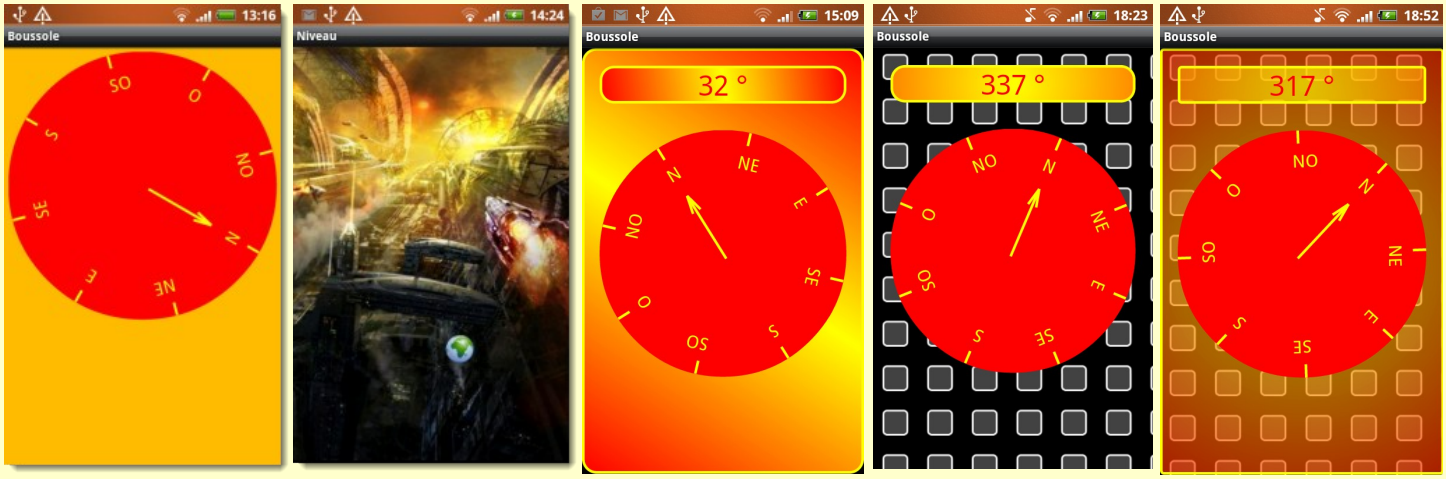
Cette étude fait suite à l'étude précédente sur les capteurs. Nous allons proposer cette fois-ci des tracés personnalisés afin d'obtenir un affichage qui va mieux correspondre aux capteurs que nous utilisons :

- Création d'une boussole.
- Déplacement d'une boule suivant l'horizontalité du mobile.
- Les **Drawables**.



TRACÉS PERSONNALISÉS - GRAPHISMES 2D

Lorsque vous devez utiliser un composant dont l'apparence est à votre libre initiative, la première chose à faire consiste à créer un nouveau composant qui hérite de la classe **View** et de redéfinir un certain nombre de méthode dont vous avez la description ci-dessous :

- ***onDraw(Canvas)*** : Méthode automatiquement appelée par le système lorsque la vue doit être affichée à l'écran. La méthode fournit un **Canvas** en paramètre qui est utilisé comme support des dessins. Je rappelle qu'il est possible d'imposer le réaffichage au travers de la méthode ***invalidate()*** du composant à réafficher.
- La méthode ***onDraw()*** contient toute la magie. Vous créez un nouveau widget à partir de zéro car vous voulez créer une interface visuelle entièrement nouvelle. La paramètre **Canvas** de la méthode ***onDraw()*** va vous permettre de donner vie à votre imagination.

*Android fournit une palette d'outils pour vous y aider, à l'aide de divers objets **Paint**. La classe **Canvas** comprend toutes les méthodes **drawXXX()** pour le dessin d'objet 2D primitifs comme les cercles, lignes, rectangles, textes et **Drawable**. Elle supporte également les transformations qui permettent la rotation, le déplacement et le redimmensionnement du **Canvas** pendant que nous dessinons.*

*Lorsque vous utilisez ces outils en les combinant à des **Drawable** et à la classe **Paint** (qui offre divers stylos et outils de remplissage personnalisables), la complexité et les détails que vos contrôles pourront afficher ne seront limités que par la taille de l'écran et la puissance du processeur.*

- ***onSizeChanged(int, int, int, int)*** : Cette méthode peut être redéfinie pour être notifié d'un changement de taille de la vue : les deux premiers paramètres définissent les anciennes largeur et hauteur, alors que les deux derniers sont les nouvelles dimensions. Elle est généralement utilisée pour initialiser certianes propriétés de la vue.
- ***onMeasure(int, int)*** : Cette méthode est automatiquement exécutée par le système lorsque ce dernier souhaite obtenir les dimensions choisies par la vue. Le fonctionnement de cette méthode est un peu particulier car il est impératif d'appeler la méthode ***setMeasuredDimension(int, int)*** lors de son exécution. Cette commande informe le système des dimensions (**largeur** et **hauteur**) choisies.
- La méthode ***onMeasure()*** est appelée lorsque le contrôle parent dispose ses contrôles enfants. Il leur demande quel est l'espace dont ils ont besoin et passe deux paramètres la **largeur** et la **hauteur**. Ils spécifient l'espace disponible pour le contrôle ainsi que quelques métadonnées décrivant cet espace. Plutôt que renvoyer un résultat, vous passez la hauteur et la largeur de la vue à la méthode ***setMeasureDimension()***.

*Les paramètres correspondant à la **largeur** et à la **hauteur** sont passés sous forme d'entiers pour des raisons d'efficacité. Avant de pouvoir être utilisés, il sdoivent être décodés à l'aide des méthodes statiques **getMode()** et **getSize()** de la classe **MeasureSpec**.*

- Selon la valeur de mode, **size** représente soit l'espace maximal (**fill_parent**) disponible pour le contrôle (si **AT_MOST**), soit la taille exacte (**wrap_content**) de votre contrôle (si **EXACTLY**). Si vous indiquez **UNSPECIFIED**, votre contrôle ne saura pas ce que la taille représente.
- En indiquant **EXACTLY**, le parent force la View à se placer dans un espace à la taille spécifiée. Dans le mode **AT_MOST**, il demande à la **View** quelle est la taille de l'espace qu'elle veut occuper, la limite haute étant spécifiée. Dans la plupart des cas, la valeur que vous renverrez sera la même.

*Dans les deux cas, vous devrez considérer ces limites comme absolues. Dans certaines circonstances, il peut être néanmoins approprié de renvoyer une dimension allant au-delà. Vous laisserez dans ce cas la parent choisir comment gérer le dépassement en utilisant des techniques comme le clipping (**bornage du résultat**) ou le scrolling (**défilement**).*

CLASSES SPÉCIFIQUES POUR DESSINER

Android fournit diverses classes permettant de dessiner. Les trois principales sont décrites ci-dessous :

- **Canvas** : En effectuant une analogie avec un peintre, nous pourrions comparer le **Canvas** à la toile (c'est d'ailleurs l'expression anglaise pour "**toile**") sur laquelle l'artiste dessine. Il autorise des opérations basiques, comme les translations, les rotations, etc. permettant de mimer les gestes du peintre sur la toile. Utilisé seul, un **Canvas** est clairement inutile. Pour tracer et dessiner sur cette feuille blanche (ou plutôt transparente), il est nécessaire d'utiliser un pinceau. Un **Canvas** n'a pas de taille ni d'existence réelle en tant que tel. Il doit être employé en conjonction d'un **Paint** et d'un **Bitmap** décrit ci-dessous.
- **Paint** : Traduit de façon littérale, le pinceau (ou **Paintbrush**) est un objet permettant de définir la forme et le style du dessin qui sera généré. **Paint** est donc porteur d'informations comme la couleur ou la largeur du trait, l'état de l'anti-aliasing, le style du texte, la façon dont est remplie la forme, etc.
- **Bitmap** : Il s'agit en quelque sorte de l'objet affiché. Un **Bitmap** est une image brute composée de pixels sur laquelle un **Canvas** est nécessaire lorsque nous souhaitons y dessiner.

*Pour se déplacer sur le **Canvas**, il est possible d'utiliser des méthodes telles que **translate(float, float)**, **rotate(float)**, **scale(float, float)**, **etc.** Dès lors qu'une de ces méthodes est appelée, l'état du **Canvas** est impacté. Ainsi, si nous passons d'un état A à B en utilisant la transformation x, puis de B à C par la transformation y, il est nécessaire de calculer les transformations inverses y' et x' pour revenir à l'état A. Cette opération est parfois difficile, il est alors possible de sauvegarder l'état actuel d'un **Canvas** grâce à la méthode **save()**. L'état du **Canvas** peut ensuite être restitué par un simple appel à **restore()**.*

- ***invalidate()*** : Cette méthode permet d'armer un bit système sur une vue décrivant son état "**impropre**". Cette méthode est donc extrêmement importante car elle **informe le système de la nécessité de redessiner** la vue en question. Lorsqu'un appel à ***invalidate()*** est effectué, Android mémorise la demande et exécutera une passe "**dessin**" dès que possible (au prochain tour de boucle événementielle).

CRÉATION D'UNE BOUSSOLE

Nous allons expérimenter ce que nous venons de découvrir en reprenant l'étude du capteur d'orientation. Je vous propose de réaliser une boussole qui nous donne le cap général.

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="fr.btsiris.compas"
        android:versionCode="1"  android:versionName="1.0">
    <application android:label="Boussole" >
        <activity android:name="Compas"  android:label="Boussole" android:screenOrientation="portrait">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

res/layout/main.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:padding="5dp"
    android:background="#FFBB00" >
  <fr.btsiris.compas.VueCompas
        android:id="@+id/boussole"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
</LinearLayout>
```

fr.btsiris.compas.Compas.java

```
public class Compas extends Activity implements SensorEventListener {
    private VueCompas boussole;
    private SensorManager gestionCapteurs;
    private Sensor orientation;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
```

```
        boussole = (VueCompas) findViewById(R.id.boussole);
        gestionCapteurs =  (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        orientation = gestionCapteurs.getDefaultSensor(Sensor.TYPE_ORIENTATION);
    }

    @Override
    protected void onStart() {
        super.onStart();
        gestionCapteurs.registerListener(this, orientation, SensorManager.SENSOR_DELAY_FASTEST);
    }

    @Override
    protected void onStop() {
        super.onStop();
        gestionCapteurs.unregisterListener(this);
    }

    public void onSensorChanged(SensorEvent evt) {
        if (evt.sensor.getType() == Sensor.TYPE_ORIENTATION) {
            boussole.setAzimut(evt.values[0]);
            boussole.invalidate();
        }
    }

    public void onAccuracyChanged(Sensor capteur, int precision) { }
}
```

Nous avons besoin d'une vue personnalisée qui s'occupe de réaliser les différents tracés pour la boussole elle-même.

fr.btsiris.compas.VueCompas.java

```
public class VueCompas extends View {
    private Paint texte, cercle, marque;
    private int hauteurTexte;
    private float azimut;

    public void setAzimut(float azimut) { this.azimut = azimut;  }

    public VueCompas(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);  init();
    }

    public VueCompas(Context context, AttributeSet attrs) {
        super(context, attrs);  init();
    }

    public VueCompas(Context context) {
        super(context);  init();
    }

    private void init() {
        cercle = new Paint(Paint.ANTI_ALIAS_FLAG);
        cercle.setColor(Color.RED);
        texte =  new Paint(Paint.ANTI_ALIAS_FLAG);
        texte.setColor(Color.YELLOW);
        texte.setTextSize(20);
        hauteurTexte = (int) texte.measureText("yY");
        marque =  new Paint(Paint.ANTI_ALIAS_FLAG);
        marque.setColor(Color.YELLOW);
        marque.setStrokeWidth(3);
    }

    @Override
    protected void onMeasure(int largeur, int hauteur) {
        int dimension = Math.min(mesure(largeur), mesure(hauteur));
        setMeasuredDimension(dimension, dimension);
    }
    private int mesure(int dimension) {
        int modeMesure = MeasureSpec.getMode(dimension);
        int tailleMesure = MeasureSpec.getSize(dimension);
        if (modeMesure == MeasureSpec.UNSPECIFIED) return 200;
        else return tailleMesure;
    }
```

```
    @Override
    protected void onDraw(Canvas canvas) {
        int px = getMeasuredWidth() / 2;
        int py = getMeasuredHeight() / 2;
        int rayon = Math.min(py, py);
        canvas.drawCircle(px, py, rayon, cercle);
        canvas.save();
        canvas.rotate(-azimut, px, py);
        int largeurTexte = (int) texte.measureText("O");
        int cardinalX = px - largeurTexte / 2;
        int cardinalY = py - rayon+hauteurTexte;
        for (int i=0; i<8; i++) {
            canvas.drawLine(px, py-rayon, px, py-rayon+15, marque);
            canvas.save();
            canvas.translate(0, hauteurTexte);
            String[] cap = {"N", "NE", "E", "SE", "S", "SO", "O", "NO"};
            if (i==0) {
                    int flecheY = 2*hauteurTexte;
                    canvas.drawLine(px, flecheY, px-5, 3*hauteurTexte, marque);
                    canvas.drawLine(px, flecheY, px+5, 3*hauteurTexte, marque);
                    canvas.drawLine(px, flecheY, px, 6*hauteurTexte, marque);
            }
            canvas.drawText(cap[i], cardinalX, cardinalY, texte);
            canvas.restore();
            canvas.rotate(45, px, py);
        }
        canvas.restore();
    }
}
```

✗ RESSOURCES DRAWABLE

Afin de faciliter la création de graphisme, Android offre une fonctionnalité extrêmement pratique : les **Drawables**. La notion de **Drawable** sous Android est assez large puisqu'elle définit tout ce qui peut être dessiné. Ainsi une couleur, une forme, un dégradé, une image, etc., sont considérés comme des **Drawables**.

*Android utilise largement les **Drawable**, il en fait donc un composant essentiel à son fonctionnement. L'utilité des **Drawables** réside dans les points donnés ci-après :*

- Un **Drawable** est une abstraction de ce qui est **"dessinable"**. Cette généralisation facilite grandement l'utilisation des graphismes tout en donnant facilement accès à des fonctionnalités plus évoluées que de simples **Bitmap** et **Canvas**.
- Les **Drawables** gèrent de façon intrinsèque leur taille et s'adaptent automatiquement à un changement de dimensions. Cette particularité permet de s'accomoder facilement des résolutions variées de l'écosystème des terminaux Android.
- Dans l'exemple précédent, il a fallu étendre **View** pour développer nos propres graphismes. Le concept de **Drawable** évite cet héritage en autorisant la modification d'une vue prédéfinie de façon native. La classe **View** autorise, par exemple, le développeur à modifier son fond par la méthode ***setBackgroundDrawable(Drawable)***. Ainsi, vous pouvez facilement imaginer une **View** disposant d'un fond dégradé, coloré ou même représentant une image.

✗ ARCHITECTURE DES DRAWABLES – PRINCIPALES FONCTIONNALITÉS

L'architecture des **Drawables** est largement inspirée de celle des **View**. En effet, un **Drawable** peut contenir d'autres **Drawables "feuille"**. Nous retrouvons donc le modèle de conception **"Composite"**.

*La principale différence entre ces deux architectures réside dans l'impossibilité d'instancier un **Drawable**. En effet, cette classe est abstraite et oblige le développeur à implémenter un certain nombre de méthodes lorqu'il souhaite créer sa propre instance de **Drawable**.*

- L'obtention d'une référence sur un **Drawable** s'effectue par l'intermédiaire d'un objet de type **android.content.res.Resources** :
- ```
  Drawable drawable = context.getResources().getDrawable(R.drawable.balle);
  ```

*Dès lors, il est possible de modifier la plupart des propriétés du **Drawable**. Parmi les fonctionnalités gérées par les **Drawables**, nous retrouvons :*

- **La taille et l'emplacement** à l'aide des méthodes suivantes:
  - ***setBounds(Rect)*** : Définit le rectangle dans lequel le Drawable sera affiché. Cette méthode permet donc de définir la taille du **Drawable**, mais également son origine.
  - ***getIntrinsicWidth(), getIntrinsicHeight()*** : Retourne, si possible, la taille intrinsèque du **Drawable**. Ainsi, les dimensions intrinsèques d'une image sont les dimensions de l'image elle-même, alors qu'un **Drawable** de type **"*couleur*"** n'a pas de taille intrinsèque (la valeur retournée par défaut est alors égale à -1).
  - ***getMinimumWidth(), getMinimumHeight()*** : Retourne les dimensions minimales suggérées par le **Drawable**.
- **L'état** : ***setState(int)*** : Un **Drawable** peut changer son apparence en fonction de l'état dans lequel il se trouve. A titre d'exemple, un bouton n'a pas la même apparence lorsqu'il est dans l'état pressé et lorsqu'aucune action n'est exécutée.
  ```

- **Le niveau** : Certains **Drawables** changent d'apparence en fonction du niveau courant. Ainsi, il est possible de définir des **Drawables** qui changent d'apparence facilement. Ces **Drawables** sont principalement utilisés pour représenter des barres de progression ou des icônes exprimant le niveau courant d'une entité graduée (qualité de la réception, état de la batterie, etc.).
- **Les propriétés d'affichage** : La classe **Drawable** autorise la modification de l'apparence générale du graphique sous-jacent. Des méthodes permettant de modifier l'opacité - ***setAlapha(int)*** - ou le filtre de couleur - ***setColorFilter(ColorFilter)*** - sont ainsi disponibles.

DÉPLACEMENT D'UNE BOULE SUIVANT L'HORIZONTALITÉ DU MOBILE

Nous allons réutiliser l'accéléromètre avec cette fois-ci une boule que se déplace suivant l'horizontalité de votre smartphone. Pour ce projet Il est nécessaire de récupérer une image de fond et une image de la boule que vous placerez dans la ressource ***drawable***.

fr.btsiris.niveau.Niveau.java


```
package fr.btsiris.niveau;

import android.view.*;

public class Accelerometre extends Activity implements SensorEventListener {
    private SensorManager gestionCapteurs;
    private Sensor accelerometre;
    private SurfaceView surface;
    private SurfaceHolder holder;
    private Bitmap boule, fond;
    private float bx, by, vx, vy, tx, ty;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        boule = BitmapFactory.decodeResource(getResources(), R.drawable.boule);
        fond = BitmapFactory.decodeResource(getResources(), R.drawable.fond);
        surface = new SurfaceView(this);
        holder = surface.getHolder();
        gestionCapteurs =  (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        accelerometre = gestionCapteurs.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        setContentView(surface);
    }

    @Override
    protected void onStart() {
        super.onStart();
        gestionCapteurs.registerListener(this, accelerometre, SensorManager.SENSOR_DELAY_GAME);
    }

    @Override
    protected void onStop() {
        super.onStop();
        gestionCapteurs.unregisterListener(this, accelerometre);
    }

    public void onSensorChanged(SensorEvent evt) {
        if (evt.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {

            // Ajout les valeurs du capteur aux variables
            vx -= evt.values[0];
            vy += evt.values[1];

            // Déplacement de la boule
            bx += vx;
            by += vy;

            // Récupération de la taille de l'écran
            tx = surface.getWidth() - boule.getWidth();
            ty = surface.getHeight() - boule.getHeight();

            // Empêcher la boule de sortir
            if (bx<0)  { bx = 0; vx = 0; }
            if (by<0)  { by = 0; vy = 0; }
            if (bx>tx) { bx = tx; vx = 0; }
            if (by>ty) { by = ty; vy = 0; }
```

```
            // Redessiner
            Canvas dessin = holder.lockCanvas();
            if (dessin==null) return;
            dessin.drawBitmap(fond, 0, 0, null);
            dessin.drawBitmap(boule, bx, by, null);
            holder.unlockCanvasAndPost(dessin);
        }
    }

    public void onAccuracyChanged(Sensor capteur, int precision) {  }
}
```

✗ TOUR D'HORIZON DES DRAWABLES

A l'instar d'une **View**, un **Drawable** peut être instancié via XML ou Java. Dans la suite de cette rubrique, nous allons principalement nous attarder sur le code XML exactement pour les mêmes raisons que celles évoquées pour la mise en œuvre des **View**.

Il est tout de même important de garder à l'esprit que l'instanciation via XML n'est en réalité qu'une surcouche de l'instanciation via du code Java. A ce titre, les fonctionnalités accessibles dans un fichier XML sont un sous-ensemble de celles offertes par Java.

✗ Android contient de nombreux types de ressources **Drawable** simples qui peuvent être entièrement définies en XML. Ces ressources comprennent les classes **ColorDrawable**, **ShapeDrawable** et **GradientDrawable**. Elles sont stockées dans le dossier ***res/drawable*** et peuvent ainsi être référencées dans du code en utilisant leurs noms de fichiers XML en minuscules.

✗ Si ces **Drawables** sont définis en XML et que vous spécifiiez leurs attributs en pixels à densité indépendante, le moteur d'exécution les mettra à l'échelle. Tout comme les graphiques vectoriels, ces **Drawables** peuvent être dynamiquement mis à l'échelle afin d'être affichés correctement et sans artifices, et ce quelles que soient la taille de l'écran, sa résolution ou sa densité en pixels. Les **Drawables gradient** constituent une exception à cette règle puisque le rayon du gradient doit être défini en pixels.

✗ COLORDRAWABLE

Le **ColorDrawable** est une objet n'affichant qu'une unique couleur. De façon assez logique, le **ColorDrawable** ne respecte pas le **ColorFilter** et les dimensions données par ***setBounds(Rect)***. Il s'affiche en réalité dans l'intégralité de la zone de clip.

*Le **ColorDrawable** n'est en fait qu'une encapsulation d'une couleur dans la notion de **Drawable**. Puisque Android gère indépendamment les ressources de type **"couleur"**, nous utiliserons rarement ce type de **Drawable**. Il est néanmoins à noter qu'un **setBackgroundColor(int)** génère en réalité un **ColorDrawable** à partir de la couleur donnée.*

✗ Le **ColorDrawable** est le plus simple des **Drawables** définis en XML. Il vous permet d'afficher une image basée sur une couleur unie. Les **ColorDrawables** sont définis dans des fichiers XML situés dans le dossier de ressources ***drawable***, en utilisant la balise **<color>**.

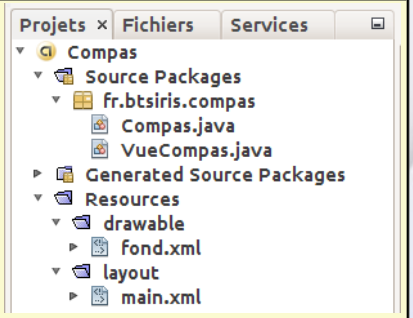
*A titre d'exemple, je vous propose de reprendre le projet sur la **boussole** afin de prévoir un fond de couleur unie personnalisée (bien que nous l'ayons mise en œuvre d'une autre façon) dans l'activité principale de l'application.*

res/drawable/fond.xml

```
<?xml version="1.0" encoding="utf-8"?>
<color
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:color="#FFCC00" />
```

res/layout/main.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:padding="5dp"
    android:background="@drawable/fond" >
   <fr.btsiris.compas.VueCompas
      android:id="@+id/boussole"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent" />
</LinearLayout>
```



✗ SHAPEDRAWABLE

Les **ShapeDrawables** (***forme***) vous permettent de définir des formes primitives simples en spécifiant leurs dimensions, fond et contour, à l'aide de la balise ***<shape>***.

*Chaque forme est constitué d'un type (spécifié par l'attribut **shape**), d'attributs de dimension et de sous-noeuds spécifiant l'épaisseur du contour et la couleur de fond. Android supporte actuellement les formes suivantes comme valeur de l'attribut **shape**.*

- **oval** :Simple ovale.
- **rectangle** : Supporte le sous-noeud **<corners>** avec l'attribut **radius** pour créer un rectangle aux angles arrondis. Il est même possible de régler chaque coin particulier au moyen des attributs spécifiques suivants : **topLeftRadius, topRightRadius, bottomLeftRadius, bottomRightRadius.**
- **ring** : Supporte les attributs **innerRadius** et **thickness** qui vous permettent de spécifier respectivement le rayon intérieur de la forme anneau ainsi que son épaisseur. Vous pouvez également utiliser **innerRadiusRatio** et/ou **thicknessRatio** pour définir le rayon intérieur de l'anneau et son épaisseur proportionnellement à sa largeur (un rayon intérieur d'un quart de la largeur aura la valeur 4).
- **line** : Ligne horizontale dont la largeur traverse la totalité du composant **View**. Cette forme particulière requiert le sous-noeud **<stroke>** pour définir la largeur et la couleur de la ligne.
- Le sous-noeud **<stroke>** permet de spécifier la bordure des formes à l'aide des attributs **width** et **color**. Il existe deux autres attributs **dashGap** et **dashWidth** qui permettent d'avoir des lignes discontinues en spécifiant respectivement la distance entre les traits et la longueur des traits.
- Vous pouvez également inclure un nœud **<padding>** pour positionner votre forme sur le **Canvas** de manière relative.
- De façon plus utile, vous pouvez inclure un sous-noeud spécifiant la couleur de fond. Le cas le plus simple consiste à utiliser le nœud **<solid>** avec l'attribut **color** pour définir la couleur de fond unie.
- Le dégradé de couleur, qui sera étudié spécifiquement dans la prochaine section, se traduit au moyen du sous-noeud **<gradient>**.
- Le sous-noeud <size> permet de préciser une dimension spécifique à l'aide des attributs **width** et **height**.
- **useLevel** : Cet attribut de la balise **<shape>**, normalement associé à **LevelListDrawable**, doit être spécifié à **false** si vous voulez qu'il soit visible

*Je reprend l'exemple précédent, mais cette fois-ci, c'est un nouveau **TextView** qui visualise la valeur de la mesure et qui profite du fond créé dans le **drawable**.*

res/drawable/fond.xml

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
        android:shape="rectangle" >
          <solid android:color="#FF8800" />
          <stroke android:width="2dp" android:color="#FF0000" />
          <corners android:radius="10dp" />
          <padding android:top="7dp" android:bottom="7dp" />
</shape>
```

res/layout/main.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:padding="20dp">
   <TextView
      android:id="@+id/cap"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"

      android:textColor="#FFFF00"
      android:textSize="32sp"
      android:gravity="center_horizontal"
      android:background="@drawable/fond" />
   <LinearLayout
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:paddingTop="30dp">
      <fr.btsiris.compas.VueCompas
          android:id="@+id/boussole"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"   />
   </LinearLayout>
</LinearLayout>
```

fr.btsiris.compas.VueCompas.java

```
package fr.btsiris.compas;

public class Compas extends Activity implements SensorEventListener {
    private VueCompas boussole;
    private TextView cap;
    private SensorManager gestionCapteurs;
    private Sensor orientation;
```

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        boussole = (VueCompas) findViewById(R.id.boussole);
        cap = (TextView) findViewById(R.id.cap);
        gestionCapteurs =  (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        orientation = gestionCapteurs.getDefaultSensor(Sensor.TYPE_ORIENTATION);
    }

    @Override
    protected void onStart() {
        super.onStart();
        gestionCapteurs.registerListener(this, orientation, SensorManager.SENSOR_DELAY_FASTEST);
    }

    @Override
    protected void onStop() {
        super.onStop();
        gestionCapteurs.unregisterListener(this);
    }

    public void onSensorChanged(SensorEvent evt) {
        if (evt.sensor.getType() == Sensor.TYPE_ORIENTATION) {
            Number valeur = evt.values[0];
            boussole.setAzimut(valeur.floatValue());
            cap.setText(valeur.intValue()+ " °");
            boussole.invalidate();
        }
    }

    public void onAccuracyChanged(Sensor capteur, int precision) { }
}
```

✗ GRADIENTDRAWABLE

Comme son nom le laisse à penser, le **GradientDrawable** permet de reproduire des dégradés de couleur. Chaque gradient définit une transition harmonieuse entre deux ou trois couleurs de façon linéaire, radiale ou circulaire.

*Les **GradientDrawable** sont définis par la balise **<gradient>** comme un sous-noeud de la définition d'un **ShapeDrawable**, au travers de la balise principale **<shape>**. Chaque **GradientDrawable** requiert au moins un attribut **startColor** et un attribut **endColor** et supporte l'attribut optionnel **middleColor**., ainsi que l'attribut **type**. L'attribut **type** vous permet de choisir votre gradient parmi les suivants :*

- ✗ **linear** : Le type par défaut. Il affiche une transition de couleurs directe de **startColor** à **endColor** suivant un angle défini par l'attribut **angle**.
- ✗ **radial** : Dessine un gradient radial de **startColor** à **endColor** depuis le bord externe de la forme jusqu'à son centre. Il requiert un attribut **gradientRadius** qui spécifie le rayon de la transition du gradient en pixels. Il supporte également optionnellement les attributs **centerX** et **centerY** pour décaler le centre du gradient. Le rayon étant défini en pixels, il ne sera pas mis à l'échelle dynamiquement pour différentes densités de pixels. Pour minimiser l'effet d'escalier, vous pourrez spécifier différentes valeurs de rayon pour différentes résolutions d'écrans.
- ✗ **sweep** : Dessine un gradient circulaire de **startColor** à **endColor** le long du bord externe de la forme parent (un anneau, typiquement).

*Voici ce que nous pouvons obtenir respectivement avec un **dégradé linéaire**, un **radial** et **radial dont le centre est choisi** sur le bord gauche en bas :*

(linéaire) res/drawable/fond.xml

```
<?xml version="1.0" encoding="utf-8"?>
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle" >
        <stroke android:width="3dp" android:color="#FFFF00" />
        <corners android:radius="15dp" />
        <gradient
                android:startColor="#FF0000"
                android:endColor="#FF0000"
                android:centerColor="#FFFF00"
                android:type="linear"
                android:angle="45" />
</shape>
```

res/layout/main.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:padding="20dp"
    android:background="@drawable/fond" >
  <TextView
      android:id="@+id/cap"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:textColor="#FF0000"
      android:textSize="32sp"
      android:gravity="center_horizontal"
      android:background="@drawable/fond" />
  <LinearLayout
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"

       android:paddingTop="30dp">
      <fr.btsiris.compas.VueCompas
          android:id="@+id/boussole"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"   />
  </LinearLayout>
</LinearLayout>
```

(radial) res/drawable/fond.xml

```
<?xml version="1.0" encoding="utf-8"?>
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle" >
        <stroke android:width="3dp" android:color="#FFFF00" />
        <corners android:radius="15dp" />
        <gradient
                android:startColor="#FFFF00"
                android:endColor="#FF0000"
                android:type="radial"
                android:gradientRadius="300"
        />


</shape>
```

(radial avec une valeur de position) res/drawable/fond.xml

```
<?xml version="1.0" encoding="utf-8"?>
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle" >
        <stroke android:width="3dp" android:color="#FFFF00" />
        <corners android:radius="15dp" />
        <gradient
                android:startColor="#FFFF00"
                android:endColor="#FF0000"
                android:type="radial"
                android:gradientRadius="700"
                android:centerX="1"
                android:centerY="1"
        />

</shape>
```

BITMAPDRAWABLE

Le **BitmapDrawable** est probablement le plus utilisé des **Drawables**. Cet objet est la représentation sous forme de Drawable d'une image de format jpg ou png. Nous récupérons donc souvent ce dernier par un simple ***getDrawable()*** sur la ressource **"brute"**.

*Il est néanmoins possible de créer un **BitmapDrawable** via XML à l'aide de la balise **<bitmap />** et d'utiliser les paramètres avancés. Voici les attributs disponibles :*

- **android:antialias** : Lors de l'affichage de l'image, il est possible de proposer un anti-aliasing en positionnant cet attribut à **true**. Cet attribut est utile dans le cas d'un rééchantillonage automatique de votre image à visualiser sur la totalité de l'écran, si cette dernière est plus petite que ce que propose l'écran.
- **android:filter** : Dans le même ordre d'idée, il est possible de valider l'attribut **filter** (**true**) lorsque vous avez besoin que votre image soit plus douce (flou glaussien).
- **android:gravity** : A l'instar du **FrameLayout**, le **BitmapDrawable** gère la notion de gravité. Il est ainsi possible de positionner une image à l'intérieur de la zone d'affichage du **Drawable** (les bounds). Par défaut, la gravité est à **fill** ce qui signifie que l'image est étirée pour remplir intégralement les bounds. Voici l'ensemble des valeurs possibles pour cet attribut : **top, bottom, left, right, center_vertical, fill_vertical, center_horizontal, fill_horizontal, center, fill, clip_vertical, clip_horizontal, start et end.**
- **android:src** : Ressource de l'image, votre fichier original.
- **android:tileMode** : Le **BitmapDrawable** gère également la notion de répétition d'un motif. Cette possibilité est en quelque sorte l'équivalent de la propriété CSS **background-repeat**. En réalité, voici les valeurs possibles pour cet attribut : **disabled** (mode par défaut), **clamp** (Réplication de la couleur de la bordure), **repeat** (répétition en x et en y) et **mirror** (répétition également, mais avec un effet miroir - symétrie).

*Toujours par rapport au projet précédent, je propose de placer une trame de fond prédéfinie dans l'activité principale, que nous pouvons retrouver dans les ressources android, dans les constantes entières qui sont à votre disposition, ici **android.R.drawable.dialog_frame** :*

res/drawable/fond.xml

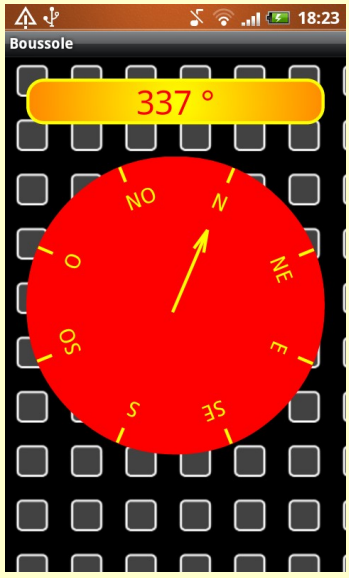
```
<?xml version="1.0" encoding="utf-8"?>
<bitmap  xmlns:android="http://schemas.android.com/apk/res/android"
      android:src="@android:drawable/dialog_frame"
      android:tileMode="repeat" />
```

res/drawable/cap.xml

```
<?xml version="1.0" encoding="utf-8"?>
<shape
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:shape="rectangle" >
          <stroke android:width="3dp" android:color="#FFFF00" />
          <corners android:radius="15dp" />
          <gradient android:startColor="#FFFF00"
                    android:endColor="#FF0000"
                    android:type="radial"
                    android:gradientRadius="300"  />
</shape>
```

res/layout/main.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:padding="20dp"
     android:background="@drawable/fond" >
    <TextView
        android:id="@+id/cap"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textColor="#FF0000"
        android:textSize="32sp"
        android:gravity="center_horizontal"
        android:background="@drawable/cap" />
    <LinearLayout
         android:layout_width="fill_parent"
         android:layout_height="wrap_content"
         android:paddingTop="30dp">
        <fr.btsiris.compas.VueCompas
            android:id="@+id/boussole"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent" />
    </LinearLayout>
</LinearLayout>
```



DRAWABLE LAYER

Le **LayerDrawable** vous permet d'assembler plusieurs ressources **Drawable** les unes sur les autres. Si vous définissez un tableau de **Drawables** partiellement transparents, vous pourrez les empiler les uns sur les autres afin de créer des combinaisons de formes dynamiques et de transformations.

*De façon similaire, vous pouvez utiliser les **LayerDrawables** comme source des **Drawables** transformables décrits dans la section précédente.*

Le listing suivant montre un **LayerDrawable**. Il est défini via la balise ***<layer-list>***. Dans cette balise, l'attribut **drawable** de chaque sous-noeud ***<item>*** définit des **Drawables** comme des composites. Chaque **Drawable** sera empilé dans l'ordre de l'index, du coup, le premier du tableau sera finalement placé au bas de la pile.

res/drawable/fond.xml

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list  xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@drawable/trame" />
    <item android:drawable="@drawable/cap" />
</layer-list>
```

res/drawable/trame.xml

```
<?xml version="1.0" encoding="utf-8"?>
<bitmap  xmlns:android="http://schemas.android.com/apk/res/android"
      android:src="@android:drawable/dialog_frame"
     android:tileMode="repeat" />
```

res/drawable/cap.xml

```
<?xml version="1.0" encoding="utf-8"?>
<shape
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:shape="rectangle" >
          <stroke android:width="3dp" android:color="#FFFF00" />
          <corners android:radius="3dp" />
          <gradient android:startColor="#AAFFFF00"
                    android:endColor="#AAFF0000"
                    android:type="radial"
                    android:gradientRadius="350"  />
</shape>
```