Fisheye View - 1ère partie

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 proposées par Volkmar Hovestadt au milieu des années 90, x étant une distance en Pixel avec le centre de la déformation :

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

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

<u>Propriétés requises</u> : 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.

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 <u>propriétés requises</u>.

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.

Étape de la réalisation du Fisheye

Vous pouvez faire ce travail en java android ou en javascript. Les étapes restent les mêmes, les principes aussi, il y a cependant des adaptations à chaque cas. Voici donc les étapes, les indications par plateformes sont données après.

Étape 1 : Déformation d'un objet décomposable en polygone

Comme il est préférable de déformer une représentation « vectorielle », nous allons travailler sur une « image » que nous allons générer. Nous ferons la version "image" plus tard (plus gourmand en calcul, avec le problème des « trous ».

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).

Il nous faudra une classe Polygon pour avoir les formes initiales et les formes déformées. Ces Polygon seront dessinés dans un « canvas ».

Dans un premier temps, vous pouvez faire une simple déformation centrée sur un point fixe, avec des paramètres de déformation également fixes, par exemple r = 350, z = 150 et o = 100.

Étape 2 - Déformation 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 compacte. 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).

Étape 3 - Déformation dynamique

Il s'agit simplement de pouvoir modifier les paramètres de la déformation. Ajoutez des interacteurs type seekbar / jauge (ou autre) et des écouteurs. À chaque changement de valeur, il faut mettre à jour la déformation et recalculer l'image.

Étape 4 – Déplacement du Fisheye

Il s'agit simplement de pouvoir modifier le centre de la déformation. Ajoutez des interacteurs type « évènements de souris » ou « déplacement du doigt » . À chaque changement de valeur, il faut mettre à jour la déformation et recalculer l'image.

Étape 5 - Déformation d'image

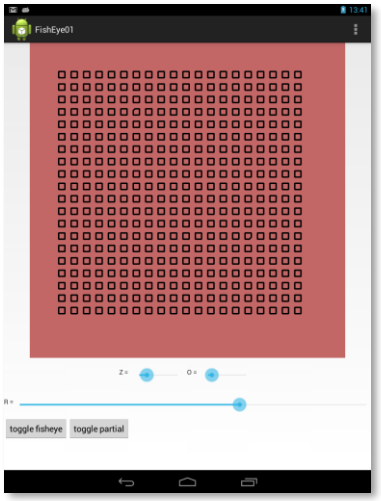
À la place de déformer une image « vectorielle à nous », essayez de déformer une image (photo). Pour cela il faut charger l'image, travailler sur ses pixels donc transformer l'image en tableau de pixel. À partir de là, il est possible de construire un autre tableau de pixel (généralement à une dimension), comprenant les pixels déformés. La valeur du pixel (i,j) généralement à l'indice i+j*width, sera affecté à (i',j') soit l'indice i'+j'*width. Une fois le nouveau tableau calculé, il faut refaire une image pour l'afficher.

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).

Indications pour Android

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



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

Attention, il y a deux limitations avec Android :

- Il n'y a pas de Polygon. Il faut refaire sa classe: MyPolygon.java.
- 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)

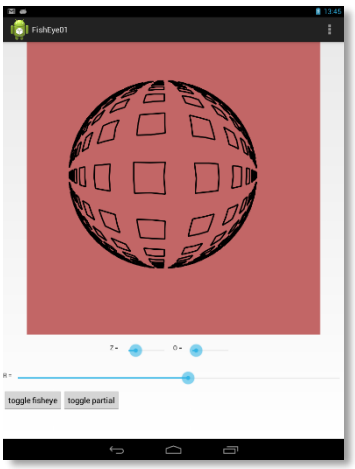
```
for(MyPolygon p :  elts)
{
    paint.setColor(p.color);
    // liste des points sous la forme d'un tableau (x1, y1, x2, y2, ...)
    // les (xi, yi) sont les sommets
    float [] pts = p.getPoints();
    for(int i = 0; i<pts.length-3; i=i+2)
         {
         g.drawLine(pts[i], pts[i+1], pts[i+2],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

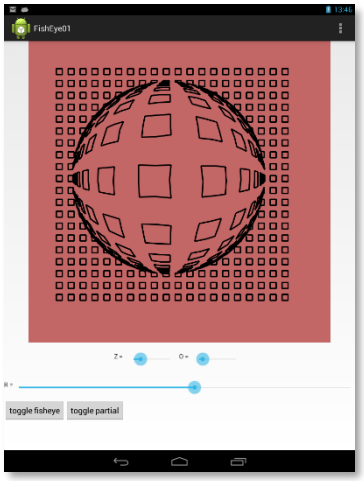
```
// création de tous les polygones
for(int i = 0; i < nb; i++)      {
       for(int j = 0; j < nb; j++)                    {
              MyPolygon p = new MyPolygon();
              float dx = w*2*i+marges;
              float dy = h*2*j+marges;


              // 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);
              }
       }

}
```



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-17\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 et 3 – Déformation partielle et dynamique



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.

Réalisation : 4 – 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 :

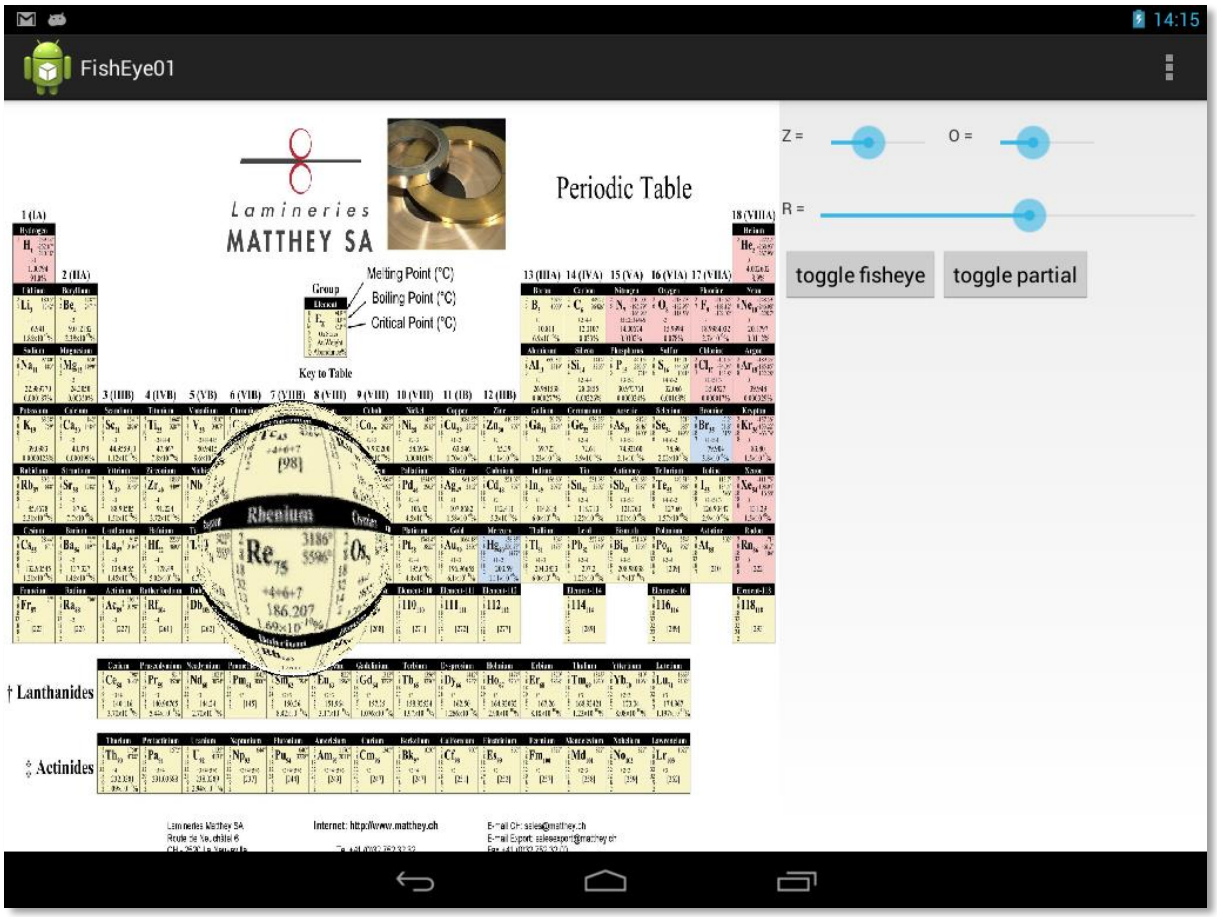
- L'endroit touché est-il directement le nouveau centre de déformation, ou faut-il passer par un calcul (avec $deform^{-1}$) pour trouver le nouveau centre ?
- Faut-il décaler les coordonnées pour ne pas masquer la déformation avec le doigt ?

Réalisation : 5 – 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);
```


14:15
FishEye01
Z =
O =
R =
toggle fisheye
toggle partial
Lamineries
MATTHEY SA
Periodic Table
Melting Point (°C)
Boiling Point (°C)
Critical Point (°C)
Key to Table
Rhenium
† Lanthanides
‡ Actinides

Indications pour Javascript

Suiviez les indications fournies en cours.

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

Voici une classe Polygone, qui inclut une méthode draw :

```
/**
*      Polygon
*  @class
*/
class Polygon {
              /**
              * @param (optionnal) {String} color
              * @param (optionnal) {Array<Point>} p
              */
              constructor (color, p) {
                     // Points array that define Polygon
                     this.points = p || [];
                     // Polygon color
                     this.color = color || "black";
              }

              /**
               * @param {Number} x
               * @param {Number} y
               */
              addPoint(x, y) {
                     this.points.push(new Point(x,y));
              }

              /**
               * Draw the polygon
               * @param {CanvasContext} ctx
               */
              draw (ctx) {
                     const p1 = this.points[0];
                     ctx.beginPath();
                     // Begin by the first point
                     ctx.moveTo(p1.x,p1.y);
                     // Draw a line for each point in polygon
                     for (let i = 0; i < this.points.length; i++) {
                            const p = this.points[i];
                            ctx.lineTo(p.x,p.y);
                     }
                     // Need to draw the last line between last point and first
                     ctx.lineTo(p1.x,p1.y);
                     // To not be
                     ctx.lineWidth = 2;
                     ctx.strokeStyle = this.color;
                     ctx.stroke();
              }
       }
```

Voici aussi du code pour générer des polygones (dans une même classe) :

```
generatePolygons (width, height, nb, marge) {
	// Calculate polygons dimension
	const w = (width - marge * 2) / (nb * 2);
	const h = (height - marge * 2) / (nb * 2);
	// Generate all polygons
	for (let i = 0; i < nb; i++) {
		for (let j = 0; j < nb; j++) {
			// X and Y Coordinate
			const dx = w * 2 * i + marge;
			const dy = h * 2 * j + marge;
			// Random polygon generation
			const rdmType = Math.floor(Math.random()*3);
			const rdmColor = Math.floor(Math.random()*3);
			const color = PolygonManager.POLYGONS_COLOR[rdmColor];
			// Polygon creation
			let poly;
			if (PolygonManager.POLYGONS_TYPE[rdmType] === "carre") {
				poly = PolygonManager.createCarre(dx,dy,w,h,color);
			}
			else if (PolygonManager.POLYGONS_TYPE[rdmType] === "triangle")
				{
				poly = PolygonManager.createTriangle(dx,dy,w,h,color);
				}
			else if (PolygonManager.POLYGONS_TYPE[rdmType] === "losange")
				{
				poly = PolygonManager.createLosange(dx,dy,w,h,color);
				}
			this.polygons.push(poly);
			this.polygonsDeform.push(poly);
			}
	}
}

// Triangle
static createTriangle(x,y,w,h,color) {
	const poly = new Polygon(color);
	poly.addPoint(x, y);
	poly.addPoint(x + w / 2, y + h);
	poly.addPoint(x + w, y);
	return poly;
	}

// Carre
static createCarre(x,y,w,h,color) {
	const poly = new Polygon(color);
	poly.addPoint(x, y);
	poly.addPoint(x + w, y);
	poly.addPoint(x + w, y + h);
	poly.addPoint(x, y + h);
	return poly;
	}

// Losange
static createLosange(x,y,w,h,color) {
	const poly = new Polygon(color);
	poly.addPoint(x + w / 2, y);
	poly.addPoint(x + w, y + h / 2);
	poly.addPoint(x + w / 2, y + h);
	poly.addPoint(x, y + h / 2);
	return poly;
	}

// Use to convert number into type or color
static get POLYGONS_TYPE() {return ["carre","triangle","losange"]}
static get POLYGONS_COLOR() {return ["red","blue","orange"]}
```

Réalisation : 2 et 3 – Déformation partielle et dynamique

Pour écouter les changements de valeurs, utilisez des addEventListener sur l'événement "input" (à adapter selon l'interacteur). Il faut alors changer les valeurs et redessiner.

Réalisation : 4 – Déplacement du Fisheye

Pour écouter le <canvas>, selon que vous voulez capter les événements souris ou « touch », il faudra s'abonner à des événements (addEventListener) pour "mousemove" ou "touchstart" + "touchmove".

Réalisation : 5 – Déformation d'image

Voici deux extraits de code, ici de la même classe, pour manipuler une image.

Pour transformer une image en tableau de pixel :

```
// Load Image
let img = new Image();
img.src = pathImg; // chemin vers l'image
// ctx est le contexte d'un canvas (où est dessinée l'image)
img.onload = () => {
	this.ctx.drawImage(img, 0, 0);
	// image source
	this.img = this.ctx.getImageData(0,0,this.canvas.width,this.canvas.height);
	// image déformée
	this.deformImg =
this.ctx.getImageData(0,0,this.canvas.width,this.canvas.height);
	// le tableau est dans this.deformImg.data
			};
```

Et pour dessiner :

```
if (this.img) {
	// ctx est le contexte d'un canvas (où est dessinée l'image)
	this.fishEyeDeform();
	this.ctx.putImageData(this.deformImg, 0, 0);
	for (let i = 0; i < this.img.data.length; i++) {
		this.deformImg.data[i] = this.img.data[i];
		}
	}
```