**Гоняем клубок с акселерометром**

Рассмотрим пример, как гонять клубок наклонами устройства, используя акселерометр.

Создадим отдельный компонент на основе **View** на котором будет происходить основное действие игры.

package ru.alexanderklimov.amazed;

import android.content.Context;

import android.graphics.Canvas;

import android.graphics.Color;

import android.graphics.Paint;

import android.graphics.Typeface;

import android.view.View;

public class BoardView extends View {

private Context mContext;

private Canvas mCanvas;

private Paint mPaint;

private Typeface mFont = Typeface.create(Typeface.SANS\_SERIF, Typeface.BOLD);

private int mTextPadding = 10;

// Размеры площадки

private int mCanvasWidth = 0;

private int mCanvasHeight = 0;

private int mCanvasHalfWidth = 0;

private int mCanvasHalfHeight = 0;

public BoardView(Context context) {

super(context);

mContext = context;

mPaint = new Paint();

mPaint.setTextSize(22);

mPaint.setTypeface(mFont);

mPaint.setAntiAlias(true);

}

@Override

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

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

// Получим размеры площадки

mCanvasWidth = w;

mCanvasHeight = h;

mCanvasHalfWidth = w / 2;

mCanvasHalfHeight = h / 2;

}

@Override

protected void onDraw(Canvas canvas) {

super.onDraw(canvas);

// Обновим ссылку на наш холст

mCanvas = canvas;

// Сделаем площадку белой

mPaint.setColor(Color.WHITE);

mCanvas.drawRect(0, 0, mCanvasWidth, mCanvasHeight, mPaint);

}

}

Мы создали шаблон для игровой площадки, которую будем постепенно усложнять. На данный момент просто заливаем всю поверность белым цветом. Также заранее объявили несколько переменных, которые понадобятся позже: размер шрифтов, размер экрана и т.п.

Подключим игровую площадку к главной активности.

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

mBoardView = new BoardView(this);

mBoardView.setFocusable(true);

mBoardView.setKeepScreenOn(true);

setContentView(mBoardView);

}

Последний штрих - зафиксируем активность в портретном режиме.

<activity

android:name=".MainActivity"

android:label="@string/app\_name"

android:screenOrientation="portrait">

Игра готова! Ладно, я пошутил.

В предыдущих примерах мы рисовали клубок в самом классе компонента. Для более сложных игр лучше создать отдельный класс для нового объекта и реализовать там все необходимые свойства и методы. Так и поступим, создав новый класс для клубка **Ball**.

Нас интересуют в первую очередь координаты клубка, а также размер и цвет. Также сделаем ограниченное число попыток на прохождение игры, добавив метод **death()**, который будет забирать жизнь у клубка.

package ru.alexanderklimov.amazed;

import android.graphics.Canvas;

import android.graphics.Color;

import android.graphics.Paint;

import android.view.View;

public class Ball {

private int mX = 0;

private int mY = 0;

private int mRadius = 0;

private int mColor = 0;

private int mLives = 1;

private View mView;

/\*\*

\* Конструктор для клубка

\* @param view Компонент, в котором используется клубок

\*/

public Ball(View view) {

this.mView = view;

init();

}

/\* Начальные координаты клубка, цвет и радиус \*/

public void init() {

mX = mRadius \* 6;

mY = mRadius \* 6;

setRadius(8);

setColor(Color.MAGENTA);

}

/\* Рисуем клубок

\* @param canvas Холст для рисования

\* @param paint Кисть для рисования

\*/

public void draw(Canvas canvas, Paint paint) {

paint.setColor(mColor);

canvas.drawCircle(mX, mY, mRadius, paint);

}

/\* @return Текущая координата X.\*/

public int getX() {

return mX;

}

/\* @return Текущая координата Y. \*/

public int getY() {

return mY;

}

/\* Обновляем позицию клубка по оси X

\* с проверкой на допустимость (не выходит за пределы экрана).

\* @param newX Новое значение, прибавляемое к текущему.

\*/

public void setX(float newX) {

mX += newX;

// Проверяем, чтобы клубок не укатился за пределы экрана

if (mX + mRadius >= mView.getWidth())

mX = mView.getWidth() - mRadius;

else if (mX - mRadius < 0)

mX = mRadius;

}

/\*Обновляем позицию клубка по оси Y

\* с проверкой на допустимость (не выходит за пределы экрана).

\* @param newY IНовое значение, прибавляемое к текущему.

\*/

public void setY(float newY) {

mY += newY;

// Проверяем, чтобы клубок не укатился за пределы экрана

if (mY + mRadius >= mView.getHeight())

mY = mView.getHeight() - mRadius;

else if (mY - mRadius < 0)

mY = mRadius;

}

public void setColor(int color) {

this.mColor = color;

}

public void setRadius(int radius) {

this.mRadius = radius;

}

/\*Число попыток для прохождения всей игры

\* @param val Число жизней

\*/

public void setLives(int val) {

mLives = val;

}

/\*@return Число оставшихся жизней

\*/

public int getLives() {

return mLives;

}

/\*Клубок умер\*/

public void death() {

mLives--;

}

}

Проверим, выводится ли клубок на экране. Инициализируем объект Клубок в классе **BoardView**:

private Ball mBall;

Инициализируем клубок в конструкторе площадки:

public BoardView(Context context) {

super(context);

...

// Инициализируем клубок

mBall = new Ball(this);

}

Выводим клубок в центре площадки.

@Override

protected void onDraw(Canvas canvas) {

super.onDraw(canvas);

...

mBall.setX(mCanvasHalfWidth);

mBall.setY(mCanvasHalfHeight);

mBall.draw(mCanvas, mPaint);

}

Проверяем и убеждаемся, что клубок выводится в центре игровой площадки.

**Подключаем акселерометр**

Пора подключать акселерометр и пробовать управлять клубком через наклоны. Перечитываем статью об [акселерометре](http://developer.alexanderklimov.ru/android/sensor-accelerometer.php) и подключаем сенсор к проекту.

package ru.alexanderklimov.amazed;

import android.content.Context;

import android.hardware.Sensor;

import android.hardware.SensorEvent;

import android.hardware.SensorEventListener;

import android.hardware.SensorManager;

import android.support.v4.view.WindowCompat;

import android.support.v7.app.ActionBarActivity;

import android.os.Bundle;

import android.util.Log;

import android.view.Menu;

import android.view.MenuItem;

import android.view.Window;

public class MainActivity extends ActionBarActivity {

private BoardView mBoardView;

private SensorManager mSensorManager; // менеджер сенсоров

private Sensor mAccelerometerSensor; // тип сенсора

// Значения, поступаемые от сенсоров

private float mAccelX = 0;

private float mAccelY = 0;

private float mAccelZ = 0; // у нас не используется

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

mBoardView = new BoardView(this);

mBoardView.setFocusable(true);

mBoardView.setKeepScreenOn(true);

setContentView(mBoardView);

// Подключаем акселерометр

mSensorManager = (SensorManager) getSystemService(Context.SENSOR\_SERVICE);

mAccelerometerSensor = mSensorManager

.getDefaultSensor(Sensor.TYPE\_ACCELEROMETER);

// Регистрируем акселерометр для получения данных

mSensorManager.registerListener(sensorEventListener, mAccelerometerSensor,

SensorManager.SENSOR\_DELAY\_GAME);

}

SensorEventListener sensorEventListener = new SensorEventListener() {

@Override

public void onSensorChanged(SensorEvent event) {

// Получаем данные от сенстора

mAccelX = event.values[0];

mAccelY = event.values[1];

mAccelZ = event.values[2];

mBoardView.updateData((int)mAccelX, (int)mAccelY);

}

@Override

public void onAccuracyChanged(Sensor sensor, int accuracy) {

// Не используется

}

};

@Override

protected void onDestroy() {

super.onDestroy();

if ( mAccelerometerSensor != null) {

mSensorManager.unregisterListener(sensorEventListener);

}

}

}

Подключив акселерометр, мы получаем данные при наклонах устройства и передаём их методу **updateData()** класса **BoardView**. Возвращаемся в этот класс, добавляем метод и другой код.

public void updateData(int x, int y){

mX = x;

mY = y;

}

public void updateBall() {

if (mX != 0) {

mBall.setX(-mX);

}

if (mY != 0) {

mBall.setY(-mY);

}

}

@Override

protected void onDraw(Canvas canvas) {

super.onDraw(canvas);

// Обновим ссылку на наш холст

mCanvas = canvas;

// Сделаем площадку белой

mPaint.setColor(Color.WHITE);

mCanvas.drawRect(0, 0, mCanvasWidth, mCanvasHeight, mPaint);

//mBall.setX(mCanvasHalfWidth);

//mBall.setY(mCanvasHalfHeight);

mBall.draw(mCanvas, mPaint);

updateBall();

invalidate();

}

Метод **updateData()** присваивает новые значения переменным **mX** и **mY**, которые затем используются в методе **updateBall()** для установки позиции клубка на экране. Я меняю у значений знак на противоположный, чтобы клубок скатывался в сторону наклона, как в реальной жизни. Если запустить проект, то можно гонять клубок по экрану.

**Строим лабиринт**

Следующим этапом станет создание лабиринта, по которому будет двигаться клубок в поисках базы. Создадим новый класс **Labyrinth**.

Построение лабиринта будет происходит через плитку, которой замостим экран. Для базы, на которую нужно прикатить клубок, создадим отдельный вид плитки другого цвета. Для этого создадим два графических ресурса в виде квадратов.

Лабиринт будет размером 20х26 плиток. Также зададим в коде и размер одной плитки. На самом деле это не слишком удачное решение. Это отчётливо стало заметно при использовании исходного кода, который писался ещё для маленьких экранов 320х240. Загрузив пример для изучения на современном смартфоне, я увидел крохотный лабиринт в углу экрана. Я не стал кардинально менять код, но вам следует позаботиться об этой проблеме и программно вычислять размер экрана и, соответственно, размер плитки. В методе **onDraw()** рисуем шесть плиток, чтобы вы поняли принцип размещения. Последняя плитка будет другого цвета.

package ru.alexanderklimov.amazed;

import android.content.Context;

import android.graphics.Bitmap;

import android.graphics.BitmapFactory;

import android.graphics.Canvas;

import android.graphics.Paint;

public class Labyrinth {

// Плитки для лабиринта

private Bitmap mPathBitmap;

private Bitmap mTargetBitmap;

// Размер плитки и самого лабиринта

private final static int TILE\_SIZE = 24; // было 16

private final static int MAZE\_COLS = 20;

private final static int MAZE\_ROWS = 26;

// Атрибуты плитки

private int mX;

private int mY;

/\*\*

\* Constructor.

\*

\* @param context Application context used to load images.

\*/

Labyrinth(Context context) {

// Загружаем картинки плиток

mPathBitmap = BitmapFactory.decodeResource(context.getApplicationContext().getResources(),

R.drawable.tile);

mTargetBitmap = BitmapFactory.decodeResource(context.getApplicationContext().getResources(),

R.drawable.target\_tile);

}

/\*\*

\* Рисуем лабиринт

\*

\* @param canvas Холст для рисования

\* @param paint Кисть для рисования

\*/

public void draw(Canvas canvas, Paint paint) {

canvas.drawBitmap(mPathBitmap, 0, 0, paint);

canvas.drawBitmap(mPathBitmap, 0, 1 \* TILE\_SIZE, paint);

canvas.drawBitmap(mPathBitmap, 0, 2 \* TILE\_SIZE, paint);

canvas.drawBitmap(mPathBitmap, 1 \* TILE\_SIZE, 0, paint);

canvas.drawBitmap(mPathBitmap, 1 \* TILE\_SIZE, 1 \* TILE\_SIZE, paint);

// Другая плитка

canvas.drawBitmap(mTargetBitmap, 1 \* TILE\_SIZE, 2 \* TILE\_SIZE, paint);

}

}

В классе **BoardView** объявляем и инициализируем объект Лабиринт и выводим плитки перед методом **mBall.draw()**, чтобы клубок рисовался поверх плиток.

private Labyrinth mLabyrinth;

public BoardView(Context context) {

super(context);

...

// Инициализируем клубок и лабиринт

mBall = new Ball(this);

mLabyrinth = new Labyrinth(mContext);

}

@Override

protected void onDraw(Canvas canvas) {

super.onDraw(canvas);

...

// Рисуем плитку лабиринта

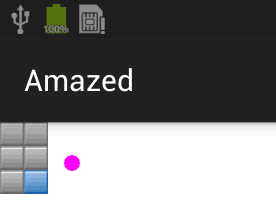
mLabyrinth.draw(mCanvas, mPaint);

// Рисуем клубок

mBall.draw(mCanvas, mPaint);

...

}



Зная, что экран будет состоять из 20 плиток в столбце и 26 плиток в ряду, мы можем замостить весь экран плитками через цикл.

for (int i = 0; i < MAZE\_COLS; i++){

for (int y = 0; y < MAZE\_ROWS; y++){

mX = i \* TILE\_SIZE;

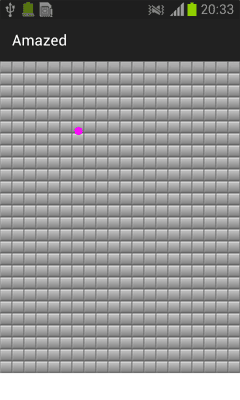
mY = y \* TILE\_SIZE;

canvas.drawBitmap(mPathBitmap, mX, mY, paint);

}

}

Код рабочий и экран будет заполнен плитками.

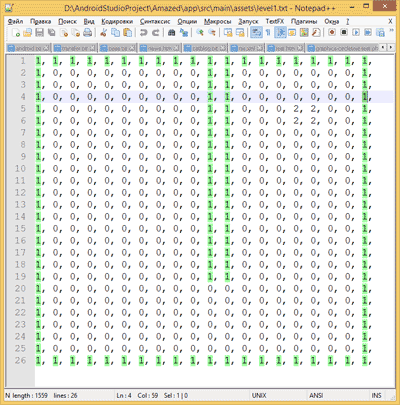


Но код не обладает нужной гибкостью. Мы пойдем другим путём. Игра будет скучной, если будет состоять из одного лабиринта. Нам нужно создать несколько уровней и подгружать их по мере прохождения. Кроме того, нужно придумать формат для хранения данных о строении лабиринта и сохранять данные в файле.

Давайте договоримся, что 0 будет отвечать за обычную плитку, 1 - стена, 2 - плитка базы.

Смысл в следующем, мы создаём в текстовом редакторе последовательность чисел, которые отвечают за позицию плиток. При считывании мы извлечём все эти числа и занесём их в массив. На основе анализа массива мы сможем восстановить внешний вид лабиринта.

Для удобства восприятия человеческим глазом данные будем разбивать на колонки и ряды, отделяя числа пробелом и запятыми. В хорошем текстовом редакторе, например, Notepad++, запись будет смотреться красиво. А если выделить какое-нибудь число, то будут подсвечены все схожие символы. Вот как выглядит первый уровень.



Сохраним уровень в текстовом файле **level1.txt** и поместим его в папку **assets**.

В классе лабиринта создадим метод для загрузки файла.

void load(Context context, int newLevel) {

// Данные хранятся в папке assets в виде level1.txt, level2.txt и т.д.

String level = "level" + newLevel + ".txt";

InputStream stream = null; // файловый поток

try {

// Конструируем массив

mMazeDataArray = new int[MAZE\_ROWS \* MAZE\_COLS];

// Открываем нужный файл

stream = context.getAssets().open(level);

// Проходим в цикле через все символы и помещаем их в массив

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

// Данные хранятся в Unicode. Конвертируем их.

mMazeDataArray[i] = Character.getNumericValue(stream.read());

// Пропускаем запятые "," и пробелы

stream.read();

stream.read();

}

} catch (Exception e) {

Log.i("Labyrinth", "Ошибка при открытии файла: " + e);

} finally {

closeStream(stream);

}

}

private static void closeStream(Closeable stream) {

if (stream != null) {

try {

stream.close();

} catch (IOException e) {

// Не используется

}

}

}

После вызова метода **load()** мы получим массив символов, по которым можно восстановить внешний вид лабиринта.

public void draw(Canvas canvas, Paint paint) {

// Проходим в цикле через элементы массива

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

// Вычисляем номер столбца и ряда для текущей плитки

mRow = i / MAZE\_COLS;

mCol = i % MAZE\_COLS;

// Конвертируем ряд и столбец в реальные координаты плитки

mX = mCol \* TILE\_SIZE;

mY = mRow \* TILE\_SIZE;

// Выводим нужный тип плитки

if (mMazeDataArray[i] == PATH\_TILE)

// Обычная плитка

canvas.drawBitmap(mPathBitmap, mX, mY, paint);

else if (mMazeDataArray[i] == EXIT\_TILE)

// База для перехода на следующий уровень

canvas.drawBitmap(mTargetBitmap, mX, mY, paint);

else if (mMazeDataArray[i] == WALL\_TILE) {

// Вместо картинки плитки-стены программно рисуем чёрный прямоугольник

mRect.left = mX;

mRect.top = mY;

mRect.right = mX + TILE\_SIZE;

mRect.bottom = mY + TILE\_SIZE;

paint.setColor(WALL\_COLOR);

canvas.drawRect(mRect, paint);

}

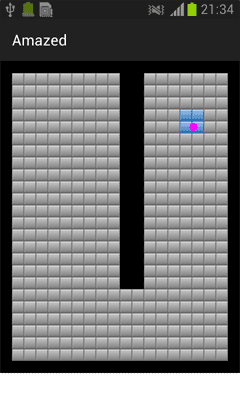
}

}

Теперь мы можем загрузить уровень и нарисовать лабиринт. Достаточно временно поместить вызов метода **load()** перед методом **draw()**.

mLabyrinth.load(mContext, mLevel);

mLabyrinth.draw(mCanvas, mPaint);



Это временное решение служит для демонстрации, что лабиринт рисуется. Но так как вызов метода **load()** происходит в методе **onDraw()**, то фактически мы постоянно открываем файл для записи при перерисовке экрана. Ничего хорошего в этом нет. Мы от этого избавимся позже.

А пока подготовим ещё девять новых уровней и сохраним их в той же папке **assets**.

Добавим новый метод, который будет увеличивать счётчик после прохождения очередного уровня.

public void initLevel() {

if (mLevel < mLabyrinth.MAX\_LEVELS) {

// Увеличиваем счётчик уровней

mWarning = true;

mLevel++;

mLabyrinth.load(mContext, mLevel);

mBall.init();

} else {

// Игрок прошёл всю игру. Обновляем состояние

switchGameState(GAME\_COMPLETE);

}

}

Переменная **MAX\_LEVELS** содержит общее количество уровней. У нас их десять. Флаг **mWarning** помогает отслеживать, что уровень пройдён и пора вывести сообщение игроку, что для продолжения игры нужно коснуться экрана.

Пора поговорить о состояниях игры. Игра может быть запущенной, завершённой, ожидающей перехода на следующий уровень. И в разных состояних используется разная логика. Поэтому нам нужно отслеживать всевозможные состояния и реагировать на них. Определим переменные для состояний игры и метод для их переключения.

// Состояния игры

private final static int NULL\_STATE = -1;

private final static int GAME\_INIT = 0;

private final static int GAME\_RUNNING = 1;

private final static int GAME\_OVER = 2;

private final static int GAME\_COMPLETE = 3;

// Текущее состояние игры

private static int mCurState = NULL\_STATE;

public void switchGameState(int newState) {

mCurState = newState;

}

В конструкторе устанавливаем стартовое состояние игры.

switchGameState(GAME\_INIT);

Создадим метод, который учитывает состояния игры.

/\*\*

\* Вызывается каждый цикл, используется для обработки текущего состояния игры.

\*/

public void gameTick() {

switch (mCurState) {

case GAME\_INIT:

// Новая игра

initNewGame();

switchGameState(GAME\_RUNNING);

case GAME\_RUNNING:

// Обновляем позицию клубка

if (!mWarning)

updateBall();

break;

}

// Перерисовываем экран при обновлении состояния игры

invalidate();

}

/\*Сбрасываем все значения для новой игры \*/

public void initNewGame() {

mBall.setLives(5);

mLevel = 0;

initLevel();

}

В нашем варианте учитываются два состояния - запускается игра и переходит в следующее состояние - игра запущена. В запущенном состоянии обновляем позицию клубка.

Возвращаемся в метод **onDraw()** и учитываем теперь возможные состояния игры.

@Override

protected void onDraw(Canvas canvas) {

super.onDraw(canvas);

// Обновим ссылку на наш холст

mCanvas = canvas;

// Сделаем площадку белой

mPaint.setColor(Color.WHITE);

mCanvas.drawRect(0, 0, mCanvasWidth, mCanvasHeight, mPaint);

// Рисуем на экране в зависимости от состояния игры

switch (mCurState) {

case GAME\_RUNNING:

// Состояние - игра запущена

// Рисуем лабиринт

mLabyrinth.draw(mCanvas, mPaint);

// Рисуем клубок

mBall.draw(mCanvas, mPaint);

// Выводим информацию

// Скоро добавим код

break;

case GAME\_OVER:

// Состояние - уровень завершен

// Скоро добавим код

break;

case GAME\_COMPLETE:

// Игра завершена

// Скоро добавим код

break;

}

gameTick();

}

Временно установите значение **true** у переменной **mWarning**, чтобы клубок мог двигаться по экрану. Убедимся, что работоспособность осталась - лабиринт первого уровня рисуется, клубок катается.

Общий каркас игры готов. Осталось довести игру до логического конца и отполировать её.

**Полировка игры**

Добавим в игру вывод информации о времени на прохождение игры, номер уровня и число жизней. Создаём метод **drawInfo()**:

private void drawInfo() {

mPaint.setColor(Color.BLUE);

mPaint.setTextAlign(Paint.Align.LEFT);

mCanvas.drawText("Время: " + (mTotalTime / 1000), mTextPadding, mCanvasHeight \* 0.96f,

mPaint);

mPaint.setTextAlign(Paint.Align.CENTER);

mCanvas.drawText("Уровень: " + mLevel, mCanvasHalfWidth, mCanvasHeight \* 0.96f,

mPaint);

mPaint.setTextAlign(Paint.Align.RIGHT);

mCanvas.drawText("Жизнь: " + mBall.getLives(), mCanvasWidth - mTextPadding,

mCanvasHeight \* 0.96f, mPaint);

// Предупреждение перед началом игры

if (mWarning) {

mPaint.setColor(Color.BLUE);

mCanvas

.drawRect(10, mCanvasHalfHeight - 25, mCanvasWidth - 10, mCanvasHalfHeight + 10,

mPaint);

mPaint.setColor(Color.WHITE);

mPaint.setTextAlign(Paint.Align.CENTER);

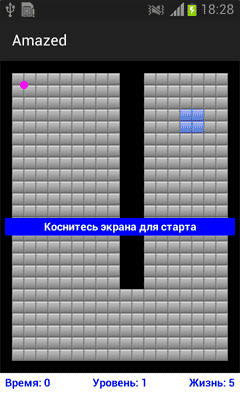
mCanvas.drawText("Коснитесь экрана для старта", mCanvasHalfWidth, mCanvasHalfHeight,

mPaint);

}

}

Метод рисует три текста внизу экрана на одной линии. А также предупреждение во время начала каждого уровня игры. Поместим вызов метода в блок **GAME\_RUNNING** метода**onDraw()**.



Чтобы убрать текст в центре экрана, нужно реализовать обработку нажатий.

@Override

public boolean onTouchEvent(MotionEvent event) {

// Нас интересует только нажатие

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

if (mCurState == GAME\_OVER || mCurState == GAME\_COMPLETE) {

// Перезапускаем игру

mCurState = GAME\_INIT;

} else if (mCurState == GAME\_RUNNING) {

// Убираем предупреждение с центра экрана для начала игры

mWarning = false;

mStartTime = System.currentTimeMillis();

}

}

return true;

}

Теперь при касании экрана флаг **mWarning** сбрасывается в значение **false**, а также начинаем отслеживать время, фиксируя начальное значение.

Игра начинается, но клубок не обращает внимания на стены лабиринта и проходит сквозь них. Непорядок. Следует отслеживать моменты, когда клубок встречается со стеной.

Добавим в класс лабиринта метод для определения типа плитки.

/\*\*

\* Определяет тип плитки, на которой находится клубок

\* @param x Текущая координата X.

\* @param y Текущая координата Y.

\* @return Возвращает тип плитки int, занятый клубком.

\*/

public int getTileType(int x, int y) {

// конвертируем координаты X, Y в ряды и столбцы.

int tileCol = x / TILE\_SIZE;

int tileRow = y / TILE\_SIZE;

// Позиция в массиве на основе координат

int location = 0;

// Умножаем на число столбцов

if (tileRow > 0)

location = tileRow \* MAZE\_COLS;

// и добавляем позицию столбца.

location += tileCol;

return mMazeDataArray[location];

}

Обновляем метод **updateBall()**, добавив проверку на тип плитки.

public void updateBall() {

if (mX != 0) {

mBall.setX(-mX);

}

if (mY != 0) {

mBall.setY(-mY);

}

// Проверяем тип плитки, которую занимает клубок

if (mLabyrinth.getTileType(mBall.getX(), mBall.getY()) == mLabyrinth.WALL\_TILE) {

// Клубок закатился на стенку.

if (mBall.getLives() > 0) {

// Если у пользователя есть жизни, то перезапускаем игру.

mBall.death();

mBall.init();

mWarning = true;

} else {

// Жизней не осталось - игра окончена

mEndTime = System.currentTimeMillis();

mTotalTime += mEndTime - mStartTime;

switchGameState(GAME\_OVER);

}

} else if (mLabyrinth.getTileType(mBall.getX(), mBall.getY()) == mLabyrinth.EXIT\_TILE) {

// Клубок докатился до базы. Запускаем новый уровень.

mEndTime = System.currentTimeMillis();

mTotalTime += mEndTime - mStartTime;

initLevel();

}

}

Как только клубок окажется на плитке типа **WALL\_TILE**, сразу сгорает жизнь и игра перезапускается (либо заканчивается). Если клубок коснулся плитки типа **EXIT\_TILE**, игрок прошёл уровень.

Фактически игра сделана. Осталось только добавить вывод сообщения при завершении игры во время прохождения уровня, когда клубок "умер" и финальное сообщение, когда игра полностью пройдена.

public void drawGameOver() {

mPaint.setColor(Color.BLACK);

mPaint.setTextAlign(Paint.Align.CENTER);

mCanvas.drawText("Игра окончена", mCanvasHalfWidth, mCanvasHalfHeight, mPaint);

mCanvas.drawText("Общее время: " + (mTotalTime / 1000) + " сек",

mCanvasHalfWidth, mCanvasHalfHeight + mPaint.getFontSpacing(), mPaint);

mCanvas.drawText("Вы прошли уровень " + (mLevel - 1) + ". Попробуйте ещё раз.",

mCanvasHalfWidth, mCanvasHalfHeight

+ (mPaint.getFontSpacing() \* 2), mPaint);

mCanvas.drawText("Коснитель экрана, чтобы продолжить", mCanvasHalfWidth, mCanvasHeight

- (mPaint.getFontSpacing() \* 3), mPaint);

}

public void drawGameComplete() {

mPaint.setColor(Color.BLACK);

mPaint.setTextAlign(Paint.Align.CENTER);

mCanvas.drawText("Конец игры", mCanvasHalfWidth, mCanvasHalfHeight, mPaint);

mCanvas.drawText("Общее время: " + (mTotalTime / 1000) + " сек.",

mCanvasHalfWidth, mCanvasHalfHeight + mPaint.getFontSpacing(), mPaint);

mCanvas.drawText("Коснитель экрана, чтобы продолжить", mCanvasHalfWidth, mCanvasHeight

- (mPaint.getFontSpacing() \* 3), mPaint);

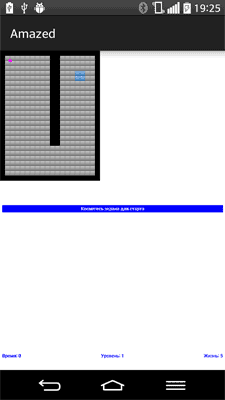
}

Данные методы вызываются в методе **onDraw()**, где были оставлены заглушки.

Теперь можно запускать игру и наслаждаться.

В статье я попытался показать последовательность действий при разработке игры с акселерометром с нуля. Естественно, при определённом опыте и квалификации часть кода можно было написать сразу, продумав процесс на этапе проектирования.

Вам осталось только сделать игру универсальной, чтобы она одинаково хорошо смотрелась на всех видах экранов. Для сравнения приведу скриншот с нового смартфона, чтобы вы увидели недостаток приложения, когда данные жёстко прописаны в коде.



Советую самостоятельно писать код по статье. Исходники можно скачать для проверки. Из-за большого размера, который генерирует студия, выложил на [Депозите](http://dfiles.ru/files/puyrfcark).