Jonathan Branger

Alix Ducros

Jo-Hakim Sayarh

Pierre Thomazon

(Albéric Boissau)

Projet 2ème SI

Candy Crush Killer

Année 2013-2014 IUT Clermont-Ferrand 1 Département Informatique

Nous remercions P. Kauffmann et J. Laffont de nous avoir encadrés pour ce projet tutoré

"Nous autorisons la diffusion de notre rapport sur l'intranet de l'IUT"

Sommaire

1) INTRODUCTION

Dans le cadre de nos études nous devions réaliser un projet pédagogique, ce projet nommé Candy Crush Killer nous a été confié par M. Kauffmann enseignant chercheur à l'IUT de Clermont Ferrand et M. Laffont enseignant à Polytech Clermont qui nous ont encadrés et accompagnés tout au long de sa réalisation. Le projet a duré 117 jours, du 18 novembre 2013 au 27 mars 2014. L'équipe initiale était composé de cinq étudiants ayant tous suivi la coloration SI : Albéric Boisseau, Jonathan Branger, Alix Ducros, Jo-Hakim Sayarh et Pierre Thomazon. Albéric étant retourné en première année, nous avons continué le projet à quatre.

Le sujet du projet était le suivant : réaliser une application Android capable de jouer de manière indépendante au célèbre jeu Candy Crush Saga[1]. Cet objectif nous a tous beaucoup motivé car il impliquait des réalisations ambitieuses et nouvelles pour nous : l'initiation à la programmation sur Android, le traitement d'une image et la conception d'une intelligence artificielle.

Par ailleurs, ce sujet nous a posé un grand nombre de soucis techniques, tels que l'apprentissage en autodidacte du langage Java, la prise en main d'outils nouveaux tels que SVN ou éclipse. Quant à ceux d'entre nous qui ont travaillé sur Android ils ont dû prendre en main son SDK. Pour les autres apprendre comment coder un Intelligence Artificielle ou des techniques de modifications d'image. En plus il nous a fallu appliquer pour la première fois des méthodes apprises en cours de gestion de projet en plus des concepts objet bien assimilés au fil de notre apprentissage.

Au fil de ce rapport nous allons présenter notre sujet ainsi que les outils que nous avons utilisés pour le réaliser. Ensuite nous développerons le travail de chacun, les perspectives d'évolution pour finir sur le bilan de ce que chaque membre de notre groupe a appris.

2) **PRESENTATION DU PROJET**

2.1) Présentation en détail du sujet

Le projet Candy Crush Killer se compose de 3 grandes parties :

- Une partie « prise d'image »
- Une partie « analyse d'image »
- Une partie « intelligence artificielle »

- "Prise d'image" : dans cette partie, l'application va prendre une photo de l'écran d'un ordinateur où se déroule une partie de Candy Crush Saga, déterminer la zone de jeu et corriger les éventuelles erreurs (luminosité, déformations ...). Cette partie a été réalisée directement sur Android.

- "Analyse d'image" : ici, on s'occupe de déterminer la nature des bonbons présents dans le jeu via un algorithme qui parcourt l'image plusieurs fois afin de bien déterminer la zone de décor et les différents types de bonbon.

- "Intelligence artificielle" : cette partie s'occupe de déterminer le meilleur coup à jouer et prend le contrôle du pointeur de la souris sur l'ordinateur afin de jouer le coup.

2.2) Présentation du matériel

Le smartphone qui nous a été confié est un Samsung Galaxy S4[2], le dernier modèle smartphone de Samsung (*cf. Figure 1 : Galaxy S4, test du support*) fonctionnant sous Android 4.2.2 et est équipé d'un appareil photo de 13 mégapixels. Mais nous savons tous qu'un même code ne marche pas forcément sur tous les smartphones, et qu'il fallait donc un autre smartphone pour tester l'application. Alix a utilisé son smartphone, un Wiko CINK Five permettant ainsi à deux personnes de travailler simultanément sur la partie Android avec un support physique.

Cependant, il fallait trouver un moyen de faire tenir le Smartphone "debout". Pour cela, nous avons réalisé un support en utilisant des pièces LEGO Technics (*cf. Figure 2 : support en LEGO Technics*) adapté au Galaxy S4 qui ne peut pas porter des smartphones plus épais.

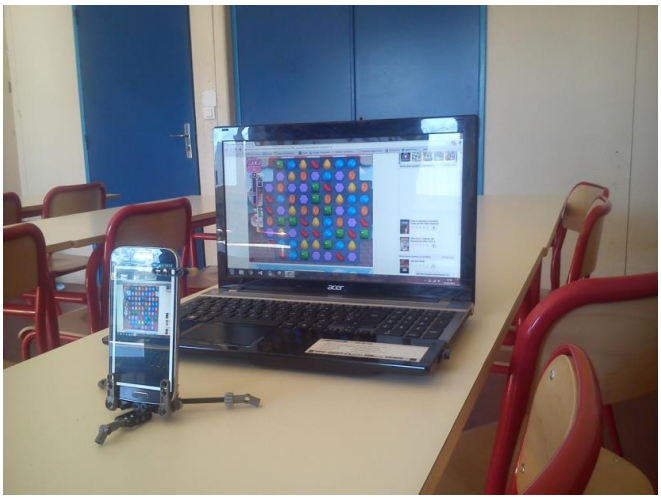

Figure 1 : Galaxy S4, test du support

Figure 2 : support en LEGO Technics

2.3) Présentation des logiciels

Nous avions à notre disposition un outil de mise en commun du code : la forge. Cette application nous a permis de mettre à jour et de récupérer le code de tous les membres du groupe grâce au gestionnaire de version SVN, et elle nous a également offert l'accès à un

forum, un wiki et à un diagramme de Gantt, dont la principale fonction est de définir des points de repère à respecter pour ne pas être en retard dans le projet.

Nous nous sommes ensuite accordés sur l'utilisation d'un IDE[3] commun. Notre choix s'est naturellement porté vers Eclipse. Nous avons téléchargé la dernière version (Kepler) et avons installé le logiciel ensemble afin d'être sûr d'avoir les mêmes modules. De plus afin de pouvoir programmer pour la plateforme Android, nous nous sommes procuré le SDK Android fourni par Google ainsi que le plugin ADT permettant d'émuler un terminal sur nos machine de développement.

2.4) Gestion du projet

Nous nous sommes réparti les tâches en fonction des affinités de chacun : Alix se chargerait de l'acquisition d'image à partir de la caméra, et Jonathan s'occuperait du calibrage et du traitement de l'image. Cette dernière serait envoyée au code fournit par Pierre, qui analyserait l'image. Une fois l'image analysée, l'IA de Jo-Hakim déterminerait le coup le plus rentable.

Nous nous sommes mis d'accord sur les types de données que chacun enverrait à l'autre de façon à pouvoir rassembler les différentes parties du code sans avoir à gérer une phase d'adaptation. De ce fait, nous avons travaillé façon plutôt indépendante : nous communiquions principalement par mail lorsqu'un problème se présentait, et nous profitions des séances prévues dans l'emploi du temps pour mettre en commun notre travail, définir les objectifs de la semaine à venir et avoir un contact régulier avec nos tuteurs de projet (une heure par semaine).

Les objectifs à réaliser au cours du projet étaient décrits dans un diagramme de Gantt (*cf. figure 3 : diagramme de Gantt prévisionnel).*

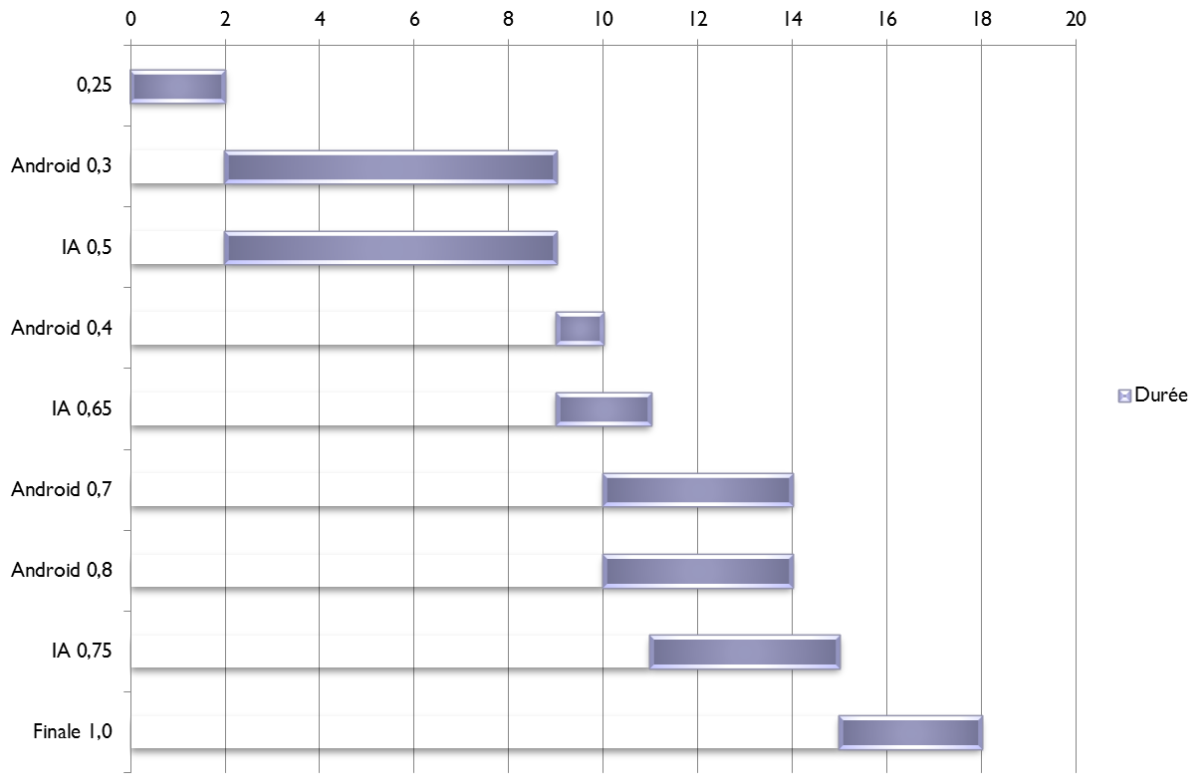

Figure 3 : diagramme de Gantt prévisionnel

0.25 : IA basique, mise en place du code sur le dépôt.

Android 0.3 : Capture d'image sur Android, traitement de l'image sur Android, correction de la distorsion de l'image.

IA 0.5 : IA favorisant les bonbons spéciaux par rapport aux bonbons normaux

Android 0.4 : Envoi de l'image

IA 0.65 : IA récursive

Android 0.7 : Portage de l'IA sous Android

Android 0.8 : Application complete

IA 0.75 : Étude d'un nouveau jeu

Finale 1.0 : Application finale

Nous avons établi les différentes versions en fonction des éléments que devait regrouper l'application finale. De cette manière, les objectifs ainsi fixés étaient plus faciles, rapides à atteindre et à appréhender.

3) Développement

3.1) Programmation Android

3.1.1) Contrôle de la caméra

La capture d'une image grâce au téléphone Android est une priorité du projet, c'est en effet en partie grâce à cette fonctionnalité que l'application peut gagner son autonomie et ne dépendre que d'elle-même en ce qui concerne l'acquisition des données qui lui sont nécessaires. Ce travail s'est déroulé en deux étapes principales.

Nous avons tout d'abord commencé en faisant appel à l'application de capture de photo d'Android, en créant un Intent[4] avec l'action `MediaStore.ACTION_IMAGE_CAPTURE`

Nous venions alors juste d'appréhender les bases d'Android et cette manière de faire était la plus abordable au vu de notre niveau à ce moment du développement : le code étant facilement compréhensible et très court.

```
Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri);
startActivityForResult(intent, PHOTO_RESULT);
```

Figure 4: Exemple de l'application

Mais si cette solution était rapide et simple à mettre en œuvre, nous nous sommes vite rendu compte qu'elle créerait rapidement des limites aux possibilités de l'application. En effet en passant ainsi la main à une application tierce pour la prise de photo, nous perdons totalement le contrôle sur les paramètres de capture et laissons la possibilité à l'utilisateur de changer la résolution de la photo, appliquer un filtre, ... Ce qui n'est évidement pas souhaitable. Nous nous sommes alors penchés sur une deuxième solution pour capturer une image : utiliser la caméra via la classe `Camera`.

Accéder de cette manière à la caméra nous donne un contrôle complet sur le périphérique, mais demande en revanche un code plus complexe et une gestion des ressources en fonction des transitions d'état de l'application. La procédure peut se décomposer en deux parties : la gestion de la caméra en elle-même et la gestion de la prévisualisation.

Le code permettant l'utilisation de la caméra est relativement simple :

```
Camera camera = Camera.open();
```

Figure 5 : code caméra

On peut ensuite récupérer les paramètres de la caméra afin d'assigner une taille d'image, un mode de flash, de focus, …

```
Camera.Parameters parameters=camera.getParameters();
parameters.setPictureSize(2048, 1152);
camera.setParameters(parameters);
```

Figure 6 : récupération paramètre

Par défaut, l'image renvoyée par la caméra est en orientation paysage, ce qui rend la prévisualisation désagréable pour l'utilisateur. Ce problème est corrigé en utilisant la méthode `setDisplayOrientation(90)` à laquelle en donne l'angle de rotation de l'image reçue.

La prise de la photo se fait en appelant la méthode :

```
takePicture(ShutterCallBack shutter, PictureCallback raw, PictureCallback
postView, PictureCallback jpeg);
```

Figure 7 : Prise photo

Les différents arguments de cette méthode sont des Callbacks[5] appelés à des moments précis du processus de prise de photo : le premier est appelé au moment précis de la capture, on peut lui assigner par exemple un bruit permettant de signaler à l'utilisateur que la photo a été prise, le deuxième argument (et les suivants) est un `PictureCallback`, il contient donc une méthode `onPictureTaken(byte[], Camera)` dans lequel on peut définir notre utilisation de l'image (contenue dans le tableau de `byte[]`). Dans notre cas nous mettrons tous les arguments à `null` sauf le dernier car il sera appelé après le redimensionnement de l'image conformément aux paramètres que nous avons précédemment passé à la caméra et la compression de l'image au format JPEG. Nous écrivons alors simplement l'image ainsi formatée sur le stockage du téléphone.

La gestion de la prévisualisation se fait par le biais d'une `SurfaceView`, via la classe `SurfaceHolder` qui permet de contrôler la taille, le format, d'éditer les pixels de la surface de prévisualisation.

```
SurfaceView su = (SurfaceView)findViewById(R.id.preview);
SurfaceHolder holder = su.getHolder();
```

Figure 8 : Contrôle pixel

Le SurfaceHolder ainsi créé va pouvoir gérer trois différents Callbacks qui correspondent au cycle de vie de la SurfaceView :

Le Callback `surfaceCreated(SurfaceHolder holder)` correspond à la création de la surface, c'est dans son corps que l'on appelle les méthodes commençant la prévisualisation.

```
camera.setPreviewDisplay(holder);
camera.startPreview();
```

Figure 9: création surface

Le Callback `surfaceChanged(SurfaceHolder holder, int format, int width, int height)` sera appelé lorsque la surface subira une transformation (par exemple la rotation du téléphone entraîne un redimensionnement automatique), il est donc nécessaire de stopper puis redémarrer la prévisualisation lorsque ce Callback est appelé.

Enfin, le Callback `surfaceDestroyed(SurfaceHolder holder)` correspondant à la destruction

de la surface, arrête la prévisualisation, et libère la caméra.

3.1.2) Création d'une IHM

L'IHM a été réalisée de façon à être simple et intuitive pour l'utilisateur. La première vue (*cf. Figure 1 : Galaxy S4, test du support*) celle que l'on voit lorsqu'on lance l'application, se compose d'un bouton, "Photo", en transparence. Ce bouton permet de prendre la photo qui sera traitée par la suite. La prévisualisation quand à elle est contenue dans une `SurfaceView` dont le fonctionnement a été développé précédemment.

Une fois la photo prise, l'application bascule sur une nouvelle vue qui affiche la photo prise et demande à l'utilisateur de placer les 4 icônes en forme de croix sur les 4 coins de la zone de jeu (*cf. Figure 2 : support en LEGO Technics*). La vue dispose également d'un bouton "Finish" qui termine l'opération de placement des icones et qui lance les opérations de redimensionnement et de morphose[6] sur l'image.
Lorsque cette vue est active, il suffit de toucher un endroit de l'écran pour y déposer l'icone active. A chaque fois qu'on dépose une des quatre icones, l'application passe à l'icone suivant, en boucle. Ainsi, il est possible de corriger une éventuelle erreur de placement.

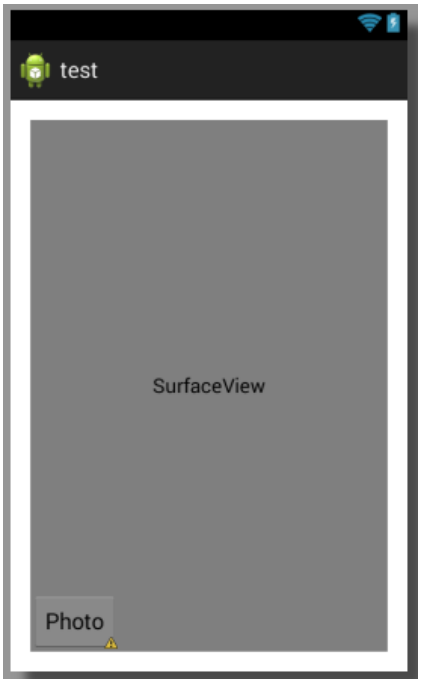

Figure 10 : application

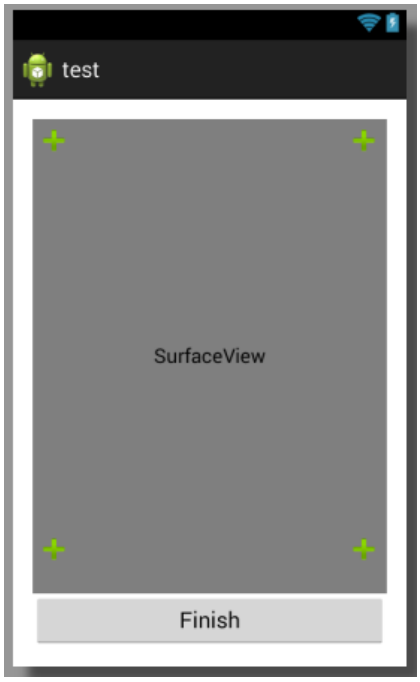

Figure 11 : application

Le résultat de cette partie de l'application est une photo qui contient uniquement la zone de jeu, mais cette image peut ressembler à un trapèze, ce qui gêne son analyse. Cette photo est donc traitée par la partie plus spécifique d'Alix qui effectue une morphose pour obtenir une image rectangulaire prête à être analysée.
La solution optimale serait d'automatiser ce processus, c'est à dire que le smartphone devrait prendre lui-même les photos et conserver uniquement la zone de jeu, mais cela demandait une analyse supplémentaire d'image longue et difficile à réaliser.
La principale caractéristique de l'IHM est le changement d'activité. C'est d'ailleurs sur la base d'un code permettant de cliquer sur un bouton pour lancer une autre activité et d'afficher "hello world" nous avons commencé à coder l'IHM.

Une activité n'est ni plus ni moins qu'un processus doté d'une interface graphique avec laquelle on peut interagir via des boutons, de l'écran tactile, ... pour effectuer diverses actions (écrire, prendre une photo ou déplacer des icones par exemple).

Lorsqu'on veut passer à une autre activité, il faut commencer par enregistrer le contexte de l'activité courante dans un Intent. On peut également enregistrer des informations dans l'intent qui seront passées à l'autre activité. Dans notre cas, cela nous a permis de passer le chemin d'enregistrement de la photo à l'autre activité afin de l'ouvrir et l'afficher dans la 2e activité. En effet, il n'est pas possible de partager directement l'objet Bitmap contenant l'image d'une activité à l'autre, la taille des extra ne pouvant excéder 1Mo.

L'idée de départ étant d'automatiser la prise de photo, la deuxième activité devrait à terme disparaître au fil des améliorations.

3.1.3) Redressement de l'image

Afin que l'algorithme de traitement d'image puisse fonctionner correctement, il est nécessaire de lui fournir une image de la zone de jeu qui soit parfaitement droite et sans bordures superflues. Malheureusement, la photo que prendra la caméra sera toujours imparfaite à cause du placement ou de l'inclinaison du smartphone et de l'environnement graphique de la zone de jeu. La solution envisagée est dès lors :

- transformer la perspective de l'image
- extraire la zone de jeu dans l'image

La transformation de la perspective peut s'appréhender de plusieurs manières, nous nous sommes plus particulièrement intéressés à la transformation par le biais d'une matrice. Pour cela, la bibliothèque `java.awt` nous semblait totalement appropriée car elle propose de nombreuses méthodes allant dans ce sens. Malheureusement, il se trouve que cette bibliothèque n'a pas d'implémentation sur Android. Les recherches dans l'API[7] Android ont fait émerger une méthode dans la bibliothèque `android.graphics` dont la finalité correspondait tout à fait à nos besoins :

```
Matrix.setPolyToPoly(float[] src, int srcIndex, float[] dst, int dstIndex, int pointCount)
```

En effet, cette méthode a pour but de créer une matrice permettant la transformation faisant en sorte que les nouvelles coordonnées des points spécifiés dans `src` correspondent aux coordonnées spécifiées dans `dst`. Les points de `src` étant les 4 points récupérés manuellement dans l'activité suivant la prise de photo.

```
Matrix transform = new Matrix() ;
//On crée la matrice de transformation
transform.setPolyToPoly(src, 0, dst, 0, src.length >> 1);
```

Il ne reste alors plus qu'à appliquer cette matrice à l'image source pour obtenir l'image redressée. Pour cela, on crée une nouvelle Bitmap :

```
Bitmap newImage = Bitmap.createBitmap(bmp,0, 0, bmp.getWidth(), bmp.getHeight(), transform, true);
```

Seulement, les résultats escomptés sont assez éloignés de ceux que nous obtenons :

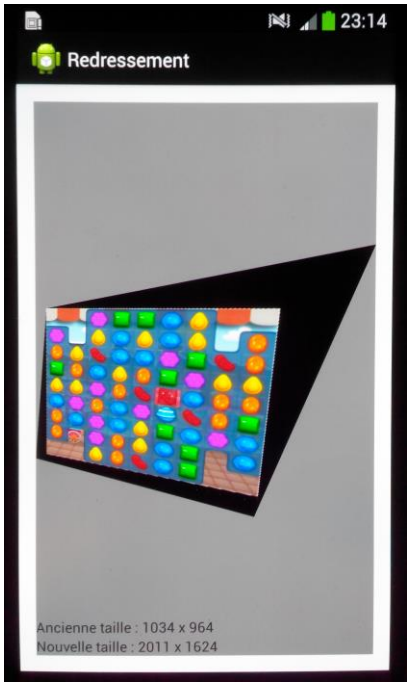

Figure 12 : Image résultante

Figure 13 : image source

Comme on peut le voir sur les figures ci-dessus, la transformation est correcte sur trois coins, mais semble incorrecte sur le coin en bas à droite. Le problème est récurrent avec d'autres images avec des coordonnées sources différentes. N'ayant à ce jour pas encore réussi à surmonter ce problème, l'étape d'extraction de la zone de jeu parmi le reste de l'image a été mise en attente.

3.1.4) Sélection de la zone de jeu

Plusieurs solutions se sont présentées pour cette partie du code :

- Une analyse d'image qui détecterait automatiquement la zone de jeu. C'était l'idée initiale, mais cela aurait demandé trop de temps à implémenter. Cette solution a été abandonnée très vite.

- Une interface munie de boutons directionnels pour placer des icones représentant les coins de la zone de jeu. N'étant pas encore très à l'aide avec Android, c'est la solution pour laquelle nous avons opté dans un premier temps. 4 flèches directionnelles permettaient de déplacer une icone, puis un bouton permettait de passer au bouton suivant. Cependant, cette solution est loin d'être ergonomique et a été remplacée par la solution définitive décrite ci-dessous.

- Une interface quasi-exclusivement vierge, composée uniquement d'un bouton "Finish". Les 4 icones se déplacent là où on touche l'écran, sachant qu'à chaque fois qu'une icone est placée, c'est l'icone suivante qui prend la main, et le compteur tourne en boucle afin de pouvoir corriger d'éventuelles erreurs de placement. Cependant, ce n'est pas encore parfait car le premier réflexe lorsqu'on fait une erreur est de la corriger, et pas de passer à la suite jusqu'à ce qu'on puisse revenir sur l'erreur. Une solution simple à implémenter serait d'intégrer un bouton qui remplacerait l'incrémentation automatique du "compteur-bouton" (un entier qui prend des valeurs de 0 à 3, chaque valeur correspondant à une icone; cf. extrait de code).

```
public boolean onTouch(View v, MotionEvent event) {
            if(event.getAction() == MotionEvent.ACTION_DOWN) {
                  float posX = event.getX();
                  tabview[currentB].setX(posX);

                  float posY = event.getY();
                  tabview[currentB].setY(posY);
                  // On change la position de l'icone dont le numéro correspond à
                  son indice dans le tableau currentB
```

```
                currentB = (currentB +1)%4;
                //incrémentation de l'indice du tableau pour changer d'icone
                "active".
            }
            return true;
        }
```

figure 14 : méthode de gestion du déplacement des icones (les commentaires ont été rajoutés pour pouvoir comprendre sans avoir le reste du code sous les yeux)

En raison de nombreux bugs rencontrés sur cette partie, le reste de cette partie de l'application (à savoir la récupération des coordonnées et leur interprétation) n'a pas pu être implémentée.

Une fois que le bouton "Finish" est pressé, chaque icone va envoyer à l'application ses coordonnées afin de déterminer la nouvelle image. Ensuite, un calcul via une matrice permettra de construire une nouvelle image à partir des pixels compris dans la zone délimitée par les coordonnées récupérées. Cette nouvelle image sera enregistrée tandis que l'ancienne sera supprimée, et le chemin de la nouvelle image est passé en paramètre aux fonctions implémentées par Alix pour réaliser la morphose de l'image.

3-2 - Traitement d'image

3-2-1) Détection du décor

La zone de jeu correspond à :

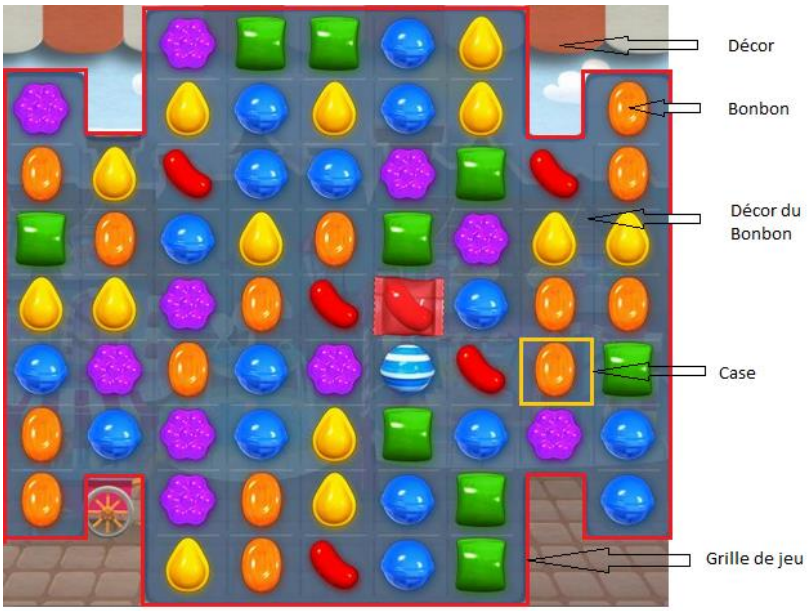

figure 15 : Grille de jeu

Dans cette zone de jeu il y a deux principales zones différentes :

- **La grille de jeu**: cette grille est séparée en cases, et à l'intérieur de ces cases, on peut distinguer à nouveau deux zones :

• *Le bonbon*, la couleur du bonbon est composée de trois composantes : rouge, bleu et vert (nous ne prenons pas en compte alpha)

• *Le décor du bonbon* , qui entoure le bonbon

- **Le décor du jeu**: c'est ce qu'il y a derrière les bonbons et autour de la grille de jeu.

Dans cette partie nous allons montrer comment on distingue le cas où c'est un bonbon et le cas où c'est du décor.

Nous avons identifié le décor de deux méthodes dédiées à la recherche de décor et une autre méthode qui permet de vérifier si c'est du décor que nous détaillerons dans *II-2-4) détection de couleur*. Nous avons préféré créer trois méthodes pour identifier le décor afin d'être

certains que ce soit du décor, car dans le cas où l'on veut déplacer un bonbon alors que c'est du décor, il y aura une boucle à l'infinie car on ne déplacera rien, la photo sera la même que la précédente, et la conclusion du meilleur coup sera le même. Dans la mesure où nous voulons éviter une boucle à l'infinie nous essayons d'être sûr des emplacements de bonbons et de décors.

1ère méthode :

figure 16 : bonbon avec bandes verticales

La figure ci-dessus représente une case de la grille avec le bonbon (de couleur bleu) et le décor (de couleur plus ou moins grise).
Dans cette méthode nous récupérons les deux parties surlignées au-dessus (à gauche et à droite du bonbon), chacune des parties a pour hauteur la hauteur d'une case et pour largeur 3 pixels (nous avons grossi les traits jaunes pour mettre en évidence les deux parties), nous expliquerons plus tard pourquoi nous avons choisi 3 pixels.

Pour chaque partie nous faisons la moyenne de tous les pixels, c'est à dire qu'on additionne tous les composants de chaque pixel, donc nous additionnons séparément tous les rouges, tous les bleus et tous les verts et nous divisons ces trois composants par le nombre total de pixels. Ensuite nous obtenons une nouvelle couleur qui correspondra à une idée de la couleur du début de cette case (ou de la fin), le début de la case (ou la fin) étant la partie jaune de gauche (ou droite) de la *figure 16 : bonbon avec bandes verticales*.

Voici en-dessous une idée du code en java pour expliquer comment calculer la couleur moyenne d'une bande jaune (droite ou gauche).

```
Couleur moyennePixel(void){
	int r = 0, g = 0, b = 0; // r : rouge (red), g : vert (green), b : bleu
(blue)
	Pixel [][] matriceDecor = new Pixel [hauteurCase][largeurCase]; // matrice
comprenant tous les pixels de la case
	for(int h = 0; h < hauteurCase; h++){
		for(int l = 0; l < 3; l++){
			r += matriceDecor[h][l].getRed();
			g += matriceDecor[h][l].getGreen();
			b += matriceDecor[h][l].getBlue();
		}
	}

	r = r / (hauteurCase * 3); // hauteurCase * 3 : taille de la matriceDecor
	g = g / (hauteurCase * 3);
	b = b / (hauteurCase * 3);

	return new Couleur(r, g, b);
}
```

figure 17 : calcul de la moyenne

Nous prenons 3 pixels de largeur pour avoir une meilleure approximation de la couleur du début de la case (ou la fin), sachant qu' aucun bonbon prend toute la largeur de la case en enlevant les 6 pixels (les 3 pixels à droite et les 3 pixels à gauche), donc la couleur que nous obtenons est uniquement la couleur du décor de la case et non quelques pixels du bonbon.

Une fois que nous avons récupéré la couleur moyenne de la partie de droite et la couleur moyenne de la partie de gauche, nous les comparons entre elles. Nous les comparons car dans une case de la grille de jeu, au milieu il y a un bonbon, ce bonbon est entouré de décor (le décor de la case), y compris à droite et à gauche du bonbon, donc si la couleur moyenne de la partie de droite est plus ou moins égale à la couleur moyenne de la partie de gauche cette case est un bonbon ("plus ou moins égale à" et non "égale à" car il faut prendre en compte un certain taux d'erreur lié à la différence de couleurs du décor de la case, car ce décor et semi-transparent et donc les couleurs peuvent varier en fonction du décor de la grille). En revanche la plupart des décors ne possèdent pas cette particularité d'avoir la partie de droite et la partie de gauche plus ou moins identique.

Nous pouvons dire que certaines cases sont du décor, mais nous ne pouvons pas affirmer avec certitude que toutes les autres cases soient des bonbons, et aussi nous ne pouvons pas affirmer avec certitude que les cases où nous avons dit que c'était du décor soient réellement du décor.

2ère méthode :

figure 18 : bonbon avec bandes horizontales

Cette méthode est similaire à la première méthode, sauf que au lieu de prendre les pixels de manière verticale nous les prenons de manière horizontale. Nous récupérons le haut de la case, nous calculons la couleur moyenne de la bande jaune en haut et le bas de la case, où nous calculons aussi la couleur moyenne de la bande jaune en bas, afin de les comparer entre eux, si les deux couleurs moyennes sont identiques alors cette case possède un bonbon sinon c'est du décor.
Comme pour la première méthode aucun bonbon n'est aussi haut que la hauteur de la case moins les 6 pixels (les 3 pixels en haut et les 3 pixels en bas), sur la *figure 16 : bonbon avec bandes verticales* aussi, nous avons épaissi la bande jaune pour mieux distinguer les deux parties.

3-2-2) Détection des bonbons identiques

Après avoir déterminer les cases comprenant uniquement du décor et les cases possédant des bonbons, nous allons déterminer tous les bonbons identiques de la grille, quand on dit bonbon identique, on parle des bonbons qui possèdent des caractères identiques, comme la couleur.
La figure en-dessous représente un bonbon, le trait jaune représente la ligne horizontale du milieu de la case d'une grille de jeu.

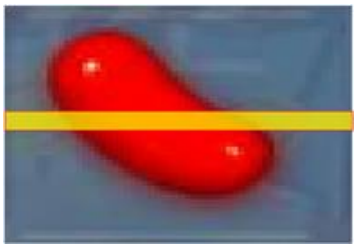

figure 19 : bonbon avec bande du milieu horizontale

Le principe est semblable aux méthodes de détection de décor, car nous récupérons la couleur moyenne du début de chaque case (voir la partie 3-*2-1 - Détection du décor)*. La couleur moyenne du début de chaque case correspond à la moyenne de tous les pixels de la bande jaune à gauche de la *figure 2 : bonbon avec bandes verticales*. La couleur moyenne du début de chaque case, cette couleur correspond à la couleur du décor de la case

Une fois la couleur du décor de la case obtenue, nous comparons cette couleur avec tous les pixels de la ligne du milieu (*cf ligne jaune de la figure 19 : bonbon avec bande du milieu horizontale*). Lorsque que nous comparons la couleur du décor de la case avec la couleur des pixels, s'ils sont égaux cela signifie que le pixel correspond au décor de la case, sinon le pixel appartient au bonbon. Par conséquence en faisant la moyenne de toutes les couleurs qui appartiennent au bonbon, on peut avoir une approximation de la couleur du bonbon.

Le code ci-dessous correspond au calcul de la couleur moyenne du bonbon par case de la grille de jeu.

```
void bonbonIdentique(void){
	// index : nombre de pixels appartenant au bonbon
	int redTotal = 0, greenTotal = 0, bleuTotal = 0, index = 0;

	// matrice comprenant tous les pixels de la case
	Pixel [][] matriceDecor = new Pixel [hauteurCase][largueurCase];

	// cf fonction moyennePixel() de la partie B - a)
	Couleur debutCase = moyennePixel();

	for(int l = 0; l < largueurCase; l++){
	//matriceDecor[hauteurCase/2][l] correspond à la ligne du milieu de la case
		int r = matriceDecor[hauteurCase/2][l].getRed();
		int g = matriceDecor[hauteurCase/2][l].getGreen();
```

```
            int b = matriceDecor[hauteurCase/2][l].getBlue();

            // si le pixel appartient au bonbon
            if(debutCase.getRed() == r && debutCase.getGreen() == g &&
debutCase.getBlue() == b){
                  redTotal += r;
                  greenTotal += g;
                  bleuTotal += b;
                  index++;
            }
      }

      return new Couleur(redTotal/index, greenTotal/index, bleuTotal/index);
}
```

figure 20 : code pour récupérer "la couleur" d'un bonbon

Pour chaque case de la grille on a une couleur moyenne ou du décor, si on a du décor, on ne s'en occupe pas, sinon on compare toutes les couleurs moyennes entre elles pour regarder s'il y a des couleurs identiques. Si les couleurs sont identiques alors on leur met un numéro unique. Comme ça nous pouvons déterminer quelle case est identique à quelles autres cases.

Comme pour la détection de décor, nous avons aussi une deuxième méthode afin de vérifier et confirmer les ressemblances. Cette deuxième méthode est identique à la précédente mais au lieu de prendre la bande jaune de gauche de la *figure 16 : bonbon avec bandes verticales*, nous prenons la bande jaune en haut de la *figure 18: bonbon avec bandes horizontales* pour déterminer la couleur du décor de la case et cette couleur nous la comparons avec la ligne horizontale jaune du milieu de la case de la *figure 21: bonbon avec bande du milieu verticale*.

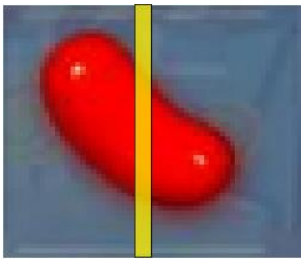

figure 21 : bonbon avec bande du milieu verticale

3-2-3) Détection de couleur

Dans cette partie nous allons traiter l'identification des couleurs, c'est à dire définir la couleur des bonbons pour chaque case de la grille de jeu.

Pour pouvoir identifier la couleur des bonbons nous avons pris des valeurs de référence par exemple on dit que **orange** correspond à *250* de rouge, *100* de vert et *0* de bleu (*cf Annexe 1 : tableau de référence des couleurs*).

Toutes ces valeurs ont été trouvées en testant, malheureusement s'il y a un reflet de soleil ou un autre événement parasite qui vient abîmer la couleur de la photo, la détection de couleur sera fausse, en conséquence il faudra prendre les photos dans des conditions optimales, c'est à dire sans reflet, avec une luminosité ni trop sombre, ni trop forte, ...

Comme pour la méthode de détection des bonbons identiques, nous regardons chaque pixel de la ligne du milieu horizontalement (*cf figure 19 : bonbon avec bande du milieu horizontale*), et nous comparons chaque pixel avec toutes les couleurs de référence.

S'il existe un certains nombres de pixels dont la valeur correspond à une même couleur de référence alors on peut dire que ce bonbon possède une certaine couleur, par exemple si nous trouvons au final qu'il y a 10 pixels dont la valeur correspond aux références de couleur rouge, alors ce bonbon est rouge (*cf Annexe 2 : code de la reconnaissance de couleurs*).

Comme pour la détection de décors et la détection de bonbons identiques, nous utilisons une deuxième méthode identique à la première, nous nous occupons de tous les pixels de la ligne verticale du milieu (*cf figure 21 : bonbon avec bande du milieu verticale*). nous comparons chaque pixel avec toutes les valeurs de référence, et comme ci-dessus nous pouvons déterminer la couleur du bonbon.

Comme nous disons dans la partie *III-2-1 Détection du décor*, sans le vouloir avec cette algorithme, on fait une autre vérification (en vérité 2 autres modifications en fonction des deux méthodes définies précédemment) du décor, car s' il y a aucun pixel qui ne correspond pas à une valeur de référence, cela signifie que c'est du décor.

Pour conclure de la détection du décor, de bonbons identiques, et de la couleur du bonbon, nous mettons ces trois méthodes en commun pour avoir avec un certain taux de certitude un tableau à deux dimensions dont chaque case du tableau correspond à une case de la grille de jeu, ce tableau est rempli par la couleur du bonbon (rouge, vert, bleu, ...).
Grâce à la détection du décor et à la vérification du décor dans cette partie, on peut connaître les emplacements des décors.

De même que si parmi les deux tableaux de couleurs que l'on obtient dans cette partie les cases ne sont pas identiques entre elles, on peut vérifier grâce à la détection de bonbons identiques, on peut déterminer quelle est la couleur.

3.3- Intelligence Artificielle

3.3.1) IA détection de possibilités

Le premier objectif de l'IA a été de pouvoir détecter les coups sur la zone de jeu, c'est-à-dire au moins deux bonbons de même couleur avec un autre bonbon de la même couleur ce situant à une case d'eux. Plusieurs exemples sont présentés sur l'image si dessous :

figure 22 : Jeu avec coups indiqués

Dans un premier temps il nous a fallu trouver une solution simple pour modéliser le damier, nous nous sommes fixés un format de sortie sur la fonction de traitement de l'image en une matrice de strings de taille 9x9, ceci étant la taille standard d'un tableau de Candy Crush sans les décors. Chaque string représentait une couleur dans la première version du code, depuis les strings ont été remplacés par une classe bonbon qui sera décrite dans la deuxième partie.

L'algorithme de l'IA est le suivant :

For int x =0 ; x<9 ; x++
For int y=0 ; y<9 ; y++
Si la case en haut ou en bas ou à droite ou à gauche de la case (Damier [x][y]) analysée
S'il existe une string de la même couleur a une case d'écart de la case égale à celle analysée
On ajoute les positions (x et y) des deux cases à échanger dans une liste

figure 23 : algorithme IA

Les coordonnées des coups sont modélisées sous la forme d'une classe *Move* composée de 5 int, 4 pour les coordonnées des deux bonbons à déplacer pour réaliser un coup et un autre appelé *score* pour comparer les *Move* entre eux selon des critères fixés par le développeur, ce sera utile pour l'IA prédictive. Le *Move* dispose d'une méthode compare(Move m) pour comparer les *Move* entre eux en fonction des positions à échanger. Cette méthode permet d'éviter des redondances dans la liste et de traiter deux fois le même coup dans l'IA prédictive.

Cette phase donne le résultat suivant :

(x_1y_1) 74 70 63 63 63 63 52 54 70 43
(x_2y_2) 73 60 62 64 53 73 53 53 71 53

figure 24 :résultat IA

3.3.2) Damier de simulation de Candy Crush

Pour pouvoir développer une IA plus évoluée il nous fallait d'abord un moyen de la tester qui soit plus performant qu'un simple tableau de String. Une bonne partie du temps dont nous disposions pour le projet est passé dans l'écriture de deux classes, Damier et Bonbon. Ces deux classes ont pour but de reproduire le jeu Candy Crush Saga, et excepté pour le score de la partie cet objectif a été atteint.

La classe *bonbon* est composée de deux strings, *couleur* et *spécial* qui représentent la couleur et le type de *bonbon*. Par défaut le champ spécial est réglé à *« vide »* ce qui signifie que le bonbon n'est pas un bonbon spécial.

La classe *bonbon* est composé de nombreuses méthodes :

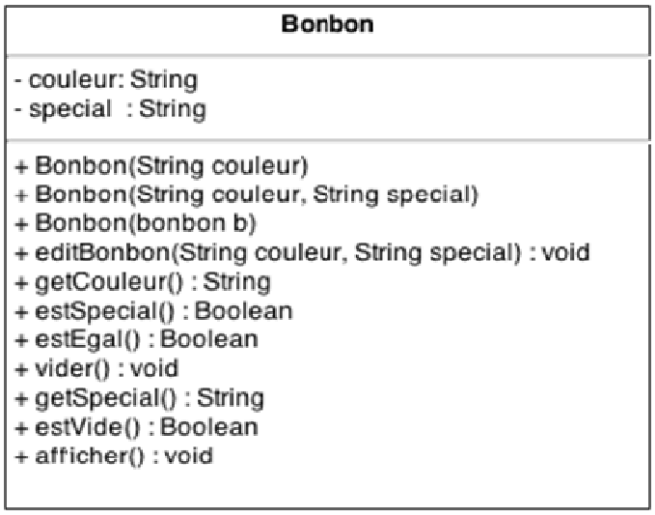

figure 25 : Classe Bonbon

- 3 constructeurs, un prenant un string, un autre deux et un constructeur par copie

- 1 méthode *editBonbon (String couleur, String special)* permettant de modifier les champs *couleur* et *spécial*

- 2 getters, un pour chaque champ

- Une méthode estEgal(Bonbon b)

- Une méthode vider() qui règle tous les champs à *« vide »*

- Une méthode estVide qui renvoie *true* si les deux champs sont égaux à *« vide »*

- Une méthode afficher() qui affiche la couleur sur la sortie standard et son champ spécial s'il est différent de *« vide »*

La méthode *estEgal()* est la méthode la plus importante de cette classe.

```
public Boolean estEgal(Bonbon b){
	if(b.getCouleur()=="vide" || couleur=="vide")
	return false;
	if(b.getCouleur()==couleur ||
b.getCouleur()=="Multicolor"||couleur=="Multicolor")
	return true;
	else
	return false;
}
```

figure 25 : Code estEgal

Cette fonction teste d'abord si l'un des deux bonbons est vide auquel cas il renvoie directement faux. Ensuite il compare les couleurs des deux bonbons entre elles, si les couleurs

sont égales la fonction renvoie *true*. La fonction revoie également *true* si un des deux bonbons est Multicolor, ceci s'explique par le fait que si un bonbon est échangé avec un bonbon Multicolor tous les bonbons de la couleur échangée sont détruits. On peut donc considérer ce bonbon comme égal aux autres pour un coup, de ce fait il est inclus dans cette fonction. Dans tous les autres cas la fonction retourne *false*.

Passons ensuite à la classe *Damier*, cette classe est essentielle pour la simulation du jeu Candy Crush Saga et le test de l'IA.

Elle est composée de deux attributs, *baseCouleurs* un tableau de 5 strings, chaque string est initialisée dans le constructeur par une des 5 couleurs de bonbon disponible dans Candy Crush saga, qui permet de générer de manière aléatoire le contenu du second attribut. Et *damier* une matrice de 9x9 bonbons, chaque case initialisée par le code suivant.

```
private Bonbon randCase(){
	Random rn = new Random();
	return baseCouleurs[rn.nextInt(5)];
}
for(int i=0;i<9;++i)
	for(int c=0;c<9;++c)
	damier[i][c]=new Bonbon(randCase().getCouleur());
```

figure 26 : Code randCase et application

randCase() retourne une string contenue dans *baseCouleurs* choisie de manière aléatoire et chaque case est construite à partir de cette case. Suite à cette génération on appelle une méthode *cleanDamier()*, qui elle-même appelle *appliRegles(int x, int y)* sur chaque case. L'algorithme d'appliRegles est le suivant.

- For (deux cases à côté de la case analysée)
	- Si la case est de la même couleur que le bonbon analysé
	On incrémente un compteur
- En fonction de la valeur du compteur on vide un certain nombre de bonbons et on génère ou pas le bonbon spécial correspondant à la configuration vidée.

figure 27 : Algorithme appliRegles

Après cette méthode la méthode *fillEmpty()* est appelée :

```
void fillEmpty(){
	for(int x=8;x>=0;x--){
		for(int y=0;y<9;y++){
			while (damier[x][y].estVide()){
				lowersweets(x,y);
//evite les boucles infinies sur des cases qui sont normalement vides
				if(x<=3)
					break;
			}
		}
	}
	for(int x=0;x<9;x++){
		for(int y=0;y<9;y++){
			if(damier[x][y].estVide()){
				damier[x][y]=new Bonbon(randCase().getCouleur());
			}
		}
	}
}
```

figure 28 : Code fillEmpty

Qui elle-même appelle *lowersweets(int x, int y)*

```
Bonbon lowersweets(int x,int y){
	Bonbon b,b1;
	if(x==0){
	b=new Bonbon(damier[x][y]);
	damier[x][y].vider();
	return b;
	}
	b=lowersweets(x-1,y);
	b1=new Bonbon(damier[x][y]);
	damier[x][y].editBonbon(b.getCouleur(), b.getSpecial());
	return b1;
}
```

figure 29 : Code lowersweets

Lowersweets est une fonction récursive qui permet de gérer la chute des bonbons effectuée par Candy Crush Saga pour remplacer les bonbons qui ont été vidés. La fonction initialise deux *bonbons* b et b1. Ensuite elle s'appelle récursivement jusqu'à atteindre le haut du tableau. Une fois en haut la fonction copie le bonbon du haut dans une variable temporaire puis la renvoie vers la fonction appelante. Une fois de retour dans celle-ci on copie le bonbon actuel dans une autre variable temporaire puis on le remplace dans le damier par le bonbon qui se situait au-dessus. On renvoie la variable temporaire récemment crée puis on rappelle la fonction jusqu'à atteindre les coordonnées x,y.

Cette fonction permet de descendre les bonbons une ligne par une c'est pour ça qu'elle est appelée dans un boucle dans la fonction *fillEmpty*. Cette fonction appelle une autre boucle qui parcourt également le damier à la recherche des cases vides, maintenant bien positionnées, à remplir avec *randCase()*. L'application de toutes ces fonctions permet d'avoir un damier conforme aux règles de Candy Crush Saga.

La dernière méthode présente est l'IA détection de possibilité expliquée dans la première partie.

Voici un damier généré par la classe damier et la liste de coups associés.

jaune	jaune	jaune	violet	violet	vert	bleu	violet	violet
bleu	vert	rouge	vert	jaune	rouge	violet	bleu	vert
vert	vert	vert	vert	jaune	vert	vert	bleu	bleu
jaune	bleu	bleu	rouge	vert	bleu	vert	rouge	vert
vert	violet	rouge	vert	bleu	rouge	rouge	jaune	vert
Bleu	violet	violet	rouge	violet	bleu	vert	vert	jaune
jaune	violet	bleu	bleu	vert	vert	rouge	violet	vert
bleu	rouge	jaune	bleu	jaune	rouge	vert	vert	jaune
violet	rouge	violet	vert	rouge	violet	rouge	vert	bleu

85 76 67 67 78 66 75 66 66 58 43 45 58 58 55 33 47 28 35 24 24 46 07 33 06
75 75 57 38 68 65 65 56 76 57 42 44 48 68 65 34 37 18 25 34 23 56 06 43 16

figure 30 : Résultat Classe Damier

3.3.3) IA prédictive

Enfin vient la classe IA qui va indiquer le meilleur coup à jouer via l'attribut *bestMove* de type *Move*. Cette classe est également composée d'un *Damier* et une *List<Move>* qui contient tous les coups détectés par *l'IA détection de possibilité*.

Le code qui estime le meilleur mouvement à jouer, nommé *estimateMove* à un algorithme très similaire à celui d'*appliRegles* :

```
estimateMove(Damier d, Move m)
	Copie du damier dans une variable temporaire
	//toute l'analyse serra effectuée dans cette variable
	Temp.swap(m) //permet d'échanger la position des deux cases indiquées par m dans le damier
	Pour les deux cases de chaque côté de celle indiquée par m.x1 et m.y1
	Si la case est de la même couleur que le bonbon analysé
	On incrémente un compteur
	En fonction de la valeur du compteur on donne une certaine valeur score au coup
```

figure 32 : Algorithme IA

Ce code est appliqué dans une fonction nommée *estimateMoves()* dont voici le code :

```
	void estimateMoves(Bonbon[][] damier){
		for(Move m : possibleMoves){
			Bonbon[][] dam = damier;
			estimateMove(dam,m);
		}
		setBestMove();
		bestMove.showMove();
	}
```

figure 33 : Code estimateMoves

```
	void setBestMove(){
		for(Move m : possibleMoves){
```

```
                    if(m.isEmpty())
                          bestMove=new Move(m);
                    else if(m.score>bestMove.score)
                          bestMove=new Move(m);
              }
        }
```

figure 34 : Code setBestMove

Dans *estimateMove* on donne à chaque coup une valeur on fonction du bonbon spécial que le coup permet d'obtenir. Un multicolor est évalué à 4 points, un emballé a 3, un rayé 2 et un normal 1. Ce système de point basique permet d'établir une hiérarchie entre les différents bonbons, elle peut être réutilisée dans un algorithme récursif que nous expliquerons dans la partie 4.4. Le coup indiqué serra celui ayant le plus haut score.

Le résultat de cette partie, appliqué au damier précédent est le suivant :

```
And the best move is ...
58
68
```

4) Reprise du projet

3.4.1) Détection de la zone de jeu sans déplacer des points

Le projet initial demande à ce que l'application soit absolument autonome et cela passe par une détection automatique de la zone de jeu. Une piste à suivre serait de créer un tableau de pixels qui correspondrait à une zone de jeu (une forme plus ou moins rectangulaire composée de carrés gris), puis, à l'aide d'un algorithme similaire à celui présenté pour distinguer les différents bonbons, comparer la photo de l'écran d'ordinateur avec ce tableau pour chercher une correspondance. Il faudra cependant faire attention car selon la photo, la zone de jeu n'occupera pas forcément le même nombre de pixels, et certains niveaux de Candy Crush Saga ont une forme particulière qu'il faudra prendre en compte.

Ce paramètre est d'ailleurs à prendre en compte également dans la partie codée par Pierre : l'analyse de l'image ne sera pas la même pour une image de 9x9 cases ou pour une image de 5x8 cases. Il faudrait donc que la détection automatique prenne systématiquement une image rectangulaire et qu'elle soit capable de compter le nombre de cases maximal en longueur et en largeur afin d'optimiser l'analyse des couleurs.

3.4.2) Perspectives programmation Android:

En ce qui concerne le développement de la partie Android du projet, la mise en place d'une solution permettant une correction efficace de la distorsion de l'image est une priorité. Dans cette optique, il est tout à fait envisageable de compléter ou corriger le code existant afin de tirer parti de la méthode *setPolyToPoly()* proposée dans la bibliothèque standard Android. Néanmoins, si cette solution s'avère trop complexe, il sera peut-être alors préférable d'envisager un algorithme plus simple n'utilisant pas de transformation matricielle mais créant une nouvelle image "manuellement" en copiant certains pixels de l'image source tout en prenant bien en compte la déformation initiale de l'image.

Une fois que ce module aie été mis en place, il serait alors envisageable d'automatiser le processus, par exemple, après avoir pris un première fois la photo et défini la zone de jeu à la main, pour peu que le téléphone reste immobile, on peut alors prendre des photos en continu et leur appliquer le même traitement que la première fois.

Enfin, en parallèle de l'automatisation du processus, il serait possible de commencer à mettre en place une communication avec l'ordinateur sur lequel tourne le jeu. Dans un premier temps une communication simple utilisant le protocole TCP/IP, mais si le temps et le téléphone le permettent, une communication bluetooth simulant une souris pourrait être un plus car cette solution s'affranchirait de la présence d'un client sur l'ordinateur de jeu pour retranscrire les commandes de souris.

3.4.3) Détection des bonbons spéciaux

Il existe de nombreuses sortes de bonbons qui n'ont pas été analysés pour différentes raisons. Par exemple, certains bonbons sont assez rares et apparaissent tard dans le jeu comme la réglisse, le chocolat, ...

figure 35: reglisse

figure 36 : Chocolat

Nous avons décidé de nous centrer sur les bonbons normaux car notre priorité et de faire fonctionner notre programme dans les premiers niveaux du jeu. La détection de ses bonbons pourrait être une partie du projet de l'année prochaine car en général, quand il y a ce type de bonbon, l'objectif de la mission est de les détruire, et non de faire un maximum de points.

Il y a aussi des bonbons spéciaux qui apparaissent en fonction des coups précédents, par exemple :

figure 37 :Bonbon multicolore

figure 38: bonbon rayé vertical

figure 39 :Bonbon rayé horizontal

figure 40: bonbon emballé

Nous avons réussi à identifier les bonbons rayés horizontalement et verticalement (*cf. figures 38 et 39*), en revanche, nous n'avons pas détecté les bonbons multicolores et les bonbons paquets (*cf. figure 37 et 40*). nous avons préféré identifier les bonbons rayés avant les autres car se sont les bonbons spéciaux qui apparaissent le plus souvent. Comme pour les bonbons chocolat ou réglisse nous avons préféré ne pas identifier les bonbons multicolores et les bonbons paquets car d'une part ces bonbons spéciaux sont assez rares et d'autre part nous préférons nous concentrer sur des objectifs plus importants tels que la couleur des bonbons par exemple.

Pour l'identification, nous avons utilisé la même technique que la détection de couleur (*cf III - 2 - 3 Détection de couleur*), c'est à dire que nous nous occupons que des pixels de la ligne du milieu, nous les comparons avec les valeurs de référence (*cf Annexe 1 : tableau de référence des couleurs*), et en plus nous les comparons avec la couleur blanche, car un bonbon rayé possède deux couleurs le blanc et une autre, nous pouvons observer les bonbons rayés horizontalement (*cf figure 39*) en comparant les valeurs de référence avec les pixels de la ligne verticale du milieu de la case. De même pour les bonbons rayés verticalement, nous les observons en comparant les valeurs de référence avec les pixels de la ligne horizontale du milieu de la case.

Si l'année prochaine le projet est continué, un des objectifs sera de détecter les bonbons multicolores et les bonbons paquets pour que l'intelligence artificielle puisse gérer ces bonbons afin d'optimiser les coups à jouer.

3.4.4 Amélioration de l'IA

Nous avons longuement réfléchi à des améliorations pour l'intelligence artificielle et les suivantes nous paraissent réalisables :

- Appliquer une fonction récursive permettant de calculer les coups induits par un coup appliqué, cette fonction utiliserais *lowersweets()* pour simuler la chute mais sans génération de bonbon aléatoire, on ne simule pas le hasard !

- Une version de l'IA pourrait détecter les formations pouvant détecter les formations pouvant résulter en un bonbon emballé ou multicolore, par exemple deux paires de bonbons de la même couleur sur la même ligne ou la même colonne séparée par un bonbon de couleur différente. Sur cette configuration l'IA se concentrait sur une partie du damier de façon à obtenir un bonbon au bon endroit pour compléter le coup.

Exemple de l'explication ci-dessus :

figure 41 : exemple formation non complète

- La gestion des objectifs est capitale à réaliser, l'algorithme de gestion de la valeur des coups, *estimateMove()*, serra à modifier il faudra qu'il prenne en compte non seulement le type de bonbon mais en plus sa position et ce de manière dynamique.

IV - Bilan technique

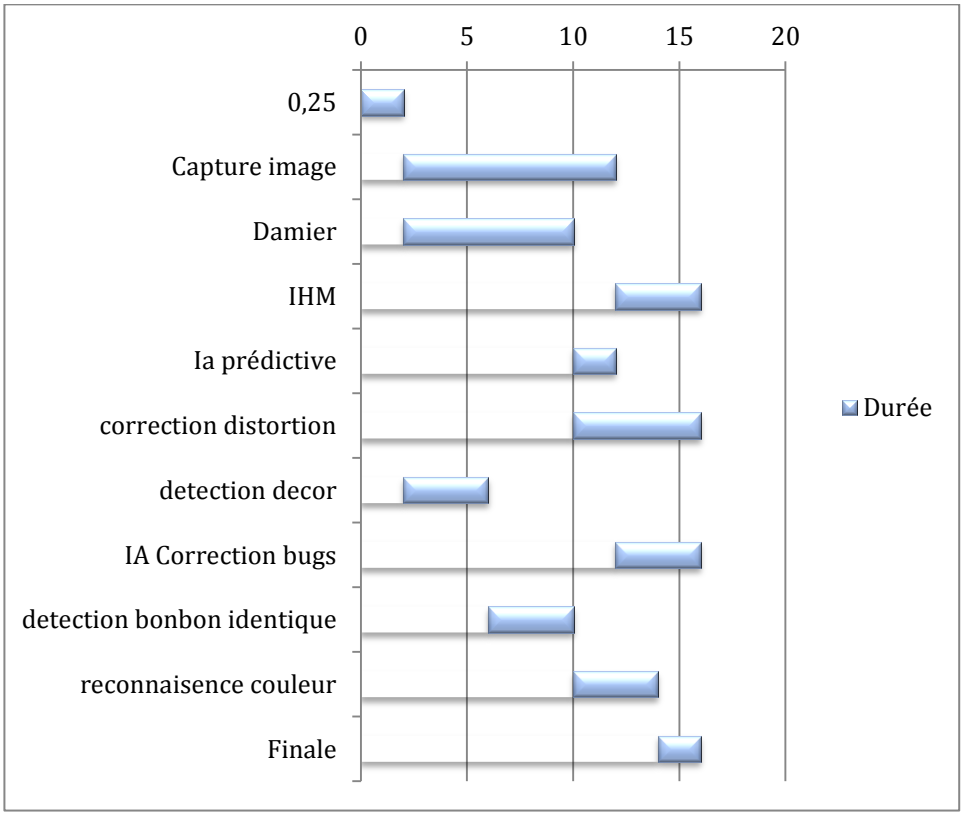

figure 40 : Diagramme gant réel

4.1) Bilan

Maintenant que nous sommes à la fin de la période, nous pouvons considérer que de nombreux objectifs ont été réalisés, malgré le fait que notre objectif final reste cependant inachevé. En effet, nous nous étions mis comme objectif final d'avoir une application Android capable de prendre des photos en continu, de les traiter de manière autonome, et enfin d'envoyer un mouvement de souris à la station sur laquelle est le jeu Candy Crush permettant de réussir les différents niveaux de manière automatique.

À l'heure d'aujourd'hui, nous avons réalisé une application Android capable de prendre une photo nativement, de transformer l'image d'une grille de jeu en un tableau de données

exploitables et de calculer un coup intéressant à jouer, le tout par le biais d'une IHM fonctionnelle.

Néanmoins, certaines parties essentielles ne marchent pas et impactent notre avancée, notamment en ce qui concerne le redressement de l'image avant son traitement, empêchant le fonctionnement *a minima* que nous aurions souhaité de l'application. De plus, nous n'avons pas encore pu envisager une automatisation du processus de capture, de redressement et de traitement des photos, étant pour l'heure limités à un traitement d'une seule image à la fois. En outre, malgré le fait que nous réussissions à calculer les coordonnées du coup à jouer, aucune communication n'est établie entre le smartphone et l'ordinateur, il faut encore que ce soit l'utilisateur qui fasse les mouvements de souris associés au coup affiché.

Les principales difficultés que nous ayons rencontrées se situent principalement autour de la maîtrise du langage Java pour lequel nous n'avions aucun cours à l'IUT, ainsi que la complexité de l'API Android, notamment dans le cycle de vie des applications ou l'utilisation Callbacks auxquels nous étions étrangers.

4.2) Perspectives

Dans l'hypothèse où le projet serait repris l'année prochaine, les tâches les plus prioritaires seraient tout d'abord celles en lien avec le fonctionnement basique de l'application : corriger le redressement de l'image ou remplacer la procédure actuelle par une autre, et établir une communication entre l'ordinateur et le smartphone permettant à ce dernier de jouer le coup calculé. Il serait alors intéressant de mettre en place une automatisation de la procédure globale, permettant ainsi à l'application de requérir moins d'attention de l'utilisateur pour fonctionner.

5 - Conclusion

5.1) Notre apprentissage au long du projet

Ce projet nous a apporté de multiples enseignements tant sur le plan technique que sur l'expérience d'un travail en équipe sur une longue durée.

Tout d'abord, cela nous a permis d'apprendre le java de manière autodidacte donc avant qu'il ne soit enseigné avec les cours de l'IUT. Nous avons décidé de coder en java avec l'IDE *Eclipse*, par conséquent nous avons aussi appris à nous servir de cet IDE.

Sachant que notre code est sur Smartphone (Samsung Galaxy S2/S4) nous nous sommes initiés au développement sur Android, en l'occurrence, faire l'IHM sous Android. Avec le travail d'Alix, nous avons apprivoisé la caméra de Smartphone sous Android qui fait appel à des aspects de Java que nous n'avons pas vu en cours. Jonathan et Alix ont géré l'écran tactile pour pouvoir capturer des points, ainsi que gérer la rotation de l'écran et le changement de vue. Ce type de développement nécessite quelques recherches pour savoir si certaines bibliothèques sont compatibles sous Android, ...

D'autres contraintes sont apparues à cause d'Android par exemple le passage de données entre les Views limité à 5Mo, ..., ce qui nous a appris diverses spécificités d'Android.

Ce projet est le projet le plus long et le plus complexe de nos deux années de DUT. Pour le gérer nous avons appris à travailler en équipe afin de répartir les tâches de façon équilibrées et en fonction des compétences de chacun. Nous avons appris aussi à gérer le temps de travail comme nous avons étudié au cours de nos années à l'IUT en gestion de projet. Nous nous sommes fixés des objectifs afin de prévoir et planifier tout au long de notre projet pour le finir à temps.

Avec le travail de Jo-Hakim, nous avons appris à concevoir une intelligence artificielle, pour cela nous avons codé un "candy crush saga" sur un terminal afin de pouvoir tester rapidement,

nous avons lu et réfléchi sur des algorithmes afin qu'ils soient à la fois performants et efficaces.

L'image prise par le Smartphone n'est pas une image traitable, car elle n'a pas une forme rectangulaire, mais plutôt de trapèze. Alix a donc appliqué une morphose sur l'image. De même que Jonathan a appris à récupérer des points sur une image afin d'obtenir juste la grille de jeu sans avoir le reste de l'écran.

Grâce au travail de Pierre, nous avons travaillé sur différents algorithmes pour pouvoir distinguer tous les bonbons, nous avons appris comment on traite une image afin de distinguer tous les éléments à l'intérieur de l'image.

5.2) Notre ressenti par rapport au projet

Nous avons eu quelques problèmes lors de ce projet, des problèmes de gestion du temps, gestion du travail, des problèmes de codage, ...

Apprendre un langage de manière autodidacte est plus difficile que de l'apprendre lors de nos cours à l'IUT, car avec des cours on peut apprendre de manière structurée et échelonnée accompagnés de travaux pratiques de temps à autre, en revanche, le fait d'apprendre par soi-même nécessite de regarder des tutoriaux qui ne sont pas souvent bien expliqués, et pas adaptés à notre niveau.

On a tous eu des problèmes de gestion de temps, on n’a pas su s'organiser pour dire exactement à quelle date on aura fini nos objectifs car on ne se rendait pas compte de l'ampleur du travail qu'il fallait effectuer. Si c'était à refaire, nous consacrerions plus de temps sur l'élaboration des objectifs à faire, et des différentes techniques utilisées tout au long du projet afin de respecter les délais des objectifs.

Nous nous sommes rendu compte (Alix et Pierre) que la bibliothèque *java.awt* n'existait pas sous Android, sachant que cette bibliothèque est utile pour la détection des couleurs

notamment avec la classe *BufferedImage* qui existe uniquement avec dans la bibliothèque *java.awt* dont nous nous servions pour traiter les images. Ce problème a été résolu avec la classe *Bitmap* qui contient l'image, et la classe *Color* qui me sert à récupérer les couleurs qui appartiennent à la bibliothèque *android.graphics*.

Certaines personnes ont été surprises par l'investissement pour le projet, nous pensions que les 4h par semaine prévues par l'emploi du temps étaient suffisantes, mais au final nous nous sommes rendu compte qu'il fallait consacrer du temps personnel pour d'une part faire avancer le projet et d'autre part résoudre des problèmes dont les solutions ne sont pas toujours évidentes et faciles à trouver sur internet.

Jonathan a dû changer d'activité sur Android, il a dû coder un programme qui devait passer de l'activité "prise de la photo" à "récupération de l'image, l'afficher et récupérer des points", ce changement d'état nécessite des connaissances que nous n'avons pas apprises, il a dû assimiler ce point de connaissance technique, comme le java, de manière autodidacte et regarder sur des tutoriaux comment résoudre ce problème.

La prise en main de la caméra avec Android a été un obstacle à la compréhension pendant un certain temps du fait de sa complexité, de codes d'exemple erronés, mais ça a permis à Alix de réutiliser ces connaissances en C# pour la réalité virtuelle.

Pour conclure, nous avons tous appris des connaissances diverses et variées telles que la gestion du temps (avec la prévision des objectifs). On a aussi appris le langage Java de manière autodidacte. Chacun a appris des connaissances différentes, comme la conception d'une intelligence artificielle, la création d'une application sur Android, la rectification de l'image, le traitement d'image, le changement d'activité...

On sait désormais qu'un projet se travaille et se prépare, notamment avec le travail en équipe, avec des outils appropriés.

Nous espérons que la lecture de ce rapport vous a éclairé sur le travail effectué et qu'il puisse être utile pour la continuation du projet.

Résumé en Anglais

This project was entrusted to us by Mr Kauffmann and Laffont. The aim was to code an application capable, from a smartphone, to take a picture of a Candy Crush Saga game, treat it and play the best move possible directly on the computer where the game is taking place.

Unfortunately not all of those objectives were reached.

The image is taken fine, Alix did a great job at correcting it and it is usable. Pierre's code translate the image in an almost perfect way, the color of the candies is detected as well as the stripped ones and the surroundings of the gaming field. However some special candies plus those used for special objectives are not treated. The IA, made by Jo-Hakim, works fine and is able to prioritize special candies over normal ones but it can't treat the special objectives the games offer. The whole application is easy to use thanks to the graphical interface coded by Jonathan.

The main untreated feature is the communication between Smartphone and PC in order to give the commands to play the game properly, instead of that our application gives the positions of the best move to play. The last feature couldn't be handled due to a lack of time.

The project allowed us to discover new technologies, the galaxy S4 provided to us by our tutor gave us the opportunity to code on Android devices, learn the basics of IA development and learn about image correction.

So as you can read there is still room for improvement yet we are quite satisfied, even though a bit frustrated, with the result we can propose you.

Bibliographie

http://chessprogramming.wikispaces.com/

http://damien-guichard.developpez.com/tutoriels/algo/introduction-techniques-IA/

http://sudokuvision.blogspot.fr

http://developer.android.com

http://stackoverflow.com

http://www.developpez.net/forums/

http://www.commentcamarche.net/forum/

http://fr.openclassrooms.com/

http://java.developpez.com/

LEXIQUE

Candy Crush Saga[(1)] : *Candy Crush Saga* est un jeu vidéo développé par **King** (en), disponible à l'origine comme une application Facebook et adapté pour les systèmes d'exploitations Android et iOS. Il s'agit d'une variation du jeu sur navigateur web *Candy Crush*. Il reprend les principes jeu concurrent *Bejeweled* sorti en 2001 sous le nom initial de *Diamond Mine* et en partie le design de *CandySwype*, sorti, en 2010.

Samsung Galaxy S4[(2)] : Le *Samsung Galaxy S4* est un smartphone haut de gamme de **Samsung**. Il succède au *Galaxy SIII*. Il fut disponible en Europe depuis le *26 avril 2013*. Il dispose de nouvelle fonctionnalités logicielles, d'un appareil phot en face arrière de 13 mégapixels et d'un écran de 5 pouces *Full HD* (1920 x 1080 px). Il est équipé d'un processeur à 4 cœurs *Qualcomm S-600* à 1.9GHz.

IDE[(3)] : *Integrated Development Environment (*Environnement de développement intégré)

Intent[(4)] : « *intention* », l'application par ce biais peut faire part au système qu'elle a l'intention de faire une action spéciale qui peut être de nature très diverse : prendre une photo, ouvrir une image, ...

Callback[(5)] : En informatique, une **fonction de rappel** (**callback** en anglais) ou fonction de post-traitement est une fonction qui est passée en argument à une autre fonction. Cette dernière peut alors faire usage de cette fonction de rappel comme de n'importe quelle autre fonction, alors qu'elle ne la connaît pas par avance.

Morphose[(6)] : Transformation, formation, mise en forme d'un objet (par exemple une image)

API[(7)] : interface de programmation (*Application Programming Interface*) est un ensemble normalisé de classes et de méthodes qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels.

Annexe

	les 3 composants		
	rouge	vert	bleu
Bleu	0	127	255
Rouge	255	0	0
Orange	255	100	0
Jaune	255	200	0
Vert	50	150	0
Violet	200	30	255
Blanc	255	255	255

Annexe 1 : tableau de référence des couleurs

```
void identificationCouleur(void){
   // Parcourt par hauteur (case)
   for(int h = 0; h < hauteur ; h++) {
      // Parcourt par Largeur (case)
      for(int l = 0; l < largeur ; l++) {
         int bleu = 0, orange = 0, jaune = 0;
         int vert = 0, violet = 0, rouge = 0, blanc = 0;
         // Parcourt par largeur (Pixel dans une case)
         for(int ll = (l * largueurCase); ll < (l + 1) * largueurCase; ll++) {
            // Récupération des pixels du milieu de la case
            Pixel p = new Pixel(ll, ((hauteurCase/2) + (h * hauteurCase)), img);

            if(p.estBleu()){
               bleu++;
            } else if (p.estOrange()){
               orange++;
            } else if (p.estJaune()){
               jaune++;
            } else if (p.estVert()){
               vert++;
            } else if(p.estViolet()){
               violet++;
            } else if(p.estRouge()){
               rouge++;
            } else if(p.estBlanc()){
               blanc++;
            }
         }
```

```
          // je met supérieur à 5 car aucune couleur ne ressemble au bleu
          if(bleu > 5){
              matriceCouleur[h][l] = "bleu";
          }



          // 15 pour bien différencier du jaune
          if(orange > 15){
              matriceCouleur[h][l] = "orange";
          }

          // 10 pour bien différencier du orange
          if(jaune > 10){
              matriceCouleur[h][l] = "jaune";
          }

          // je met supérieur à 5 car aucune couleur ne ressemble au vert
          if(vert > 5){
              matriceCouleur[h][l] = "vert";
          }

          // je met supérieur à 5 car aucune couleur ne ressemble au violet
          if(violet > 5){
              matriceCouleur[h][l] = "violet";
          }

          // je met supérieur à 5 car aucune couleur ne ressemble au rouge
          if(rouge > 5){
              matriceCouleur[h][l] = "rouge";
          }
      } // fin for Largeur
   } // fin for Hauteur
}
```

Annexe 2 : code de la reconnaissance de couleurs

```
      void estimateMove(Bonbon[][] damier,Move m){
            bord = new Damier(damier);
            bord.swap(m);
            int xp=1,xm=1,yp=1,ym=1;

            for(int i =1 ; i<3; i++){
                  if(m.x1<7){

      if(bord.damier[m.x1][m.y1].estEgal(bord.damier[(m.x1+i)][m.y1])){
                              xp++;
                        }
                        else
                              break;
                  }
            }
            for(int i =1 ; i<3; i++){
                  if(m.x1>2){
                        if(bord.damier[m.x1][m.y1].estEgal(bord.damier[(m.x1-
i)][m.y1])){
```

```
                                xm++;
                        }
                        else
                                break;
                }
        }
        for(int i =1 ; i<3; i++){
                if(m.y1<7){

    if(bord.damier[m.x1][m.y1].estEgal(bord.damier[m.x1][(m.y1+i)])){
                                yp++;
                        }
                        else
                                break;
                }
        }
        for(int i =1 ; i<3; i++){
                if(m.y1>2){

    if(bord.damier[m.x1][m.y1].estEgal(bord.damier[m.x1][(m.y1-i)])){
                                ym++;
                        }
                        else
                                break;
                }
        }

        if(ym==3){
                if(yp==3){
                        m.score=4;
                }
                else if(xp==3){
                        m.score=3;
                }
                else if(xm==3){
                        m.score=3;
                }
                else if(yp==2){
                        m.score=2;
                }
                else{
                        m.score=1;
                }
        }
        if (yp==3){
                if(ym==3){
                        m.score=4;
                }
                else if(xp==3){
                        m.score=3;
                }
                else if(xm==3){
                        m.score=3;
                }
                else if(ym==2){
                        m.score=2;
                }
                else{
                        m.score=1;
```

```
                }
            }
            if(xp==3){
                if(xm==3){
                    m.score=4;
                }
                else if(yp==3){
                    m.score=3;
                }
                else if(ym==3){
                    m.score=3;
                }
                else if(xm==2){
                    m.score=2;
                }
                else {
                    m.score=1;
                }
            }
            if(xm==3){
                if(xp==3){
                    m.score=4;
                }
                else if(yp==3){
                    m.score=3;
                }
                else if(ym==3){
                    m.score=3;
                }
                else if(xp==2){
                    m.score=2;
                }
                else {
                    m.score=1;
                }
            }
        }
```

Annexe 3 : code de estimateMove