M3G Chapter 5. The Beginnings of a FPS

It's time to track down those penguins from earlier chapters and treat them with extreme prejudice.

GunnerM3G is the beginnings of a FPS (First-Person Shooter), where you roam about looking for two penguins to shoot at. Once a penguin's been blasted three times, it disappears from the scene. Each time you fire the gun, there's a flash of light and a pleasing laser beam-like noise. If the gun's pointing at a penguin, they're enveloped in a nasty looking animated fireball, accompanied by an explosion sound.

Figures 1 to 3 shows the user amongst the penguins, moving in on one of them and testing the gun, and finally blasting away.

Figure 1. First Encounter.

Figure 2. Moving in, Testing your Gun.

Figure 3. Shooting a Penguin.

The new programming elements in GunnerM3G:

- The camera has two 2D images attached to it. The "gun hand" is permanently visible, while the "shot flash" appears briefly after the user presses the fire key.
- We shoot a *pick ray* into the scene and obtain a reference to the object that it hits. If it's a penguin then an explosion occurs at the intersection point. The ray is a M3G level construct, not a 'death' ray.

- The explosion is a series of 2D images. An image briefly becomes the texture for a square mesh, before being replaced by the next image in the sequence.
- This example supports layered sounds, where multiple laser blasts and explosions can be heard together.
- A penguin mesh contains a PenguinInfo object as its *user object*, which allows us to store extra details in the mesh, such as the number of 'lives' the penguin has left.

A guiding principle is to fake the 3D effects as much as possible. The "gun hand", the "shot flash", the explosions, the floor, and the background are all 2D visuals wrapped over simple rectangular meshes. Only the penguins are complicated models: they must have 3D depth, since the user can move around them.

Readers may recognize some elements from earlier chapters. The camera controls are based on those from the MobileCamera class, but we've (mercifully) dispensed with modes. The camera only moves forwards, backwards, and turn left or right. The floor is a tiled texture, created using TiledFloor from M3G chapter 4.

A final touch is that the penguins are randomly positioned on the floor each time the MIDlet begins, and the floor and background images are randomly chosen from a small selection of alternatives. Figure 4 shows another execution of GunnerM3G.

Figure 4. Another Day, and the Penguins are Back.

There are numerous ways GunnerM3G could be improved. One of them is to add actual laser beams (or bullets, or whatever). Although the visuals include a "shot flash" and an explosion, there's no representation of the shot itself.

1. Class Diagrams for GunnerM3G

Figure 5 shows the UML diagrams for all the classes in the GunnerM3G application. Only the public and protected methods are included.

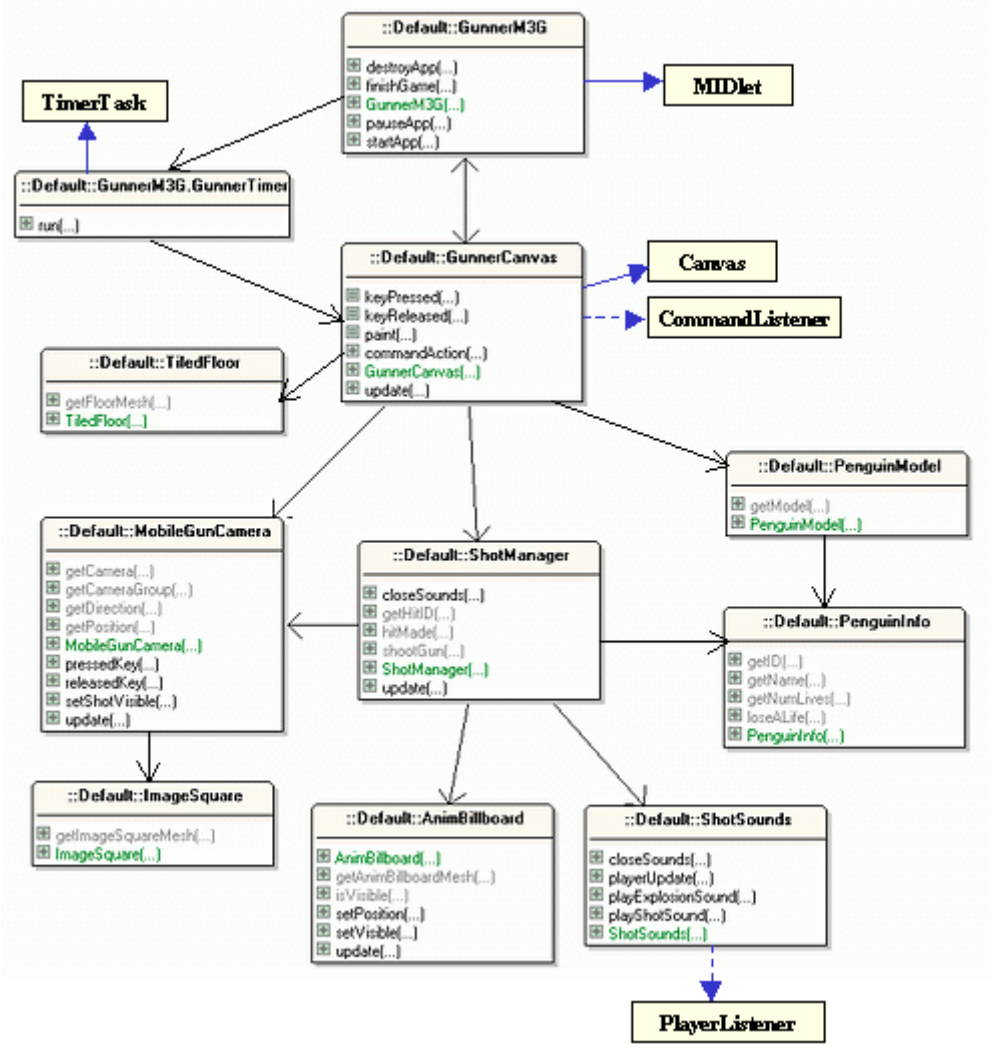

Figure 5. Class Diagrams for GunnerM3G.

GunnerM3G is the top-level MIDlet for the application, and similar to our earlier examples: it uses GunnerTimer to periodically update the canvas by calling update() in GunnerCanvas.

GunnerCanvas creates the 3D scene, using TiledFloor for the floor, and two instances of PenguinModel for the penguins. MobileGunCamera manages the camera, and utilizes ImageSquare instances for its attached "gun hand" and "shot flash" images.

A ShotManager object responds to the user pressing the fire key by making MobileGunCamera's "shot flash" appear, and playing a laser noise through ShotSounds. ShotManager sends a pick ray into the scene, which may intersect with a penguin model. Relevant penguin data is stored in a PenguinInfo object stored in the model, which is used by ShotManager to report on the hit. The animated fireball is controlled by AnimBillboard, and the explosion sound by ShotSounds.

2. The GunnerCanvas Class

As in previous 'canvas' classes, the main tasks are to build the scene, respond to key presses, and to periodically update the scene. Two new elements are the display of a hit message image when a penguin is shot (shown in Figure 3) and the random selection of the floor and background images, and the penguins' positions.

2.1. Building the Scene

The scene graph built in GunnerCanvas appears in Figure 6.

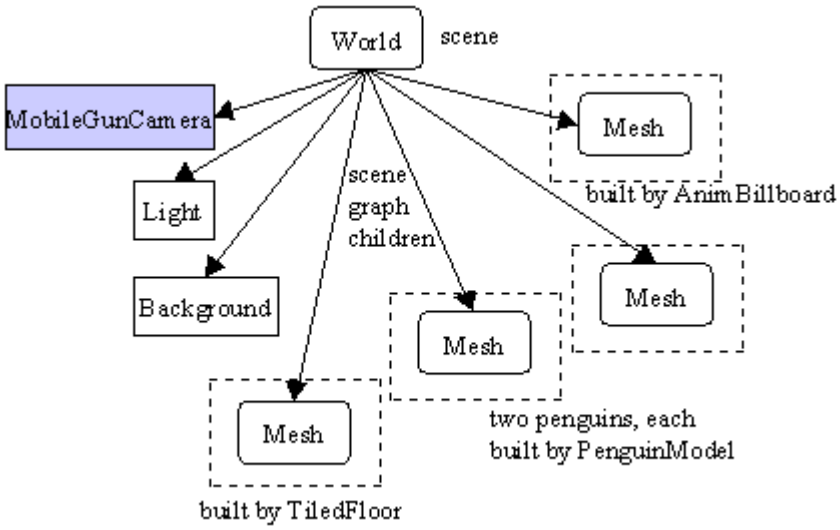

Figure 6. Scene Graph for GunnerM3G.

Figure 6 doesn't show the details for the sub-graph built by MobileGunCamera, which is explained later. The mesh constructed by AnimBillboard is for displaying the animated explosion, and is usually invisible.

GunnerCanvas uses buildScene() to construct the scene:

```
private void buildScene()
{ addCamera();
  addLights();
  addBackground();
  addFloor();
  addPenguins();
  shotMng = new ShotManager(scene, mobCam);
}
```

addCamera() creates a MobileGunCamera object, and addLights() adds a directional light. The ShotManager instance is in control of displaying the explosion animation, and it instantiates the AnimBillboard and adds it's mesh to the scene.

addBackground() and addFloor() both randomly select an image to decorate their floor and background. addFloor() shows the general approach:

```
private void addFloor()
{ Image2D floorIm = null;
```



```
    int choice = rnd.nextInt(4);
    System.out.println("Adding floor image " + choice);

    switch (choice) {
      case 0: floorIm = loadImage("/floor/grass.gif"); break;
      case 1: floorIm = loadImage("/floor/stone.jpg"); break;
      case 2: floorIm = loadImage("/floor/cobbles.jpg"); break;
      case 3: floorIm = loadImage("/floor/plate.jpg"); break;
      default: System.out.println("No floor image"); break;
    }

    TiledFloor f = new TiledFloor( floorIm, FLOOR_LEN);
      // FLOOR_LEN by FLOOR_LEN size, made up of 1 by 1 tiles

    scene.addChild( f.getFloorMesh() );
  }
```

The randomly selected image is tiled over a `FLOOR_LEN` by `FLOOR_LEN` sized floor (`FLOOR_LEN` is 8). The Random object, rnd, is created in GunnerCanvas' constructor. loadImage() loads the image using Loader.load().

addPenguins() creates `NUM_PENGUINS` (2) penguins, positioned at random on the floor:

```
  private void addPenguins()
  {
    PenguinModel pm;
    float x, z;
    for (int i=0; i < NUM_PENGUINS; i++) {
      x = rndCoord(); z = rndCoord();
      System.out.println("Adding Penguin " + i +
                         " at (" + x + ", " + z + ") ...");
      pm = new PenguinModel("Penguin " + i, i, x, z);
      scene.addChild( pm.getModel() );   // add the model
    }
  }  // end of addPenguins()


  private float rndCoord()
  // generate a coordinate in the range -FLOOR_LEN/2 to FLOOR_LEN/2
  {
    float f = rnd.nextFloat();  // 0.0f to 1.0f
    float coord = (f* FLOOR_LEN) - FLOOR_LEN/2.0f;
    return coord;
  }
```

rndCoord() uses the rnd Random object to generate a float in the range 0-1, and this is scaled to a value between `-FLOOR_LEN/2` to `FLOOR_LEN/2` (the extent of the floor). Two such numbers specify the (x, z) location of the penguin. No effort is made to avoid the penguins overlapping, but the chances of this are fairly unlikely.

2.2. Key Processing

Key presses and releases are sent from GunnerCanvas onto the MobileGunCamera, mobCam, where they control the camera's movement. However, the pressing of the fire key is sent to the ShotManager instance, to shoot the gun.

```
// global variables
private MobileGunCamera mobCam;
private ShotManager shotMng;


protected void keyPressed(int keyCode)
{
  int gameAction = getGameAction(keyCode);
  if (gameAction == Canvas.FIRE)
    shotMng.shootGun();
  else
    mobCam.pressedKey(gameAction);
}


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


2.3. Updating and Repainting

Figure 7 summarizes the stages in updating and repainting the scene.

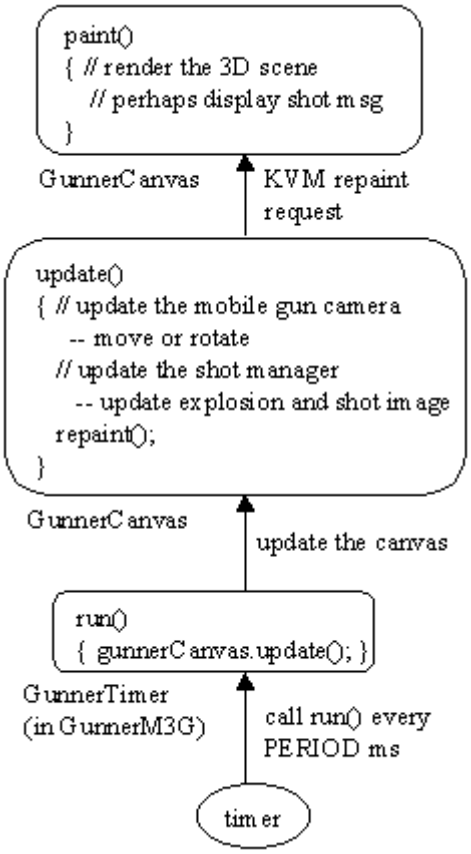

Figure 7. Updating and Repainting in GunnerM3G.

The update() method in GunnerCanvas:

```
public void update()
{ mobCam.update();     // update the camera
  shotMng.update();    // update the shot manager
  repaint();
}
```

The camera update causes a repeating key setting, such as a forward step, to be executed again. The shot manager updates the status of the "shot flash" and animated explosion images. The "shot flash" should only be visible for a short time before disappearing, and the explosion image is changed to the next one in the sequence.

The paint() method renders the 3D scene, then decides whether to show the hit message:

```
// global variables
private Image hitIm;     // hit image
private int xHit;        // x-coord of hit image on screen


protected void paint(Graphics g)
```



```
  {
    iG3D.bindTarget(g);
    try {
      iG3D.render(scene);
    }
    catch(Exception e)
    { e.printStackTrace();  }
    finally {
      iG3D.releaseTarget();
    }

    // maybe show a hit message (an image and text)
    if ( shotMng.hitMade() ) {
      g.drawImage( hitIm, xHit, 10, Graphics.TOP|Graphics.LEFT);

      // large and bold
      g.setFont(Font.getFont(Font.FACE_PROPORTIONAL,
                             Font.STYLE_BOLD, Font.SIZE_LARGE));
      g.setColor(0xFF0000);  // red
      g.drawString( "ID of penguin: " + shotMng.getHitID(), 5,5,
                                        Graphics.TOP|Graphics.LEFT);
    }
  } // end of paint()
```

The message is a preloaded image, stored in hitIm, and the string "ID of penguin: " plus a number, with the value coming from calling getHitID() in ShotManager.

3. The MobileGunCamera Class

MobileGunCamera creates the sub-graph representing the camera, "gun hand" and "shot flash". It also responds to key presses to move the camera forward, backwards, and turn it left or right. The class is a relative of MobileCamera from M3G chapter 3, without the modes. The four sorts of movement can be handled by the four arrow keys alone.

The general idea is to place 2D images in front of the camera, and move them as the camera moves. One image is the "gun hand", the other the "shot flash", which is only rendered momentarily when the user presses the fire key.

Figure 8 is a simplified picture of the camera's configuration, including the relative position of things along the negative z- and y-axes.

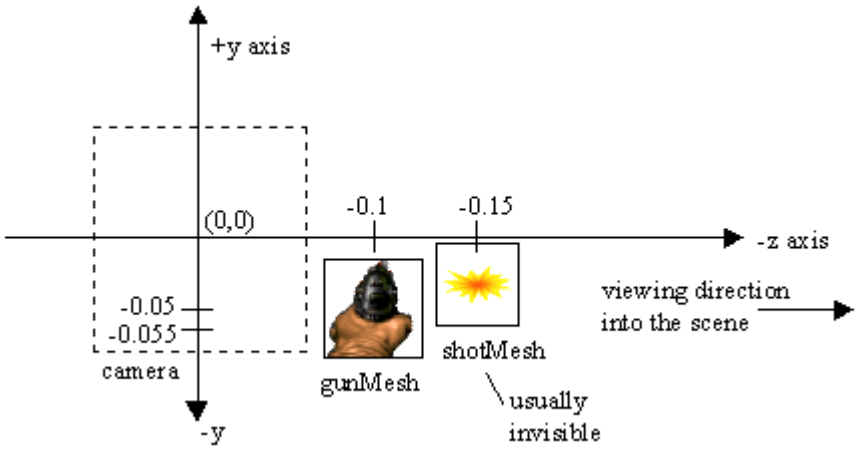

Figure 8. Camera Elements along the -Z-Axis.

The camera initially faces along the -z axis, so the 2D images must be positioned further along that axis. The "gun hand", encoded in the gunMesh object, is as close to the camera as possible, so nothing is likely to appear between it and the camera. The "shot flash" image, managed by the shotMesh object, is further along the -z axis so that when it's made visible, it appears beyond the gun.

The two images must be linked to the camera, so that when the camera moves, they do as well. This requirement leads to the sub-graph shown in Figure 9.

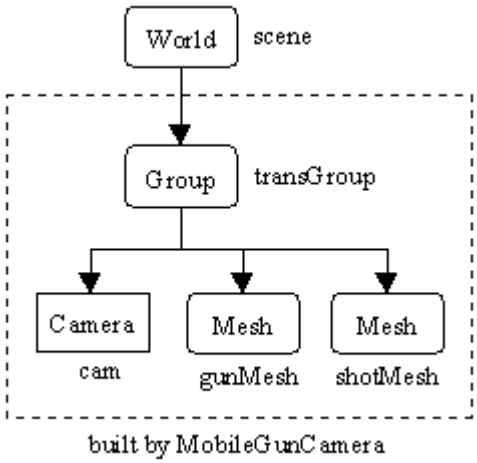

Figure 9. The Sub-graph for the Camera.

The camera and meshes are grouped together under a single Group node, transGroup. Camera translations and rotations are applied to that node, thereby moving and rotating everything in unison.

3.1. Building the Sub-graph

The scene graph shown in Figure 9 is built by MobileGunCamera's constructor:

```
  // initial camera position
  private static final float X_POS = 0.0f;
  private static final float Y_POS = 0.4f;
  private static final float Z_POS = 2.0f;

  private Transform trans = new Transform();

  // the camera's current y-axis rotation
  private float yAngle;

  // scene graph elements for the camera
  private Group transGroup;
  private Camera cam;
  private Mesh shotMesh;


  public MobileGunCamera(int width, int height)
  {
    cam = new Camera();
    float aspectRatio = ((float) width) / ((float) height);
    cam.setPerspective(70.0f, aspectRatio, 0.1f, 50.0f);
    cam.postRotate(-2.0f, 1.0f, 0, 0);  // x-axis rot down

    // gun image mesh
    Image2D imGun = loadImage("/gun.gif");
    ImageSquare imSqGun =
             new ImageSquare(imGun, 0, -0.055f, -0.1f, 0.05f);
    Mesh gunMesh = imSqGun.getImageSquareMesh();
    gunMesh.setPickingEnable(false);

    // shot image mesh
    Image2D imShot = loadImage("/shot.gif");
    ImageSquare imSqShot =
             new ImageSquare(imShot, 0, -0.05f, -0.15f, 0.1f);
    shotMesh = imSqShot.getImageSquareMesh();
    shotMesh.setPickingEnable(false);
    shotMesh.setRenderingEnable(false);  // invisible initially

    // put everything together;
    // transGroup handles both camera translation and y-axis rots
    transGroup = new Group();      // no initial rotation
    trans.postTranslate(X_POS, Y_POS, Z_POS);
    transGroup.setTransform(trans);
    transGroup.addChild(cam);
    transGroup.addChild(gunMesh);
    transGroup.addChild(shotMesh);

    // store initial rotation
    yAngle = 0.0f;
  }  // end of MobileGunCamera()
```

The ImageSquare class used to create the "gun hand" and "shot flash" image meshes will be explained shortly, but its constructor's arguments are the image, the (x,y,z) location of the mesh's center, and a scaling factor for its sides. The values were arrived at by trial-and-error, starting with a -z axis position that was far away from the camera, then moving slowly closer, accompanied by scaling to make the image smaller.

Both the meshes are made unpickable by calling Node.setPickingEnable(false). As we'll see later, the ShotManager shoots a pick ray out of the camera into the scene, and we want it to pass through the images unimpeded.

The shotMesh Mesh is made invisible by Node.setRenderingEnable(false). It's visibility is toggled with the public MobileGunCamera method setShotVisible():

```
public void setShotVisible(boolean b)
{  shotMesh.setRenderingEnable(b);  }
```

setShotVisible() is called by the ShotManager at various times to make the shot flash appear and disappear.

3.2. Handling Key Presses and Releases

Four booleans are manipulated in response to key presses and releases.

```
// global booleans for remembering key presses
private boolean upPressed = false;
private boolean downPressed = false;
private boolean leftPressed = false;
private boolean rightPressed = false;


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;
  }
}

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;
  }
}
```

The current values of the booleans determine the translation or rotation carried out when the camera is updated:

```
public void update()
{
  if (upPressed || downPressed)
    updateMove();
  else if (leftPressed || rightPressed)
    updateRotation();
}
```


updateMove() applies a simple translation to the transGroup Group node, but updateRotation() is a little more involved.

The basic problem with rotation is that it must be applied to the camera when it's positioned at the origin, or the rotation will change the camera's location.

Our solution is to maintain a global, yAngle, which records the total y-axis rotation of the camera since the camera was created. When it comes time to rotate the camera, it's current location is saved, then a transformation is applied which sends the camera to the origin, rotates it, then moves it back to its original location.

```
// globals for storing the camera's current y- rotations
private float yAngle;
private Transform rotTrans = new Transform();


private void updateRotation()
{
  if (leftPressed) {   // rotate left around the y-axis
    rotTrans.postRotate(ANGLE_INCR, 0, 1.0f, 0);
    yAngle += ANGLE_INCR;
  }
  else {   // rotate right around the y-axis
    rotTrans.postRotate(-ANGLE_INCR, 0, 1.0f, 0);
    yAngle -= ANGLE_INCR;
  }

  // angle values are modulo 360 degrees
  if (yAngle >= 360.0f)
    yAngle -= 360.0f;
  else if (yAngle <= -360.0f)
    yAngle += 360.0f;

  // apply the y-axis rotation to transGroup
  storePosition();
  trans.setIdentity();
  trans.postTranslate(xCoord, yCoord, zCoord);
  trans.postRotate(yAngle, 0, 1.0f, 0);
  transGroup.setTransform(trans);
}  // end of updateRotation()
```

rotTrans holds the current rotation transformation, used when calculating the pick ray direction. This is explained in more detail below.

storePosition() extracts the (x,y,z) location of the transGroup node, and stores it in xCoord, yCoord, and zCoord.

```
private void storePosition()
{
  transGroup.getCompositeTransform(trans);

  trans.get(transMat);
  xCoord = transMat[3];
  yCoord = transMat[7];
  zCoord = transMat[11];
}  // end of storePosition()
```


The call to Transformable.getCompositeTransform() returns a 4-by-4 matrix, with the translation component in the fourth column.

A confusing aspect of updateRotation() is that we don't seem to translate the camera to the origin. The relevant code fragment is:

```
trans.setIdentity();
trans.postTranslate(xCoord, yCoord, zCoord);
trans.postRotate(yAngle, 0, 1.0f, 0);
transGroup.setTransform(trans);
```

The transformation stored in trans is a rotation followed by a translation (operations are carried out right-to-left). This *overwrites* the existing transform in transGroup when Transformable.setTransform() is called. This is equivalent to changing an unrotated transGroup starting at the origin.

3.3. Supplying Pick Ray Information

MobileGunCamera is contacted by the ShotManager when it needs to generate a pick ray. ShotManager requires the camera's current location and forward facing direction, which are supplied by getPosition() and getDirection().

```
public float[] getPosition()
{ storePosition();
  return new float[] { xCoord, yCoord, zCoord };
}


public float[] getDirection()
{ float[] zVec = {0, 0, -1.0f, 0};
  rotTrans.transform(zVec);
  return new float[] { zVec[0], zVec[1], zVec[2] };
}
```

getPosition() calls storePosition(), then returns an array of the latest xCoord, yCoord, and zCoord values.

getDirection() applies the rotTrans Transform (which stores the current y-axis rotation) to the unit vector (0,0,-1). This vector represents the starting direction of the camera – facing along the -z axis. The result, written back to zvec[], is the camera's current direction.

A small catch with both these methods is that they should return new float arrays, rather than references to existing arrays in the MobileGunCamera. Arrays stored as globals in MobileGunCamera could be changed after their references have been passed to ShotManager, affecting ShotManager's calculations.

4. The ImageSquare Class

The ImageSquare constructor is:

```
ImageSquare(Image2D bbImage, float x, float y, float z, float size);
```


It creates a square mesh of length size, with its center at (x,y,z), facing along the +z axis. The image wrapped over it (bbImage) can have transparent elements which won't be rendered when the mesh is displayed.

The code is virtually the same as the Billboard class in M3G chapter 4, except that ImageSquare doesn't align with anything. I won't bother going through its implementation.

Although we only use ImageSquare to hold the "gun hand" and "shot flash" images in MobileGunCamera, the class could be used anywhere that we need a picture (perhaps with transparent parts). For example, ImageSquare objects could display buildings in the background.

5. The PenguinModel Class

The PenguinModel constructor is:

```
PenguinModel(String name, int ID, float xCoord, float zCoord);
```

It creates a penguin model, standing on the XZ plane, at location (xCoord, 0, zCoord). The name and the ID can be anything; in GunnerCanvas, the name is "Penguin " plus the ID number.

The constructor:

```
// global variable
private Mesh model;

public PenguinModel(String name, int ID,
                            float xCoord, float zCoord)
{ // build the mesh
  VertexBuffer vb = makeGeometry();
  IndexBuffer ib =  new TriangleStripArray(0, getStripLengths());
  Appearance app = makeAppearance("/penguin.gif");
  model = new Mesh(vb, ib, app);
  model.setUserObject( new PenguinInfo(name,ID));  //store name, ID

  // reposition the model's start position and size
  model.setTranslation(xCoord+0.25f, 0.25f, zCoord+0.25f);
  model.scale(0.5f, 0.5f, 0.5f);

  model.setPickingEnable(true);   // so can fire a pick ray at it
}
```

The geometry is constructed in makeGeometry(), the appearance in makeAppearance (). We've seen these methods before in earlier chapters so I'll skip them here. The model is constructed using data methods generated by ObjView, (see M3G Chapter 1) and pasted into this class:

```
short[] getVerts();
byte[] getNormals();
short[] getTexCoords();
int[] getStripLengths();
Material setMatColours();
```


The only new feature is the setting of the model's user object to contain an instance of PenguinInfo. ShotManager uses the presence of a PenguinInfo object in a mesh to identify it as a penguin, and employs the data inside the object to calculate when it's time for the penguin to die (disappear).

6. The PenguinInfo Class

A penguin can suffer three laser blasts before it disappears. PenguinInfo records the penguin's name, ID, and the number of lives it has left.

```
public class PenguinInfo
{
  private static final int MAX_LIVES = 3;

  private String name;
  private int id;
  private int numLives;


  public PenguinInfo(String nm, int id)
  { name = nm;
    this.id = id;
    numLives = MAX_LIVES;
  }

  public String getName()
  { return name;  }

  public int getID()
  { return id;  }

  public int getNumLives()
  {  return numLives;  }

  public boolean loseALife()
  { if (numLives > 0) {
      numLives--;
      return true;
    }
    else  // can't set to negative
      return false;
  } // end of loseALife()

}  // end of PenguinInfo class
```

This number of lives is reduced by ShotManager calling loseALife().

7. The ShotManager Class

The ShotManager manages the animated explosion (created by AnimBillboard), the playing of sounds (laser beams and explosions in ShotSounds), and the visibility of

the "shot flash" graphic in MobileGunCamera. However, it's most important job is to respond to the user pressing the fire key to shoot the gun.

The ShotManager constructor sets up the animated explosions in AnimBillboard, and the sounds in ShotSounds.

```
// global variables
private World scene;
private MobileGunCamera mobCam;

private AnimBillboard exploBoard;   // animated explosions
private ShotSounds shotSounds;
      // manages the fired shot and explosions sounds


public ShotManager(World s, MobileGunCamera mc)
{
  scene = s;
  mobCam = mc;

  // set up animated explosions billboard
  System.out.println("Adding explosion billboard");
  Image2D[] ims = loadImages("/explo/explo", 6);
  Group camGroup = mobCam.getCameraGroup();
  exploBoard = new AnimBillboard(ims, camGroup, 1.5f);
  scene.addChild( exploBoard.getAnimBillboardMesh() );

  // initialise sound resources for shooting
  shotSounds = new ShotSounds();
}  // end of ShotManager()
```

AnimBillboard receives its images in an Image2D[] array, and needs a reference to the camera so it can stay aligned with it.

7.1. Shooting the Gun

When GunnerCanvas is passed a Canvas.FIRE game action, it calls shootGun() in ShotManager. The "shot flash" image appears briefly, and a laser sound plays. If a pick ray from the camera, along its current forward direction, hits something, then an animated explosion is shown at the hit coordinates, and an explosion sound is played. If a penguin's been hit, it loses a life. When all its lives have been used up, the penguin is removed from the scene.

The components of a pick ray are illustrated in Figure 10.

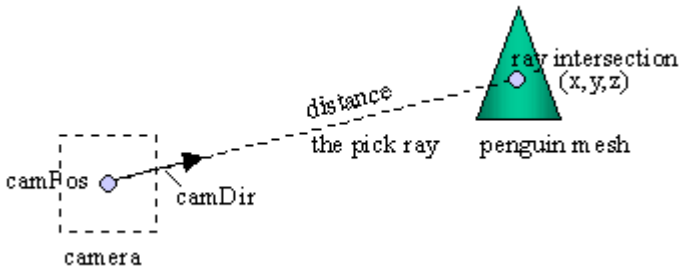

Figure 10. Pick Ray from Camera into Scene.

The pick ray requires a starting point and direction vector (camPos and camDir in Figure 10), and progresses in that direction until it intersects a pickable mesh. The distance traveled is obtained from a RayIntersection object, and the (x,y,z) intersection point on the mesh can then be calculated.

shootGun()'s implementation:

```
// Counter to control how long the fired shot image is displayed
private int firedShotCounter = 0;
private int hitPenguinID = -1;   // ID of last penguin hit


public int shootGun()
{
  // show fired shot image and play a sound
  firedShotCounter = FIRED_SHOT_MAX;
  mobCam.setShotVisible(true);
  shotSounds.playShotSound();

  // fire a picking ray out in the camera's forward direction
  RayIntersection ri = new RayIntersection();
  float[] camPos = mobCam.getPosition();   //camera's current posn
  float[] camDir = mobCam.getDirection();  // current fwd dir

  if (scene.pick(-1, camPos[0], camPos[1], camPos[2],
              camDir[0], camDir[1], camDir[2], ri)) {   // hit?
    float distance = ri.getDistance();  // normalized dist to hit

    checkHit(ri);

    // calculate the coords of the hit thing
    float x = camPos[0] + (camDir[0] * distance);
    float y = camPos[1] + (camDir[1] * distance);
    float z = camPos[2] + (camDir[2] * distance);
```



```
      // show explosion at (x,y,z) and play a sound
      exploBoard.setPosition(x,y,z);
      exploBoard.setVisible(true);
      shotSounds.playExplosionSound();
    }
    return hitPenguinID;
  }  // end of shootGun()
```

The camera position (camPos) and camera direction (camDir) are retrieved from MobileGunCamera, then plugged into Group.pick(). The integer argument, -1, means that every mesh in the scene will be tested. Meshes have picking enabled by default, so it's a good idea to disable it for as many of them as possible, to reduce the likelihood of hitting the wrong thing, and to speed up ray testing.

I used Node.setPickingEnable(false) to switch off picking for the camera's ImageSquares and the floor.

If the pick ray intersects with a mesh, then Group.pick() returns true, and fills the RayIntersection object (ri) with information. RayIntersection.getDistance() returns the distance from the pick ray origin (camPos) to the intersection point. This distance is normalized to the length of the direction vector (i.e. the length of camDir). This makes it easy to obtain the intersection point, as shown in the code.

We need the intersection point (x,y,z) in order to position the AnimBillboard, exploBoard, prior to making it visible.

Mysterious Integers

There are two somewhat mysterious integers used in shootGun(): firedShotCounter and hitPenguinID.

firedShotCounter starts with the value `FIRED_SHOT_MAX` (2) and is decremented by each call to ShotManager's update() method:

```
public void update()
{
  exploBoard.update();

  if (firedShotCounter == 0)     // hide the shot image?
    mobCam.setShotVisible(false);   // yes, hide it
  else
    firedShotCounter--;   // keep counting down
}
```

When firedShotCounter reaches 0, after only two calls to update(), the "shot flash" image disappears.

The other job of update() is to make the explosion animation progress to its next frame, which is does by calling AnimBillboard's update() method. When the end of the animation sequence is reached, further calls to update() will have no effect.

The other integer is hitPenguinID, which identifies which penguin was hit. When no penguin was shot, its value is set to -1. hitPenguinID is returned to GunnerCanvas (the caller of shootGun()), where it's used as part of the hit message.

7.2. What did I Hit?

Although Group.pick() tells us whether the pick ray intersected something, it doesn't say *what* was hit. We have to delve into the instantiated RayIntersection object to find that out. checkHit() decides if the user hit a penguin or not.

```
private void checkHit(RayIntersection ri)
{
   hitPenguinID = -1;   // reset to default value
   Node selected = ri.getIntersected();
   if (selected instanceof Mesh) {  // we hit a mesh
     Mesh m = (Mesh) selected;
     PenguinInfo pInfo = (PenguinInfo) m.getUserObject();
     if (pInfo != null) {  // we hit a penguin
       hitPenguinID = pInfo.getID();  // save ID
       hitPenguin(pInfo, m);
     }
   }
}
```

A reference to the Node object hit by the pick ray is obtained with RayIntersection.getIntersected(), and we test to see if it's a Mesh (it may also be a Camera, Light, or Group object). Even if we determine that it's a mesh, is it a penguin mesh? This is where the user object field comes in handy, since only penguin meshes contain objects. The PenguinInfo object holds all the details we need (including the penguin's ID).

A shot makes a penguin disappear if it's already been hit twice before. This behaviour is implemented in hitPenguin():

```
private void hitPenguin(PenguinInfo pInfo, Mesh m)
{
  int numLives = pInfo.getNumLives() - 1;  // since just shot
  if (numLives == 0) {    // last life gone
    System.out.println("Killed " + pInfo.getName() +
                       "; ID: " + hitPenguinID);
    m.setRenderingEnable(false);
    m.setPickingEnable(false);
  }
  else {
    System.out.println("Hit " + pInfo.getName() +
                       "; ID: " + hitPenguinID +
                       "; lives left: " + numLives);
    pInfo.loseALife();
  }
}  // end of hitPenguin()
```

If its time to say 'bye bye' to the penguin, then it's made invisible and unpickable, otherwise we decrement the number of lives.

The user object field (which is present in any subclass of Object3D) is a good way of augmenting an object's data and behaviour.

8. The AnimBillboard Class

AnimBillboard is a variant of the Billboard class of M3G chapter 4. It creates a square of length size, resting on the XZ plane at the origin, which stays z-axis aligned with the camera position. The image wrapped over the board can have transparent elements, which won't be rendered when the mesh is displayed.

Where AnimBillboard differs from Billboard is that its position can be adjusted by calling setPosition(), and its visibility toggled with setVisible().

The important new feature is that AnimBillboard will display an animation, a sequence of images used as textures, in quick succession on the face of its board. The sequence begins when the board is made visible, and the next frame is shown whenever AnimBillboard's update() method is called. At the end of the sequence, the board will automatically become invisible again.

The animation mechanism is illustrated by Figure 11.

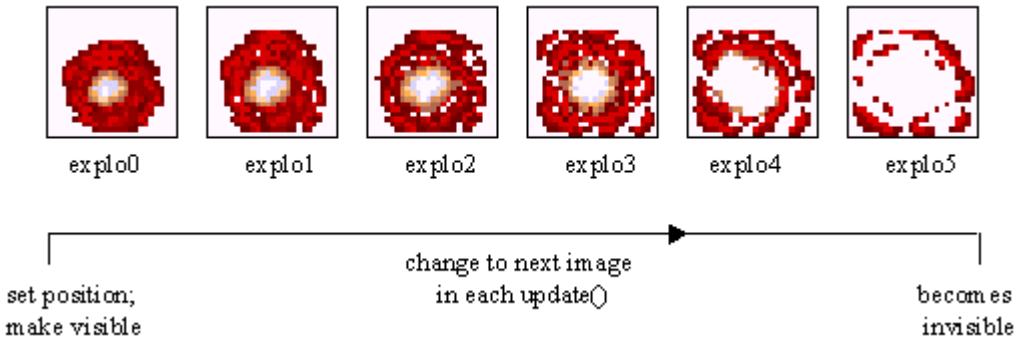

Figure 11. The Animation Sequence in AnimBillboard.

AnimBillboard's constructor creates a square mesh in the same way as the mesh in ImageSquare.

```
// globals
private Mesh bbMesh;
private Appearance app;

private Texture2D[] textures;   // holds the images
private int numImages, currImIdx;

private boolean isVisible;  // is the animated board visible?
private Group cameraGroup;   // used for camera alignment


public AnimBillboard(Image2D[] bbIms, Group camGroup, float size)
{
  cameraGroup = camGroup;

  // build the mesh
  VertexBuffer vb = makeGeometry();

  int[] indicies = {1,2,0,3};  // one quad
  int[] stripLens = {4};
  IndexBuffer ib = new TriangleStripArray(indicies, stripLens);

  numImages = bbIms.length;
```



```
    currImIdx = 0;
    textures = makeTextures(bbIms, numImages);

    app = makeAppearance( textures[currImIdx] );

    bbMesh = new Mesh(vb, ib, app);

    // scale and position the mesh
    float scale2 = size * 0.5f;   // to reduce 2 by 2 image to 1 by 1
    bbMesh.scale(scale2, scale2, scale2);
    bbMesh.setTranslation(0, scale2, 0);

    bbMesh.setAlignment(cameraGroup, Node.Z_AXIS, null, Node.NONE);
             // only use z-axis alignment with the camera
    bbMesh.setPickingEnable(false);

    setVisible(false);
  }  // end of AnimBillboard()
```

makeGeometry() builds a square centered at the origin on the XZ plane with sides of 2 units (which explains the additional scale factor of 0.5 later on in the constructor).

makeTextures() takes the array of images supplied as input, and creates an array of textures. Each texture's colours and alpha will replace those of the mesh.

makeAppearance() returns a texture appearance which only shows the opaque parts. The first texture in the texture array is used initially (its index is stored in currImIdx).

The mesh is made unpickable, so it's not possible to shoot at the explosion.

8.1. Positioning and Visibility

Unlike the Billboard class of M3G chapter 4, AnimBillboard can be moved and its visibility changed:

```
public void setPosition(float x, float y, float z)
{  bbMesh.setTranslation(x,y,z); }

public void setVisible(boolean visible)
{ isVisible = visible;
  if (isVisible) {
    bbMesh.setRenderingEnable(true);
    currImIdx = 0;    // reset the image display index
  }
  else
    bbMesh.setRenderingEnable(false);
}
```

When the board is made visible, its image display index (currImIdx) is reset. This prepares it for the next update() call.

Typical usage for these methods can be seen in shootGun() in ShotManager: the board is positioned at the ray intersection point with setPosition(), then made visible with setVisible(true).

8.2. Updating the Animation

AnimBillboard's update() method is called from GunnerCanvas's update(), which is called periodically by the GunnerTimer timer task. If the explosion is visible, then the current image in the sequence is shown, and the image index is incremented, ready for the next update.

When the end of the sequence is reached, the board becomes invisible.

```
public void update()
{
  if (isVisible) {
    bbMesh.align(cameraGroup);   // update alignment
    if (currImIdx == numImages)  // at end of sequence
      setVisible(false);
    else
      app.setTexture(0, textures[currImIdx]);  // show image
    currImIdx = currImIdx+1;   // set to next image index
  }
}
```

The call to Node.align() keeps the billboard facing the camera. Even if the user moves while the explosion is playing, the board will rotate to follow the camera.

9. The ShotSounds Class

The ShotSounds class manages laser beams sounds (emitted by our high-tech blaster), and explosions (as a beam hits a mesh). We want to hear overlapping audio (e.g. multiple laser beam noises and explosions), triggered when the user rapidly presses the fire key, and several shots hit the penguin.

This means that the ShotSounds implementation can't utilize a single Player object to generate all the sounds, because Player can only handle a single sound at a time. Also, the initialisation of a player, including the loading of a new sound, takes a small, but noticeable, amount of time, which we want to avoid.

These requirements point us towards a ShotSounds implementation containing a *pool* of Player objects, with their sounds ready to play when we need them. We hide this complexity, so the public methods for playing sounds, playShotSound() and playExplosionSound(), don't require any pool-related arguments. ShotSounds chooses an available player for itself when one of these methods is called.

The ShotSounds() constructor stores the Player objects in two arrays: one for playing laser beam sounds, the other for explosions.

```
// number of fired shot and explosion sounds used
private static final int NUM_SHOT_SOUNDS = 3;
private static final int NUM_EXPLO_SOUNDS = 5;


// for playing sounds (fired shots and explosions)
private Player[] shotPlayers;
private Player[] exploPlayers;
private int currExploPlayer, currShotPlayer;
```



```
public ShotSounds()
{
  // load fired shot sounds
  System.out.println("Loading sounds");
  shotPlayers = loadSounds("/sounds/laser.wav", NUM_SHOT_SOUNDS);
  currShotPlayer = 0;

  loadExploSounds("/sounds/explo");
}  // end of ShotSounds()
```

loadSounds() uses loadSound() to load multiple copies of the *same* laser sound ("sounds/laser.wav"), creating a separate Player for each one.

```
private Player[] loadSounds(String fn, int num)
{
  Player[] ps = new Player[num];
  for (int i=0; i < num; i++)
    ps[i] = loadSound(fn);
  return ps;
}
```

Explosions are more interesting if they sound a bit different from each other, so several explosion clips are loaded by loadExploSounds():

```
private void loadExploSounds(String fn)
// load fn + ((0 to (NUM_EXPLO_SOUNDS-1) + ".wav" sounds
{
  exploPlayers = new Player[NUM_EXPLO_SOUNDS];
  currExploPlayer = 0;
  for (int i=0; i < NUM_EXPLO_SOUNDS; i++)
    exploPlayers[i] = loadSound(fn+i+".wav");
}
```

A prefix is supplied ("sounds/explo" in this case), and an actual file obtained by adding a number and ".wav" extension.

Both loadSounds() and loadExploSounds() load an individual sound using loadSound (), which is explained below.

9.1. Playing a Sound

playShotSound() plays a laser beam audio clip, and playExplosionSound() an explosion:

```
public void playShotSound()
{ playSound( shotPlayers[currShotPlayer] );
  currShotPlayer = (currShotPlayer+1)%NUM_SHOT_SOUNDS;
}

public void playExplosionSound()
{ playSound( exploPlayers[currExploPlayer] );
  currExploPlayer = (currExploPlayer+1)%NUM_EXPLO_SOUNDS;
```



```
}
```

They use the currShotPlayer and currExploPlayer counters to cycle through the pool of Player objects stored in shotPlayers[] and exploPlayers[].

Both playShotSound() and playExplosionSound() use playSound() to play a particular sound. playSound() is described below.

9.2. Player States

To understand how we use a player, it helps to consider a (simplified) version of its possible states, and the transitions between them, as shown in Figure 12. A complete version of this diagram can be found in the MIDP documentation for the Player interface.

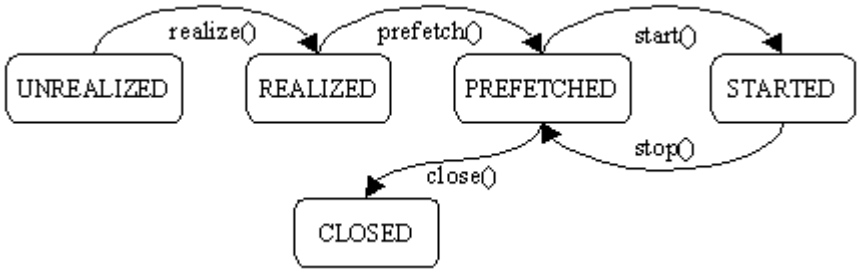

Figure 12. A Simplified State Transition Diagram for Player.

The names attached to the transitions are the methods which move a player from one state to the next.

A Player object only starts playing a sound when it's moved from the PREFETCHED state to STARTED by a call to Player.start(). It's possible to call start() when the Player has just been created, but the object still has to pass through the UNREALIZED, REALIZED, and PREFETCHED states first, which involves loading the resource and creating in-memory buffers.

A useful optimization is to transfer the Player object to its PREFETCHED state as part of the sound loading phase, so that the call to Player.start() only requires a quick transition from PREFETCHED to STARTED.

A player stops when it reaches the end of the sound, or when Player.stop() is invoked. When that happens, the Player object moves from the STARTED state back to PREFETCHED. It's then ready to play again.

Player.close() causes the Player instance to become CLOSED (it terminates).

9.3. Player States in ShotSounds

The ShotSounds methods which affect a Player's state are shown in the modified state transition diagram in Figure 13.

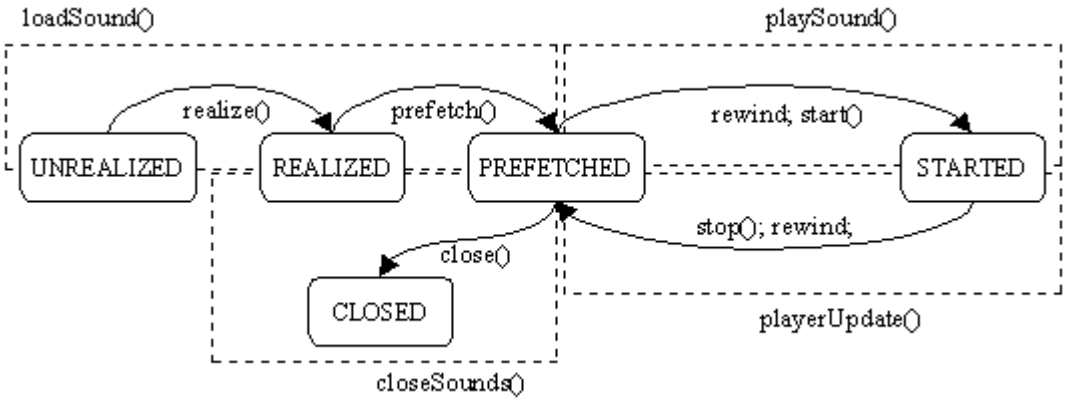

Figure 13. Player State Transition Diagram and ShotSounds Methods.

loadSound() moves a player to its PREFETCHED state, playSound() moves it to STARTED, playerUpdate() makes the player return to its PREFETCHED state, and closeSounds() terminates all the players.

The loadSound() code:

```
private Player loadSound(String fn)
{
  Player p = null;
  try{
    InputStream in = getClass().getResourceAsStream(fn);
    p = Manager.createPlayer(in,"audio/x-wav");
    p.realize();
    p.prefetch();    // move player to PREFETECHED state
    p.addPlayerListener(this);
  }
  catch(Exception ex)
  { System.out.println("Could not load sound in " + fn);  }

  return p;
}
```

playSound() includes two extra tests around its call to Player.start()

```
private void playSound(Player p)
// Only start a player if it's in a PREFETECHED state.
{
  if (p != null) {
    if (p.getState() == Player.PREFETCHED) {
      try{
        p.setMediaTime(0);    // rewind (a safety precaution)
        p.start();
      }
      catch(Exception ex)
      { System.out.println("Could not play sound ");  }
    }
  }
}  // end of playSound()
```


The check of the Player's state deals with the possibility that the Player is already playing its sound. This may occur if the Player was called previously by playShotSound() or playExplosionSound(), and the sound hasn't yet finished.

This state test seems unnecessary since the Player documentation specifies that if Player.start() is called when the Player is in the STARTED state, then the request will be ignored. However, in practice, this usually causes strange behaviour in the player, with fragments of the sound being output, or the Player going dead. The avoidance of possible problems is why the sound is rewound by calling Player.setMediaTime().

loadSound() attaches ShotSounds to the Player object as a listener, which means that playerUpdate() will be called when the Player changes state, and when other events occur. We're only interested in responding to the arrival of an END_OF_MEDIA string.

```
public void playerUpdate(Player p, String event, Object eventData)
// reset the player, ready for its next use
{
  if (event == END_OF_MEDIA) {
    try {
      p.stop();           // back to PREFETECHED state
      p.setMediaTime(0);  // rewind to the beginning
    }
     catch(Exception ex) {}
  }
}  // end of playerUpdate()
```

The use of playerUpdate() to explicitly stop the Player isn't required according to the Player documentation, since the object should automatically return to the PREFETCHED state when the sound finishes. However, code which relies on the automatic state change often behaves erratically. The rewinding of the sound to the beginning by Player.setMediaTime() is yet another example of defensive programming.

closeSounds() loops through the two Players arrays, calling close() on all the players.

```
public void closeSounds()
// tidy up by closing all the players down
{
  for(int i=0; i < NUM_SHOT_SOUNDS; i++)
    shotPlayers[i].close();
  for(int i=0; i < NUM_EXPLO_SOUNDS; i++)
    exploPlayers[i].close();
}
```

