4

Java\textsuperscript{TM} Mobile 3D Graphics

4.1 M3G

This section introduces the Mobile 3D Graphics API, M3G also known as JSR-184\textsuperscript{1} \cite{40}.

Even if a 3D Graphics API for Java already exists (JAVA3D), most mobile devices have limited memory and processor power; thus Java 3D is unsuitable for them. Therefore, a proposal for a more suitable API was put together by a group of experts. The need was for a scalable, small-footprint, interactive 3D API for mobile devices that could work as an optional package for J2ME\textsuperscript{TM} to allow 3D graphics. Java Platform, Micro Edition or Java ME (formerly referred to as Java 2 Platform, Micro Edition or J2ME), is a collection of Java API for the development of software for resource constrained-devices such as PDAs, cell phones, and other consumer appliances. M3G is a software package for providing 3D graphic functionalities to a wide range of devices(Figure 4.1).

M3G is designed to be a 3D API suitable for the J2ME platform and CLDC \cite{41}/MIDP \cite{42}.

Since it uses floats, it cannot be implemented on top of CLDC 1.0 but must be implemented on at least version 1.1 of CLDC. The Connected Limited Device Configuration (CLDC) is a specification of a framework for Java ME applications targeted at devices with very limited resources such as pagers and mobile phones. It could possibly be implemented on MIDP\textsuperscript{2} 1.0, but most devices supporting M3G will likely also support MIDP 2.0 \cite{37}. It is integrated with components of MIDP to allow efficient rendering to its Image and Canvas classes.

\textsuperscript{1} Java\textsuperscript{TM} Specification Requests (JSRs) are formal documents that describe proposed specifications and technologies to be added to the Java platform.

\textsuperscript{2} Mobile Information Device Profile (MIDP) is a specification published for the use of Java on embedded devices.
We consider, now, the question about a need for a new mobile 3D Graphics standard when OpenGL ES is already available.

OpenGL ES is a low-level API standard, since it’s based upon OpenGL; in fact, even for building simple 3D scenes it requires developing many lines of code, while there’s a need for having a compact version of the final application. M3G is a high-level library designed to be compatible with OpenGL ES API. It is not a competitor, but it’s more a complement to the OpenGL ES API set. This design choice has many advantages:

- Enhances developer’s productivity.
- Minimizes code size of graphics applications.
- Increases applications’ performance.

JSR-184 must not be mixed up with Java 3D standard API, which extends 3D capabilities to standard Java applications. Java 3D was designed for desktop computers and its completely unsuited for M3G. Many solutions and technicalities used in Java 3D has been modified and reused in JSR-184, as for example the support for a scene graph, which is used for representing in a compact and hierarchical structure all the elements that are part of a 3D scene. The scene graph represents a tree structure and includes definitions of each kind of physical or abstract object in the 3D world (cameras, lights, animations, etc.). The root of this scene tree is in fact represented by a World node object.
Moreover, JSR-184 specifications describe a new standard file format (.m3g), used for including all data related to a specific scene (the scene graph) and loading these data in applications coded to support the M3G standard. In this way, the scene data, including animations, can be created by using common 3D modeling programs (Maya, 3D Studio, ...) available on the market. These models can then be saved in M3G format and imported in a M3G application program that, by using few lines of code, can visualize and animate the imported scene. The product life cycle is thus tremendously accelerated by a clear separation between graphics design and code development of applications. In fact, graphic artists can create their own look and feel for the scene, including animations, and then export them as an M3G file to application developers.

4.2 MIDP Applications

The applications supporting the M3G standard are MIDP applications, and they are also known as MIDlets. The MIDP specification defines the minimum hardware, software, and network requirements for an application to run on a certain kind of device. MIDP applications are co-resident with other applications and executed in the Mobile Information Device (MID) framework, as shown in Figure 4.2.
We describe now some basic concepts useful for designing and implementing a MIDP application. A life cycle of a MIDlet application is shown in Figure 4.3.

![MIDlet life cycle diagram](image)

**Fig. 4.3.** MIDlet life cycle.

*Application Management Software* (AMS) is an environment where a MIDlet is installed, executed, stopped, and uninstalled. AMS creates every new instance of a MIDlet and manages its status during the execution process. A generic MIDlet can be in one of the following statuses: *Paused, Active, Destroyed*. When created and initialized, it is in the *Paused* status; if there is an exception raised by MIDlet constructor, it goes into *Destroyed* status. MIDlet goes into the *Active* status when a call of the `startAPP()` method is completed.

Figure 4.3 shows the statuses of a MIDlet and the functions that manage transitions from one status to another.

The user interface in MIDP applications is built by using two API functions. The first API, which is a low-level function, is extended from the abstract class *Canvas*, and the second, which is a high-level function, uses *Alert, Form, List, and TextBox* classes, extended from a *Screen* abstract class as shown in Figure 4.4.

High-level API classes are designed to provide portability among software components on different MIDs. The *Canvas* class allows applications to have direct control over the user interface but delegates to programmers porta-
bility implementation among MIDs interfaces (display size, supported colors number, different kinds of keyboards, etc.).

Usually, one or more MIDlets are packed in a JAR file called a MIDlet suite, which is then used by AMS. MIDlets included in the same suite share the same execution environment (virtual machine) and thus can interact among one another. Every MIDlet suite can be associated with an application descriptor used for describing its content. The descriptor file extension should be .jad. It is used for managing MIDlets and storing application configuration properties. These properties can be modified in a JAD file without changing the associated JAR. MIDP specification provides detailed information on building and developing application descriptors and their attributes.

In the latest versions of MIDP standard, there are some interesting new features:

- HTTP secure protocol support (HTTPS)
- Enhanced network management
- Support for application distribution
- New graphical user interface (GUI) components
- Gaming support
- Multimedia functions support (audio and animations)
- A security model (trusted MIDlets)

The main enhancement is a package entirely dedicated to gaming development for J2ME framework, called javax.microedition.lcdui.game. It includes
several classes that enable game developing for mobile devices [43]. In par-
icular, a *GameCanvas* class can be used in conjunction with an M3G standard
on devices supporting MIDP version 2.0.

### 4.3 Immediate and Retained Mode

The main class for drawing a scene with M3G standard is the *Graphics3D*
class. It is defined as a singleton ³ and a unique instance can be accessed via
the `getInstance()` method. To draw a scene, it is necessary to link a *Graphics3D*
instance to a *target* object, draw the scene by an appropriate method, and
release the *target*, as shown in the following snippet of code.

```java
Graphics3D g3d = Graphics3D.getInstance();
World = world; ...
Graphics g = ...;
boolean bound = false;
try {
    g3d.bindTarget(g);
    bound = true;
    g3d.render(world);
}
finally {
    if(bound) g3d.releaseTarget();
}
```

A target object is a common *Graphics* object, the same as used in the
`paint()` method with a *Canvas* or a *GameCanvas* class.

Graphics3D can also draw on top of an Image2D object. In this way a
developer can draw a three-dimensional scene and use it as texture. Note that
target objects must be released after using them; otherwise a unique instance
of Graphics3D cannot be linked to other objects, and buffers cannot be sent
to the screen for visualization.

Graphics3D supports two different drawing modalities:

1. **Immediate mode**
   - This is a low-level modality that allows defining each detail of a draw-
     ing process.
   - It draws an individual node, a group of nodes, or a submesh in a scene
     graph.
   - Cameras, lights, and background are managed separately.

2. **Retained mode**

³ A one-time instantiated class with a unique point of access.
• This mode hides low-level details by loading and visualizing three-dimensional scenes by means of a few lines of code.
• It directly draws the World object, at the root of a scene graph.
• It manages cameras, lights, and background by accessing them directly with a World object.

The retained mode allows developers to use already-made, complex, three-dimensional models; for instance, a developer can easily manage a scene graph in order to build a car model. Nodes representing wheels can rotate around their axes and are constrained to be parallel with respect to the car body orientation. All this information can be used by specifying it during modeling as additional information to nodes. The retained mode simplifies 3D world design by hiding low-level technical details from developers.

The overall control of a 3D scene can be obtained only by using low-level functions, and by accessing the graphics pipeline, and thus, for this reason, JSR-184 supports also the immediate mode, where drawing functions could be invoked on single objects. Moreover, the retained mode can take advantage of graphics acceleration because it is built on low-level immediate mode functions. Both modalities can be used in conjunction with each other, allowing developers to balance drawing performance with resources by choosing the appropriate modality with respect to their target.

4.4 Scene Graph

The retained mode uses a scene graph for linking all geometric objects in a three-dimensional world made of a tree structure. Each node of the graph represents a geometric object and contains information on appearance, \(^4\) positioning in space, and function with respect to other nodes.

To build a 3D world, objects are used as subclasses of the Node base class. Then the Group class contains many objects, and the World class is a special case of the group class that includes all nodes in a scene. A World node is root of the scene graph and it is different from a regular node, in that all specified transformations are ignored during scene rendering.

A 3D world can be created from scratch, and new nodes can be linked after that, but a more convenient procedure is storing a scene in an .m3g file and then loading that scene to manage it in a scene graph. A complete and basic scene graph includes at least a World object and a Camera object. Figure 4.5 shows a generic scene graph describing different node characteristics.

It also possible to share components among different nodes of a scene graph, thus reducing memory usage. There are, anyway, some exceptions:

• Nodes can belong only to one group.
• Cycles among nodes are not allowed.

\(^4\) It includes all geometric information concerning the node.
To uniquely identify and address an object within a scene graph, a field of Object3D class is used, the userID. Each node in a tree holds its own userID, while World object has 0 by default; all other objects can have arbitrary values. It is important to observe that userID values are not unique within a tree, and so different nodes could have the same userID.

To build a scene graph from an .m3g file, the Loader class is invoked; it manages object extraction from files and builds all necessary classes. It creates all animation controllers, and it initializes the tree structure and all group of nodes, lights, and cameras. All these functionalities are included in a single load() method. More specifically, all classes stored in the .m3g file are deserialized in a vector containing Object3D objects returned as the result of a method call.

For example, in order to use a first element of an M3G file called test.m3g, as an instance of a World class, we need to write the following code:
Object3D [] o = null;

try {
    o = Loader.load("test.m3g");
}

catch (Exception e) {}

World loaderWorld = (World) o[0];

This class can also load many types of image formats, such as PNG, and in this case the result of the load method would be an Image2D object.

4.5 Transformations

The abstract class Transformable defines all geometric transformations that can be applied on nodes. There are four types of transformations:

- Translations (T)
- Rotations (R)
- Nonuniform scale (S).
- Generic nonhomogeneous matrix $4 \times 4$ (M)

Given a point in the space $p = (x, y, z, w)$, representing a vertex coordinate or a texture coordinate, its transformation can be defined (with respect to a coordinate system) as follows:

$$p' = TRSM \times p$$

A Transformable class defines methods for setting these components, also individually, as for example with the methods setTranslation() or setScale().

4.6 Nodes of the Scene Graph

The node class is an abstract class representing all kinds of nodes included in a scene graph, such as: lights, cameras, meshes, sprites, and groups. A node defines a local coordinate system that can be transformed with respect to its ancestor coordinates system. Nodes can also be lined up with other nodes or point to a reference node; in this way we can force, for instance, a light or a camera node to point to a fixed object.

Another interesting characteristic of a node is the ScopeID parameter. This field is used for setting the visibility levels of a node, and in general is used for computing the visibility of a set of objects. Many different kinds of
masks can be defined for the visibility of parts of a scene and to modify the scope of the camera in order to match the parts of a scene that are visible. If the scope of a camera and the nodes do not match, the nodes aren’t drawn on the screen, thus saving resources especially at the computational level.

Moreover, this parameter can be used for speeding up computations on lighting. Usually, in a three-dimensional environment, the lights have a certain radius, determined by the type of light and its intensity. By setting different scopes for lights and objects corresponding to their distances, it can be computed if a light has effects on that object or not. This allows the use of many different light sources in the same scene without affecting the speed of performances and saving computational resources.

A set of nodes can be grouped together by using a *Group* class. Grouping different objects can help in the case of managing different objects with the same kind of operations. A typical group example is a car model with four wheels. In fact, by defining a car as a group of nodes, it is possible to move the whole car without moving each wheel individually.

### 4.7 Camera Class

A camera class is represented by a node in a scene graph, which sets the position of observers in the scene and the projection of a 3D perspective on a two-dimensional display.

The camera is generally pointing toward negative values of the z axis. It can be positioned and oriented in the same way as other nodes, namely by using transformations available at each node. It uses classical projections and clipping rules that apply for *OpenGL*, with the exception of the user-defined clipping planes, which are not supported. It is, instead, possible to define many cameras, and thus it is possible to have many different viewpoints.

### 4.8 Managing Illumination

The *JSR-184* specification supports four kinds of lights, each having different computational complexity and thus performance. The equations used for light computation are directly imported from the *OpenGL* standard ones. Light types are:

- **Ambient light**: defines the general intensity of objects in a scene. Ambient lights illuminate a scene with the same illumination quantity; thus position and direction are ignored during computations.
- **Directional light**: defines only the source direction of light. Position or distance from an object has no effect on the latter, even if it can be set anywhere in the scene.
- **Omni light**: defines a light source point. The omni lights affect objects in each direction. A curve can be set to adjust the intensity variable according to the distance from objects.
- **Spot light**: defines the position, direction, and radius of a light cone. This light doesn’t have any effect on objects out of its light cone.

The computations needed to manage a light require a considerable amount of CPU time. It is thus crucial to choose the right kind of light related to the scope of a scene and to avoid putting lights on every object by using a *scope node* and thus saving computational performance. Every light has a color determined by the RGB components and has an intensity value, but the exact effect of light hitting a surface is also function of that surface’s material.

### 4.9 Meshes and Sprites

A mesh object is a node in a scene graph that represents a three-dimensional 3D object specified by a set of polygons. The object itself is made of several sub-meshes, each having its own appearance, as shown in Figure 4.6.

![Mesh object structure](image)

**Fig. 4.6.** A mesh object structure.

A submesh is an array of *triangle strips* defined by an *IndexBuffer* object. *Triangle strips* are made by indexing vertex coordinates and other attributes
of a *VertexBuffer* associated with an *IndexBuffer*. *VertexBuffer* contains information about vertex positions, normals, and texture coordinates. Each submesh in a mesh shares the same *VertexBuffer*.

The components of an appearance object are:

- **Material**, which defines the colors to be used in lighting computations.
- **CompositingMode**, which allows per-pixel composition attributes such as transparencies and z-buffer.
- **PolygonMode**, which contains attributes at the polygonal level including settings for face visibility (back and front) and perspective corrections.
- **Fog**, which contains all attributes for setting a fog effect.
- **Texture2D**, which incorporates all 2D images and attributes for specifying how an image can match the related submeshes.

The *mesh* class also includes two subclasses used for managing dynamic meshes, which can change their shapes according to certain parameters: *MorphingMesh* and *SkinnedMesh*.

An object of *MorphingMesh* type is equivalent to an ordinary mesh, except that its vertices are drawn and computed as a weighted linear combination. It is a combination of a *VertexBuffer* and *VertexBuffers*, which are targets of the morphing operation. All target *VertexBuffer*s, also called *morph targets*, include the same properties: the same number of vertices for each array, the same number of components per vertices, and the same component size.

By denoting a base mesh by \(B\), morph targets by \(T_i\), and weights for each morph target by \(w_i\), a resulting mesh can be represented by the following equation:

\[
R = B + \sum_i w_i(T_i - B)
\]

Morphing can be computed on every vertex attribute:

- Vertex positions
- Colors
- Normals
- Texture coordinates

The *SkinnedMesh* class represents a skeleton animated polygonal mesh. In contrast with a normal mesh class, it includes a skeleton structure. The skeleton is built by means of a hierarchical structure, by using scene graph nodes. Each node belonging to a skeleton represents a *bone*, which is a transformation. Each vertex can be linked to one or more skeleton bones. In this way a mesh is extended and linked to a structure that can manage it.
The *Sprite3D* class represents a 2D image with a position in three-dimensional space. The structure of a Sprite3D object is shown in Figure 4.7.

Images are stored in *Image2D* objects. Their *appearance* contains attributes for fog and composite effects. There are two modalities for appearance:

- *Scaled mode*, in which the width and height of a sprite on the screen are computed, as it is a rectangle with one unit thick and based on the *XY* plane centered in its local coordinate system origin.
- *Unscaled mode*, in which the width and height of a sprite are measured in pixels and are equal to a rectangle defined by setting its size.

### 4.10 Animations

Each object extended from a basic *Object3D* class can be animated. The most relevant classes for managing animations are:

- *KeyFrameSequence*
- *AnimationController*
- *AnimationTrack*

*KeyFrameSequence* contains all animation data as a time sequence of values called *keyframes*. A *keyframe* represents a value of an attribute at a certain instant of time. It contains a vector of components, specified by its constructor, which has the same size for each *keyframe* in a sequence. Since *keyframe*
values can be distant in time, interpolation functions are provided to manage them.

A KeyFrameSequence object can be associated with different animation targets by using an AnimationTrack class. It associates a KeyFrameSequence with an AnimationController object and a property that can be animated. This kind of property consists of a scalar value or a variable vector that can be updated by an animation system. An example of a property that can be animated is the orientation of a node. Animated properties are identified by a symbolic constant, and sometimes they’re related only to a restricted class of values, like the SHINESS property of a Material object.

Classes derived from Object3D include one or more animated properties. An Object3D with animated properties is called an animated object. Each animated property of an animated object is an animation target. Each animated object can include references to zero or more AnimationTracks, which are activated by their related AnimationControllers.

An AnimationController manages the position and speed of an animation sequence. An animation sequence can be defined as a set of AnimationTracks managed by a single AnimationController. Each AnimationTrack contains all the data needed to manage an animated property on an animated object. By using an AnimationController, operations like pausing, stopping, playing, and speeding-up an animation sequence are available.

Figure 4.8 shows the classes related to the animation process.

Fig. 4.8. Classes related to the animation process.
4.11 Ray Intersections

The \textit{RayIntersection} class represents an infinite line starting from an origin (in a coordinate reference system) and pointing in a fixed direction. It is used for storing references to all \textit{Mesh} or \textit{Sprite3D} objects intersected by this line. Not only intersections but also distances between this line and intersected objects are stored. The \textit{RayIntersection} object is created at run time and cannot be loaded by the \textit{Loader()} class. It is used in conjunction with the \textit{pick()} method of a \textit{Group} class. This method returns information on this first object in a group intersected by the line passed as a parameter to this method. Information on intersected objects is then returned by the \textit{RayIntersection} object. This class is used to manage collisions among objects or to simulate, for example, a gun-shot hitting a target placed at a fixed distance.

4.12 Building an M3G Demo

In this section we explore how to use some high-level classes provided by M3G API to create a simple demo program with a car model moving on the screen and avoiding obstacles by a collision detection method.

First we need to install a \textit{Sun Java Wireless Toolkit}, and create a new project by using a \textit{Ktoolbar} interface. This will help in understanding applications of the described concepts and classes.
4.12 Building an M3G Demo

Fig. 4.9. A snapshot of a wireless toolkit interface.

After defining both the project and MIDlet name, we need to choose a CLDC 1.1 configuration, which supports float data type, and choose Mobile 3D Graphics for J2ME (jsr 184) as additional API.

Recall that Figure 4.3 showed the life cycle of a MIDlet and methods used for changing MIDlet status, which will be used below.

The example of source code for a MIDlet includes not only a declaration of these methods, but also some functions for timing animations.

```java
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import java.io.*;

public class CarDemo extends MIDlet {
    private static final int PERIOD = 50;  // in ms
    private Timer timer;
    private CarDemoCanvas canvas = null;
```
```java
private Display display;

public void pauseApp() {}

public void destroyApp(boolean b) {}

public Display getDisplay()
{
    return display;
}

public void startApp()
{
    // check whether m3g is supported or not
    String version = System.getProperty("microedition.m3g.version");
    if (version == null) {
        finishGame();
    }
    else {
        display = Display.getDisplay(this);
        canvas = new CarDemoCanvas(this);
        timer = new Timer();

        display.setCurrent(canvas);
        timer.schedule(new AnimTimer(), 0, PERIOD);
    }
}

public void finishGame()
{
    timer.cancel();  // stop the timer
    notifyDestroyed();
}
```

Inside the `StartApp()` method a `microedition` version property is checked to indicate if the device supports additional `m3g` API. In case of a positive answer, the `CarDemoCanvas` class is initialized as the core class of our demo application.

In the same source code must be inserted a class called `Animtimer` that is used for managing animations. This class contains a `run` method, which is in charge of updating animations. To set the timing for animations, a `Timer` class has been used.
// Class for managing timing

class AnimTimer extends TimerTask
{
    public void run()
    {
        if (canvas != null)
            canvas.update();
    }
}

4.12.1 DemoCarCanvas Class

The DemoCarCanvas class extends the Canvas class and implements a CommandListener for managing the code to be executed in response to the EXIT and BACK events generated by the device user’s interface.

The following snippet of code is a class constructor.

public CarDemoCanvas(CarDemo carDemo)
{
    this.carDemo = carDemo;

    exitCmd = new Command("Exit", Command.EXIT, 0);
    addCommand(exitCmd);
    setCommandListener(this);

    g3d = Graphics3D.getInstance();

    width = getWidth();
    height = getHeight();

    scene = new World();
    createScene();

    // start the animation
    nextTimeToAnimate = scene.animate(appTime);
}
First a reference to the MIDlet CarDemo is stored, and then the \textit{EXIT} command is set to allow users to close the application by pressing a device key.

After the \textit{Graphics3D} object is instantiated, it represents a 3D graphics context and provides a method for scene drawing, called \textit{render}.

The device screen height and width are set, and after the \textit{scene} object is created (the type of this object is \textit{World}), it will contain all the three-dimensional scene objects (lights, cameras, and meshes).

Finally the \textit{createScene} method is invoked for creating and setting all objects included in the scene.

```java
private void createScene() {
    createCar();
    createCamera();
    createLight();
    createBackground();
    createFloor();
    createCone();
}
```

In \textit{CreateScene}, many methods are called, one for each object included in the three-dimensional scene.

In the scene we include:

- One camera (normCamera)
- Two lights (light and light2)
- One background (background)
- Three meshes for visualizing a car, some cones (alias the obstacles), and a floor

To load the mesh models in a three-dimensional scene, a technique has been developed by Andrew Davison \cite{Davison} which converts a \textit{wavefront OBJ} model into a Java class containing M3G code. We thus generated three different classes, Car, Floor, and Cone, including all visualization code for these three models.

A \textit{createCamera} class sets up a simple camera, and we will use some transformations on this camera (mainly two 90-degree rotations with respect to the \textit{x} and \textit{y} axis) for visualizing the scene from the right perspective. To have a better perspective, there is also a \textit{setOrientation} method, which slides the scene down a little bit.

Finally a camera is added to the \textit{scene} object and set as active.

```java
private void createCamera() {
    float aspectRatio = (float) width / (float) height;
    ```
// normCamera
normCamera = new Camera();
normCamera.setPerspective(60.f, aspectRatio, 1.0f, 100000.f);

// camera transformations
Transform normCameraTransform = new Transform();
normCameraTransform.postRotate(90, 1f, 0f, 0f);
normCameraTransform.postRotate(90, 0f, 1f, 0f);
normCameraTransform.postTranslate(0f, 0f, 200.0f);
normCamera.setTransform(normCameraTransform);

// angles downward slightly
normCamera.setOrientation(-50.0f, 0f, 1f, 0f);

scene.addChild(normCamera);
scene.setActiveCamera(normCamera);
}

Code that manages the lights also sets and uses the methods and properties of M3G API as shown below.

private void createLight() {

    // 1 omni light (ahead)
    Light light = new Light();
    light.setColor(0xffffff);
    light.setIntensity(1.0f);
    light.setMode(Light.OMNI);
    light.setTranslation(0, 0, 100);

    // 1 omni light (behind)
    Light light2 = new Light();
    light2.setMode(Light.OMNI);
    light2.setIntensity(1.0f);
    light2.setTranslation(100, 100, 100);

    scene.addChild(light);
    scene.addChild(light2);
}

Very similar to the method above is the createBackground method coded as follows:
private void createBackground () {
    Background background = new Background ();
    background . setColor (0x004080C0);
    scene . setBackground (background );
}

This method creates a background object, setting it to a light blue color. The color format is managed by M3G as 0xAARRGGBB, where:

- A stands for the alpha channel
- R stands for the red channel
- G stands for the green channel
- B stands for the blue channel

The remaining methods, createCar, createCone, and createFloor, use their respective classes: Cone, Floor, and Car.

private void createFloor () {
    Image2D floorIm = loadImage ("/piano.png");
    Plane plane = new Plane ( floorIm , 1000,1000);
    scene . addChild ( plane . getPlaneMesh () );
}

private void createCone () {
    Cone cone = new Cone ();
    Cone cone1 = new Cone ();
    cone . getMesh () . setTranslation (100,130,30f);
    cone1 . getMesh () . setTranslation (−150,−100,30f);
    scene . addChild ( cone . getMesh () );
    scene . addChild ( cone1 . getMesh () );
}

The createFloor and createCone classes, respectively, build a plane and two cones (obstacles), similar to street cones positioned in the scene.

The Plane class creates a 2D plane, and its constructor takes three parameters as input:
• An `Image2D` object for managing textures
• An integer representing x axis elongation
• An integer representing y axis elongation

All images are loaded by the `loadImage` method, which performs some consistency checks before returning an `Image2D` object as result.

```java
private Image2D loadImage(String fn) {
    Image2D im = null;
    try {
        im = (Image2D)Loader.load(fn)[0];
    } catch (Exception e) {
        System.out.println("Cannot make image from "+fn);
    }
    return im;
}
```

The `createCar` method does not contain a `Car` class constructor, since a `car` object is created during the initialization phase; it contains instead methods for managing collisions among car and other objects (street cones) in the scene:

• The `setScene` method takes a World object pointer to use it with the `Car` class.
• The `setPickingEnable` method enables/disables a car mesh during collision computations.

```java
private void createCar() {
    // Disable collision detection for car mesh
    car.getMesh().setPickingEnable(false);

    car.setScene(scene);
    carGroup = car.getCarGroup();
    scene.addChild(carGroup);
}
```

The `update` method is invoked for updating the car position on the screen after checking for collisions by using the `updateCar` method of `Car` class. Once a car is positioned, the screen is repainted by the `repaint` method.
public void update() {
    appTime++;
    if (appTime >= next_time_to_animate) {
        next_time_to_animate = scene.animate(appTime) + appTime;
        System.out.println("next time to animate: \
        + next_time_to_animate);
    }
    car.updateCar();
    posCar = car.getPosition();
    normCamera.setTranslation(posCar[0], posCar[1], posCar[2]);
    repaint();
}

The paint method is in charge of drawing the final three-dimensional scene. As already mentioned, the only method provided by the M3G standard for drawing is the render method; it can be called after linking a graphics context to a canvas.

We also display on the screen the car speed (top left side of the screen).

protected void paint(Graphics g) {
    g3d.bindTarget(g);
    g3d.render(scene);
    g3d.releaseTarget();
    g.drawString("Speed: \
    + car.getSpeed(), 5, 5, Graphics.TOP | Graphics.LEFT);
}

The last two methods of the CarDemoCanvas class are used for managing the keyboard.

protected void keyPressed(int keyCode) {
    int gameAction = getGameAction(keyCode);
    car.pressedKey(gameAction);
4.12 Building an M3G Demo

protected void keyReleased(int keyCode) {
    int gameAction = getGameAction(keyCode);
    car.releasedKey(gameAction);
}

Both methods check which key has been pressed, store it in a gameAction variable, and pass it to Car class methods for updating the car position.

4.12.2 Car Class

Car class not only contains code for visualizing a car model (i.e., Floor and Clone classes) but also has methods for animating cars, and manages (by means of the RayIntersect class) a basic collision detection algorithm.

The car class constructor takes an Image2D parameter for textures. It also is in charge of building a car model and managing a group of transformations (trans) for positioning a car in the scene. The model is then linked to a group; in this way if we modify a transformation each of the children nodes is affected by this change.

```java
public Car(Image2D img) {
    this.scene = scene;
    model = makeModel(img);
    transGroup = new Group();
    trans.postTranslate(X_POS, Y_POS, Z_POS);
    transGroup.setTransform(trans);
    transGroup.addChild(model);
}
```

The storePosition method extracts the current car position from the transformations group.

```java
private void storePosition() {
    // extract the current (x,y,z) position from transGroup
    transGroup.getCompositeTransform(trans);
    trans.get(transMat);
    xCoord = transMat[3];
}
```
The `transMat` object represents a $4 \times 4$ float matrix:

```java
private float[] transMat = new float[16];
```

The methods used for managing position and direction of the car are:

- `updateMove`, which takes a transformation parameter (`trans`), performs all checks for collision detection, and moves the car according to a space attribute computed by the current speed of the car.
- `updateRotation`, which by means of key pressed (left and right arrow keys) rotates the car (using a `Transform` object called `rotTrans`).

Both these methods are invoked by the `updateCar` method, which manages the following:

- Increasing speed until the up arrow key is released, and decreasing speed when holding down the arrow key (or releasing both keys)
- Executing the `updateMove` method
- Executing the `updateRotation` method only if the left or right arrow keys are pressed

```java
public void updateCar() {
    if (upPressed) {
        if(speed<MAX_SPEED) speed+=2f;
    }
    if(downPressed) {
        if(speed>0.0f) speed-=4f;
        if(speed<0.0f) speed=0.0f;
    }
    if(!upPressed && !downPressed) {
        if(speed>0.0f) speed-=2f;
        if(speed<0.0f) speed=0.0f;
    }
    updateMove();
    if (leftPressed || rightPressed) 
```
updateRotation();
else if ( !leftPressed && !rightPressed )
    angle = 2.0f;
}

All necessary attributes for managing key pressing (Boolean) are updated by the pressedKey and releasedKey methods:

```java
public void pressedKey(int gameAction) {
    switch (gameAction) {
        case Canvas.UP: upPressed = true; break;
        case Canvas.DOWN: downPressed = true; break;
        case Canvas.LEFT: leftPressed = true; break;
        case Canvas.RIGHT: rightPressed = true; break;
        default: break;
    }
}
```

```java
public void releasedKey(int gameAction) {
    switch (gameAction) {
        case Canvas.UP: upPressed = false; break;
        case Canvas.DOWN: downPressed = false; break;
        case Canvas.LEFT: leftPressed = false; break;
        case Canvas.RIGHT: rightPressed = false; break;
        default: break;
    }
}
```

We now describe the updateRotation method as coded below.

```java
private void updateRotation() {
    if (angle < MAX_ANGLE) angle += 1.0f;
    if (leftPressed) { // rotate left around
        // the z-axis
        rotTrans.postRotate(angle, 0, 0, 1.0f);
        zAngle += angle;
    } else { // rotate right around the z-axis
        rotTrans.postRotate(-angle, 0, 0, 1.0f);
        zAngle -= angle;
    }
```
Rotation takes place on the axis by changing the \textit{zAngle} attribute, which is increased or decreased by pressing the appropriate key. It is important to store the rotation status in the \textit{Transform rotTrans} object for keeping the information useful for the car direction vector.

The \textit{getDirection} method computes the direction of the car.

```java
public float[] getDirection() {
    // zVec contains the initial direction of the car
    float[] zVec = {-1, 0, 0, 0};

    // the exact direction is given after computing the applied rotations
    rotTrans.transform(zVec);

    return new float[] { zVec[0], zVec[1], zVec[2] };
}
```

The car direction is obtained by computing all the applied rotations. Car direction is used for computing collision detection in the \textit{updateMove} method.

Collisions are managed by using the \textit{RayIntersect} class provided by the M3G standard. A \textit{RayIntersect} object is set by the \textit{pick} method, which is part of the \textit{Group} objects. \textit{RayIntersection} stores a pointer to the intersected \textit{Mesh} or \textit{Sprite3D}, and to all the relevant information about the intersection point.
The `pick` method first takes `Mesh` or `Sprite3D` into the group and enables it for picking, which is intersected by a pick ray passed as a parameter (a ray is a line in our case).

The following is the code for the `updateMove` method. It is self-explanatory, as it includes comments:

```java
private void updateMove() {

    transGroup.getTransform(trans);

    // computing space from speed
    space = speed * 0.5f;

    // check collisions
    RayIntersection ri = new RayIntersection();

    // car direction updating
    dir = getDirection();

    // check whether there is something in front
    // (Mesh or Sprite3D) excluding the car itself
    // and the floor

    if (scene.pick(-1, xCoord, yCoord, zCoord, dir[0], dir[1], dir[2], ri)) {

        // object distance
        float distance = ri.getDistance();

        // 38 is cone size

        if (distance > 38.0f + space) {
            // move
            trans.postTranslate(-space, 0, 0);
            transGroup.setTransform(trans);
        }
    } else {
        // stop the car
        speed = 0.0f;
        return;
    }
}
```
// move
trans.postTranslate(-space, 0, 0);
transGroup.setTransform(trans);
}

Figure 4.10 shows a snapshot of the application described with coding examples, and includes many of the API described in the chapter.

Fig. 4.10. DemoCar screen shot.
4.13 Summary

This chapter introduced M3G and the Java Mobile 3D Graphics library, and described how an application could be developed for mobile devices supporting this standard.

We described also the frameworks (CLDC/MIDP) used by Java for managing mobile devices and applications. M3G is consider an extension of these libraries and thus it is included in their development process. We also discussed the modalities of M3G, Immediate and Retained mode, explaining when and how to choose between the two. We then described elements of the M3G scene graph, which is a hierarchical structure used by this library for representing and managing a 3D scene.

We finally provided a comprehensive example, called CarDemo, including all the concepts, elements, and API that clarify the functionalities.