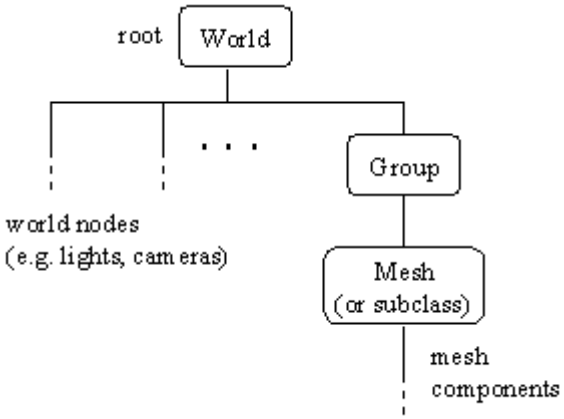

Figure 4. Format of a Scene Graph with a World Node at its Root.

The 3D shape is below a Group node, which is directly below the World node. The scene graph may be considerably more complex than this, with a very large tree of nodes below the Group node, forming dense 3D scenery with AnimationTracks, KeyframeSequences, and the like.

We aren't interested in utilizing the entire loaded scene graph, since ViewM3G already has a World node with lighting, a background, and a camera. Consequently, ViewM3G detaches the top-most Group node, and treats it as the 'model' to be added to its scene graph.

The other kind of M3G file has a format shown in Figure 5.

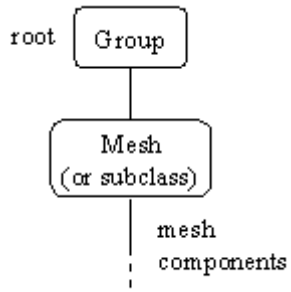

Figure 5. Format of a Scene Graph with a Group Node at its Root.

This kind of scene graph is typically created when a single 3D shape is exported from a modeling package. At present, only 3D Studio Max offers an M3G exporter, but I expect that'll change fairly soon. As with Figure 4, the model is below a Group node, and there may be considerably more branches and sub-branches surrounding it, representing component shapes.

This type of scene graph is already in a suitable form to be attached to ViewM3G's scene graph. We just connect the root Group node to the scene.

2. Scene Graph Examination

The simplest way of examining a scene graph is to recursively traverse over its nodes starting from the root.

Figure 3 shows the output from ViewM3G when it examines the scene graph loaded from tube.m3g. The scene graph is shown diagrammatically in Figure 6.

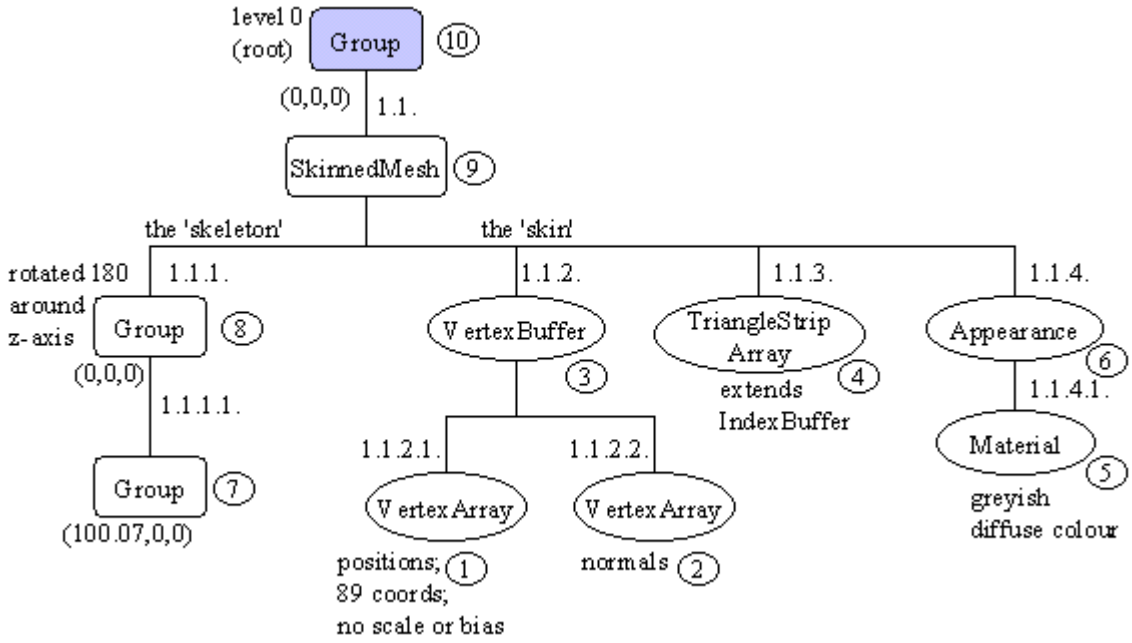

Figure 6. Scene Graph Loaded from tube.m3g.

The 'section/sub-section' numbers in Figure 3 label the branches of the scene graph in Figure 6. This numbering scheme is generated by ViewM3G, it isn't part of the scene graph.

The numbered ovals next to the nodes in Figure 6 are user IDs, which can be attached to nodes by the scene graph author. These values are printed after the "ID:" strings in Figure 3. If a node contains a user ID, then the number can be employed by Object3D.find() to retrieve a reference to that node. Unfortunately, there's no requirement that nodes be given IDs, and an ID doesn't need to be unique.

Other details of the examination in Figure 3, such as the matrices printed for the Groups, will be explained when we consider the ViewModel class.

3. Class Diagrams for ViewM3G

Figure 7 shows the class diagrams for all the classes in the ViewM3G application, including their public and protected methods.

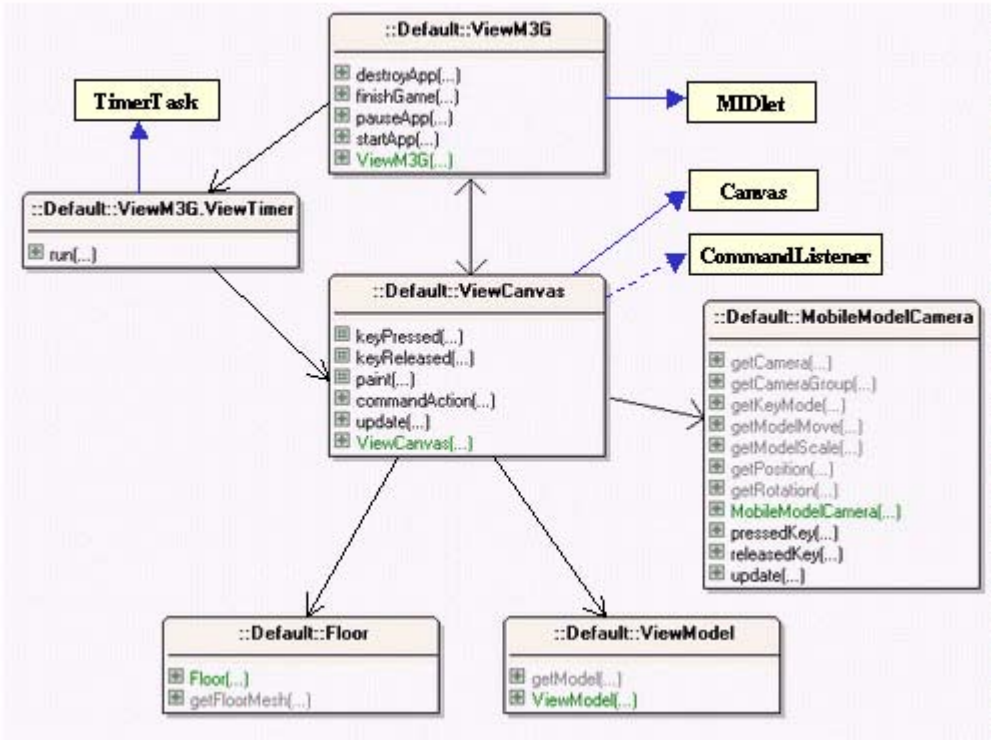

Figure 7. Class Diagrams for ViewM3G.

The top-level MIDlet, ViewM3G, and its TimerTask, ViewTimer, perform the same tasks as earlier examples, so I'll skip merrily pass them. The same goes for the Floor class, which is the same as the one in AnimM3G (M3G Chapter2); it's used to display a 8-by-8 green grid on the XZ plane.

This leaves three classes: ViewCanvas, MobileModelCamera, and ViewModel.

ViewCanvas creates the scene: a blue background, a light, the floor (using Floor); it utilizes ViewModel to load a model from a specified M3G file.

MobileModelCamera has two functions: it adds mobility to the camera *and* to the loaded model. The mobile camera part is a copy of the MobileCamera class introduced in MorphM3G (M3G Chapter 3), joined by a small amount of new code to affect the model's scale and position.

4. The ViewCanvas Class

ViewCanvas calls buildScene() to create the various parts of its scene graph:

```
// private static final String MODEL_FN = "/tube.m3g";
// M3G file to load


private void buildScene()
```



```
{
  ViewModel vm = new ViewModel(MODEL_FN);
  Group modelGroup = vm.getModel();
  scene.addChild( modelGroup );  // add the model

  addCamera(modelGroup);
  addLights();
  addBackground();
  addFloor();
}
```

The comments in the ViewCanvas class list many alternative values for MODEL_FN. I found the files in Sun's Wireless Toolkit, and in Nokia's M3G introductory article, "3-D Game Development on JSR-184", available from http://www.forum.nokia.com/main/1,6566,040,00.html?fsrParam=2-3-/main.html&fileID=4554. The tube example was kindly sent to me by Leon HongKong.

The modelGroup node returned by ViewModel's getModel() method is the top-level Group node of the loaded model, as shown in Figures 4 and 5.

addCamera() creates a MobileModelCamera instance:

```
private void addCamera(Group modelGroup)
{
  mobCam = new MobileModelCamera(modelGroup,
                              getWidth(), getHeight());
       // the mobile camera can affect modelGroup
  scene.addChild( mobCam.getCameraGroup() );
  scene.setActiveCamera( mobCam.getCamera() );
}
```

A reference to the loaded model's Group node is passed to MobileModelCamera, so it can be scaled and translated.

4.1. Update and Repaint

ViewCanvas's update() method, called by the ViewTimer TimerTask object, is pleasantly short:

```
public void update()
{ mobCam.update();
  repaint();
}
```

Unlike many of our earlier examples, there's no call to animate(), since nothing is using an animation track. The mobile camera's update() method is called so that the effect of a repeating key is felt in each update.

paint() is similar to previous examples: it draws the 3D scene, then writes information taken from the mobile camera to the screen. In ViewM3G, this information includes model scaling and translation details, drawn in the top-right corner of the screen:


```
int rightSide = getWidth() - 5;
g.drawString( mobCam.getModelScale(), rightSide, 18,
                              Graphics.TOP|Graphics.RIGHT);
g.drawString( mobCam.getModelMove(), rightSide, 31,
                              Graphics.TOP|Graphics.RIGHT);
```

The results can be seen in the screenshots in Figures 1 and 2, and Figure 8 below.

5. The MobileModelCamera Class

MobileModelCamera builds upon MobileCamera, so it can move, rotate, and 'float' the camera, and also translate and scale the supplied Group node.

The advantages of having camera mobility are illustrated when the tube model is first loaded. Figure 8 shows the scene, but where's the tube?

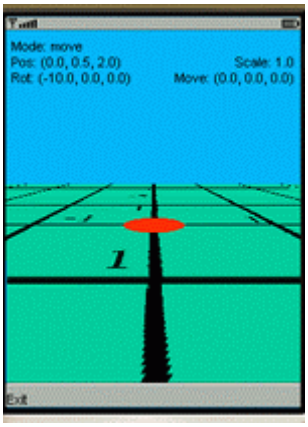

Figure 8. The Missing Tube.

The camera's initial position coincides with the inside right edge of the tube, and since the model employs backface culling, the tube is invisible. It comes into view when the viewpoint is moved 0.1 units to the right, away from the z-axis.

I won't explain the parts of MobileModelCamera unchanged from MobileCamera; I politely refer you to M3G Chapter 3 for descriptions of those features.

Perhaps the worst thing about MobileModelCamera is the increase in the number of modes: the original MobileCamera class has three for moving, rotating, and raising the camera; MobileModelCamera adds another two to deal with the model's translation and scaling:

```
// key mode constants
private static final int MOVE = 0;   // move left, right, fwd, back
private static final int ROTATE = 1; // turn left, right, up, down
private static final int FLOAT = 2;  // move up, down
private static final int SCALE_YTRANS = 3;  // model scale, y-trans
private static final int XZTRANS = 4;       // model x- or z- trans
private static final int NUM_MODES = 5;
```


This translates into numerous presses of the "select" key by the finger-weary user, just to cycle through the modes. Five presses are needed before a mode is revisited, which seems a tad excessive.

Another drawback of my moded approach is the combination of the scaling and y-axis translation in SCALE_YTRANS, which was done solely to reduce the number of modes. It isn't possible to add y- axis moves to the x- and z- axis translation mode (XZTRANS), because there's only four arrow keys to play with in any given mode.

When update() is called, the current key mode determines what will happen:

```
public void update()
{
  switch(keyMode) {
    case MOVE: updateMove();  break;
    case ROTATE: updateRotation();  break;
    case FLOAT: updateFloating();  break;
    case SCALE_YTRANS: scaleTransModel(); break;
    case XZTRANS: transModel(); break;
    default: break;
  }
}  // end of update()
```

The updateMove(), updateRotation() and updateFloating() methods are unchanged from MobileCamera; scaleTransModel() and transModel() are new.

Scaling and translation are applied to the top-level Group node for the model, which is passed to the MobileModelCamera instance through its constructor:

```
// globals for model manipulation
private Group modelGroup;  // Group node for the model
private float scaleFactor;
private float xMove, yMove, zMove;


public MobileModelCamera(Group mg, int width, int height)
{
  modelGroup = mg;
  scaleFactor = 1.0f;
  xMove = 0.0f;  yMove = 0.0f; zMove = 0.0f;
     : // the rest of the constructor
}
```

Scale changes are recorded in scaleFactor, and translations in xMove, yMove, and zMove. Their values can be accessed by ViewCanvas calling getModelScale() and getModelMove().

scaleTransModel() uses Transformable.scale() to adjust the Group node's size, and Transformable.translate() to move it along the y-axis.

```
// global translation and scaling increments for the model
private static final float MOVE_INCR = 0.1f;
private static final float SCALE_UP = 1.1f;
```



```
  private static final float SCALE_DOWN = 1.0f/SCALE_UP;


  private void scaleTransModel()
  // scale or y-translate the model
  {
    if (upPressed) {          // scale up
      modelGroup.scale(SCALE_UP, SCALE_UP, SCALE_UP);
      scaleFactor *= SCALE_UP;
    }
    else if (downPressed) {   // scale down
      modelGroup.scale(SCALE_DOWN, SCALE_DOWN, SCALE_DOWN);
      scaleFactor *= SCALE_DOWN;
    }
    else if (leftPressed) {   // move down the y-axis
      modelGroup.translate(0, -MOVE_INCR, 0);
      yMove -= MOVE_INCR;
    }
    else if (rightPressed) {   // move up the y-axis
      modelGroup.translate(0, MOVE_INCR, 0);
      yMove += MOVE_INCR;
    }
  }  // end of scaleTransModel()
```

transModel() utilizes the same approach as the y-axis translation code in scaleTransModel(). The up/down keys move the model along the z-axis, and the left/right keys shift it left and right.

6. The ViewModel Class

ViewModel performs three main tasks:

1. It loads the scene graph stored in the specified M3G file.
2. It finds and stores a reference to the top-most Group node above the scene graph's model.
3. It traverses the scene graph, printing out information that the programmer may find useful when manipulating the model later.

1. The loading part is straightforward: it's just a call to Loader.load().
2. Finding the top-most group node is a little tricky since the model may be in a scene graph with a World node at its root (see Figure 4) or on its own (as in Figure 5).

   We deal with both cases by storing the first Group node found during the top-down traversal of the scene graph, since this is always above the model (at least in all the examples we've tried).

   If the Group node is connected to a World node, then it has to be detached, so it can be added to ViewM3G's scene.

   We also take the opportunity to apply an x-axis rotation of -90 degrees to the Group node. This is motivated by the observation that all the existing World-

based scenes seem to have been created in 3DS Max, where the positive z-axis is pointing downwards instead of straight out of the screen. The consequence is that when the model is loaded into M3G, it's face-down. The rotation turns the model upwards, to proudly face front.

3. The examination phase is a recursive graph traversal utilizing Object3D.getReferences(). This method returns an array of links leaving a given node, which can be examined in the same manner. The following example, which is based on one in the getReferences() documentation, illustrates the general idea:

```
void traverseGraph(Object3D obj)
{  // a top-down recursive traverse
   int numRefs = obj.getReferences(null);
   if (numRefs > 0) {
      Object3D[] objArray = new Object3D[numRefs];
      obj.getReferences(objArray);
      for (int i = 0; i < numRefs; i++) {
         processObject(objArray[i]);   // process object i
         traverseGraph(objArray[i]);   // and its children
      }
   }
}
```

The first call to getReferences() returns the number of links out of the node, which is used to size an array. The array is populated by the second call to getReferences ().

processObject() is often a giant switch statement which distinguishes between all the possible subclasses of Object3D. In our version of this code, we only examine the Group, VertexBuffer, and Material nodes.

The examination details are sent to standard output, which is the KToolbar message window in Sun's Wireless Toolkit. This is adequate for very simple model's like the tube shown in Figure 3, but more realistic models generate many, many screens of text. One alternative is to display the output using MIDP's GUI capabilities, an approach taken by Ben Hui's M3G viewer, being developed at http://www.benhui.net/modules.php?name=Mobile3D.

6.1. Loading the M3G File

The M3G file is loaded in ViewModel's constructor, and its top-level node is passed to the examination method, examineObjects().

```
// globals
private Group modelGroup;     // top-level Group node for the model

private boolean foundGroup;   // Group node found during search?
private boolean isSceneRoot;
                // is found Group node the root of the scene graph?


public ViewModel(String fn)
// load and examine the model
{
  System.out.println("Loading model from " + fn);
```



```
    modelGroup = null;
    foundGroup = false;    // Group node not found yet
    isSceneRoot = true;    // assume it will be the root of the graph

    Object3D[] parts = null;
    try {
      parts = Loader.load(fn);
    }
    catch (Exception e)
    { e.printStackTrace(); }

    examineObjects(parts, "", null, 0);
    adjustModelPosition();
  } // end of ViewModel()
```

examineObjects() traverses the graph, printing out useful information. It also looks for the top-level Group node, and stores a reference to it in modelGroup (and sets foundGroup to be true). If the scene graph is a World scene, then isSceneRoot is set to false, which triggers an x-axis rotation of modelGroup in adjustModelPosition().

Loader.load() returns an array of Object3Ds, since a graph may have multiple roots (i.e. nodes not referenced by other nodes). In practice, this generality doesn't seem to be used in real M3G files: either there's one World node or a single Group node.

6.2. Examining the Scene Graph

examineObjects() carries out a basic examination of each of the objects in the parts[] array, printing the object's class name and its user ID.

```
  private void examineObjects(Object3D[] parts, String section,
                                      Object3D parent, int level)
  {
    String name, subsection;
    for (int i=0; i < parts.length; i++) {
      name = parts[i].getClass().getName();
      subsection = section + (i+1) + ".";
      printlnIndent(subsection + " " + name + ", ID: " +
                                 parts[i].getUserID(), level);
      examineNode(parts[i], name, parent, level);
      examineRefs(parts[i], subsection, level+1);
    }
  }  // end of examineObjects()
```

The level integer and section String are used by printlnIndent() to indent its output, and prefix it with a section label (see Figure 3 for examples).

It's useful to know a node's user ID, since Object3D.find() can return a reference to a node based on a supplied ID. However, nodes don't have to be assigned IDs (by default they have the value 0), and the value may not be unique.

Figures 3 and 6 show that the scene graph for tube.m3g has two 'joints' (Group nodes) in its SkinnedMesh, with unique ID's 7 and 8. These joints can be accessed with find ():

```
    try {
      Object3D[] parts = Loader.load("/tube.m3g");
```



```
    }
    catch (Exception e)
    {  e.printStackTrace();  }

    Group mGroup = (Group) parts[0];
    Group joint1 = (Group) mGroup.find(8);
    Group joint2 = (Group) mGroup.find(7);
```

examineRefs() bears a striking resemblance to the traverseGraph() method from earlier:

```
private void examineRefs(Object3D obj, String section, int level)
{
  int numRefs = obj.getReferences(null);
  if (numRefs > 0) {
    Object3D[] parts = new Object3D[numRefs];
    obj.getReferences(parts);
    examineObjects(parts, section, obj, level);
  }
}
```

The method obtains the object's links in an array, and recursively calls examineObjects().

6.2.1. Examining a Node

examineNode() is the starting point for more detailed examinations of particular types of nodes.

```
private void examineNode(Object3D part, String name,
                                Object3D parent, int level)
{
  if (name.endsWith("Group")) {
    storeGroup((Group) part, parent, level);
    examineGroup((Group) part, level);
  }
  else if (name.endsWith("VertexBuffer"))
    examineVertexBuffer((VertexBuffer) part, level);
  else if (name.endsWith("Material"))
    examineMaterial( (Material) part, level);
}
```

The multi-way branch calls different methods depending on if the node is a Group, VertexBuffer, or Material. The choice is based on the fully qualified class name (obtained in examineObjects() with getClass().getName()). The use of endsWith() allows us to ignore the package details before the class name.

6.2.2. Examining a Group

examineGroup() prints the composite transform matrix for the Group, which combines the node's generic matrix M, non-uniform scale S, orientation R, and translation T components into a single 4-by-4 matrix, of the form:

$$\begin{bmatrix} r11 & r12 & r13 & tx \\ r21 & r22 & r23 & ty \\ r31 & r32 & r33 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The code:

```
// globals for looking at the transform in a Group node
private float trans[] = new float[16];
private Transform modelTransform = new Transform();


private void examineGroup(Group g, int level)
{
  g.getCompositeTransform(modelTransform);
  modelTransform.get(trans);
  for(int i=0; i < 16; i= i+4)
    printlnIndent( round2dp(trans[i]) + " " +
                   round2dp(trans[i+1]) + " " +
                   round2dp(trans[i+2]) + " " +
                   round2dp(trans[i+3]), level);
}
```

Figure 3 includes output for the two Group nodes that form the skinned mesh's skeleton. The nodes are numbered 7 and 8, with node 8 the parent of node 7.

The matrix for node 8 is:

$$\begin{bmatrix} -0.99 & 0 & 0 & 0 \\ 0 & -0.99 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix for node 7 is:

$$\begin{bmatrix} 1 & 0 & 0 & 100.07 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A Group's (x,y,z) position is stored in the matrix's fourth column, so node 8 is at the origin (0,0,0), and node 7 at (100.07,0,0). The node 7 numbers seem a bit strange, since the tube lies along the negative x-axis.

The answer comes by looking at the rotation components of the matrices. The rotation information is located in the top-left 3-by-3 corner of the matrix; it combines the x-, y-, and z- rotations applied to the node, so can be hard to interpret. However, the rotations for nodes 7 and 8 are straightforward.

Node 7 has an identity matrix for its rotation component, so isn't rotated at all, while node 8 is rotated 180 degrees around the z-axis. We know this since a general rotation about the z-axis takes the form:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In other words, cos θ = -1 in node 8 (after rounding), which means that θ = 180 degrees.

This explains why node 7's position is (100.07,0,0), but the node appears on the negative x-axis: its parent node (node 8) has been turned around the z-axis, so node 7 has been flipped as well. The scene-level position of node 7 is actually (-100.07,0,0).

Scaling information can be extracted from the main diagonal of the matrix. If the scaling factors in the x-, y-, and z- directions are Sx, Sy, and Sz, then they'll appear in the matrix as:

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

However, these values are often obscured by the presence of rotations.

Usually, it's easier to retrieve the transformation components of a node (the M, S, R, and T elements) as separate values.

6.2.3. Examining a VertexBuffer

The general form of a VertexBuffer is shown in Figure 9.

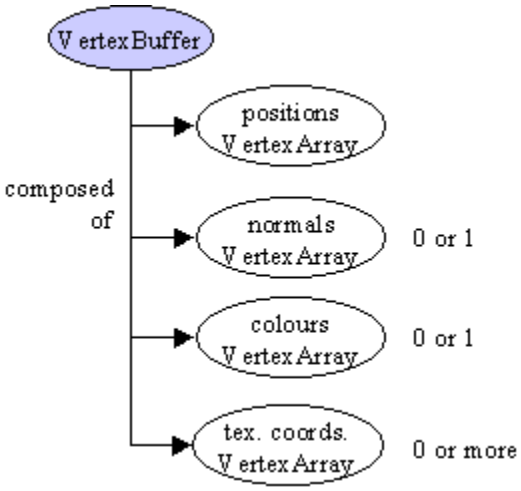

Figure 9. VertexBuffer Elements.

Access to the positions VertexArray would be of great use, for example to calculate a bounding box for the shape. Unfortunately, the M3G API offers no way of accessing the data in the arrays. This means that the VertexBuffer examination has to content itself with checking for the existence of the various arrays (the normals, colours and texture coordinates VertexArrays are optional).


```
private void examineVertexBuffer(VertexBuffer vb, int level)
{
  float[] scaleBias = new float[4];
  VertexArray va = vb.getPositions(scaleBias);

  printlnIndent("Positions: " + (va != null) +
            ";  No. Coords: " + (vb.getVertexCount()/3), level);
  if (va != null)
    printlnIndent("Posn ID: " + va.getUserID() +
       ";  User Object: " + (va.getUserObject() != null), level);

  printlnIndent("Scale/Bias: " + round2dp(scaleBias[0]) + " " +
                                 round2dp(scaleBias[1]) + " " +
                                 round2dp(scaleBias[2]) + " " +
                                 round2dp(scaleBias[3]), level);
  int texCount = 0;
  while (true) {
    if (vb.getTexCoords(texCount, scaleBias) == null)
      break;
    texCount++;
  }

  printlnIndent("Colours: " + (vb.getColors() != null) +
                ";  Normals: " + (vb.getNormals() != null) +
                ";  No. of textures: " + texCount, level);
}  // end of examineVertexBuffer()
```

6.2.4. Examining the Material

The Material information for a shape consists of its ambient, diffuse, emissive, and specular colours, and its shininess. Reporting the colours in hexadecimal form is preferable to decimal, since the alpha, red, green, and blue components of the colour are separated out.

```
private void examineMaterial(Material m, int level)
{
  int ambColour = m.getColor(Material.AMBIENT);
  int diffColour = m.getColor(Material.DIFFUSE);
  int emisColour = m.getColor(Material.EMISSIVE);
  int specColour = m.getColor(Material.SPECULAR);
  printlnIndent("Colours: " +  ambColour + ", " + diffColour+", " +
                               emisColour + ", " + specColour+", " +
                               m.getShininess(), level);
  printlnIndent("Colours in Hex: " + Integer.toHexString(ambColour)
                + ", " + Integer.toHexString(diffColour) + ", "
                + Integer.toHexString(emisColour) + ", "
                + Integer.toHexString(specColour), level);
}
```

A glance at Figure 3 shows that the tube's ambient colour is 0x333333, which means that it's red, green, and blue components combine to be nearly black. The ambient, emissive, and specular colours don't use alphas. The tube's diffuse colour is 0xffcccccc, which means no alpha, and a grey combination of red, green, and blue.

The availability of a user ID for the tube's material (ID == 5) permits us to make the tube a more interesting colour when it's loaded into another application:

```
try {
  Object3D[] parts = Loader.load("/tube.m3g");
}
catch (Exception e)
{  e.printStackTrace();  }

Group mGroup = (Group) parts[0];
Material m = (Material) mGroup.find(5);
m.setColor(Material.DIFFUSE, 0xFFC3C3);   // pink
```

Hexadecimals for colours, such as 0xFFC3C3, can be obtained with ColorCalc (http://labrocca.com/colorcalc/): you choose the colour by moving sliders, and it supplies the corresponding hexadecimal.

Figure 10 shows the pink tube in the LoaderM3G application described later:

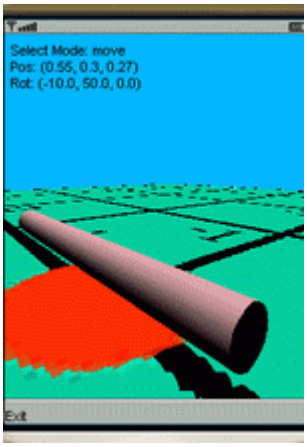

Figure 10. The Tube Turned Pink.

6.3. Extracting the Top-level Group

examineNode() calls storeGroup() whenever it encounters a Group node. If it's the first one, then a reference is stored in modelGroup. If the node isn't the root of the graph, then we attempt to detach it from its parent.

The code utilizes three globals:

```
private Group modelGroup;    // top-level Group node for the model
private boolean foundGroup;  // Group node found during the search?
private boolean isSceneRoot;
                // is found Group node the root of the scene graph?


public ViewModel(String fn)
{
  modelGroup = null;
  foundGroup = false;    // Group node not found yet
  isSceneRoot = true;    // assume it will be the root of the graph
     // more code :
}
```


The foundGroup boolean, which is initially false, is set to true when the first Group node is found. isSceneRoot is set true at the start, but is switched to false if the found Group node isn't the root of the scene graph.

The storeGroup() method:

```
private void storeGroup(Group g, Object3D parent, int level)
{
  if (!foundGroup) {   // if not already found a Group node
    modelGroup = g;
    System.out.println("modelGroup assigned Group with ID " +
            g.getUserID() + "; at level " + level);
    foundGroup = true;
    if (level != 0) {   // the Group is not the root node
      isSceneRoot = false;
      if (parent instanceof Group) {
        Group gPar = (Group) parent;
        gPar.removeChild(modelGroup);
        System.out.println("Detached Group from parent with ID " +
                                    gPar.getUserID() );
      }
      else {
        System.out.println("Could not detach Group; set to null");
        modelGroup = null;   // since cannot attach it to our scene
      }
    }
  }
}  // end of storeGroup()
```

The notion of 'at the root' is implemented by the level integer passed around during the graph traversal. If the node is at the root, then the level value will be 0.

Node detachment requires that the node's parent be passed into storeGroup(), so that Group.removeChild() can be called on it.

6.4. Rotating the Model

If modelGroup isn't the root node then it's probably been created in 3DS Max, and so is probably facing downwards. adjustModelPosition() rotates it to face forwards, and lifts it a little off the floor.

```
private void adjustModelPosition()
{
  if ((modelGroup != null) && (!isSceneRoot)) {
    System.out.println("Rotating model -90 degrees around x-axis");
    System.out.println("Raising the model 0.1 units up y-axis");
    Group mGroup = modelGroup;
    modelGroup = new Group();
    modelGroup.setOrientation(-90.0f, 1.0f, 0, 0);  // rotate
    modelGroup.setTranslation(0.0f, 0.1f, 0.0f);    // raise
    modelGroup.addChild(mGroup);
  }
}
```

adjustModelPosition() is called from the ViewModel constructor, after the scene traversal has finished.

Figure 11 shows the beautiful model from pond_vilkutus2.m3g, created for a Nokia example. No scaling or translation is required when it's loaded into ViewM3G, but the automatic rotation is necessary.

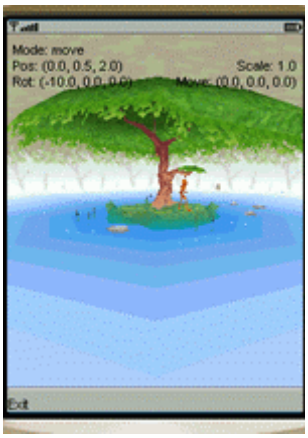

Figure 11. The Model from pond_vilkutus2.m3g

7. Using Models in Other Applications

The principle aim of ViewM3G is to gather information about the model in a M3G file, so it can be more easily used in other applications. The information includes user IDs, and suitable scaling and translation values.

This extra information is utilized in two methods:

```
Group loadRootModel(String fn, float scale,
                    float xMove, float yMove, float zMove);
Group loadSceneModel(String fn, int id, float scale,
                    float xMove, float yMove, float zMove);
```

These methods should be added to an application if model loading is needed. They both return the model's top-level Group node, and the specified scaling and translations are applied to the node.

There are two methods to distinguish between the two kinds of model we may encounter: loadRootModel() manipulates a Group node which is the root of the loaded scene graph. loadSceneModel() handles a Group node in a scene graph with a World root node.

loadRootModel() loads the scene graph, treats the root node as the Group, and applies scaling and translations to it:

```
private Group loadRootModel(String fn, float scale,
                            float xMove, float yMove, float zMove)
{
  System.out.println("Loading model from " + fn);

  Object3D[] parts = null;
  try {
    parts = Loader.load(fn);
  }
```



```
    catch (Exception e)
    {  e.printStackTrace();  }

    Group mGroup = (Group) parts[0];   // use Group node at the root
    mGroup.setTranslation(xMove, yMove, zMove);
    mGroup.scale(scale, scale, scale);
          // translations and scale obtained from ViewM3G
    return mGroup;
  }  // end of loadRootModel()
```

A typical call would be to load a M3G file containing a single model, such as the tube in tube.m3g:

```
Group modelGroup =
    loadRootModel("/tube.m3g", 0.0048f, 0.4f, 0.1f, 0);
Material m = (Material) modelGroup.find(5);
m.setColor(Material.DIFFUSE, 0xFFC3C3);   // make the material pink
```

The scale factor (0.0048), and the (x,y,z) translations (0.4, 0.1, 0) were taken from the ViewM3G display. We also use the Material node's user ID value (obtained from the model's examination) to make the tube pink.

One weakness of loadRootModel() is the assumption that the Group node is the first element in the array returned by Loader.load(). This could be improved by the user supplying a position index as an argument when the node isn't in the first array cell.

loadSceneModel() loads a model that's beneath the root in the M3G file. The model's top-level Group node is detached from its parent, and rotated (since it's probably a 3DS Max creation). Then any additional scaling and translations are applied.

```
private Group loadSceneModel(String fn, int id, float scale,
                             float xMove, float yMove, float zMove)
{
  System.out.println("Loading model from " + fn);

  Object3D[] parts = null;
  try {
    parts = Loader.load(fn);
  }
  catch (Exception e)
  {  e.printStackTrace();  }

  Group root = (Group) parts[0];
  Group mGroup = (Group) root.find(id); //use Group with user id

  root.removeChild(mGroup);
       // we're assuming that mGroup is directly below the root

  // reposition the non-root model
  // translations and scaling obtained from ViewM3G
  Group posGroup = new Group();
  posGroup.setOrientation(-90.0f, 1.0f, 0, 0);
         // rotate -90 around x-axis
  posGroup.setTranslation(xMove, yMove+0.1f, zMove);
        // translation + small lift
  posGroup.scale(scale, scale, scale);
```



```
    posGroup.addChild(mGroup);

    return posGroup;
  }  // end of loadSceneModel()
```

loadSceneModel() takes an additional input argument compared to loadRootModel(): the user ID of the Group node. This makes it possible to find the node without recursively searching over the graph, but it means that the node must have been assigned an ID by the model creator.

The scaling, translation, and rotation of the model is done through a new Group node, posGroup. This is to avoid changing the model's Group node, which may contain it's own transformations.

In ViewM3G, the rotation of the Group is accompanied by a small lift (0.1 units), so the model is a raised above the XZ plane. In loadSceneModel(), this value is added to the translation applied with Transformable.setTranslation().

A typical call to loadSceneModel() would be to load a model from a M3G file containing a complete World-base scene graph, such as the Nokia pond example:

```
Group modelGroup = loadSceneModel("/pond_vilkutus2.m3g",
                                  421721094, 1.0f, 0, 0, 0);
```

The Group's ID (421721094) was found by looking at the examination output from ViewM3G. No scaling or translation are required.

There's a few shortcomings with this simple method. One of them is that I've assumed there's only one root node returned by Loader.load(). Another is that I've trusted that the Group node is always directly underneath the root, so the root is it's parent. These issues can be fixed, but would overly complicate the code.

8. LoaderM3G

The loadRootModel() and loadSceneModel() methods can be seen in action in the LoaderM3G application. It offers the usual 3D world: a floor covered with a green grid, a blue background, lighting, and a mobile camera. However, it uses one of our load methods to load the desired model.

Figure 12 shows the Nokia 'ant on ice' model, from otokka_jump2.m3g. The model is the ant, the ice floor, and hill in the background. The green grid is still present, but hidden underneath the ice. The left hand screenshot shows the initial view, and the right hand view is after the camera has been moved, rotated, and raised a tad.

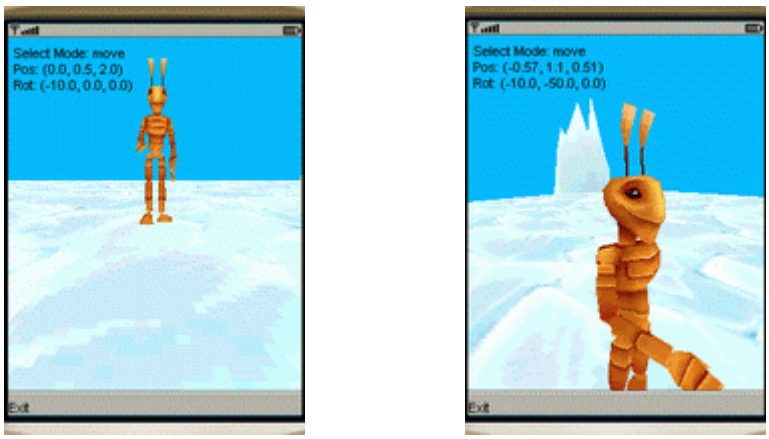

Figure 12. An Ant on Ice in LoaderM3G.

The camera in LoaderM3G isn't the extended version used in ViewM3G: it only translates, rotates, and 'floats' the user's viewpoint.

The class diagrams for LoaderM3G are given in Figure 13.

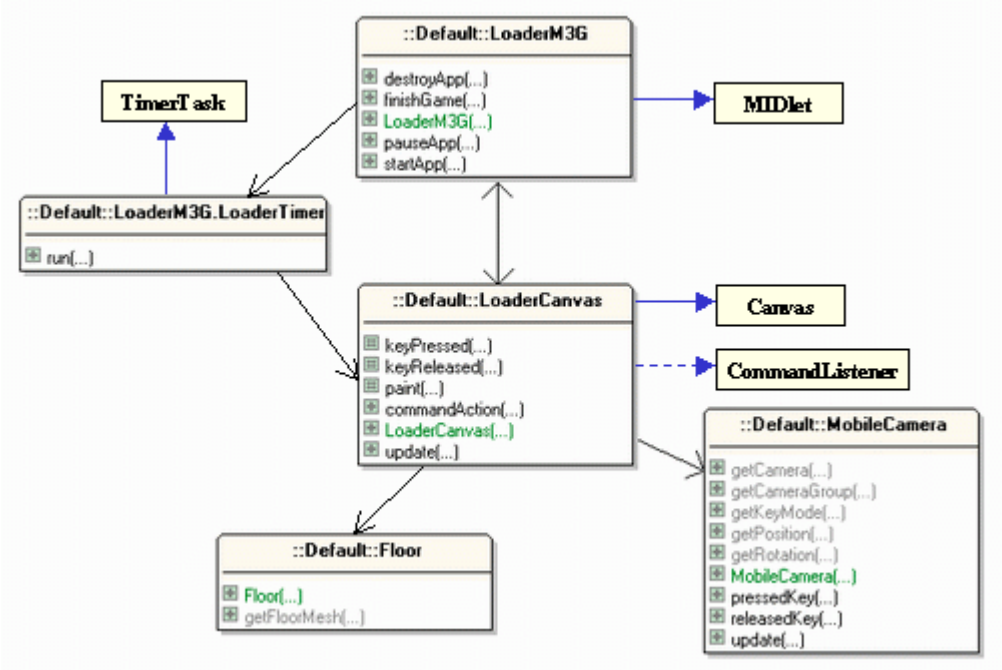

Figure 13. Class Diagrams for LoaderM3G.

The essential point about LoaderM3G is that it's a copy of MorphM3G from M3G chapter 3, but without the morphing model. The only change comes in LoaderCanvas, when it's time to load the model. buildScene() is defined as:

```
private void buildScene()
// add elements to the scene
{
  addCamera();
  addLights();
  addBackground();
  addModel();
  addFloor();
}
```

The new method is addModel(), which wraps up a call to loadRootModel() or loadSceneModel() depending on the choice of file.

```
private void addModel()
{
  Group modelGroup = null;

  // example from Leon HongKong
  // modelGroup = loadRootModel("/tube.m3g", 0.0048f,0.4f,0.1f, 0);
  // Material m = (Material) modelGroup.find(5);
  // m.setColor(Material.DIFFUSE, 0xFFC3C3);   // pink

  // examples from WTK 2.2
  // modelGroup = loadSceneModel("/pogoroo.m3g", 421721094,
  //                                             1.0f, 0, 0, 0);
  // modelGroup = loadSceneModel("/skaterboy.m3g", 421721094,
  //                                     0.2897f, -0.19f, 0, 0);

  // examples from Nokia "3-D Game Development on JSR-184"
  // modelGroup = loadSceneModel("/pond_vilkutus2.m3g", 421721094,
  //                                             1.0f, 0, 0, 0);

  modelGroup = loadSceneModel("/otokka_jump2.m3g", 421721094,
                                            0.424f, 0, 0, 0);

  // modelGroup = loadSceneModel("/nokia_on_ice.m3g", 421721094,
  //                                         0.1635f, 0, 1.1f, 0);
  // modelGroup = loadSceneModel("/tunnel.m3g", 421721094,
  //                                         0.0431f, 0, 0, 0);

  scene.addChild( modelGroup );  // add the model
}
```

There's no sophisticated mechanism for choosing between the load methods; it's up to the programmer to call the correct one based on his/her analysis of ViewM3G's output.

loadRootModel() and loadSceneModel() are private methods in the LoaderCanvas class, implemented in the way described earlier.

