PART 4

Android: Build a tilting maze

Our expert

Juliet Kemp enjoys fiddling with code, and has a bizarre fondness for alarming neon colours on screens.

Juliet Kemp has some more fun with the accelerometer and creates some custom XML attributes.

In the last tutorial, we looked at sensors. This time we're going to play about a bit more with the accelerometer, which is particularly useful if you want to write games. We'll also look at more complicated and multi-layered XML layouts, creating your own custom attributes, and views which also contain code logic. Our project to put all this into practice is a game where the player tips the device to navigate a little ball around a maze.

Setting up the maze

We want to create a 'perfect' maze, with a single path from any point to any other point, no loops, and no inaccessible sections. This means that there's a single path for the player to navigate, from the entrance (top left) to the exit (bottom right); the key is not to end up in a dead end.

There are a bunch of different ways to generate a maze but the algorithm we're using is 'depth first', starting at a given cell and then traversing the whole of the maze, choosing at random which walls to knock down to create a path.

We're using a basic, recursive version of this algorithm; unfortunately this limits the size of the maze you can generate, as you run out of stack quite quickly.

If you want to fill the whole device, you'll also encounter the problem of differently-sized screens. For now, we're going to set the maze size as an absolute. Once again, the orientation is locked as portrait, with this attribute in the **application** entity in **AndroidManifest.xml**:

```
android:screenOrientation="portrait"
```

To draw the maze, we're going to use a custom view. Android best practice is to keep as much layout as possible in XML, so let's set up the **res/layout/main.xml** layout file:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  >
  <com.example.maze.MazeView
        id="@+id/maze"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
</FrameLayout>
```

Add this line to **MazeActivity.onCreate**, to use the layout:

```
setContentView(R.layout.main);
```

On to the **MazeView** class. Because the constructor is being called ('inflated') from the XML layout, the constructor must take two specific arguments, (**Context** and **AttributeSet**):

```
public MazeView(Context context, AttributeSet attrs) {
  super(context);
  initVars(attrs);
  initMaze();
  generate(1, 1);
}
```

We can't pass the cell width and maze dimensions directly into the constructor from **MazeActivity**. Instead, we'll set them up as custom XML attributes, and use the **initVars(attrs)** method to get the values into the code.

To create a custom attribute, you first need to declare its format. Create a file **res/values/attrs.xml** that looks like this:

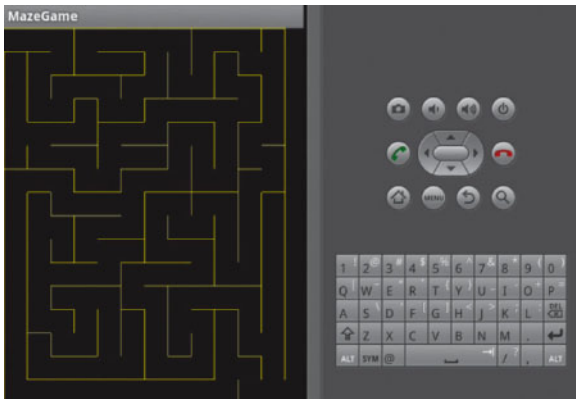
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <declare-styleable name="Maze">
    <attr name="cellWidth" format="dimension"/>
    <attr name="pixelWidth" format="dimension"/>
    <attr name="pixelHeight" format="dimension"/>
    <attr name="ballDiam" format="dimension"/>
  </declare-styleable>
</resources>
```

The name **Maze** is arbitrary, as are the attribute names. Open up **res/layout/main.xml** again to make two important changes. First add an extra schema line to the overall layout:

```
<FrameLayout
  xmlns:android="http://schemas.android.com/apk/res/
android"
  xmlns:maze="http://schemas.android.com/apk/res/com.
example.maze"
  ...
>
```

The **xmlns:maze** line is important: the final part of that URL should be the same as the 'package' line in your **AndroidManifest.xml** (you don't need to actually create that URL; it just needs to have the correct structure). Next, in the same file, add the attributes to the **MazeView** layout setup:

```
<com.example.maze.MazeView
  android:id="@+id/maze"
  ...
  maze:cellWidth="50px"
  maze:pixelWidth="300px"
  maze:pixelHeight="400px"
```



A 'perfect' maze (generated on the emulator) – it's probably the sort that you're most familiar with.

```
  maze:ballDiam="39px"
/>
```

The first part of each **maze:cellWidth** attribute name must be the same as the ID you used in the **xmlns:maze** line. It's important to put the 'px' value (pixels); since these attributes have dimension format, bare integers will throw a compile error.

Now back to the **MazeView** code to grab these values in the **initVars()** method:

```
private void initVars(AttributeSet attrs) {
  TypedArray a = getContext().obtainStyledAttributes(attrs, R.
styleable.Maze);
  cellWidth      = a.getDimension(R.styleable.Maze_cellWidth, 0);
  int pixelWidth  = (int)a.getDimension(R.styleable.Maze_
pixelWidth, 0);
  int pixelHeight = (int)a.getDimension(R.styleable.Maze_
pixelHeight, 0);
  a.recycle();
  width  = pixelWidth/(int)cellWidth;
  height = pixelHeight/(int)cellWidth;
}
```

A **TypedArray** is a container for an attribute array. It's important to call **recycle()** on it once you're done with it. **obtainStyledAttributes** grabs the attributes as defined in **R.styleable.Maze** (the ones we set up in **attrs.xml**) rather than the whole lot. Then we grab the values we need with the **getDimension** method (use the right method for your attribute type, eg, **getString** or **getColor**). We'll use **cellWidth** later as a float, and **width** and **height** as ints, hence the casting.

The 0 in the arguments to **getDimension** is the default value to use if an attribute is not defined or not a resource. If the value is 0, though, the maze isn't going to work very well. Add this line after **a.recycle()** to provide basic error-catching:

```
if (cellWidth <= 0 || pixelWidth <= 0 || pixelHeight <= 0) {
  throw new RuntimeException("Valid dimensions for the
maze must be passed into this class via XML.");
}
```

That's the layout all set up in XML; now onto the maze logic.

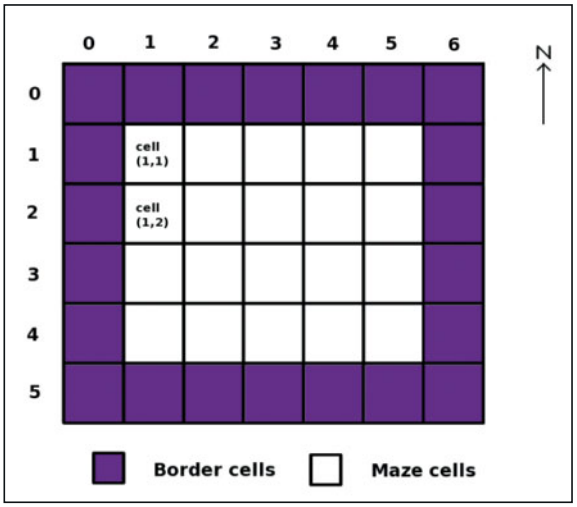
The **init()** method initialises the maze:

```
private void init() {
  // border cells have already been visited
  visited = new boolean[width+2][height+2];
  for (int x = 0; x < width+2; x++) {
    visited[x][0] = visited[x][height+1] = true;
  }
  for (int y = 0; y < height+2; y++) visited[0][y] = visited[width+1]
[y] = true;
  // every cell has all its walls intact to start with
  north = new boolean[width+2][height+2];
  // ... east, south, west similarly ...
  for (int x = 0; x < width+2; x++)
    for (int y = 0; y < height+2; y++)
      north[x][y] = east[x][y] = south[x][y] = west[x][y] = true;
  west[1][1] = false;
  east[width][height] = false;
}
```

You can think of each of the boolean arrays of arrays as a 2D grid which matches the maze grid. To create the 'border' of the maze, the grid is actually set up as having one extra cell all around it (so it's two cells wider and two cells taller than the maze itself, as in the picture below).

The first section of the **init()** method records the border cells as having already been visited, so they won't be iterated over in the main algorithm. Next, the grids representing the cell walls are set up. Each grid cell defines whether a particular cell has that wall or not. So **north[1][2]=true** would mean that the cell marked as 1,2 on the picture had its north wall intact.

Finally, we set up the entrance and the exit, by knocking out the east wall of the top-left-corner cell, and the west wall of the bottom-right-corner cell.



Maze cells with a border all around.

Dimensions in XML

In the layout in the main text, the integer dimension values are hard-coded. This isn't very maintainable, especially as we're going to use those same values again when we set up the **MazeBall** later.

A better solution is to set them up as dimension resources. Create a file **res/values/dimensions.xml** (it must be in this directory, but the file name is arbitrary) that looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="cellWidth">50px</
dimen>
  <dimen name="pixelWidth">300px</
dimen>
  <dimen name="pixelHeight">400px</
dimen>
  <dimen name="ballDiam">39px</
dimen>
</resources>
```

Yes, that is 'dimen', not 'dimension' as in the attribute declaration.

Android will pick up the type of the resource from the XML; other resource types include string, integer, boolean, and color. Check out the developer documentation for more.

Now edit the lines in **res/layout/main.xml**:

```
maze:cellWidth="@dimen/cellWidth"
```

We'll still retrieve these in the code in the same way, but this makes the XML clearer and more maintainable.

Note that you can also grab the dimension values from the code directly from their declaration as resources, using **int cellWidth =(int) getResources().getDimension(R.dimen.cellWidth);**. But it's better practice to keep it all in the layout and to use the code only for the parts that require logic and/or user interaction.

Quick tip

To make this more testable, you could break out the drawing and the maze generation into two separate classes, and call one from the other.

The next method, **generate**, does the bulk of the work.

```
private void generate(int x, int y) {
  visited[x][y] = true;
  while (!visited[x][y+1] || !visited[x+1][y] || !visited[x][y-1] ||
!visited[x-1][y]) {
    while (true) {
      double r = Math.random();
      if (r < 0.25 && !visited[x][y-1]) {
        north[x][y] = south[x][y-1] = false;
        generate(x, y-1);
        break;
      }
      // see code DVD for the rest of the method
    }
  }
}
```

The method is called as **generate(1,1)**, so we start off in the entrance cell on the top left corner, and set that as 'visited'. Next, look for any unvisited neighbouring cells. Assuming one of them is indeed unvisited, pick one neighbour at random (using **Math.random()**) and, if it hasn't been visited, knock both walls down between these two cells (the one belonging to the current cell, and the one belonging to the neighbour cell). Move into the neighbour cell, and run **generate()** on that cell. Eventually, this recurses through all of the available cells. Every cell has been visited, and every cell can be reached from one cell before it in the generation tree. Thus, there is a tree from the first cell to any other cell; and specifically, a tree (a path) to the exit cell.

Finally, we need to draw it on the screen. **onDraw** is a method which any View must have, which will be called automatically whenever the View is drawn or regenerated.

```
protected void onDraw(Canvas canvas) {
  paint.setColor(Color.YELLOW);
  for (int x = 1; x <= width; x++) {
    for (int y = 1; y <= height; y++) {
      if (south[x][y]) {
        canvas.drawLine(cellWidth*(x-1),
                        cellWidth*y,
                            cellWidth*x,
                            cellWidth*y,
                            paint);
              }
      // see code on DVD for other directions
    }
  }
}
```

Pick any visible colour for your maze, then run through each cell, drawing any walls it has. To get the pixel location of each cell, we multiply its grid reference by the cell width.

Quick tip

It's not easy to find out what formats are valid for custom attributes. The list is: __reference__, __string__, __color__, __dimension__, __boolean__, __integer__, __float__, __fraction__, __enum__, and __flag__. (With thanks to Bill Woody at http://chaosinmotion.com/blog/?p=179)

Moving the ball around

Next, we have to generate the ball and move it around. We don't want to redraw the maze background every time the ball moves, so we use a separate View, **MazeBall**. The constructor looks like this:

```
public MazeBall(Context context, AttributeSet attrs) {
  super(context);
  initVars(attrs);
  init();
}
```

initVars works in much the same way as the **initVars** method in the **MazeView** class, to get our custom attributes (here, cell width and ball diameter) from the XML layout. We want to draw the two views on top of each other, so **res/layout/main.xml** should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout .... >
  <com.example.maze.MazeView .... />
  <com.example.maze.MazeBall
            id="@+id/mazeball"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
    maze:cellWidth="@dimen/cellWidth"
    maze:ballDiam="@dimen/ballDiam"
    />
</FrameLayout>
```

This is a good reason to use the **dimen** resources rather than hard-coding them in **main.xml**; they're used in both the **MazeView** and the **MazeBall** elements.

To initialise the ball, we draw it in the top left corner of the maze:

```
private ShapeDrawable ball;
....
private void init() {
  ball = new ShapeDrawable(new OvalShape());
  ball.getPaint().setColor(Color.GREEN);
  ball.setBounds(0, 1, diameter, diameter+1);
}
```

ShapeDrawables are drawn by creating a bounding box for them by specifying the top left and lower right corners of the box, as in the **setBounds** line here.

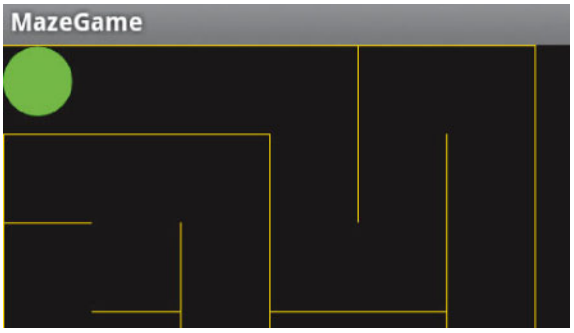
Compile and run this and you'll get a nice maze, with a ball sitting in the top left corner. Next we need to get the ball to move around when the device is tilted.

Set **MazeActivity** up to listen for sensor events, by implementing **SensorEventListener** and getting a sensor:

```
public class MazeActivity extends Activity implements
SensorEventListener
{
  private SensorManager manager;
  private Sensor accel;

  public void onCreate(Bundle savedInstanceState) {
            ....
            manager = (SensorManager)
getSystemService(SENSOR_SERVICE);
            accel = manager.getDefaultSensor(Sensor.TYPE_
ACCELEROMETER);
  }
```

To implement **SensorEventListener**, we also need to write **onResume**, **onPause**, and **onAccuracyChanged** methods. As



› Our maze with the ball ready to go at the start.

these were covered in the last tutorial, we won't go into them here – check out the code on the disc to see them.

The **onSensorChanged** method is the one that picks up the sensor values and does something with them. Declare the **MazeBall** private variable at the top of the **MazeActivity** class, then add a line to **onCreate** to grab the View.

```
private MazeBall ball;
...
//in onCreate
ball = (MazeBall)findViewById(R.id.mazeball);
```

The **onSensorChanged** method itself is straightforward:

```
public void onSensorChanged(SensorEvent event) {
  ball.moveBall(event.values[0], event.values[1]);
  ball.invalidate();
}
```

The first line passes the X and Y sensor values across to the **moveBall** method, and the second line kicks off the redrawing of the **MazeBall** view. Note that we don't touch the maze itself.

The **moveBall** method in **MazeBall** moves the ball's bounding box according to the values passed in:

```
protected void moveBall(float x, float y) {
  topX = topX - (int)x;
  topY = topY + (int)y;
  ball.setBounds(topX, topY, diameter+topX, diameter+topY);
}
```

Compile with **ant debug**, plug your device into the USB port with USB debugging turned on, and install the app with **adb -d install -r bin/maze-debug.apk** to try it out.

We're still missing one more step. You can now tilt the ball around the screen, but it doesn't react to where the walls of the maze are; it just floats over them.

Remembering the maze

To check whether the ball can be moved in any particular direction, add a test to the **moveBall** method:

```
protected void moveBall(float x, float y) {
  int newX = topX - (int)x;
  int newY = topY + (int)y;
  boolean movable = canBallMove(newX, newY);
  if (movable) {
    topX = newX;
    topY = newY;
  }
  else { // ball stays where it is }
  ball.setBounds(topX, topY, diameter+topX, diameter+topY);
}
```

The **canBallMove** method is moderately logically complicated. We compare the whereabouts of the middle of the ball (the 'hotspot') between the old position and the new:

```
int newMidX = x + diameter/2;
int newMidY = y + diameter/2;
int oldCellX = (topX + diameter/2)/cellWidth + 1;
int oldCellY = (topY + diameter/2)/cellWidth + 1;
int newCellX = newMidX/cellWidth + 1;
int newCellY = newMidY/cellWidth + 1;
```

You can get the grid reference (as used in the **MazeView** setup) of any given point by dividing its pixel value by the maze cell width, then adding 1 (since the [0][x] and [x][0] cells form the borders of the maze).

The next section of the code checks whether any of the rest of the ball is in a different cell from its hotpoint:

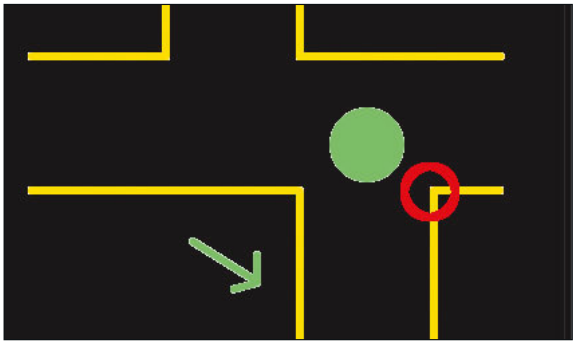
```
if ((newMidX % cellWidth) + diameter/2 >= cellWidth) {
newCellX++; }
if ((newMidX % cellWidth) - diameter/2 <= 0) { newCellX--; }
... repeat with Y values ...
```

The **%** is the modulo divisor, which we use to find out where in the cell the midpoint is. We can then work out whether the edges of the bounding box overlap a new cell. If they do, the next question is whether there's a wall between:

```
return !(MazeActivity.maze.isWall(oldCellX, oldCellY,
newCellX, newCellY));
```

All the cell and wall information is kept in the **MazeView**, so we call over to that. (Note that **isWall** returns true if there *is* a wall, whereas **canBallMove** must return true if there *isn't* a wall. You could rename one of these and have the return values both be the same, if you prefer.) If the ball is moving north, west, south or east, it's easy to check:

```
if    ((oldX == newX) && (oldY == newY)) { wallPresent = false; }
```



› **There's a wall where the red ring is, but no wall to the north or east of the ball.**

```
else if (newX == oldX - 1) { wallPresent = west[oldX][oldY]; }
...
else { wallPresent = false; }
```

The slightly more complicated part involves checking the diagonal corners. As in the picture, if a ball is moving diagonally north-east, there might be no wall to the north or east of that cell, but walls at the corner-adjacent cell. We need to check for those as well. Here's a sample of that code (check out the disc for the remainder):

```
if ((oldX != newX) && (oldY != newY)) {
  if ((newX == oldX + 1) && (newY == oldY + 1) &&
     (north[newX][newY] || west[newX][newY]) ) {
    wallPresent = true;
  } ....
}
```

If the new cell that the ball is heading into is south-east of the old cell (remember that on this grid, we count down and right from the origin in the top left corner, to match up with the Android 2D co-ordinates), we check the north and west walls of this new, diagonally adjacent, cell.

The same check is then repeated for the other three diagonally-adjacent cells. Once all that's returned, **canBallMove** also returns true or false, and the ball moves accordingly. Compile and run, and you should see the ball stopping whenever you tilt it into a wall. LXF

Quick tip

The problem with this code is that we're effectively using the four corners of the box bounding the (round) ball to check whether it's bumping into walls. This can give rise to slightly odd behaviour.

Final tutorial

Next tutorial, we'll have a look at the start and end of a game; on-screen pop-ups; timing and keeping score; fitting the maze to the screen and making it all a bit smoother and threading and UI.