

**SWE 6120 – Mobile Applications Development**

**Lab#5 - Adding Graphics to the Sudoku Game**

Given on: 9th June 2016

Lecturer: Dr. G. Chege [gchege@usiu.ac.ke;](mailto:gchege@usiu.ac.ke;)

Here we continue to build the Sudoku game first by adding some graphics to the user interface, then the game logic. Recall the target User Interface is as shown in the figure below.



At the end of Lab#2, the Sudoku game had an opening screen, an About dialog box, and a way to start a new game. But it was missing one very important part: the game! We’ll use the native 2D graphics library to implement that part.

Starting the Game first we need to ﬁll in the code that starts the game. ***startGame()*** takes one parameter, the index of the difﬁculty name selected from the list. Here’s the new deﬁnition:

/\*\* Start a new game with the given difficulty level \*/

private void startGame(int i) {

Log.d(TAG, "clicked on " + i);

Intent intent = new Intent(Sudoku.this, Game.class); intent.putExtra(Game.KEY\_DIFFICULTY, i);

startActivity(intent);

}

The game part of Sudoku will be another activity called Game, so we create a new intent to kick it off. We place the difﬁculty number in an extraData area provided in the intent, and then we call the startActivity() method to launch the new activity.

The extraData area is a map of key/value pairs that will be passed along to the intent. The keys are strings, and the values can be any primitive type, array of primitives, Bundle, or a subclass of Serializable or Parcelable.

Deﬁning the Game Class Here’s the outline of the Game activity:

package com.example.sudoku;

import android.app.Activity;

import android.app.Dialog;

import android.os.Bundle;

import android.util.Log;

import android.view.Gravity;

import android.widget.Toast;

public class Game extends Activity {

private static final String TAG = "Sudoku";

public static final String KEY\_DIFFICULTY =

"com.example.sudoku.difficulty";

public static final int DIFFICULTY\_EASY = 0;

public static final int DIFFICULTY\_MEDIUM = 1; public static final int DIFFICULTY\_HARD = 2;

private int puzzle[] = new int[9 \* 9];

private PuzzleView puzzleView;

@Override

protected void onCreate(Bundle savedInstanceState){

super.onCreate(savedInstanceState);

Log.d(TAG, "onCreate");

int diff = getIntent().getIntExtra(KEY\_DIFFICULTY,

DIFFICULTY\_EASY);

puzzle = getPuzzle(diff); calculateUsedTiles();

puzzleView = new PuzzleView(this); setContentView(puzzleView); puzzleView.requestFocus();

} // ...

}

The onCreate() method fetches the difficulty number from the intent and selects a puzzle to play. Then it creates an instance of the PuzzleView class, setting the PuzzleView as the new contents of the view. Since this is a fully customized view, it was easier to do this in code than in XML.

The *calculateUsedTiles()* method, which is defined later, uses the rules of Sudoku to figure out, for each tile in the nine-by-nine grid, which numbers are not valid for the tile because they appear elsewhere in the horizontal or vertical direction or in the three-by-three subgrid.

This is an activity, so we need to register it in AndroidManifest.xml:

<activity

android:name=".Game" android:label="@string/game\_title"/>

We also need to add a few more string resources to **res/values/strings.xml**:

<string name="game\_title">Game</string>

<string name="no\_moves\_label">No moves</string>

<string name="keypad\_title">Keypad</string>

#### **Defining the PuzzleView Class**

Next we need to define the **PuzzleView** class. Instead of using an XML layout, this time let’s do it entirely in Java.

package com.example.sudoku;

import android.content.Context;

import android.graphics.Canvas;

import android.graphics.Paint;

import android.graphics.Rect;

import android.graphics.Paint.FontMetrics;

import android.graphics.Paint.Style;

import android.util.Log;

import android.view.KeyEvent;

import android.view.MotionEvent;

import android.view.View;

import android.view.animation.AnimationUtils;

public class PuzzleView extends View {

private static final String TAG = "Sudoku"; private final Game game; public PuzzleView(Context context) { super(context); this.game = (Game) context; setFocusable(true); setFocusableInTouchMode(true);

} // ...

}

In the constructor we keep a reference to the Game class and set the option to allow user input in the view. Inside PuzzleView, we need to implement the onSizeChanged() method. This is called after the view is created and Android knows how big everything is.

private float width; // width of one tile

private float height; // height of one tile

private int selX; // X index of selection

private int selY; // Y index of selection

private final Rect selRect = new Rect();

@Override

protected void onSizeChanged(int w, int h, int oldw, int oldh) {

width = w / 9f;

height = h / 9f;

getRect(selX, selY, selRect);

Log.d(TAG, "onSizeChanged:width" + width + ",height" + height);

super.onSizeChanged(w, h, oldw, oldh);

}

private void getRect(int x, int y, Rect rect) {

rect.set((int) (x \* width), (int) (y \* height), (int) (x \* width + width),

(int) (y \* height + height));

}

We use onSizeChanged() to calculate the size of each tile on the screen (1/9th of the total view width and height). Note this is a floating-point number, so it’s possible that we could end up with a fractional number of pixels. selRect is a rectangle we’ll use later to keep track of the selection cursor.

At this point we’ve created a view for the puzzle, and we know how big it is. The next step is to draw the grid lines that separate the tiles on the board.

#### Drawing the Board

Android calls a view’s onDraw( ) method every time any part of the view needs to be updated. To simplify things, onDraw( ) pretends that you’re re-creating the entire screen from scratch. In reality, you may be drawing only a small portion of the view as defined by the canvas’s clip rectangle. Android takes care of doing the clipping for you.

Start by defining a few new colors to play with in **res/values/colors.xml**:

<color name="puzzle\_background">#ffe6f0ff</color>

<color name="puzzle\_hilite">#ffffffff</color>

<color name="puzzle\_light">#64c6d4ef</color>

<color name="puzzle\_dark">#6456648f</color>

<color name="puzzle\_foreground">#ff000000</color>

<color name="puzzle\_hint\_0">#64ff0000</color>

<color name="puzzle\_hint\_1">#6400ff80</color>

<color name="puzzle\_hint\_2">#2000ff80</color>

<color name="puzzle\_selected">#64ff8000</color>

Here’s the basic outline for onDraw( ):

@Override protected void onDraw(Canvas canvas) {

// Draw the background...

Paint background = new Paint();

background.setColor(getResources().getColor(

R.color.puzzle\_background));

canvas.drawRect(0, 0, getWidth(), getHeight(), background);

// Draw the board...

// Draw the numbers...

// Draw the hints...

// Draw the selection...

}

The first parameter is the Canvas on which to draw. In this code, we’re just drawing a background for the puzzle using the puzzle\_background color.

Now let’s add the code to draw the grid lines for the board:

// Draw the board...

// Define colors for the grid lines

Paint dark = new Paint(); dark.setColor(getResources().getColor(R.color.puzzle\_dark));

Paint hilite = new Paint(); hilite.setColor(getResources().getColor(R.color.puzzle\_hilite));

Paint light = new Paint();

light.setColor(getResources().getColor(R.color.puzzle\_light));

// Draw the minor grid lines for (int i = 0; i < 9; i++) {

canvas.drawLine(0, i \* height, getWidth(), i \* height, light);

canvas.drawLine(0, i \* height + 1, getWidth(), i \* height

+ 1, hilite);

canvas.drawLine(i \* width, 0, i \* width, getHeight(), light);

canvas.drawLine(i \* width + 1, 0, i \* width + 1, getHeight(), hilite);

}

// Draw the major grid lines

for (int i = 0; i <9; i++) {

if (i % 3 != 0) continue;

canvas.drawLine(0, i \* height, getWidth(), i \* height, dark);

canvas.drawLine(0, i \* height + 1, getWidth(), i \* height

+ 1, hilite);

canvas.drawLine(i \* width, 0, i \* width, getHeight(), dark);

canvas.drawLine(i \* width + 1, 0, i \* width + 1, getHeight(), hilite);

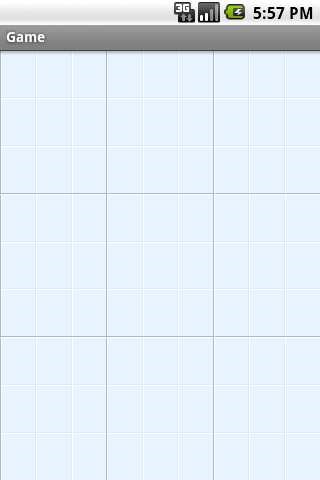
}

The code uses three different colors for the grid lines: a light color between each tile, a dark color between the three-by-three blocks, and a highlight color drawn on the edge of each tile to make them look like they have a little depth.

The order in which the lines are drawn is important, since lines drawn later will be drawn over the top of earlier lines. You can see what this will look like in the next Figure. Next, we need some numbers to go inside those lines.

#### **Drawing the Numbers**

The following code draws the puzzle numbers on top of the tiles. The tricky part here is getting each number positioned and sized so it goes in the exact center of its tile.



Drawing the grid lines with three colors for effect

// Draw the numbers...

// Define color and style for numbers

Paint foreground = new Paint(Paint.ANTI\_ALIAS\_FLAG); foreground.setColor(getResources().getColor(R.color.puzzle\_foreground));

foreground.setStyle(Style.FILL);

foreground.setTextSize(height \* 0.75f); foreground.setTextScaleX(width / height); foreground.setTextAlign(Paint.Align.CENTER);

// Draw the number in the center of the tile

FontMetrics fm = foreground.getFontMetrics();

// Centering in X: use alignment (and X at midpoint)

float x = width / 2;

// Centering in Y: measure ascent/descent first

float y = height / 2 - (fm.ascent + fm.descent) / 2;

for (int i = 0; i < 9; i++) {

for (int j = 0; j < 9; j++) {

canvas.drawText(this.game.getTileString(i, j), i

\* width + x, j \* height + y, foreground);

}

}



##### Centering the numbers inside the tiles

We call the getTileString( ) method (defined later) to find out what numbers to display. To calculate the size of the numbers, we set the font height to three-fourths the height of the tile, and we set the aspect ratio to be the same as the tile’s aspect ratio. We can’t use absolute pixel or point sizes because we want the program to work at any resolution.

To determine the position of each number, we center it in both the x and y dimensions. The x direction is easy—just divide the tile width by 2. But for the y direction, we have to adjust the starting position downward a little so that the midpoint of the tile will be the midpoint of the number instead of its baseline. We use the graphics library’s FontMetrics class to tell how much vertical space the letter will take in total, and then we divide that in half to get the adjustment. You can see the results in the next Figure.

That takes care of displaying the puzzle’s starting numbers (the givens). The next step is to allow the player to enter their guesses for all the blank spaces.

### **Handling Input**

One difference in Android programming—as opposed to, say, iPhone programming—is that Android phones come in many shapes and sizes and have a variety of input methods. They might have a keyboard, a D-pad, a touch screen, a trackball, or some combination of these. A good Android program, therefore, needs to be ready to support whatever input hardware is available, just like it needs to be ready to support any screen resolution.

#### **Defining and Updating the Selection**

First we’re going to implement a little cursor that shows the player which tile is currently selected. The selected tile is the one that will be modified when the player enters a number. This code will draw the selection in onDraw( ):

// Draw the selection...

Log.d(TAG, "selRect=" + selRect);

Paint selected = new Paint();

selected.setColor(getResources().getColor(R.color.puzzle\_selected)); canvas.drawRect(selRect, selected);

We use the selection rectangle calculated earlier in onSizeChanged() to draw an alpha-blended color on top of the selected tile.

Next we provide a way to move the selection by overriding the onKeyDown() method:

@Override

public boolean onKeyDown(int keyCode, KeyEvent event) {

Log.d(TAG, "onKeyDown:keycode=" + keyCode + ",event=" + event);

switch (keyCode) { case KeyEvent.KEYCODE\_DPAD\_UP: select(selX, selY - 1);

break;

case KeyEvent.KEYCODE\_DPAD\_DOWN:

select(selX, selY + 1);

break;

case KeyEvent.KEYCODE\_DPAD\_LEFT:

select(selX - 1, selY);

break;

case KeyEvent.KEYCODE\_DPAD\_RIGHT:

select(selX + 1, selY);

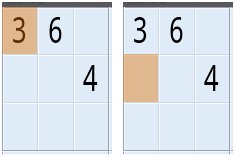
break;

default:

return super.onKeyDown(keyCode, event);

} return true;

}



##### Drawing and moving the selection

If the user has a directional pad (D-pad) and they press the up, down, left, or right button, we call select( ) to move the selection cursor in that direction.

How about a trackball? We could override the onTrackballEvent() method, but it turns out that if you don’t handle trackball events, Android will translate them into D-pad events automatically. Therefore, we can leave it out for this example.

Inside the select( ) method, we calculate the new x and y coordinate of the selection and then use getRect( ) again to calculate the new selection rectangle:

private void select(int x, int y) {

invalidate(selRect); selX = Math.min(Math.max(x, 0), 8);

selY = Math.min(Math.max(y, 0), 8);

getRect(selX, selY, selRect);

invalidate(selRect);

}

Notice the two calls to invalidate( ). The first one tells Android that the area covered by the old selection rectangle (on the left of the Figure above) needs to be redrawn. The second invalidate( ) call says that the new selection area (on the right of the figure) needs to be redrawn too. We don’t actually draw anything here.

This is an important point: never call any drawing functions except in the onDraw( ) method. Instead, you use the invalidate( ) method to mark rectangles as *dirty*. The window manager will combine all the dirty rectangles at some point in the future and call onDraw( ) again for you. The dirty rectangles become the clip region, so screen updates are optimized to only those areas that change.

Now let’s provide a way for the player to enter a new number on the selected tile.

#### **Entering Numbers**

To handle keyboard input, we just add a few more cases to the onKeyDown() method for the numbers 0 through 9 (0 or space means erase the number).

|  |  |
| --- | --- |
| case KeyEvent.KEYCODE\_0:  case KeyEvent.KEYCODE\_SPACE: setSelectedTile(0); break; | |
| case KeyEvent.KEYCODE\_1: | setSelectedTile(1); break; |
| case KeyEvent.KEYCODE\_2: | setSelectedTile(2); break; |
| case KeyEvent.KEYCODE\_3: | setSelectedTile(3); break; |
| case KeyEvent.KEYCODE\_4: | setSelectedTile(4); break; |
| case KeyEvent.KEYCODE\_5: | setSelectedTile(5); break; |
| case KeyEvent.KEYCODE\_6: | setSelectedTile(6); break; |
| case KeyEvent.KEYCODE\_7: | setSelectedTile(7); break; |
| case KeyEvent.KEYCODE\_8: | setSelectedTile(8); break; |
| case KeyEvent.KEYCODE\_9: | setSelectedTile(9); break; |

case KeyEvent.KEYCODE\_ENTER:

case KeyEvent.KEYCODE\_DPAD\_CENTER: game.showKeypadOrError(selX, selY); break;

To support the D-pad, we check for the Enter or center D-pad button in onKeyDown() and have it pop up a keypad that lets the user select which number to place.

For touch, we override the onTouchEvent( ) method and show the same keypad, which will be defined later:

@Override

public boolean onTouchEvent(MotionEvent event) {

if (event.getAction() != MotionEvent.ACTION\_DOWN)

return super.onTouchEvent(event);

select((int) (event.getX() / width), (int) (event.getY() / height)); game.showKeypadOrError(selX, selY);

Log.d(TAG, "onTouchEvent:x" + selX + ",y" + selY);

return true;

}

Ultimately, all roads will lead back to a call to setSelectedTile( ) to change the number on a tile:

public void setSelectedTile(int tile) {

if (game.setTileIfValid(selX, selY, tile)) {

invalidate(); // may change hints

} else {

// Number is not valid for this tile

Log.d(TAG, "setSelectedTile:invalid:" + tile);

} }

The showKeypadOrError() and setTileIfValid( ) methods will be defined in later in the lab. Note the call to invalidate( ) with no parameters. That marks the whole screen as dirty, which violates good practice However, in this case, it’s necessary because any new numbers added or removed might change the hints that we are about to implement in the next section.

#### **Adding Hints**

How can we help the player out a little without solving the whole puzzle for them? How about if we draw the background of each tile differently depending on how many possible moves it has. Add this to onDraw( ) before drawing the selection:

// Draw the hints...

// Pick a hint color based on #moves left

Paint hint = new Paint();

int c[] = { getResources().getColor(R.color.puzzle\_hint\_0), getResources().getColor(R.color.puzzle\_hint\_1), getResources().getColor(R.color.puzzle\_hint\_2), };

Rect r = new Rect(); for (int i = 0; i < 9; i++) {

for (int j = 0; j < 9; j++) {

int movesleft = 9 - game.getUsedTiles(i, j).length;

if (movesleft < c.length) {

getRect(i, j, r);

hint.setColor(c[movesleft]);

canvas.drawRect(r, hint);

}

}

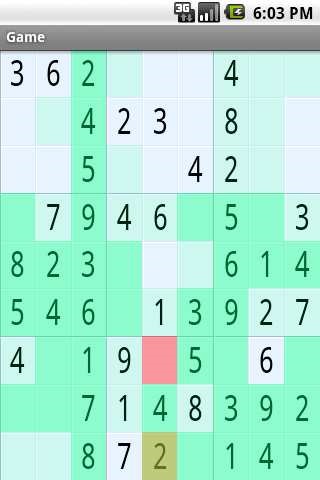
}

We use three states for zero, one, and two possible moves. If there are zero moves, that means the player has done something wrong and needs to backtrack.

The result will look like the next Figure. Can you spot the mistake(s) made by the player?[[1]](#footnote-1)

#### **Shaking Things Up**

What if the user tries to enter an obviously invalid number, such as a number that already appears in the three-by-three block? Just for fun, let’s make the screen wiggle back and forth when they do that. First we add a call to the invalid number case in setSelectedTile():



Tiles are highlighted based on how many possible values the tile can have.

// Number is not valid for this tile

Log.d(TAG, "setSelectedTile:invalid:" + tile);

startAnimation(AnimationUtils.loadAnimation(game, R.anim.shake));

This loads and runs a resource called R.anim.shake, defined in res/anim/ shake.xml, that shakes the screen for 1,000 milliseconds (1 second) by 10 pixels from side to side.

<?xml version="1.0" encoding="utf-8"?>

<translate xmlns:android="http://schemas.android.com/apk/res/android" android:fromXDelta="0" android:toXDelta="10" android:duration="1000" android:interpolator="@anim/cycle\_7" />

The number of times to run the animation and the velocity and acceleration of the animation are controlled by an animation interpolator defined in the XML File.

<?xml version="1.0" encoding="utf-8"?>

<cycleInterpolator xmlns:android="http://schemas.android.com/apk/res/android" android:cycles="7" />

This particular one will cause the animation to be repeated seven times.

### The Rest of the Game

Now let’s go back and tie up a few loose ends, starting with the Keypad class. These pieces are necessary for the program to compile and operate but have nothing to do with graphics.

#### **Creating the Keypad**

The keypad is handy for phones that don’t have keyboards. It displays a grid of the numbers 1 through 9 in an activity that appears on top of the puzzle. The whole purpose of the keypad dialog box is to return a number selected by the player.

Here’s the user interface layout from res/layout/keypad.xml:

<?xml version="1.0" encoding="utf-8"?>

<TableLayout xmlns:android="http://schemas.android.com/apk/res/android" android:id="@+id/keypad" android:orientation="vertical" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:stretchColumns="\*">

<TableRow>

<Button android:id="@+id/keypad\_1" android:text="1">

</Button>

<Button android:id="@+id/keypad\_2" android:text="2">

</Button>

<Button android:id="@+id/keypad\_3" android:text="3">

</Button>

</TableRow>

<TableRow>

<Button android:id="@+id/keypad\_4"

android:text="4">

</Button>

<Button android:id="@+id/keypad\_5" android:text="5">

</Button>

<Button android:id="@+id/keypad\_6" android:text="6">

</Button>

</TableRow>

<TableRow>

<Button android:id="@+id/keypad\_7" android:text="7">

</Button>

<Button android:id="@+id/keypad\_8" android:text="8">

</Button>

<Button android:id="@+id/keypad\_9" android:text="9">

</Button>

</TableRow>

</TableLayout>

Next let’s define the Keypad class. Here’s the outline:

package com.example.sudoku;

import android.app.Dialog;

import android.content.Context;

import android.os.Bundle;

import android.view.KeyEvent;

import android.view.View;

public class Keypad extends Dialog {

protected static final String TAG = "Sudoku";

private final View keys[] = new View[9]; private View keypad;

private final int useds[]; private final PuzzleView puzzleView;

public Keypad(Context context, int useds[], PuzzleView puzzleView) { super(context); this.useds = useds; this.puzzleView = puzzleView;

}

@Override

protected void onCreate(Bundle savedInstanceState) {



Invalid values are hidden in the keypad view.

super.onCreate(savedInstanceState);

setTitle(R.string.keypad\_title); setContentView(R.layout.keypad); findViews(); for (int element : useds) {

if (element != 0) keys[element - 1].setVisibility(View.INVISIBLE);

}

setListeners();

}

// ...

}

If a particular number is not valid (for example, the same number already appears in that row), then we make the number invisible in the grid so the player can’t select it (see Figure above).

The findViews( ) method fetches and saves the views for all the keypad keys and the main keypad window:

private void findViews() {

keypad = findViewById(R.id.keypad);

keys[0] = findViewById(R.id.keypad\_1);

keys[1] = findViewById(R.id.keypad\_2);

keys[2] = findViewById(R.id.keypad\_3);

keys[3] = findViewById(R.id.keypad\_4);

keys[4] = findViewById(R.id.keypad\_5);

keys[5] = findViewById(R.id.keypad\_6);

keys[6] = findViewById(R.id.keypad\_7);

keys[7] = findViewById(R.id.keypad\_8);

keys[8] = findViewById(R.id.keypad\_9);

}

setListeners( ) loops through all the keypad keys and sets a listener for each one. It also sets a listener for the main keypad window:

private void setListeners() {

for (int i = 0; i < keys.length; i++) {

final int t = i + 1;

keys[i].setOnClickListener(new View.OnClickListener(){

public void onClick(View v) { returnResult(t);

}});

}

keypad.setOnClickListener(new View.OnClickListener(){

public void onClick(View v) { returnResult(0);

}});

}

When the player selects one of the buttons on the keypad, it calls the returnResult( ) method with the number for that button. If the player selects a place that doesn’t have a button, then returnResult( ) is called with a zero, indicating the tile should be erased.

onKeyDown() is called when the player uses the keyboard to enter a number:

@Override

public boolean onKeyDown(int keyCode, KeyEvent event) {

int tile = 0;

switch (keyCode) {

case KeyEvent.KEYCODE\_0:

case KeyEvent.KEYCODE\_SPACE: tile = 0; break;

|  |  |
| --- | --- |
| case KeyEvent.KEYCODE\_1: | tile = 1; break; |
| case KeyEvent.KEYCODE\_2: | tile = 2; break; |
| case KeyEvent.KEYCODE\_3: | tile = 3; break; |
| case KeyEvent.KEYCODE\_4: | tile = 4; break; |
| case KeyEvent.KEYCODE\_5: | tile = 5; break; |
| case KeyEvent.KEYCODE\_6: | tile = 6; break; |
| case KeyEvent.KEYCODE\_7: | tile = 7; break; |
| case KeyEvent.KEYCODE\_8: | tile = 8; break; |
| case KeyEvent.KEYCODE\_9: | tile = 9; break; |

default:

return super.onKeyDown(keyCode, event);

}

if (isValid(tile)) { returnResult(tile);

} return true;

}

If the number is valid for the current tile, then it calls returnResult( ); otherwise, the keystroke is ignored.

The isValid( ) method checks to see whether the given number is valid for the current position:

private boolean isValid(int tile) {

for (int t : useds) {

if (tile == t)

return false;

}

Return true;

}

If it appears in the used array, then it’s not valid because the same number is already used in the current row, column, or block. The returnResult( ) method is called to return the number selected to the calling activity:

/\*\* Return the chosen tile to the caller \*/ private void returnResult(int tile) { puzzleView.setSelectedTile(tile); dismiss();

}

We call the PuzzleView.setSelectedTile method to change the puzzle’s current tile. The dismiss call terminates the Keypad dialog box. Now that we have the activity, let’s call it in the Game class and retrieve the result:

/\*\* Open the keypad if there are any valid moves \*/ protected void showKeypadOrError(int x, int y) {

int tiles[] = getUsedTiles(x, y);

if (tiles.length == 9) {

Toast toast = Toast.makeText(this, R.string.no\_moves\_label, Toast.LENGTH\_SHORT);

toast.setGravity(Gravity.CENTER, 0, 0); toast.show();

} else {

Log.d(TAG, "showKeypad:used=" + toPuzzleString(tiles));

Dialog v = new Keypad(this, tiles, puzzleView); v.show();

}

}

To decide which numbers are possible, we pass the Keypad a string in the extraData area containing all the numbers that have already been used.

#### **Implementing the Game Logic**

The rest of the code in Game.java concerns itself with the logic of the game, in particular with determining which are and aren’t valid moves according to the rules. The setTileIfValid( ) method is a key part of that. Given an x and y position and the new value of a tile, it changes the tile only if the value provided is valid.

/\*\* Change the tile only if it's a valid move \*/ protected boolean setTileIfValid(int x, int y, int value) {

int tiles[] = getUsedTiles(x, y);

if (value != 0) {

for (int tile : tiles) {

if (tile == value) return false;

}

}

setTile(x, y, value); calculateUsedTiles(); return true;

}

To detect valid moves, we create an array for every tile in the grid. For each position, it keeps a list of filled-in tiles that are currently visible from that position. If a number appears on the list, then it won’t be valid for the current tile.

The getUsedTiles() method retrieves that list for a given tile position:

/\*\* Cache of used tiles \*/

private final int used[][][] = new int[9][9][];

/\*\* Return cached used tiles visible from the given coords \*/

protected int[] getUsedTiles(int x, int y) {

return used[x][y];

}

The array of used tiles is somewhat expensive to compute, so we cache the array and recalculate it only when necessary by calling calculateUsedTiles():

/\*\* Compute the two dimensional array of used tiles \*/

private void calculateUsedTiles() { for (int x = 0; x < 9; x++) {

for (int y = 0; y < 9; y++) {

used[x][y] = calculateUsedTiles(x, y);

// Log.d(TAG, "used[" + x + "][" + y + "] = "

// + toPuzzleString(used[x][y]));

}

}

}

calculateUsedTiles( ) simply calls calculateUsedTiles(x, y) on every position in the nine-by-nine grid:

/\*\* Compute the used tiles visible from this position \*/

private int[] calculateUsedTiles(int x, int y) {

int c[] = new int[9];

// horizontal

for (int i = 0; i < 9; i++) {

if (i == y)

continue;

int t = getTile(x, i);

if (t != 0)

c[t - 1] = t;

}

// vertical

for (int i = 0; i < 9; i++) {

if (i == x)

continue;

int t = getTile(i, y);

if (t != 0)

c[t - 1] = t;

}

// same cell block

int startx = (x / 3) \* 3;

int starty = (y / 3) \* 3;

for (int i = startx; i < startx + 3; i++) {

for (int j = starty; j < starty + 3; j++) {

if (i == x && j == y)

continue;

int t = getTile(i, j);

if (t != 0)

c[t - 1] = t;

}

}

// compress

int nused = 0;

for (int t : c) {

if (t != 0)

nused++;

}

int c1[] = new int[nused];

nused = 0;

for (int t : c) {

if (t != 0)

c1[nused++] = t;

}

return c1;

}

We start with an array of nine zeros. On line 5, we check all the tiles on the same horizontal row as the current tile, and if a tile is occupied, we stuff its number into the array:

On line 13, we do the same thing for all the tiles on the same vertical column, and on line 21, we do the same for tiles in the three-by-three block.

The last step, starting at line 33, is to compress the zeros out of the array before we return it. We do this so that array.length can be used to quickly tell how many used tiles are visible from the current position.

#### **Miscellaneous**

Here are a few other utility functions and variables that round out the implementation. easyPuzzle, mediumPuzzle, and hardPuzzle are our hardcoded Sudoku puzzles for easy, medium, and hard difficulty levels, respectively.

private final String easyPuzzle = "360000000004230800000004200" + "070460003820000014500013020" + "001900000007048300000000045";

private final String mediumPuzzle =

"650000070000506000014000005" + "007009000002314700000700800" +

"500000630000201000030000097";

private final String hardPuzzle = "009000000080605020501078000" +

"000000700706040102004000000" + "000720903090301080000000600";

getPuzzle( ) simply takes a difficulty level and returns a puzzle:

/\*\* Given a difficulty level, come up with a new puzzle \*/

private int[] getPuzzle(int diff) {

String puz;

// TODO: Continue last game switch (diff) {

case DIFFICULTY\_HARD: puz = hardPuzzle; break;

case DIFFICULTY\_MEDIUM: puz = mediumPuzzle;

break;

case DIFFICULTY\_EASY: default: puz = easyPuzzle; break;

}

return fromPuzzleString(puz);

}

Later we’ll change getPuzzle( ) to implement a continue function. toPuzzleString( ) converts a puzzle from an array of integers to a string. fromPuzzleString( ) does the opposite.

/\*\* Convert an array into a puzzle string \*/ static private String toPuzzleString(int[] puz) { StringBuilder buf = new StringBuilder();

for (int element : puz) { buf.append(element);

}

return buf.toString();

}

/\*\* Convert a puzzle string into an array \*/

static protected int[] fromPuzzleString(String string) {

int[] puz = new int[string.length()];

for (int i = 0; i < puz.length; i++) {

puz[i] = string.charAt(i) - '0';

}

return puz;

}

The getTile( ) method takes x and y positions and returns the number currently occupying that tile. If it’s zero, that means the tile is blank.

/\*\* Return the tile at the given coordinates \*/

private int getTile(int x, int y) { return puzzle[y \* 9 + x];

}

/\*\* Change the tile at the given coordinates \*/

private void setTile(int x, int y, int value) { puzzle[y \* 9 + x] = value;

}

getTileString( ) is used when displaying a tile. It will return either a string with the value of the tile or an empty string if the tile is blank.

/\*\* Return a string for the tile at the given coordinates \*/

protected String getTileString(int x, int y) {

int v = getTile(x, y);

if (v == 0)

return "";

else return String.valueOf(v);

}

Once all these pieces are in place, you should have a playable Sudoku game. Give it a try to verify it works.

Later on we will include some multimedia (music in the background) and local storage (where we will program the UI continue button – recall the User Interface for the game).

1. 2. The two numbers on the bottom row, middle block, are wrong. [↑](#footnote-ref-1)