

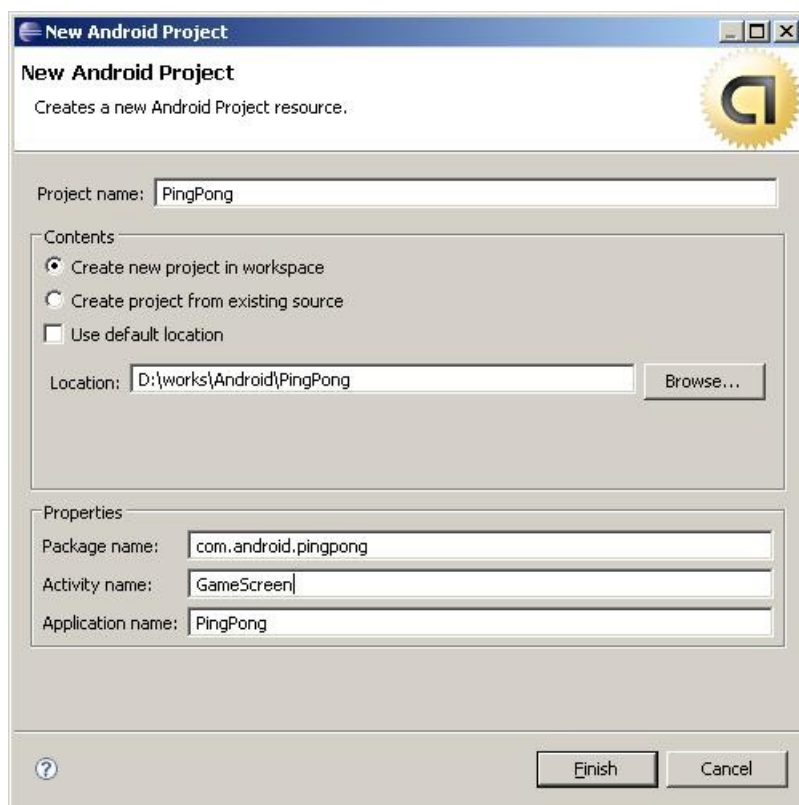
Пишем игру для Android. Часть 1. Surface

На прошлую статью про Android я получила хорошие отзывы, вдохновилась и решила продолжать. Эта серия статей будет посвящена написанию игры для Android.

Писать мы будем игру в пинг-понг. Изначально задумывался арканойд, но для мануала получалось слишком громоздко, так что я решила упростить до пинг-понга. Итак, есть прямоугольное поле, на нем две ракетки, управляемые игроками, и мячик. Мячик летает, отражаясь от ракеток и боковых стенок. Когда один игрок не успевает отбить мячик, его противнику засчитывается очко. Игра продолжается, пока один из игроков не наберет определенное число очков. Вот такую игру мы и будем писать. Одна ракетка будет управляться пользователем, другая — компьютером.

Создание проекта

Проект будем делать, как и в прошлый раз, в Eclipse. Создаём:



Получили автоматически сгенерившийся HelloWorld. На форме у нас единственный элемент управления — `TextView`. Но нам нужно разместить на форме компонент, который бы отрисовывал игровое поле и обрабатывал нажатия клавиш. Среди стандартных такого нет, так что придется создавать свой.

Surface

Прежде, чем создавать такой класс, рассмотрим некоторые базовые понятия и классы.

SurfaceView

SurfaceView унаследован от `View` и является элементом управления, предоставляющим

область для рисования (**Surface**). Суть в том, чтобы дать отдельному потоку возможность рисовать на Surface, когда он захочет, а не только тогда, когда приложению вздумается обновить экран. Понятие Surface очень похоже на Canvas, но все же немного не то. Canvas — это область рисования на компоненте, а Surface сам является компонентом, т.е. у Surface есть Canvas.

SurfaceView является элементом управления, т.е. можно его непосредственно разместить на форме. Однако, в этом случае толку толку от него будет мало. Так что мы будем писать свой класс, унаследованный от SurfaceView, а также класс для потока, который будет на нем рисовать.

SurfaceHolder

Интерфейс, с помощью которого происходит вся непосредственная работа с областью рисования. Выглядит это примерно так:

```
SurfaceHolder surfaceHolder;  
...  
Canvas canvas = surfaceHolder.lockCanvas(); // начали рисовать  
// рисуем  
surfaceHolder.unlockCanvasAndPost(canvas); // закончили рисовать
```

SurfaceHolder.Callback

Интерфейс содержит функции обработки изменения состояния Surface:

- **surfaceCreated(SurfaceHolder holder)** — первое создание Surface. Здесь можно, например, запускать поток, который будет рисовать на Surface.
- **surfaceChanged(SurfaceHolder holder, int format, int width, int height)** — любое изменение Surface (например, поворот экрана).
- **surfaceDestroyed(SurfaceHolder holder)** — уничтожение Surface. Здесь можно останавливать процесс, который рисует на Surface.

Класс для отображения игры

Итак, узнав, что такое Surface, можно двигаться дальше. Создаем класс `GameView.java`, унаследованный от `SurfaceView` и реализующий интерфейс `SurfaceHolder.Callback`. Добавим интерфейсные функции и перегрузим конструктор. Кроме того, следует завести в этом классе ссылку на `SurfaceHolder`. В результате получится что-то вроде того:

GameView.java

```
public class GameView extends SurfaceView implements SurfaceHolder.Callback  
{  
    /**  
     * Область рисования  
     */  
    private SurfaceHolder mSurfaceHolder;  
  
    /**  
     * Конструктор  
     * @param context  
     * @param attrs  
     */  
}
```

```

public GameView(Context context, AttributeSet attrs)
{
    super(context, attrs);

    // подписываемся на события Surface
    mSurfaceHolder = getHolder();
    mSurfaceHolder.addCallback(this);
}

@Override
/**
 * Изменение области рисования
 */
public void surfaceChanged(SurfaceHolder holder, int format, int width, int height)
{
}

@Override
/**
 * Создание области рисования
 */
public void surfaceCreated(SurfaceHolder holder)
{
}

@Override
/**
 * Уничтожение области рисования
 */
public void surfaceDestroyed(SurfaceHolder holder)
{
}
}

```

Теперь мы можем запросто писать в разметке формы такое:

main.xml

```

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <com.android.pingpong.GameView
        android:id="@+id/game"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"/>
</FrameLayout>

```

И, запустив программу, увидим пустой экран. Теперь давайте что-нибудь нарисуем.

Класс для рисования

Поставим себе первую цель: нарисовать на экране прямоугольное поле размером 300 x 250.

Как было уже ранее сказано, все рисование должно производиться из отдельного потока. Создадим класс, GameManager, унаследованный от Thread.

GameManager.java

```
public class GameManager extends Thread
{
    private static final int FIELD_WIDTH = 300;
    private static final int FIELD_HEIGHT = 250;

    /** Область, на которой будем рисовать */
    private SurfaceHolder mSurfaceHolder;

    /** Состояние потока (выполняется или нет. Нужно, чтобы было удобнее
прибивать поток, когда потребуется) */
    private boolean mRunning;

    /** Стили рисования */
    private Paint mPaint;

    /** Прямоугольник игрового поля */
    private Rect mField;

    /**
     * Конструктор
     * @param surfaceHolder Область рисования
     * @param context Контекст приложения
     */
    public GameManager(SurfaceHolder surfaceHolder, Context context)
    {
        mSurfaceHolder = surfaceHolder;
        mRunning = false;

        mPaint = new Paint();
        mPaint.setColor(Color.BLUE);
        mPaint.setStrokeWidth(2);
        mPaint.setStyle(Style.STROKE);

        int left = 10;
        int top = 50;
        mField = new Rect(left, top, left + FIELD_WIDTH, top + FIELD_HEIGHT);
    }

    /**
     * Задание состояния потока
     * @param running
     */
    public void setRunning(boolean running)
    {
        mRunning = running;
    }

    @Override
    /** Действия, выполняемые в потоке */
    public void run()
    {
        while (mRunning)
        {
            Canvas canvas = null;
            try
            {
                // подготовка Canvas-a
                canvas = mSurfaceHolder.lockCanvas();
                synchronized (mSurfaceHolder)
                {
                    // собственно рисование
                }
            }
        }
    }
}
```

```

        canvas.drawRect(mField, mPaint);
    }
}
catch (Exception e) { }
finally
{
    if (canvas != null)
    {
        mSurfaceHolder.unlockCanvasAndPost(canvas);
    }
}
}
}
}
}

```

Стоит отдельно упомянуть о классе `Paint`. Этот класс используется для хранения всяческих используемых при рисовании стилей — цветов, толщины и стиля линий, шрифтов (это мы рассмотрим позже) и тому подобного. В остальном код достаточно прозрачен. Собственно рисование проходит всегда одинаково — лочим `Canvas`, рисуем, разлочиваем.

Теперь надо запустить рисовательный поток в нашем контроле. Добавляем в класс соответствующее поле:

GameView.java

```

/**
 * Поток, рисующий в области
 */
private GameManager mThread;

```

В конструкторе `GameView`:

GameView.java

```

mThread = new GameManager(mSurfaceHolder, context);

```

При создании области рисования надо будет запустить наш поток:

GameView.java

```

public void surfaceCreated(SurfaceHolder holder)
{
    mThread.setRunning(true);
    mThread.start();
}

```

А при удалении — прибить:

GameView.java

```

public void surfaceDestroyed(SurfaceHolder holder)
{
    boolean retry = true;
    mThread.setRunning(false);
    while (retry)

```

```

    {
        try
        {
            // ожидание завершения потока
            mThread.join();
            retry = false;
        }
        catch (InterruptedException e) { }
    }
}

```

Теперь, запустив программу, видим следующее:



Поворот экрана

Как уже было упомянуто, при повороте экрана вызывается обработчик `surfaceChanged`. Впрочем, при создании `surface` он тоже вызывается. В параметрах можно получить размеры доступной части экрана, что очень приятно, потому что с помощью класса `DisplayMetrics` можно получить только полный размер экрана, куда еще входит верхнее поле, на котором рисовать нельзя).

Итак, в `surfaceChanged` мы будем пересчитывать положение нашего поля на экране. Добавим в `GameManager` такую функцию:

GameManager.java

```

/**
 * Инициализация положения объектов, в соответствии с размерами экрана
 * @param screenHeight Высота экрана
 * @param screenWidth Ширина экрана
 */
public void initPositions(int screenHeight, int screenWidth)
{
    int left = (screenWidth - FIELD_WIDTH) / 2;
    int top = (screenHeight - FIELD_HEIGHT) / 2;

    mField.set(left, top, left + FIELD_WIDTH, top + FIELD_HEIGHT);
}

```

Эта функция ставит наше игровое поле в центр экрана. Инициализацию положения `mField` в конструкторе `GameManager` можно вовсе убрать, оставив только:

GameManager.java

```

public GameManager(SurfaceHolder surfaceHolder, Context context)
{
    mSurfaceHolder = surfaceHolder;
    mRunning = false;

    mPaint = new Paint();
    mPaint.setColor(Color.BLUE);
    mPaint.setStrokeWidth(2);
    mPaint.setStyle(Style.STROKE);
}

```

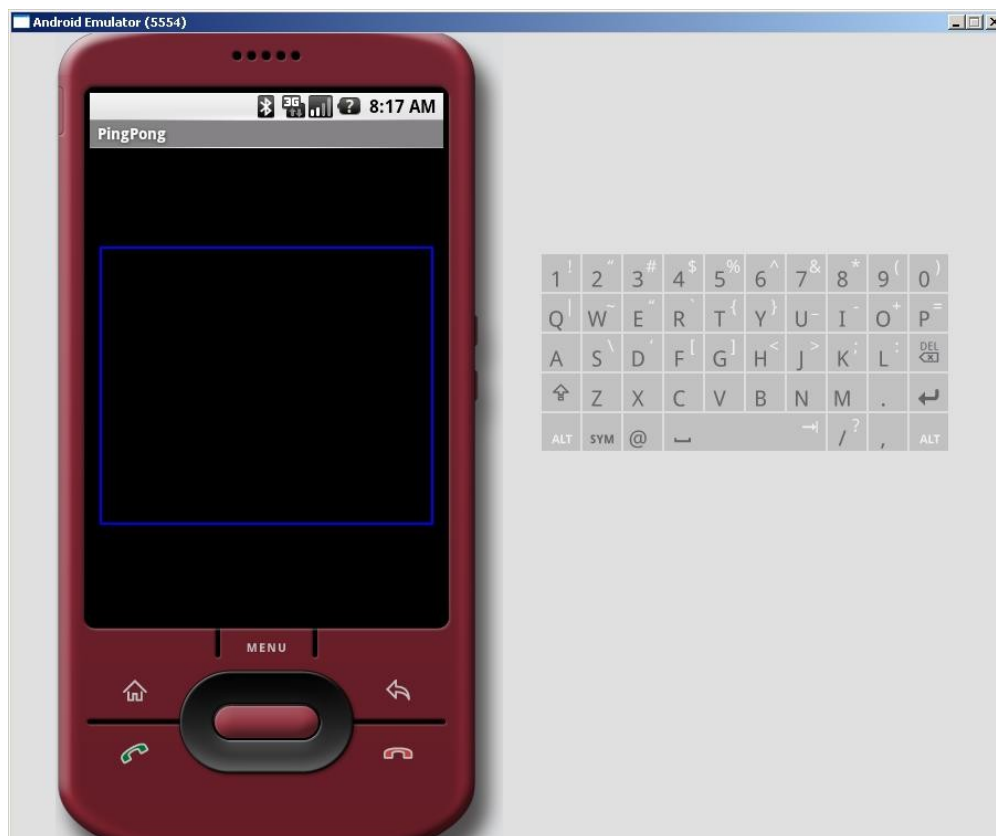
```
mField = new Rect();  
}
```

Теперь в `surfaceChanged` можно написать следующее:

GameView.java

```
@Override  
/**  
 * Изменение области рисования  
 */  
public void surfaceChanged(SurfaceHolder holder, int format, int width, int  
height)  
{  
    mThread.initPositions(height, width);  
}
```

Теперь при изменении `Surface` (в том числе при его создании) будет пересчитываться положение поля. Так что приложение будет выглядеть так:



Или так:

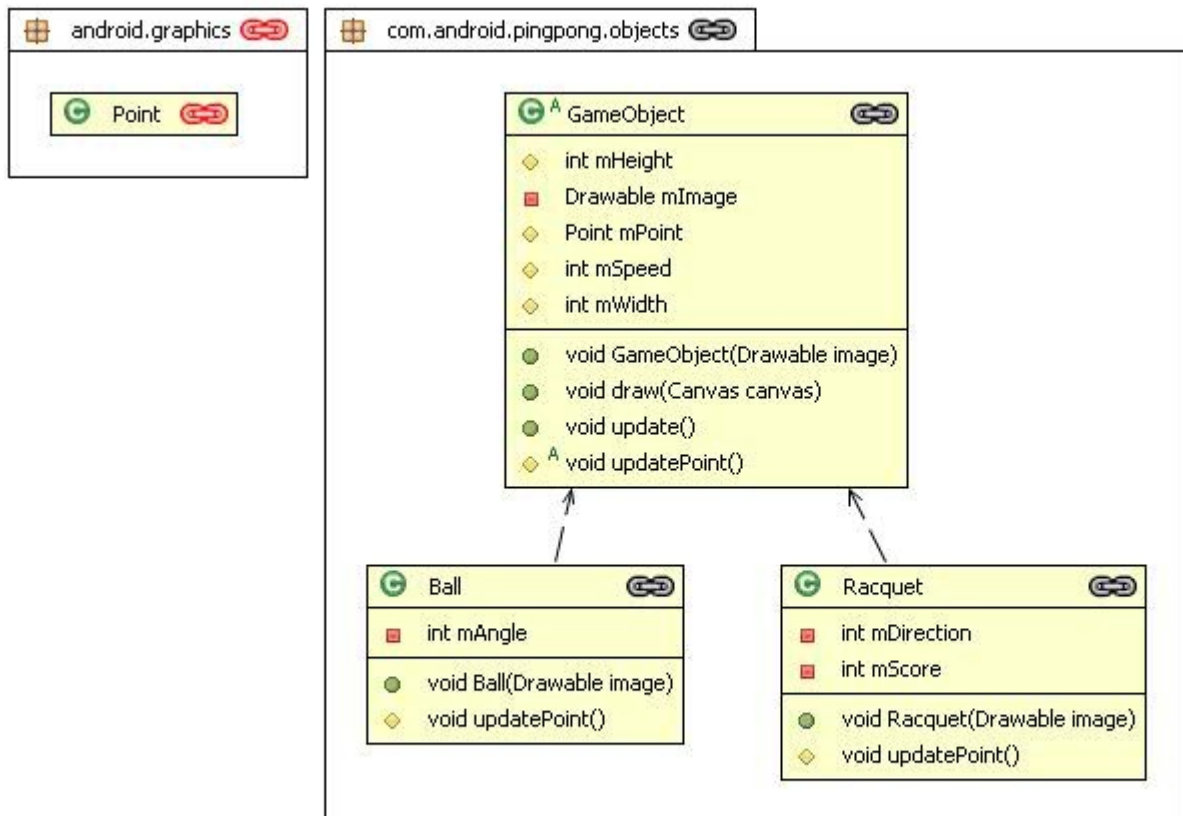


Итак

Мы рассмотрели необходимые понятия, написали простое приложение, которое умеет что-то рисовать, обработали поворот экрана. Исходники, как обычно, [прилагаются](#).

Пишем игру для Android. Часть 2. Игровые объекты

Добавим в нашу игру экшена. Как известно, в пинг-понге три действующих лица: мячик и две ракетки. Имеет смысл реализовать соответствующие классы — `Ball` и `Racquet`. А, поскольку имеются некоторые свойства, присущие обоим этим сущностям (как то: расположение, изображение, размеры и т.д.), то можно сделать базовый класс под названием `GameObject`. Диаграмма классов будет такая:



У всех игровых объектов есть:

- `mPoint` — левый верхний угол прямоугольника, ограничивающего объект. С её помощью однозначно определяется положение объекта на плоскости
- `mImage` — изображение объекта. Изображения обычно берутся из ресурсов приложения — `/res/drawable`
- `mHeight`, `mWidth` — размеры изображения объекта. Вынесены как поля класса только для того, чтобы не загромождать код всякими `mImage.getIntrinsicHeight()` и `mImage.getIntrinsicWidth()`
- `mSpeed` — скорость перемещения объекта (фактически, на сколько перемещается объект за один шаг)

Кроме того, все игровые классы умеют рисовать себя на указанном Canvas-е (метод `draw`) и вычислять своё следующее состояние (`update`). Причем, перемещение происходит так: вычисляется следующее состояние опорной точки (`updatePoint`), а потом туда переносится изображение. А, поскольку мячик и ракетка перемещаются по-разному, метод `updatePoint` сделан абстрактным.

В класс `Ball` добавлено поле `mAngle` — угол (в градусах) к оси `Ox`, под которым летит мячик. В класс `Racquet` — поля `mDirection` (направление, куда движется ракетка — вправо или влево, или вообще не движется) и `mScore` (количество очков у игрока).

Реализации классов

GameObject

GameObject.java

```
public abstract class GameObject
{
    // Константы для направлений
    public final int DIR_LEFT = -1;
    public final int DIR_RIGHT = 1;
    public final int DIR_NONE = 0;

    /** Координаты опорной точки */
    protected Point mPoint;

    /** Высота изображения */
    protected int mHeight;

    /** Ширина изображения */
    protected int mWidth;

    /** Изображение */
    private Drawable mImage;

    /** Скорость */
    protected int mSpeed;

    /**
     * Конструктор
     * @param image Изображение, которое будет обозначать данный объект
     */
    public GameObject(Drawable image)
    {
        mImage = image;
        mPoint = new Point(0, 0);
        mWidth = image.getIntrinsicWidth();
        mHeight = image.getIntrinsicHeight();
    }

    /** Перемещение опорной точки */
    protected abstract void updatePoint();

    /** Перемещение объекта */
    public void update()
    {
        updatePoint();
        mImage.setBounds(mPoint.x, mPoint.y, mPoint.x + mWidth, mPoint.y +
mHeight);
    }

    /** Отрисовка объекта */
    public void draw(Canvas canvas)
    {
        mImage.draw(canvas);
    }

    /** Задает левую границу объекта */
    public void setLeft(int value) { mPoint.x = value; }

    /** Задает правую границу объекта */
}
```

```

public void setRight(int value) { mPoint.x = value - mWidth; }

/** Задает верхнюю границу объекта */
public void setTop(int value) { mPoint.y = value; }

/** Задает нижнюю границу объекта */
public void setBottom(int value) { mPoint.y = value - mHeight; }

/** Задает абсциссу центра объекта */
public void setCenterX(int value) { mPoint.x = value - mHeight / 2; }

/** Задает левую ординату центра объекта */
public void setCenterY(int value) { mPoint.y = value - mWidth / 2; }
}

```

Ball

Ball.java

```

public class Ball extends GameObject
{
    private static final int DEFAULT_SPEED = 3;
    private static final int PI = 180;

    /** Угол, который составляет направление полета шарика с осью Ox */
    private int mAngle;

    /**
     * @see com.android.pingpong.objects.GameObject#GameObject(Drawable)
     */
    public Ball(Drawable image)
    {
        super(image);

        mSpeed = DEFAULT_SPEED; // задали скорость по умолчанию
        mAngle = getRandomAngle(); // задали случайный начальный угол
    }

    /**
     * @see com.android.pingpong.objects.GameObject#updatePoint()
     */
    @Override
    protected void updatePoint()
    {
        double angle = Math.toRadians(mAngle);

        mPoint.x += (int)Math.round(mSpeed * Math.cos(angle));
        mPoint.y -= (int)Math.round(mSpeed * Math.sin(angle));
    }

    /** Генерация случайного угла в промежутке [95, 110]U[275,290] */
    private int getRandomAngle()
    {
        Random rnd = new Random(System.currentTimeMillis());

        return rnd.nextInt(1) * PI + PI / 2 + rnd.nextInt(15) + 5;
    }
}

```

Racquet

Racquet.java

```
public class Racquet extends GameObject
{
    private static final int DEFAULT_SPEED = 3;

    /** Количество заработанных очков */
    private int mScore;

    /** Направление движения */
    private int mDirection;

    /** Задание направления движения */
    private void setDirection(int direction)
    {
        mDirection = direction;
    }

    /**
     * @see com.android.pingpong.objects.GameObject#GameObject(Drawable)
     */
    public Racquet(Drawable image)
    {
        super(image);

        mDirection = DIR_NONE; // Направление по умолчанию - нет
        mScore = 0; // Очков пока не заработали
        mSpeed = DEFAULT_SPEED; // Задали скорость по умолчанию
    }

    /**
     * @see com.android.pingpong.objects.GameObject#updatePoint()
     */
    @Override
    protected void updatePoint()
    {
        mPoint.x += mDirection * mSpeed; // двигаем ракетку по оси Ох в
        соответствующую сторону
    }
}
```

Отображение игровых объектов

Рисуем картинки

Ну вроде классы наши готовы, можно теперь их использовать в программе. Но сначала надо нарисовать картинки, для наших объектов. Картинки должны быть в png. У меня получилось так:



Мячик

Наша ракетка

Ракетка противника

Берем все это счастье, и кидаем в **/res/drawable**, где у нас хранятся всякие такие ресурсы.

Создаем игровые объекты

Теперь нам надо где-то создать экземпляры наших классов, чтобы они там жили, обновлялись и отображались на экране. Очевидно, что тут нам поможет GameManager. Итак, добавим в него такие поля:

GameManager.java

```
/** Ресурсы приложения */
private Resources mRes;

/** Мячик */
private Ball mBall;

/** Ракетка, управляемая игроком */
private Racquet mUs;

/** Ракетка, управляемая компьютером */
private Racquet mThem;
```

В конструкторе инициализируем их:

GameManager.java

```
public GameManager(SurfaceHolder surfaceHolder, Context context)
{
    mSurfaceHolder = surfaceHolder;
    mRunning = false;

    // инициализация стилей рисования
    mPaint = new Paint();
    mPaint.setColor(Color.BLUE);
    mPaint.setStrokeWidth(2);
    mPaint.setStyle(Style.STROKE);
    Resources res = context.getResources();

    mField = new Rect();
    mBall = new Ball(res.getDrawable(R.drawable.ball));
    mUs = new Racquet(res.getDrawable(R.drawable.us));
    mThem = new Racquet(res.getDrawable(R.drawable.them));
}
```

Расставляем их по местам

Однако, этого мало. У всех игровых объектов опорная точка задана в начале координат, т.е. они сейчас все в куче, и надо бы их растащить по местам. И самый лучший метод, где можно это сделать — `initPositions()`:

GameManager.java

```
public void initPositions(int screenHeight, int screenWidth)
{
    int left = (screenWidth - FIELD_WIDTH) / 2;
    int top = (screenHeight - FIELD_HEIGHT) / 2;

    mField.set(left, top, left + FIELD_WIDTH, top + FIELD_HEIGHT);
}
```

```

        // мячик ставится в центр поля
        mBall.setCenterX(mField.centerX());
        mBall.setCenterY(mField.centerY());

        // ракетка игрока - снизу по центру
        mUs.setCenterX(mField.centerX());
        mUs.setBottom(mField.bottom);

        // ракетка компьютера - сверху по центру
        mThem.setCenterX(mField.centerX());
        mThem.setTop(mField.top);
    }

```

Заставляем их что-то делать

Итак, объекты мы создали, теперь надо заставить их что-то делать. Понятно, что все изменения должны происходить в цикле, который работает в методе `run()`. На каждом шаге цикла мы должны обновить состояния объектов и отобразить их. Добавим две функции:

GameManager.java

```

/** Обновление объектов на экране */
private void refreshCanvas(Canvas canvas)

```

```

{
    // рисуем игровое поле
    canvas.drawRect(mField, mPaint);

    // рисуем игровые объекты
    mBall.draw(canvas);
    mUs.draw(canvas);
    mThem.draw(canvas);
}

```

```

/** Обновление состояния игровых объектов */
private void updateObjects()

```

```

{
    mBall.update();
    mUs.update();
    mThem.update();
}

```

А в методе `run()` будут вызовы этих методов:

GameManager.java

```

public void run()
{
    while (mRunning)
    {
        Canvas canvas = null;
        try
        {
            // подготовка Canvas-a
            canvas = mSurfaceHolder.lockCanvas();
            synchronized (mSurfaceHolder)
            {
                updateObjects(); // обновляем объекты
                refreshCanvas(canvas); // обновляем экран
            }
        }
        catch (Exception e)
        {
            // обработка исключений
        }
    }
}

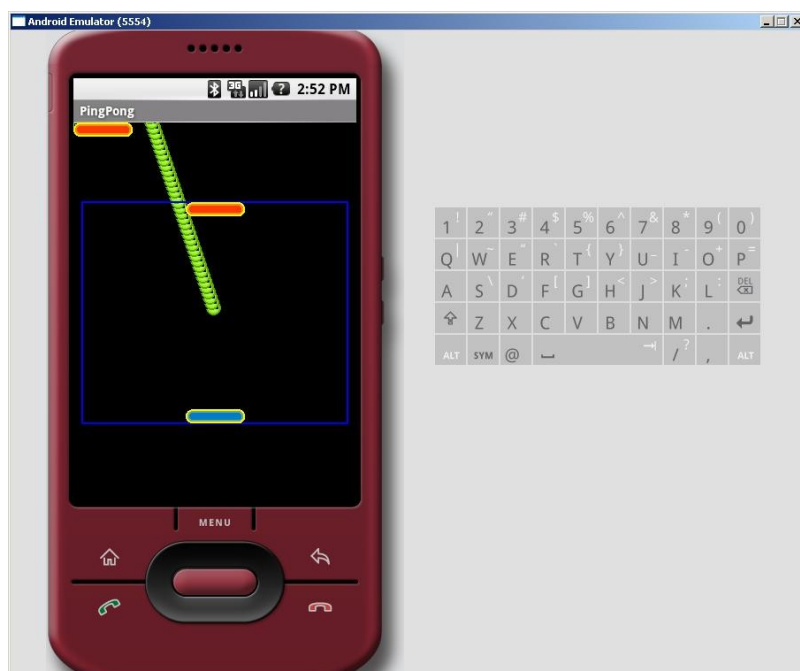
```

```

        sleep(20);
    }
}
catch (Exception e) { }
finally
{
    if (canvas != null)
    {
        mSurfaceHolder.unlockCanvasAndPost(canvas);
    }
}
}
}

```

Ну все, вроде сделали, можно запускать. Ракетки не двигаются, потому что ими пока никто не управляет, а вот шарик вполне себе летает. Правда, оставляет при этом за собой шлейф:



Делаем фон

Это всё из-за того, что мы не очищаем экран перед очередной отрисовкой. А очищать экран это тоже не так-то просто. Ничего вроде `canvas.clear()` я не нашла, так что пришлось извращаться. Суть в том, что мы заводим специальный `Bitmap`, при инициализации `Surface` задать ему размеры на весь экран, а потом в `refreshCanvas` выводить его. При желании можно загрузить в этот `Bitmap` какое-нибудь изображение из ресурсов.

Итак, заводим поле:

GameManager.java

```

/** Фон */
private Bitmap mBackground;

```

Инициализируем его в `initPositions`

GameManager.java

```
public void initPositions(int screenHeight, int screenWidth)
{
    int left = (screenWidth - FIELD_WIDTH) / 2;
    int top = (screenHeight - FIELD_HEIGHT) / 2;

    mField.set(left, top, left + FIELD_WIDTH, top + FIELD_HEIGHT);
    mBackground = Bitmap.createBitmap(screenWidth, screenHeight,
Bitmap.Config.RGB_565);
    // мячик ставится в центр поля
    mBall.setCenterX(mField.centerX());
    mBall.setCenterY(mField.centerY());

    // ракетка игрока - снизу по центру
    mUs.setCenterX(mField.centerX());
    mUs.setBottom(mField.bottom);

    // ракетка компьютера - сверху по центру
    mThem.setCenterX(mField.centerX());
    mThem.setTop(mField.top);
}
```

И отрисовываем в refreshCanvas:

GameManager.java

```
private void refreshCanvas(Canvas canvas)
{
    // вывод фонового изображения
    canvas.drawBitmap(mBackground, 0, 0, null);
    // рисуем игровое поле
    canvas.drawRect(mField, mPaint);

    // рисуем игровые объекты
    mBall.draw(canvas);
    mUs.draw(canvas);
    mThem.draw(canvas);
}
```

И получаем примерно вот такую картину:



Итак

Мы реализовали игровые объекты, написали код для их перемещения. Мячик у нас успешно летает, но не отражается от стенок поля, а ракетки теоретически тоже могут перемещаться, но код для их управления еще не написан. Этими вещи мы рассмотрим в следующей статье.

[Исходники примера](#)

Пишем игру для Android. Часть 3. Управление игровыми объектами

В этой статье мы рассмотрим две темы: управление игровыми объектами и их взаимодействие. Мячик у нас уже летает, осталось сделать, чтобы он отражался от стен и ракеток; также стоит реализовать управление нижней ракетки игроком, а верхней — неким алгоритмом. Итак, приступим.

Движение мячика

Для начала добавим в `GameObject` следующие полезные функции:

GameObject.java

```
/** Верхняя граница объекта */  
public int getTop() { return mPoint.y; }
```

```

/** Нижняя граница объекта */
public int getBottom() { return mPoint.y + mHeight; }

/** Левая граница объекта */
public int getLeft() { return mPoint.x; }

/** Правая граница объекта */
public int getRight() { return mPoint.x + mWidth; }

/** Центральная точка объекта */
public Point getCenter() { return new Point(mPoint.x + mWidth / 2, mPoint.y + mHeight / 2); }

/** Высота объекта */
public int getHeight() { return mHeight; }

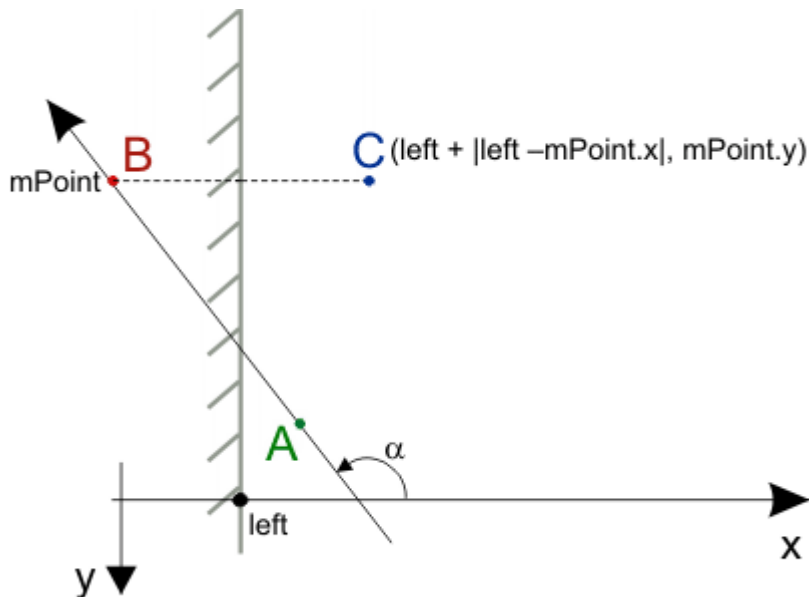
/** Ширина объекта */
public int getWidth() { return mWidth; }

/** @return Прямоугольник, ограничивающий объект */
public Rect getRect() { return mImage.getBounds(); }

/** Проверяет, пересекаются ли два игровых объекта */
public static boolean intersects(GameObject obj1, GameObject obj2)
{
    return Rect.intersects(obj1.getRect(), obj2.getRect());
}

```

Игровые объекты ничего не знают ни о друг друге, ни об игровом поле, поэтому все столкновения будут обрабатываться GameManager-ом. Итак, рассмотрим сначала такую ситуацию:



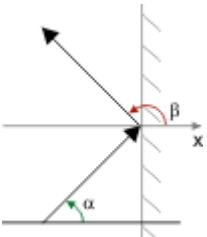
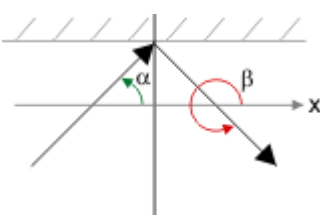
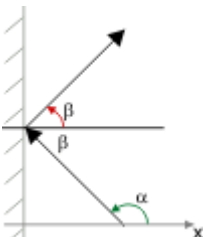
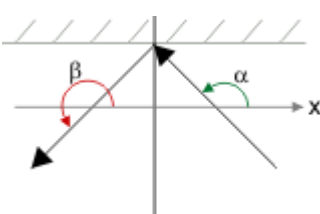
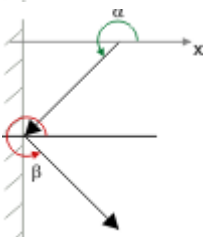
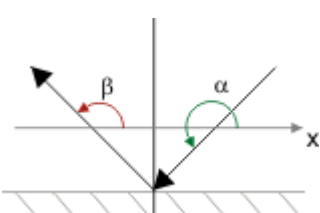
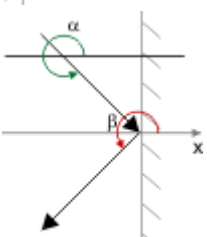
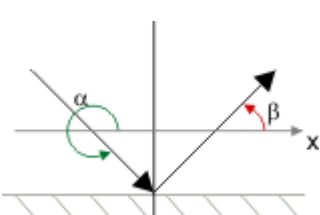
Итак, наш мячик был в некотором состоянии A, потом, пройдя расстояние mSpeed в заданном направлении, перешел в состояние B, и оказалось, что он вышел за пределы поля. Тут надо сделать следующее: поместить шарик в правильное состояние C, получившееся при отражении соответствующей координаты от стены, и изменить направление движения так, чтобы угол падения был равен углу отражения.

Расчет координат при столкновении

На рисунке выше показано, какие координаты будет иметь мячик при столкновении со стеной слева. Аналогично можно рассчитать и остальные случаи.

Вычисление нового направления движения

Рассмотрим варианты столкновения. Пусть α — угол, под которым движется мячик, а β — угол, получившийся после столкновения. Посмотрим, как β зависит от α в различных случаях:

Значения α	Столкновение с вертикалью	Столкновение с горизонталью
1 $0 < \alpha < \pi/2$		
$\beta = \pi - \alpha$		$\beta = 2\pi - \alpha$
2 $\pi/2 < \alpha < \pi$		
$\beta = \pi - \alpha$		$\beta = 2\pi - \alpha$
3 $\pi < \alpha < 3\pi/2$		
$\beta = 3\pi - \alpha$		$\beta = 2\pi - \alpha$
4 $3\pi/2 < \alpha < 2\pi$		
$\beta = 3\pi - \alpha$		$\beta = 2\pi - \alpha$

Выяснив всё это, можно добавить в класс `Ball` следующие функции:

Ball.java

```
/** Отражение мячика от вертикали */
public void reflectVertical()
{
    if (mAngle > 0 && mAngle < PI)
        mAngle = PI - mAngle;
    else
        mAngle = 3 * PI - mAngle;
}
```

```
/** Отражение мячика от горизонтали */
public void reflectHorizontal()
```

```
{  
    mAngle = 2 * PI - mAngle;  
}
```

Обновление же в `GameManager` изменится таким образом:

GameManager.java

```
private void updateObjects()  
{  
    mBall.update();  
    mUs.update();  
    mThem.update();  
  
    // проверка столкновения мячика с вертикальными стенами  
    if (mBall.getLeft() <= mField.left)  
    {  
        mBall.setLeft(mField.left + Math.abs(mField.left - mBall.getLeft()));  
        mBall.reflectVertical();  
    }  
    else if (mBall.getRight() >= mField.right)  
    {  
        mBall.setRight(mField.right - Math.abs(mField.right - mBall.getRight()));  
        mBall.reflectVertical();  
    }  
  
    // проверка столкновения мячика с горизонтальными стенами  
    if (mBall.getTop() <= mField.top)  
    {  
        mBall.setTop(mField.top + Math.abs(mField.top - mBall.getTop()));  
        mBall.reflectHorizontal();  
    }  
    else if (mBall.getBottom() >= mField.bottom)  
    {  
        mBall.setBottom(mField.bottom - Math.abs(mField.bottom -  
mBall.getBottom()));  
        mBall.reflectHorizontal();  
    }  
}
```

Запускаем и видим, что мячик летает по полю и отражается от всех стен. Вообще-то ему надо отражаться только от вертикальных стен и от ракеток, но управление ракетками у нас пока не реализовано, так что для наглядности пока так, а потом мы этот кусок кода уберем.

Управление ракеткой

Перемещать ракетку можно двумя способами. Первый — отлавливать нажатие кнопок вправо и влево, и при нажатии смещать ракетку в нужную сторону. Однако, как показала практика, такой способ не очень хорош, так как ракетка двигается резко и подтормаживает при первом нажатии. Более прогрессивным способом оказался другой: при нажатии клавиши назначать ракетке соответствующий `Direction`, а при отпуске обнулять его. А в методе `updateObjects` ракетка сама перемещается в том направлении, которое у нее указано.

Итак, теперь надо бы написать код для обработки нажатия клавиш. Вообще, нажатие клавиш ловит `View`, и для обработки нужно перегрузить функции `onKeyDown` и `onKeyUp`. Однако, игровыми объектами у нас ведаёт `GameManager`, так что фактическая обработка будет происходить именно там. Так что добавляем в `GameView` следующее:

GameView.java

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event)
{
    return mGameManager.doKeyDown(keyCode);
}

@Override
public boolean onKeyUp(int keyCode, KeyEvent event)
{
    return mGameManager.doKeyUp(keyCode);
}
```

А в GameManager нужно добавить методы doKeyDown и doKeyUp, который будут выполнять всю работу:

GameView.java

```
/**
 * Обработка нажатия кнопки
 * @param keyCode Код нажатой кнопки
 * @return Было ли обработано нажатие
 */
public boolean doKeyDown(int keyCode)
{
    switch (keyCode)
    {
        case KeyEvent.KEYCODE_DPAD_LEFT:
            mUs.setDirection(GameObject.DIR_LEFT);
            return true;
        case KeyEvent.KEYCODE_DPAD_RIGHT:
            mUs.setDirection(GameObject.DIR_RIGHT);
            return true;
        default:
            return false;
    }
}

/**
 * Обработка отпускания кнопки
 * @param keyCode Код кнопки
 * @return Было ли обработано действие
 */
public boolean doKeyUp(int keyCode)
{
    if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT ||
        keyCode == KeyEvent.KEYCODE_DPAD_RIGHT)
    {
        mUs.setDirection(GameObject.DIR_NONE);
        return true;
    }
    return false;
}
```

Однако, если мы запустим сейчас наше приложение, окажется, что обработка клавиш не работает. View попросту не получает эти события. Чтобы исправить положение, нужно в конструктор GameView добавить одну строчку:

GameView.java

```
public GameView(Context context, AttributeSet attrs)
{
    super(context, attrs);

    // подписываемся на события Surface
    mSurfaceHolder = getHolder();
    mSurfaceHolder.addCallback(this);

    mGameManager = new GameManager(mSurfaceHolder, context);
    setFocusable(true);
}
```

Теперь все работает, ракетка управляется.

Искусственный интеллект для противника

У компьютера логика будет простая: по возможности не допускать, чтобы мячик выходил за пределы ракетки. Если мячик окажется правее ракетки, ракетка едет направо, если левее — налево. Реализуем всю эту логику в методе `moveAI()` у `GameManager`-а:

GameManager.java

```
private void moveAI()
{
    if (mThem.getLeft() > mBall.getRight())
        mThem.setDirection(GameObject.DIR_LEFT);
    else if (mThem.getRight() < mBall.getLeft())
        mThem.setDirection(GameObject.DIR_RIGHT);
    mThem.update();
}
```

A `GameManager.updateObjects()` будет выглядеть так:

GameManager.java

```
private void updateObjects()
{
    mBall.update();
    mUs.update();
    moveAI();
    // обработка столкновений
    ...
}
```

Еще пара доработок

Ракетки выходят за пределы экрана

Собственно, никто им этого делать не запрещает. Напишем в `GameManager` такой метод:

GameManager.java

```
private void placeInBounds(Racquet r)
{
    if (r.getLeft() < mField.left)
        r.setLeft(mField.left);
    else if (r.getRight() > mField.right)
        r.setRight(mField.right);
}
```

А в `GameManager.updateObjects()` добавим:

GameManager.java

```
private void updateObjects()
{
    mBall.update();
    mUs.update();
    moveAI();
    // чтобы ракетки не выходили за пределы поля
    placeInBounds(mUs);
    placeInBounds(mThem);
    // обработка столкновений
    ...
}
```

Теперь все стало хорошо, ракетки не выходят за пределы поля.

Столкновение мячика с ракетками

Сейчас у нас мячик летает, отражаясь от стен, а хотелось бы, чтобы он летал, отражаясь только от вертикальных стен и от ракеток. Убираем из `GameManager.updateObjects()` весь код, осуществляющий отражение от горизонтальных стен, и пишем код для отражения от ракеток. Код достаточно прост:

GameManager.java

```
private void updateObjects()
{
    // Обновление положений объектов
    ...

    // Обработка столкновений с вертикальными стенами
    ...
    // проверка столкновений мячика с ракетками
    if (GameObject.intersects(mBall, mUs))
    {
        mBall.setBottom(mUs.getBottom() - Math.abs(mUs.getBottom() -
mBall.getBottom()));
        mBall.reflectHorizontal();
    }
    else if (GameObject.intersects(mBall, mThem))
    {
        mBall.setTop(mThem.getTop() + Math.abs(mThem.getTop() - mBall.getTop()));
        mBall.reflectHorizontal();
    }
}
```

Физика тут простейшая: везде угол падения равен углу отражения. Надо сказать, смотрится это местами довольно несуразно, да и искусственный интеллект в таких условиях практически непобедим. Но более реалистичное движение шарика делать не хочется. Во-первых, я в физике не очень сильна, а во-вторых, статья получится совсем уж большая и совсем не про андроид. Так что оставляю эту часть читателям в качестве легкого домашнего упражнения :)

Итак

Итак, мы написали код для управления ракетками, причем управления как человеком, так и компьютером, реализовали обработку столкновений. В принципе, уже можно играть.

[Исходники для этой части](#)

Пишем игру для Android. Часть 4. Игровой процесс

В этой части мы напишем обработку выигрышей-проигрышей, реализуем подсчет очков, а также сделаем, чтобы игру можно было ставить на паузу. Собственно, пауза тут несколько не в тему, но девать ее некуда, так что сделаем ее в этой части.

Обработка проигрыша

Помнится, мы заводи́ли в классе `Racquet` поле `mScore`, в котором собирались хранить количество очков у игрока. Теперь самое время начать использовать это поле.

Итак, в начале игры количество очков у обоих игроков пусто. Когда игрок не успевает отбить мяч, его противнику засчитывается очко, мячик и ракетки возвращаются на исходные позиции. Игра продолжается, пока какой-нибудь из игроков не наберет `N` очков. `N` мы пока что объявим константой, а в следующей части вынесем в настройки.

Проверка проигрыша должна осуществляться также в методе `updateObjects()` `GameManager`-а. Описанная нами логика запишется так:

GameManager.java

```
private void updateObjects()
{
    ...
    // проверка проигрыша
    if (mBall.getBottom() < mThem.getBottom())
    {
        mUs.incScore();
        reset();
    }

    if (mBall.getTop() > mUs.getTop())
    {
        mThem.incScore();
        reset();
    }
}
```

`Racquet.incScore()` увеличивает на 1 количество очков у игрока:

Racquet.java

```
/** Увеличить количество очков игрока */
public void incScore()
{
    mScore++;
}
```

`GameManager.reset()` расставляет ракетки и мячик на исходные позиции, задает мячику новый случайный угол, а также делает паузу (чтобы игрок успел понять, что произошло).

GameManager.java

```
private void reset()
{
    // ставим мячик в центр
    mBall.setCenterX(mField.centerX());
    mBall.setCenterY(mField.centerY());
    // задаем ему новый случайный угол
    mBall.resetAngle();

    // ставим ракетки в центр
    mUs.setCenterX(mField.centerX());
    mThem.setCenterX(mField.centerX());

    // делаем паузу
    try
    {
        sleep(LOSE_PAUSE);
    }
    catch (InterruptedException iex)
    {
    }
}
```

`LOSE_PAUSE` — это константа класса `GameManager`, в которой задается длина паузы в миллисекундах (у меня она равна 2000). Метод же `resetAngle()` класса `Ball` выглядит следующим образом:

Ball.java

```
/** Задает новое случайное значение угла */
public void resetAngle()
{
    mAngle = getRandomAngle();
}
```

Если теперь запустить приложение, то увидим, что, если упустить мячик, то он никуда не улетит, а через некоторое время восстановится в центре. А про очки пока ничего сказать нельзя, потому что они нигде не выводятся. Что ж, будем выводить.

Вывод количества очков

Вывод текста на экран производится с помощью метода `drawText(String text, float x, float y, Paint paint)` класса `Canvas`. Как можно заметить, стили текста

задаются с помощью экземпляра класса `Paint`. Где-то в первой части мы создавали в `GameManager` такое поле `mPaint`, где хранились стили для рисования игрового поля. Для вывода текста можно использовать это же поле, и при каждой перерисовке экрана задавать ему стили сначала для игрового поля, а потом для текста. А можно завести отдельный экземпляр `Paint` для хранения стилей текста:

GameManager.java

```
private Paint mScorePaint;
```

Инициализировать его в конструкторе:

GameManager.java

```
public GameManager(SurfaceHolder surfaceHolder, Context context)
{
    mSurfaceHolder = surfaceHolder;
    Resources res = context.getResources();
    mRunning = false;

    // стили для рисования игрового поля
    mPaint = new Paint();
    mPaint.setColor(Color.BLUE);
    mPaint.setStrokeWidth(2);
    mPaint.setStyle(Style.STROKE);
    // стили для вывода счета
    mScorePaint = new Paint();
    mScorePaint.setTextSize(20);
    mScorePaint.setStrokeWidth(1);
    mScorePaint.setStyle(Style.FILL);
    mScorePaint.setTextAlign(Paint.Align.CENTER);
    // игровые объекты
    mField = new Rect();
    mBall = new Ball(res.getDrawable(R.drawable.ball));
    mUs = new Racquet(res.getDrawable(R.drawable.us));
    mThem = new Racquet(res.getDrawable(R.drawable.them));
}
```

А непосредственно вывод счета игры производится в методе, где происходит вся отрисовка текущей игровой ситуации — `refreshCanvas`

GameManager.java

```
private void refreshCanvas(Canvas canvas)
{
    // вывод фонового изображения
    canvas.drawBitmap(mBackground, 0, 0, null);

    // рисуем игровое поле
    canvas.drawRect(mField, mPaint);

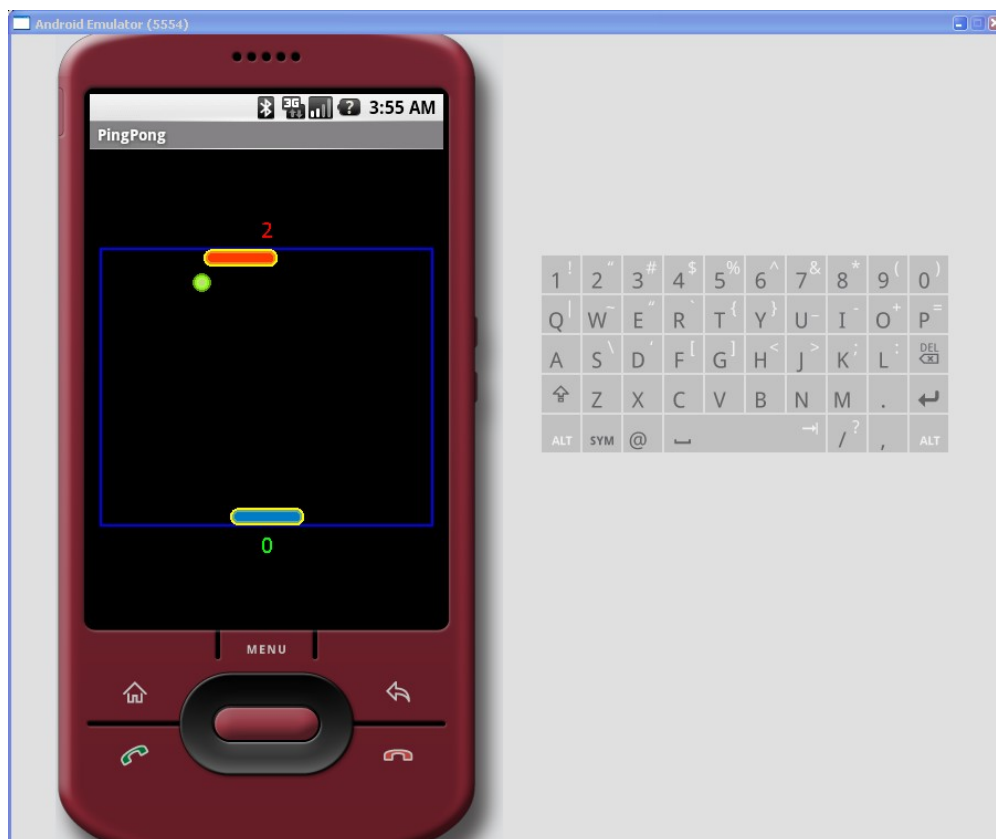
    // рисуем игровые объекты
    mBall.draw(canvas);
    mUs.draw(canvas);
    mThem.draw(canvas);
    // вывод счета
```

```

mScorePaint.setColor(Color.RED);
canvas.drawText(String.valueOf(mThem.getScore()), mField.centerX(),
mField.top - 10, mScorePaint);
mScorePaint.setColor(Color.GREEN);
canvas.drawText(String.valueOf(mUs.getScore()), mField.centerX(),
mField.bottom + 25, mScorePaint);
}

```

Правда, совсем уж без изменения стиля не обошлось. Наши очки мы рисуем зелёным, а очки противника — красным. Запустив, увидим примерно такую картину:



Использование пользовательских шрифтов

А теперь нам захотелось использовать для вывода счета какой-нибудь наш красивый шрифт. Рассмотрим, как это можно сделать.

В нашем проекте есть такая папка **assets**, там хранятся такие ресурсы, как TrueType-шрифты, возможно, какие-то большие тексты и т.д.. Основное отличие их от ресурсов, которые хранятся в папке **res** — это то, что используются они гораздо реже, и доставать их оттуда сложнее. Ресурсы из **res** можно запросто достать с помощью класса **R**, а **assets** вытаскиваются с помощью специального класса **AssetManager**.

Итак, создадим в папке **assets** папку **fonts** и кинем туда шрифт под названием **Mini.ttf**. Теперь, чтобы достать этот шрифт и использовать его для вывода количества очков, достаточно добавить в инициализацию **mScorePaint** в конструкторе одну строчку:

GameManager.java

```

mScorePaint.setTypeface(Typeface.createFromAsset(context.getAssets(),
"fonts/Mini.ttf"));

```

`context.getAssets()` получит менеджер ресурсов (`AssetManager`) для данного приложения, откуда потом будет можно загрузить шрифт по указанному пути. Стоит обратить внимание, что путь является case-sensitive, т.е. "fonts/mini.ttf" уже ничего не загрузит.



Неприятность

И всё бы хорошо, но теперь время от времени стали возникать ситуации, когда в начале игры у одного из игроков выводится не 0 очков, а 1. Я так понимаю, что проблемы возникают в самом начале программы, перед `initPositions`, когда у игровых объектов координаты еще не заданы, а `updateObjects` уже вызывается. Чтобы исправить положение, заведем в классе `GameManager` еще одно булево поле `mInitialized`, в конструкторе зададим как `false`, а в `initPositions` присвоим ему `true`. Тогда в `run` можно написать так:

GameManager.java

```
public void run()
{
    while (mRunning)
    {
        Canvas canvas = null;
        try
        {
            // подготовка Canvas-a
            canvas = mSurfaceHolder.lockCanvas();
            synchronized (mSurfaceHolder)
            {
                if (mInitialized)
                {
                    updateObjects(); // обновляем объекты
                }
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

```

        refreshCanvas(canvas); // обновляем экран
        sleep(20);
    }
}
}
catch (Exception e) { }
finally
{
    if (canvas != null)
    {
        mSurfaceHolder.unlockCanvasAndPost(canvas);
    }
}
}
}
}

```

Теперь гарантированно не будет происходить никаких проверок, пока не будут инициализированы координаты игровых объектов. Проблема решена.

Обработка окончания игры

Прежде всего, нам следует завести в `GameManager` переменную, где бы хранилось количество очков, до которого идет игра. Заведем такую переменную и сразу сеттер к ней. Итак:

GameManager.java

```

/** Максимальное число очков, до которого идет игра */
private static int mMaxScore = 5;

public static void setMaxScore(int value)
{
    mMaxScore = value;
}

```

Саму проверку на окончание игры можно поместить как в метод `updateObjects()`, так и прямо в `run()`. Но, думаю, правильнее именно в `updateObjects()`:

GameManager.java

```

/** Обновление состояния игровых объектов */
private void updateObjects()
{
    // Обновление состояния игровых объектов
    ...

    // обработка столкновений
    ...

    // проверка проигрыша
    ...

    // проверка окончания игры
    if (mUs.getScore() == mMaxScore & mThem.getScore() == mMaxScore)
    {
        this.mRunning = false;
    }
}

```

```
}
```

Напомню, что метод `run` выглядит так:

GameManager.java

```
public void run()
{
    while (mRunning)
    {
        // обновление и отрисовка объектов
    }
}
```

То есть, когда `mRunning` станет равным `false`, поток завершится. Раз он завершился — игра закончена, и надо вывести на экран ее результаты. Так что логично видеть в методе `run()` что-то вроде:

GameManager.java

```
public void run()
{
    while (mRunning)
    {
        // обновление и отрисовка объектов
    }
    // рисование GameOver
    ...
}
```

А теперь разберемся, как это рисование может выглядеть. Как известно, при рисовании мы лочим `Canvas`, рисуем, и затем разлочиваем. При этом еще нужно отловить возможные исключения. Получается куча кода, которая появляется при каждом рисовании и сильно загромождает текст программы. Естественно, хочется вынести все это в метод-обертку и передавать туда ссылку на функцию, осуществляющую собственно рисование. На `C#` это выглядело бы так:

```
delegate void DrawFunction(Canvas canvas);

private void draw(DrawFunction something)
{
    Canvas canvas = null;
    try
    {
        {
            canvas = mSurfaceHolder.lockCanvas();
            synchronized (mSurfaceHolder)
            {
                something(canvas);
            }
        }
    }
    catch (Exception e) { }
    finally
    {
        if (canvas != null)
        {
            mSurfaceHolder.unlockCanvasAndPost(canvas);
        }
    }
}
```

```
}  
}
```

Но здесь нам не C#, здесь климат иной, и делегатов нет. Однако, как мне подсказал товарищ **хеуе**, подобный код можно написать. Итак, добавим в `GameManager` такой интерфейс:

GameManager.java

```
private interface DrawHelper  
{  
    void draw(Canvas canvas);  
}
```

И такой метод, куда мы вынесем всю работу по подготовке canvas-a:

GameManager.java

```
private void draw(DrawHelper helper)  
{  
    Canvas canvas = null;  
    try  
    {  
        // подготовка Canvas-a  
        canvas = mSurfaceHolder.lockCanvas();  
        synchronized (mSurfaceHolder)  
        {  
            if (mInitialized)  
            {  
                helper.draw(canvas);  
            }  
        }  
    }  
    catch (Exception e) { }  
    finally  
    {  
        if (canvas != null)  
        {  
            mSurfaceHolder.unlockCanvasAndPost(canvas);  
        }  
    }  
}
```

Теперь можно завести конкретные реализации `DrawHelper` на все случаи жизни. Я добавляю их две:

GameManager.java

```
/** Хелпер для перерисовки экрана */  
private DrawHelper mDrawScreen;  
  
/** Хелпер для рисования результата игры */  
private DrawHelper mDrawGameOver;
```

Инициализирую в конструкторе таким образом:

GameManager.java

```
public GameManager(SurfaceHolder surfaceHolder, Context context)
{
    ...

    // функция для рисования экрана
    mDrawScreen = new DrawHelper()
    {
        public void draw(Canvas canvas)
        {
            refreshCanvas(canvas);
        }
    };

    // функция для рисования результатов игры
    mDrawGameOver = new DrawHelper()
    {
        public void draw(Canvas canvas)
        {
            // Вывели последнее состояние игры
            refreshCanvas(canvas);

            // смотрим, кто выиграл и выводим соответствующее сообщение
            String message = "";
            if (mUs.getScore() > mThem.getScore())
            {
                mScorePaint.setColor(Color.GREEN);
                message = "You won";
            }
            else
            {
                mScorePaint.setColor(Color.RED);
                message = "You lost";
            }
            mScorePaint.setTextSize(30);
            canvas.drawText(message, mField.centerX(), mField.centerY(),
mScorePaint);
        }
    };
}
```

После этого метод `run()` преобразуется до неузнаваемости:

GameManager.java

```
/** Действия, выполняемые в потоке */
public void run()
{
    while (mRunning)
    {
        if (mInitialized)
        {
            updateObjects(); // обновляем объекты
            draw(mDrawScreen);
        }
        draw(mDrawGameOver);
    }
}
```


И сразу результат:



Пауза

Тут совсем кратко. Объявим в классе GameManager поле:

GameManager.java

```
/** Стоит ли приложение на паузе */  
private boolean mPaused;
```

Если приложение на паузе, поток работает "вхолостую", т.е. состояния объектов не меняются и вообще ничего не происходит. Это значит, в методе run () будет следующее:

GameManager.java

```
public void run()  
{  
    while (mRunning)  
    {  
        if (mPaused) continue;  
  
        if (mInitialized)  
        {  
            updateObjects(); // обновляем объекты  
            draw(mDrawScreen);  
        }  
    }  
    draw(mDrawGameOver);  
}
```

Будем ставить приложение на паузу, если нажата средняя клавиша джойстика:

GameManager.java

```
public boolean doKeyDown(int keyCode)
{
    switch (keyCode)
    {
        case KeyEvent.KEYCODE_DPAD_LEFT:
        case KeyEvent.KEYCODE_A:
            mUs.setDirection(GameObject.DIR_LEFT);
            return true;
        case KeyEvent.KEYCODE_DPAD_RIGHT:
        case KeyEvent.KEYCODE_D:
            mUs.setDirection(GameObject.DIR_RIGHT);
            return true;
        case KeyEvent.KEYCODE_DPAD_CENTER:
            mPaused = !mPaused;
            draw(mDrawPause);
            return true;
        default:
            return false;
    }
}
```

`mDrawPause` — хелпер для рисования паузы. Я уже не буду приводить к нему листинг, там все просто.

Итак

У нас уже совсем готовая игра. Можно играть, выигрывать, проигрывать, ставить на паузу.

[Исходники примера](#)

Пишем игру для Android. Часть 5. Хранение настроек

Вот мы и добрались до конца. Осталось сделать только главное меню приложения, а также сделать игре настройки (собственно, меню только для того и нужно, чтобы было откуда настройки вызывать). Ну первое мы еще с прошлой статьи умеем, так что особых сложностей быть не должно. А вот второе следует рассмотреть подробнее. Итак, приступим.

Окно приветствия

В [одной из прошлых статей](#) подробно рассматривался вопрос, как создавать формы в приложении для Android и делать переходы между ними. Так что особо останавливаться я не буду, и так все ясно.

На нашей форме приветствия будет какая-нибудь картинка и три кнопки: "Начать игру", "Настройки" и "Выход". Картинку в формате `png`, которую мы назовем `start.png` нужно положить в папку `/res/drawable`. Названия кнопок нужно вынести в `strings.xml`, добавив следующие строки:

res/values/strings.xml

```
<resources>
    <string name="app_name">PingPong</string>
    <string name="start_title">Start Game</string>
    <string name="settings_title">Settings</string>
```

```
<string name="exit_title">Exit</string>
</resources>
```

Тогда разметка для новой формы будет выглядеть так:

res/layout/start.xml

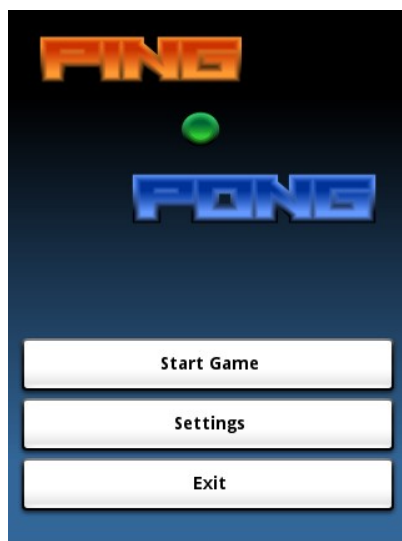
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="bottom"
    android:background="@drawable/start"
    android:padding="8dip">

    <Button android:id="@+id/StartButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textStyle="bold"
        android:text="@string/start_title" />

    <Button android:id="@+id/SettingsButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textStyle="bold"
        android:text="@string/settings_title" />

    <Button android:id="@+id/ExitButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textStyle="bold"
        android:text="@string/exit_title" />
</LinearLayout>
```

Фоновое изображение можно задать экрану с помощью полезного свойства `android:background`. Собственно, так можно задавать фон и кнопкам, и вообще чему угодно. Получили вот такую разметку:



Добавим соответствующий этой разметке класс `StartScreen.java`. Сразу обработаем нажатия всех кнопок:

StartScreen.java

```
public class StartScreen extends Activity implements OnClickListener
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.start);

        // Кнопка "Start"
        Button startButton = (Button)findViewById(R.id.StartButton);
        startButton.setOnClickListener(this);

        // Кнопка "Exit"
        Button exitButton = (Button)findViewById(R.id.ExitButton);
        exitButton.setOnClickListener(this);

        // Кнопка "Settings"
        Button settingsButton = (Button)findViewById(R.id.SettingsButton);
        settingsButton.setOnClickListener(this);
    }

    /** Обработка нажатия кнопок */
    public void onClick(View v)
    {
        switch (v.getId())
        {
            case R.id.StartButton:
            {
                Intent intent = new Intent();
                intent.setClass(this, GameScreen.class);
                startActivity(intent);
                break;
            }

            case R.id.SettingsButton:
            {
                break;
            }

            case R.id.ExitButton:
            {
                finish();
                break;
            }

            default:
            {
                break;
            }
        }
    }
}
```

По нажатию на кнопку **Start** происходит переход на экран с игрой. Обратите внимание, что StartScreen при этом не закрывается. Это значит, что, когда закроется StartScreen, мы попадем обратно на экран приветствия. По нажатию на **Settings** пока что ничего не происходит, по **Exit** — приложение закрывается.

Осталось только зарегистрировать эту форму в приложении и сделать ее главной. Для этого идем в `AndroidManifest.xml`:

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.pingpong"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".GameScreen"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
                <category android:name="android.intent.category.SAMPLE_CODE" />
            </intent-filter>
        </activity>
        <activity android:name=".StartScreen">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Теперь приложение начинается с StartScreen, все кнопки работают.

Настройки

Я сделаю в настройках два параметра — максимальное количество очков и сложность. Сложность игры будем менять, варьируя скорость мячика и ракеток.

Сама форма с настройками делается достаточно просто. Есть специальный наследник класса Activity — PreferenceActivity, который именно для этого и предназначен. Когда мы хотим сделать форму с настройками, нужно унаследоваться именно от него, и он возьмет на себя большую часть рутины.

Разметка

Разметка для формы с настройками выглядит несколько необычно:

res/xml/preferences.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android" >
    <PreferenceCategory
        android:title="@string/prefs_title">

        <ListPreference android:key="@string/pref_difficulty"
            android:title="@string/difficulty_title"
            android:entries="@array/difficulty"
            android:entryValues="@array/difficulty"
            android:defaultValue="1"
            />

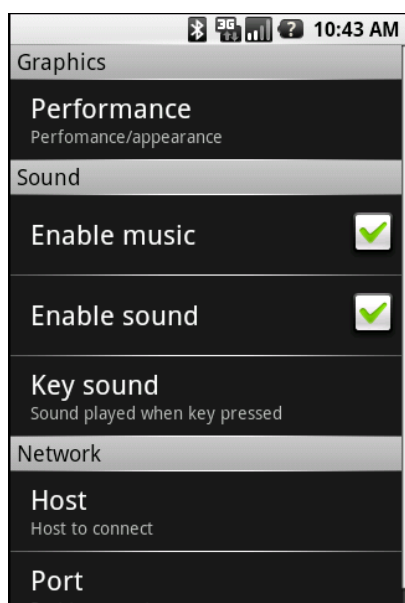
        <EditTextPreference android:key="@string/pref_max_score"
            android:title="@string/score_title"
```

```
        android:defaultValue="10"
    />
</PreferenceCategory>
</PreferenceScreen>
```

Настолько необычно, что, если поместить этот XML в папку layout, eclipse начнет ругаться, что не может разрешить такие классы. Собственно, это не просто разметка: там также содержатся ключи настроек и значения по умолчанию. Поэтому мы создадим отдельную папку **xml**, и поместим этот файл туда. А теперь обо всем по порядку.

PreferenceCategory

Ну, PreferenceScreen все понятно, а что такое PreferenceCategory? Как ни удивительно, это категория настроек. Например, у какой-нибудь игры могут быть настройки графики, настройки звука, настройки сети и т.д.. Удобно отобразить их сгруппированными, вот так:



А можно обойтись без групп: PreferenceScreen уже сам по себе является контейнером для настроек. В нашем случае, например, настроек мало и группировать нечего. Но это я так, для полноты картины.

Какие можно сделать настройки

Как видно даже не прошлой картинке, настройки могут быть разными. И флажки, и текстовые поля, и списки. Все они происходят от одного класса Preference, и наследуют от него всякие полезные атрибуты, которые можно задавать в разметке, как то:

android:title

Заголовок настройки или контейнера настроек.

android:summary

Краткое описание. Проще говоря, это то, что пишется под заголовком мелким шрифтом.

android:defaultValue

Значение по умолчанию

android:key

Ключ, с которым данная настройка будет храниться и с которым можно будет к ней обращаться.

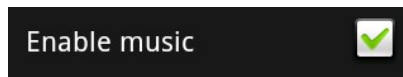
android:dependency

Задаёт зависимость от другого контрола. Например, можно поставить эдитору зависимость от флажка, и тогда, если флажок не установлен, но эдитор будет неактивен.

Ну и ещё кое-что. Рассмотрим некоторые конкретные виды настроек.

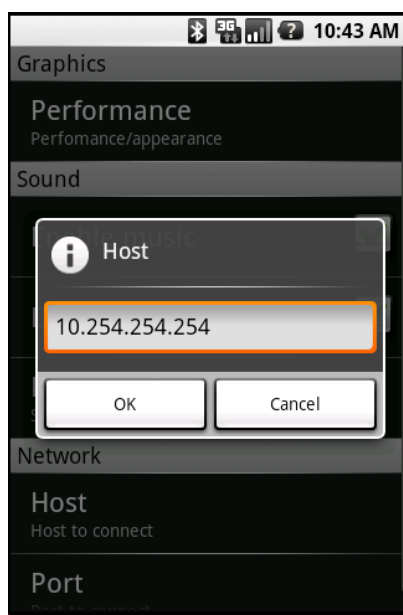
CheckBoxPreference

Вот такой флажок:



EditTextPreference

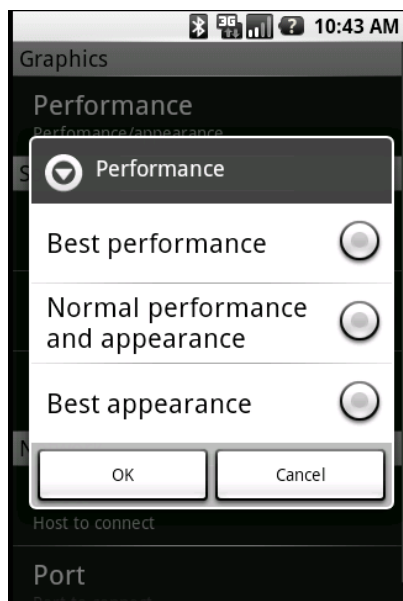
Редактор текста.



Честно говоря, у редактора текста очень хотелось бы валидатор, а ещё лучше маску. Например, IP-адрес имеет специфический формат, и хотелось бы запрещать пользователю вводить там что попало. Но мне такую функциональность так и не удалось обнаружить.

ListPreference

Своеобразная реализация Dropdown-а. Хотя, на телефоне наверно и вправду так удобнее.



На этом контроле хотелось бы остановиться подробнее. А конкретнее, рассказать, откуда он, собственно, берет элементы списка. А берет он их из ресурсов с помощью таких атрибутов.

`android:entries`

Здесь хранится ссылка на ресурс, в котором хранятся отображаемые элементы списка. Все значения, которые хочется вынести в ресурсы, хранятся в папке **values**. До сих пор там была только одна XML-ка — `strings.xml`. Но теперь надо добавить еще одну — **arrays.xml**. И добавить в узел `resources` следующее:

```
<string-array name="performance">
    <item>Best performance</item>
    <item>Normal performance and appearance</item>
    <item>Best appearance</item>
</string-array>
```

После этого можно смело указывать в `android:entries` этот ресурс, список будет загружен.

Кстати говоря, в `item`-ах может быть не непосредственно строка, а ссылка на строку в `strings.xml`. Например, в нашем случае будет так (разумеется, стоит добавить соответствующие значения в `strings.xml`):

res/values/arrays.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="difficulty">
        <item>@string/difficulty_easy</item>
        <item>@string/difficulty_normal</item>
        <item>@string/difficulty_hard</item>
    </string-array>
</resources>
```

`android:entryValues`

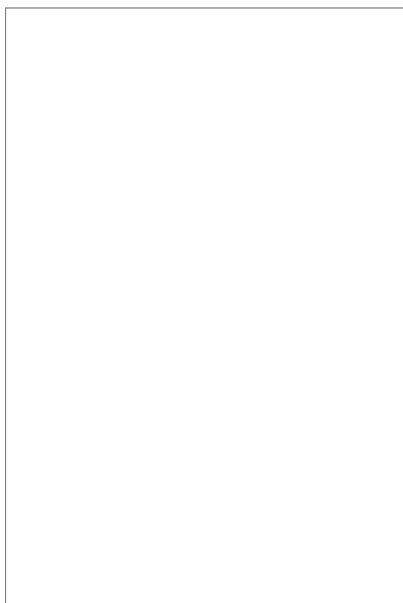
Список действительных значений. Также ссылка на ресурс, и задавать можно так же.

Если кодов будет меньше, чем значений, приложение будет валиться с исключением. Если больше — не будет. В нашем случае можно в `entries` и `entryValues` смело задавать одно и то же, но бывает, когда имеет смысл их разделять.

А еще мне никак не удалось победить у `ListPreference` атрибут `android:defaultValue`. Не работает и все.

RingtonePreference

Настройка звукового сигнала. В нашем приложении, например, можно было бы добавить звук, с которым мячик ударяется о стенку, и выбирать этот звук с помощью такого контрола. Выглядит это так:



Класс формы

Класс для формы с настройками будет выглядеть так:

SettingsScreen.java

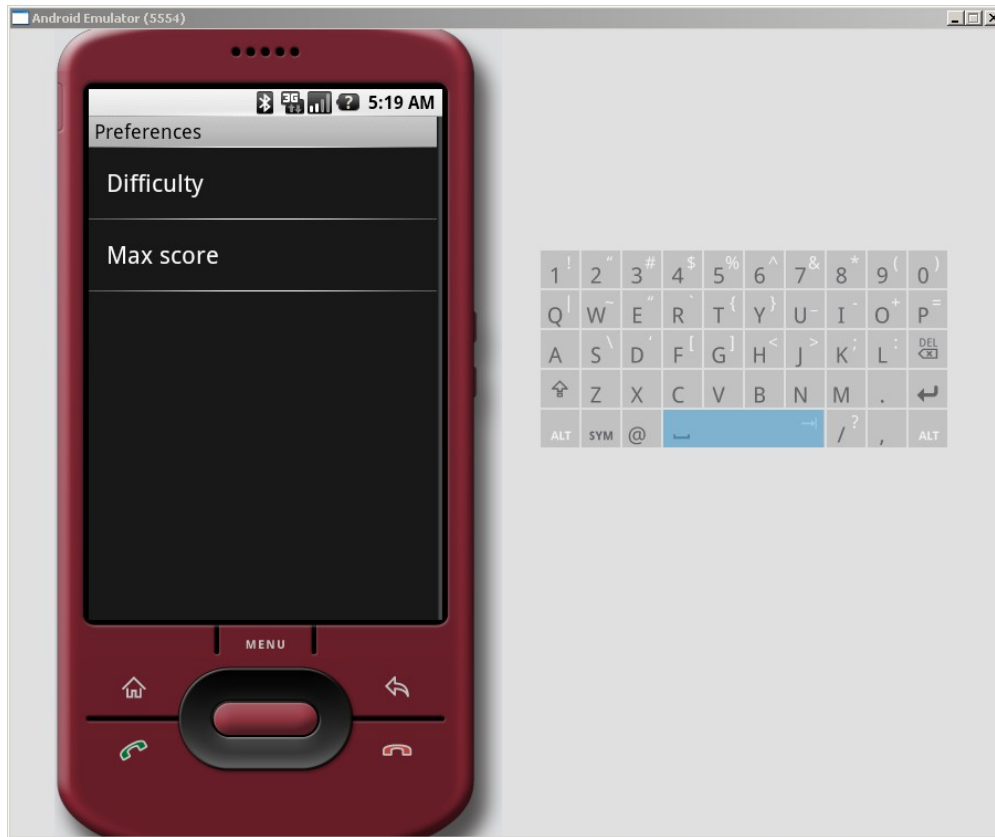
```
public class SettingsScreen extends PreferenceActivity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);

        // Настройки и их разметка загружаются из XML-файла
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

Кстати, настройки необязательно загружать из XML-ки, можно добавлять все эти настройки прямо в коде конструктора. В сэмплах, которые идут с Android SDK, такие примеры есть.

Добавляем в `StartScreen` код для открытия формы настроек, прописываем `SettingsScreen` в `AndroidManifest.xml`. (Все выглядит точно так же, как и для

GameScreen, так что листинги не привожу). И увидим мы следующее:



Не знаю, кому как, а мне нравится, когда рядом с названием опции написано ее значение. Но как это сделать автоматически, я так и не нашла, пришлось все делать руками, используя для этого поле `summary`. Итак, `summary` настройки должно обновляться при изменении значения. По счастью, есть такое событие `OnPreferenceChange`. Итак, пишем:

SettingsScreen.java

```
public class SettingsScreen extends PreferenceActivity implements
Preference.OnPreferenceChangeListener
{
    /* Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);

        // Load the preferences from an XML resource
        addPreferencesFromResource(R.xml.preferences);
        ListPreference difficulty =
(ListPreference) this.findPreference("pref_difficulty");
        difficulty.setSummary(difficulty.getEntry());
        difficulty.setOnPreferenceChangeListener(this);

        EditTextPreference maxScore =
(EditTextPreference) this.findPreference("pref_max_score");
        maxScore.setSummary(maxScore.getText());
        maxScore.setOnPreferenceChangeListener(this);
    }
    public boolean onPreferenceChange(Preference preference, Object newValue)
    {

```

```

        preference.setSummary((CharSequence)newValue);

        return true;
    }
}

```

Думаю, все понятно без слов. Теперь мы видим такую картину:



И, если мы будем менять настройки, изменения сразу же будут отображаться в `summary`.

Использование настроек в других формах

Ну все, настройки мы сделали, они как-то сами где-то сохранились. Теперь возникла необходимость их прочитать и что-нибудь с ними сделать. Читать и делать мы будем в классе `GameManager`, а конкретно, в конструкторе. Для работы с сохраненными настройками приложения используется класс `SharedPreferences`. Вся работа по чтению и применению настроек выглядит так:

SettingsScreen.java

```

public GameManager(SurfaceHolder surfaceHolder, Context context)
{
    ...

    // стили для рисования игрового поля
    ...

    // игровые объекты
    ...
    // применение настроек
    SharedPreferences settings =

```

```
PreferenceManager.getDefaultSharedPreferences(context);

    String difficulty =
settings.getString(res.getString(R.string.pref_difficulty),
res.getString(R.string.difficulty_normal));
    setDifficulty(difficulty);

    int maxScore =
Integer.parseInt(settings.getString(res.getString(R.string.pref_max_score),
"10"));
    setMaxScore(maxScore);
}
```

Метод `setDifficulty` приводить не буду, там ничего особо умного не написано.

Настройки из `SharedPreferences` можно читать с помощью методов `getString`, `getInt`, `getBoolean` и т.п. Все они принимают два параметра — ключ к настройке (то, что мы задавали с помощью атрибута `android:key`) и значение по умолчанию. Однако, воспользоваться чем-то кроме `getString` мне так и не удалось.

Заключение

Итак, мануал закончен. Получился он огромным, но при этом собственно про андроид оказалось не так уж и много, что самое-то обидное :(Спасибо, если кто дочитал до конца. Буду рада услышать любые мнения.

Отдельное спасибо **xeve** и **std.denis**

[А вот и все исходники](#)