Fisheye View - 1ère partie
- version Java/Android-

Nous allons appliquer le principe des fisheye views, en nous appuyant sur la thèse de Frédéric Vernier (http://www.limsi.fr/Individu/vernier/). Voici les formules proposée par Volkmar Hovestadt au milieu des années 90 :

$$FDeform(x) = x \times \frac{\sqrt{(x^2+z^2)(r^2-(z-o)^2)+z^2(z-o)^2}+z(z-o)}{x^2+z^2}$$

$$FDeform^{-1} = \frac{z \times x}{\sqrt{r^2-x^2}+(z-o)}$$

La fonction FDeform permet de transformer l'espace : FDeform(0) = 0 (pas de déformation du centre) pour x proche de 0, FDeform(x) sera plus grande (en valeur absolue) et FDeform tend vers une valeur finie quand x tend vers l'infini.

x est une distance en Pixel avec le centre de la déformation.

Citons Frédéric Vernier : « Les paramètres r, o et z permettent d'adapter la fonction aux besoins de l'interaction. Le paramètre r fixe la valeur limite de la fonction. Il fixe donc la surface de visualisation à l'écran […] Les paramètres o et z fixent les proportions de la zone agrandie et de la zone rétrécie par rapport à la surface totale de visualisation ».

Le principe est donc le suivant : étant donnée une image, il faut calculer la place de chaque pixel après transformation. Ainsi, pour chaque pixel p(i,j) nous calculons les nouvelles coordonnées image pour y affecter la valeur du pixel.

Soit i et j les coordoonnées (en pixel) du point p(i,j) à déformer. Soit dist la distance (en pixel) entre ce point et le centre de la transformation c(i_{centre}, j_{centre}). Le nouveau point sera à la distance FDeform(dist) et sur la même ligne que (c(i_{centre}, j_{centre}), p(i,j)). Le calcul est donc :

scale = FDeform(dist)/dist;

$i' = i_{centre} + (i - i_{centre}) * scale$

$j' = j_{centre} + (j - j_{centre}) * scale$

Ainsi les nouvelles coordonnées de p(i,j) sont (i', j');

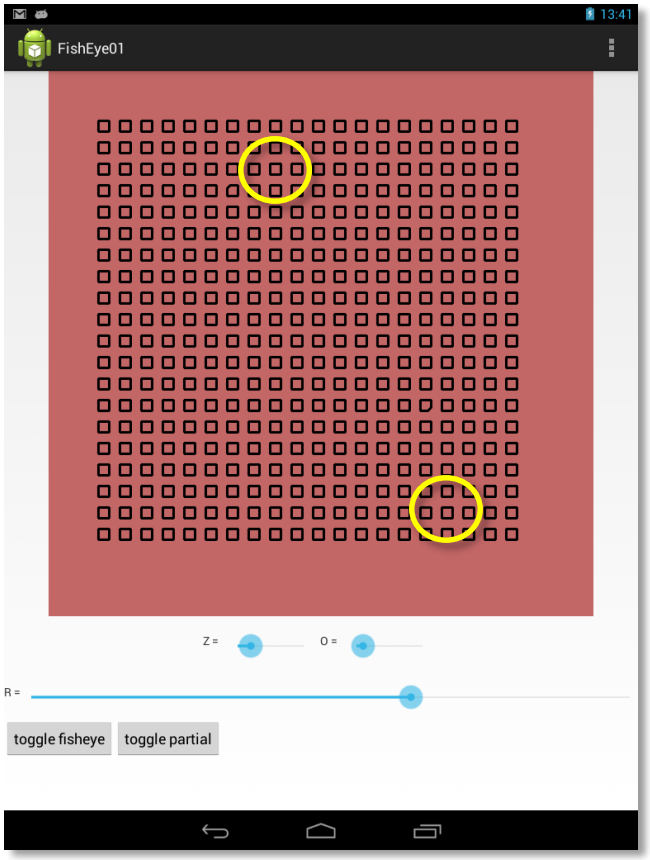
Optimisation :

Pour des raisons d'optimisation nous remarquerons que pour tous les points équidistants du centre, la valeur de "scale" sera la même. Pour une image de taille *n* x *m* pixels, pour les abscisses,

FDeform(i) sera calculée $\sqrt{n^2+m^2}$ fois (car c'est la distance la plus grande possible) plutôt que n*m fois.

Toujours pour des raisons d'implémentation, dans le calcul de FDeform, nous remarquerons aussi qu'une partie des termes peuvent être calculé.

Réalisation : 1 - Déformation d'un objet décomposable en polygone



Partez d'un nouveau projet, avec une "blank activity".

Comme il est préférable de déformer une représentation « vectorielle », nous allons travailler sur une « image » que nous allons générer. Nous ferons la version "image" à la fin du tp.

Cette image sera « re-créée » à partir d'une liste de formes géométriques (carrés ou polygones) dont on stockera les coordonnées des sommets (dans l'image non déformée, à l'échelle 1).

Attention, il y a deux limitations avec Android :

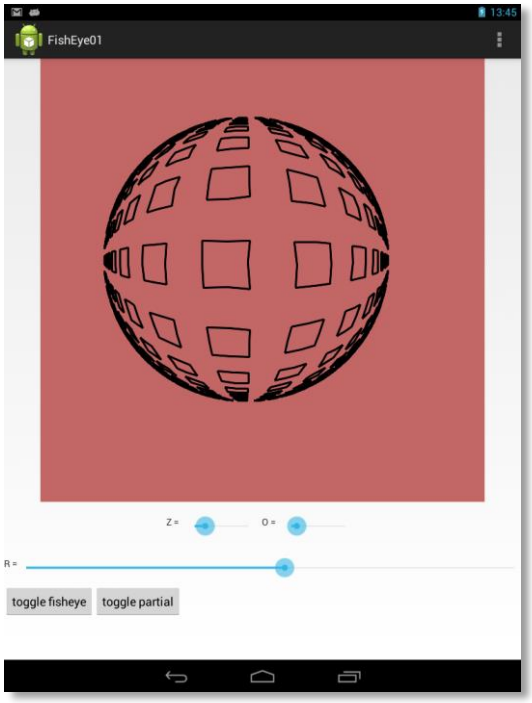
- Il n'y a pas de Polygon (contrairement en awt / swing). Il faut refaire sa classe.
- Il n'y a pas de méthode dans Canvas pour dessiner un polygone (il faut dessiner toutes les lignes), avec une boucle ou utiliser un objet Path, en utilisant moveTo puis lineTo, c.f. http://developer.android.com/reference/android/graphics/Path.html (ce dernier semble être un peu plus lent que la boucle ci-dessous)

```
for(MyPolygon p :  elts)
{
    paint.setColor(p.color);
    // liste des points sous la forme d'un tableau (x1, y1, x2, y2, ...)
    // les (xi, yi) sont les sommets
    float [] pts = p.getPoints();
    for(int i = 0; i<pts.length-3; i=i+2)
          {
          g.drawLine(pts[i], pts[i+1], pts[i+2], pts[i+3], paint);
          }
    // ligne entre le dernier sommet et le premier somment
    g.drawLine(pts[pts.length-2], pts[pts.length-1], pts[0], pts[1], paint);
}
```

Pour faire nos propres composants graphiques sous android, nous pouvons nous baser sur les explications disponibles en lignes :
http://developer.android.com/guide/topics/ui/custom-components.html

La génération des carrés / polygones peut ressembler à cela :

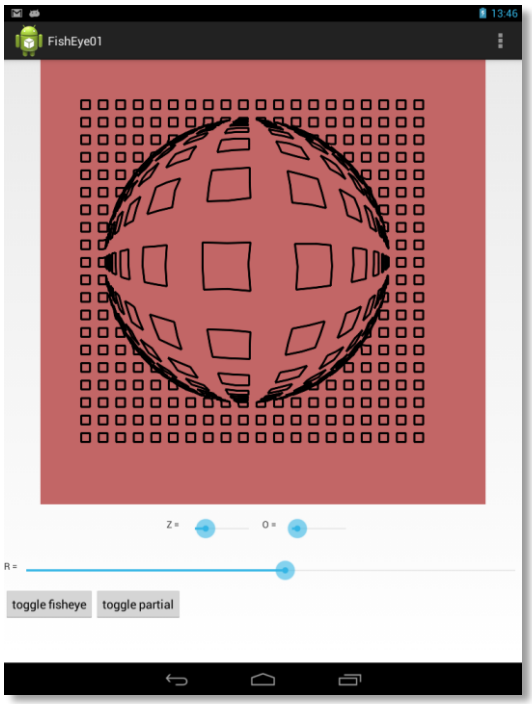
```
protected void generatePolygons()       {
       elements = new ArrayList<MyPolygon>();

       // dimension dans laquelle s'inscrit un polygone
       float w = (originalSize[0]-marges*2) / (nb*2);
       float h = (originalSize[1]-marges*2) / (nb*2);

        // pour faire quelques carres différents
       int tiers = nb /3 ;
        int sixieme = nb /6 ;
        int deuxtiers = 2*nb /3 ;
        int troisquarts = 3*nb /4 ;

        float pasW = w/4;
        float pasH = h/4;


       // création de tous les polygones
       for(int i = 0; i < nb; i++)      {
              for(int j = 0; j < nb; j++)                    {
                     MyPolygon p = new MyPolygon();
                     float dx = w*2*i+marges;
                     float dy = h*2*j+marges;


                     // ajout des points constituants les polygones
                     if ((i == tiers) && (j==sixieme)) {
                            p.addPoint(dx, dy+pasH);
                            p.addPoint(dx+pasW, dy);

                     } else {
                            p.addPoint(dx, dy);
                     }

                     p.addPoint(dx+w/2, dy);
                     p.addPoint(dx+w, dy);
                     p.addPoint(dx+w, dy+h/2);

                     if ((i == troisquarts) && (j==deuxtiers)) {
                      p.addPoint(dx+w, dy-pasH+h);
                            p.addPoint(dx-pasW+w, dy+h);

                     } else {
                            p.addPoint(dx + w, dy + h);
                     }

                     p.addPoint(dx+w/2, dy+h);
                     p.addPoint(dx, dy+h);
                     p.addPoint(dx, dy+h/2);

                     p.color = Color.BLACK;
                     // ou une autre couleur qui dépend de i et de j

                     elements.add(p);
                     }
              }

       }
```



Dans un premier temps, vous pouvez faire une simple déformation centrée sur un point fixe.

`Pour gérer la taille de la vue créée, il faudra bien surcharger la méthode onMeasure(int widthMeasureSpec, int heightMeasureSpec) qui fera appel à setMeasuredDimension(int width, int height)`. Comme le montre l'exemple `dans le dossier android-sdk\samples\android-10\ApiDemos\src\com\example\android\apis\view\LabelView.java` les paramètres de `onMeasure()` ne sont pas directement la taille mais une spécification de la taille…

Réalisation : 2 - Déformation dynamique et partielle



Il est possible de limiter la déformation là où c'est nécessaire c'est quand la distance n'est plus changée : FDeform(x) = x, en dehors de la solution 0. Il faut donc résoudre :

$$x\frac{\sqrt{(x^2+z^2)(r^2-(z-o)^2)+z^2(z-o)^2}+z(z-o)}{x^2+z^2} = x$$

Ce qui se "simplifie" en :

$$x^4 + x^2(2zo - r^2 + (z-o)^2) + z^2(o^2 - r^2) = 0$$

Il s'agit donc d'une équation du second degré en x^2, qui peut donc se résoudre par discriminent et en écartant la solution négative : $x^2 = \frac{-(2zo-r^2+(z-o)^2)+\sqrt{(2zo-r^2+(z-o)^2)^2-4z^2(o^2-r^2)}}{2}$

Au-delà de cette solution, FDeform(x) < x, et l'espace « information » se conpact. L'information est donc moins lisible. Modifiez donc votre objet déformable pour autoriser une déformation restreinte en appliquant la déformation que pour les points proches du centre de déformation (distance inférieure ou égale à la solution).

Vous pourrez remarquer que pour o très petit (0 ou 1), alors la solution est très proche de r.
Ce calcul ne vaut que pour les formules de Volkmar Hovestadt.

Finalement, finalisez votre application :

- Permettez le changement des valeurs des paramètres, le choix de la formule, etc.
  - Pour changer les valeurs des paramètres une SeekBar (et les événements OnSeekBarChange) peut être utilisée.

Pensez à rendre votre application le plus proche possible d'une application « utilisable » : changement d'orientation, etc.

Note: d'autres formules

Il n'y a pas qu'une seule formule de Fisheye View. Implémentez aussi d'autres formules, par exemple une formule simplifiée (c.f. thèse de Frédéric Vernier)

$$FDeform(x) = \frac{R \times Z}{-Z - x} + R$$

$$FDeform^{-1}(x) = \frac{R \times Z}{R - x} - Z$$

Pour cette formule, FDeform(x) = x est vraie pour x = R-Z.

Chaque formule donnera évidemment un résultat différent. Ce qui compte, c'est de respecter les propriétés requises :

La fonction FDeform permet de transformer l'espace : FDeform(0) = 0 (pas de déformation du centre) pour x proche de 0, FDeform(x) sera plus grande (en valeur absolue) et FDeform tend vers une valeur finie quand x tend vers l'infini.

Réalisation : 3 – Déplacement du Fisheye

Avec la méthode onTouchEvent de View, permettez le déplacement de la fisheye en changeant le centre de la déformation. Posez-vous les questions suivantes :

- L'endroit touché est-il directement le nouveau centre de déformation, ou faut-il passer par un calcul (avec $deform^{-1}$) pour trouver le nouveau centre ?
- Faut-il décaler les coordonnées pour ne pas masquer la déformation avec le doigt ?

Réalisation : 4 – Déformation d'image

En faisant attention à la mémoire, déformez une image (jpg ou png).

Pour cela, vous pouvez utiliser des Bitmap :

- Pour initialiser votre image :

```
Resources res = getResources();
source = BitmapFactory.decodeResource(res, R.drawable.id_de_l_image);
```

- Pour récupérer le tableau de pixel :

```
originalWidth = source.getWidth();
originalHeight = source.getHeight();
pixelsSource = new int[originalHeight * originalWidth];
```

```
source.getPixels(pixelsSource, 0, source.getWidth(), 0, 0, originalWidth,
originalHeight);
```

- Pour faire une Bitmap à partir d'un tableau de pixel :
```
calculatedBitmap = Bitmap.createBitmap(pixelsDest, originalWidth,
originalHeight, Config.ARGB_8888);
```
- Pour afficher une Bitmap :
```
// g est un Canvas, dans onDraw
g.drawBitmap(toDraw, rectSrc, rectDst, paint);
```

Vous pouvez combler les trous dans l'image calculée en faisant la moyenne des pixels affectés autours des pixels non affectés par le calcul. Ceci fera un effet de "flou", mais comblera les trous.
Attention à la mémoire requise et aux temps de calculs (dépend de la dimension de l'image).