



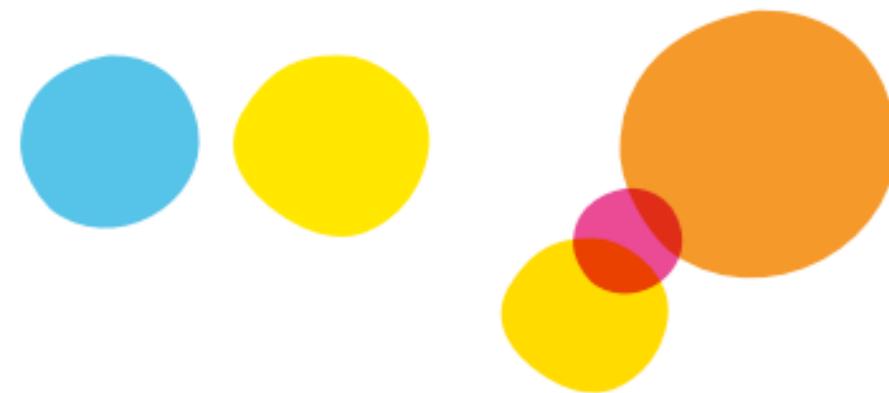
Morphing

RYAN BOUCHOU, X
X, X
X

CURSUS INGÉNIEUR
PREMIÈRE ANNÉE
GÉNIE INFORMATIQUE

20 janvier 2024

Tuteur entreprise :
ÉLISABETH X
prénom.nom@entreprise.com



Résumé

Table des matières

1	Introduction	4
1.1	Contexte	4
1.2	Objectif	4
2	Organisation au sein du groupe	5
2.1	Espace de travail : GitHub	5
2.2	Répartition des tâches	5
3	Cas des polygones simples	6
3.1	Généralités	6
3.2	Morphing naïf	6
3.2.1	Principe et notation	6
3.2.2	Cas des images matricielles	7
3.3	Méthode L-A	10
3.3.1	Principe et notation	10
3.3.2	Cas des images matricielles	11
4	Morphing de formes courbes	12
4.1	Propédeutique au morphing de courbes splines	12
4.1.1	Splines	12
4.1.2	Courbes B-Splines	12
4.2	Morphing de courbes B-splines	14
4.2.1	Calcul d'une courbe B-spline	14
5	Morphing d'images quelconques	16
5.1	Déformation des images	17
5.1.1	Préliminaires	17
5.1.2	Cas simple : un seul vecteur de contrôle	17
5.1.3	Cas général : plusieurs vecteurs de contrôle	18
5.2	Interpolation des images	20
6	Analyse et conception	21
6.1	Diagramme de cas d'utilisation	21
6.2	Diagramme de classe	21
6.3	Diagramme d'activité	21
7	Testing	26
8	Bilans personnels	28
8.1	Bilan Rubens	28
8.2	Bilan Alex	28
8.3	Bilan Paul	28
8.4	Bilan Romain	28
8.5	Bilan Ryan	29
	Annexes	32
	A Splines	32

B Tests unitaires	32
Références	39

1 Introduction

1.1 Contexte

Le morphing est une technique d'animation et d'effets visuels utilisée pour créer une transformation fluide entre deux images ou objets distincts. Ce processus permet de voir une image se métamorphoser progressivement en une autre, en donnant l'illusion que la première se transforme directement en la seconde.

Le morphing est couramment utilisé dans diverses applications, notamment :

Cinéma et Télévision Pour créer des effets spéciaux impressionnants où un personnage ou un objet change de forme de manière spectaculaire ;

Vidéo musicale Les clips musicaux utilisent souvent le morphing pour des transitions visuelles créatives et artistiques ;

Publicité Pour attirer l'attention et illustrer des transformations de produits ou de services ;

Logiciels de photographie et d'animation Offrent des outils de morphing pour les artistes et les animateurs afin de créer des effets visuels captivants.

Un exemple célèbre de morphing dans le cinéma est celui utilisé dans le film "Terminator 2 : Judgment Day" (1991), où le personnage du T-1000 change de forme de manière impressionnante. Cette technique a depuis été perfectionnée et est devenue un outil standard dans les effets visuels.

1.2 Objectif

L'objectif de ce projet est de réaliser une application de morphing permettant de créer une animation d'une image *source* à une image de *destination*. Pour ce faire, trois exercices différents sont proposés par le sujet :

1. **Formes simples** : deux polygones de même couleur ;
2. **Formes courbées** : deux formes quelconques avec des courbures ;
3. **Visages** : deux visages.

2 Organisation au sein du groupe

Pour ce projet, nous avons été réparti en groupe de cinq étudiants (GI). Il faut donc s'organiser pour mener à bien le projet pour le 31 mai 2024, date à laquelle la soutenance est prévue.

2.1 Espace de travail : GitHub

Ce projet a été l'occasion de découvrir GitHub, un logiciel de partage et de stockage de fichiers optimisé et pensé pour le développement informatique. Il permet de s'échanger des fichiers à distance, voir l'historique des modifications et de gérer les conflits en cas de fichiers modifiés par deux personnes différentes en même temps. C'est pourquoi nous avons choisi cet outil pour nous aider dans la réalisation de notre application de morphing.

2.2 Répartition des tâches

Une fois l'environnement de travail pris en main, il nous a fallu nous répartir des tâches à réaliser.

D'abord, il nous fallait nous mettre d'accord sur la méthode à suivre. Pour cela, tout le monde a fait des recherches de son côté durant la première semaine en notant les sources consultées qui seront notées dans la bibliographie de ce rapport. Après plusieurs recherches, nous avons opté pour l'implémentation de la méthode des lignes plutôt que la triangulation de Delaunay.

Globalement, deux groupes se sont formés. Romain et Rubens se sont occupés de la partie IHM de l'application ainsi que de la mise en place des contrôleurs. Quant à Ryan et Paul, ils sont focalisés sur la partie back-end du Java avec l'implémentation des algorithmes et des fonctions mathématiques nécessaires. Alexandre, a faisait la liaison entre les deux pôles. Néanmoins, quand l'un d'entre nous a une question, il est possible de s'échanger certaines tâches pour s'entraider. Notre organisation est plutôt flexible.

3 Cas des polygones simples

En premier lieu, nous allons nous intéresser au cas des polygones simples. Ce faisant, ceci sera l'occasion pour nous de faire notre premier pas dans l'*informatique graphique*, en traitant des cas élémentaires du problème de la morphose.

3.1 Généralités

Conventions Pour toute la suite, et sauf mention contraire, on se place dans le \mathbb{R} -evn \mathbb{R}^2 muni de la norme euclidienne. On notera pour tout vecteur $x \in \mathbb{R}^2$, $X \in \mathcal{M}_{2,1}(\mathbb{R})$ la matrice colonne associée à x dans la base canonique de \mathbb{R}^2 .

Définition 1.1. Soit n un entier naturel non nul. On appelle *polygone simple* de \mathbb{R}^2 tout n -uplet $P = (p_1, \dots, p_n)$ de \mathbb{R}^2 tels que :

1. Les segments ne se croisent pas, c'est-à-dire que pour chaque paire de segments $[p_i, p_{i+1}]$ et $[p_j, p_{j+1}]$ (où p_{n+i} est p_i), les segments ne partagent pas de points autres que les sommets.
2. Chaque sommet p_i est partagé par exactement deux segments.

On notera p_{i+} le segment $[p_i, p_{i+1}]$ et p_{i-} le segment $[p_i, p_{i-1}]$. Enfin, on notera \mathbb{P} l'ensemble des polygones simples de \mathbb{R}^2 .

Définition 1.2. Soit $P \in \mathbb{P}$ un polygone simple de \mathbb{R}^2 .

1. On appelle *ordre* de P le nombre n de composantes de P .
2. On appelle *arête* de P tout segment p_{i+} ou p_{i-} pour $i \in \{1, \dots, n\}$.
3. On appelle *sommet* de P tout vecteur p_i pour $i \in \{1, \dots, n\}$.

Subséquemment, pour n un entier naturel non nul, on note \mathbb{P}_n l'ensemble des polygones simples de \mathbb{R}^2 d'ordre n .

Définition 1.3. Soit $P \in \mathbb{P}$ un polygone simple de \mathbb{R}^2 . On appelle *intérieur* de P , noté $\overset{\circ}{P}$, l'ensemble des points $x \in \mathbb{R}^2$ tels que x est à gauche de chaque arête de P .

Situation À ce stade, l'enjeu de cette première partie est de déterminer un algorithme permettant la morphose d'un polygone simple P vers un autre polygone simple Q . Pour ce faire, il nous faut nous intéresser aux conditions d'une telle transformations, ainsi qu'à sa réalisation.

3.2 Un premier algorithme de morphing

3.2.1 Principe et notation

Principe L'idée de cet algorithme est de déformer progressivement le polygone P en un autre polygone Q . Pour ce faire, une première approche consiste à déformer chaque sommet p_i de P en un sommet q_i de Q par une interpolation linéaire. Ainsi, on obtient une suite de polygones $(P_k)_{0 \leq k \leq N}$ où N est le nombre d'images intermédiaires voulues et tels que $P_0 = P$, $P_N = Q$.

Pour que l'algorithme soit correct, il est nécessaire que les polygones P et Q soient de même ordre.

Notation Soit $n, N > 0$ et $(P_k)_{0 \leq k \leq N}$ une suite de polygones de $(\mathbb{P}_n)^{\mathbb{N}}$. On notera $p_1^{(k)}, \dots, p_n^{(k)}$ les sommets de P_k .

Ce faisant, pour le calcul des images intermédiaires on donne l'algorithme suivant :

Données : $P, Q \in \mathbb{P}_n$ deux polygones, $N > 0$ le nombre de frames

Résultat : Une suite de polygones (P_k)

$P^* \leftarrow (P_1, \dots, P_N)$

pour $k \leftarrow 0$ à N faire

$t \leftarrow \frac{k}{N}$
 $P_k = (p_1^{(k)}, \dots, p_n^{(k)})$
pour $i \leftarrow 1$ à n faire
 | $p_i^{(k)} \leftarrow (1 - t) * p_i + t * q_i$
fin

fin

retourner P^*

Algorithme 0 : générationFramesNaif

3.2.2 Cas des images matricielles

Principe Dans le cadre de l'exercice, la donnée du problème est constituée de deux images matricielles P et Q de taille $L \times l$ représentant respectivement des polygones simples de couleur même couleur. Naturellement, plusieurs méthodes algorithmiques peuvent être utilisées pour encoder l'information géométrique portée par une image. Entre autre, des algorithme de *détection de contours* ou de *segmentation* peuvent être utilisés. Toutefois, et par soucis de simplicité, nous allons considérer que l'information géométrique est directement encodée par l'utilisateur lors de la saisie des *points de contrôle*. De plus, pour chaque pixel de coordonnées (i, j) , on a :

$$\begin{cases} \mathbb{I}_P(i, j) = 1 & \text{si } (i, j) \in \overset{\circ}{P} \\ \mathbb{I}_P(i, j) = 0 & \text{sinon} \end{cases}$$

Pour que l'algorithme soit correct, il est nécessaire que les images P et Q soient de même taille. De plus, si les points de contrôle saisis par l'utilisateur ne correspondent pas aux sommets des polygones, les résultats de l'algorithme peuvent être inattendus.

Propriété 1.1. Soit $u, v \in \mathbb{R}^2$ deux vecteurs. Alors u est à gauche de v si et seulement si $\det(u, v) > 0$.

DÉMONSTRATION : Admis.

o.ε.δ.

Propriété 1.2. Soit $p = [u, v], (u, v) \in \mathbb{R}^2$ un segment et $x \in \mathbb{R}^2$ un vecteur. Alors x est à gauche de p si et seulement si $\det(p, [u, x]) > 0$.

DÉMONSTRATION : Soit $u, v \in \mathbb{R}^2$ deux vecteurs. On note θ l'angle orienté entre u et v . Alors,

$$\begin{aligned}
 u \text{ est à gauche de } v &\iff \det(u, v) > 0 & (1) \\
 &\iff \|u\| \cdot \|v\| \sin(\theta) > 0 & (2) \\
 &\iff \sin(\theta) > 0 & (3) \\
 &\iff \theta \in]0, \pi[& (4) \\
 &\iff u \text{ est dans le demi-espace à gauche de } v & (5) \\
 && \text{o.}\varepsilon.\delta.
 \end{aligned}$$

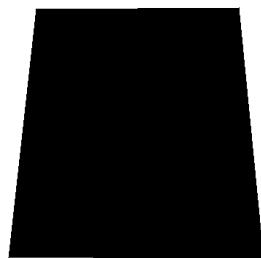
Ce faisant, pour le calcul des images intermédiaires, et la determination de l'intérieur du polygone, de on donne la suivante :

Données : Une liste de points tab représentant le polygone, un point P
Résultat : Un booléen indiquant si le point P est à l'intérieur du polygone
 $nbp \leftarrow$ taille de tab **pour** $i \leftarrow 0$ à $nbp - 1$ **faire**
 | $A \leftarrow tab[i]$ $B \leftarrow tab[(i + 1) \bmod nbp]$
 | $D \leftarrow (B.x - A.x, B.y - A.y)$ $T \leftarrow (P.x - A.x, P.y - A.y)$
 | $d \leftarrow \det(D, T)$
 | **si** $d < 0$ **alors**
 | | **retourner** Faux
 | **fin**
fin
retourner Vrai ;
Algorithme 1 : isInside

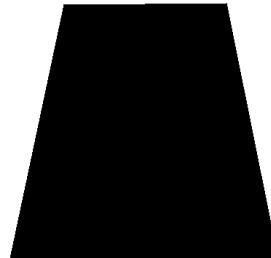
On en déduit l'algorithme naif pour le mophing de formes unies simples :

Données : Deux images matricielles P et Q de taille $L \times l$, un entier N
Résultat : Une suite d'images matricielles P^*
 $nbp \leftarrow$ taille de tab
 $P^* \leftarrow$ générationFramesNaif(P, Q)
pour $i \leftarrow 0$ à $nbp - 1$ **faire**
 | **pour** $x \leftarrow 0$ à $L - 1$ **faire**
 | | **pour** $y \leftarrow 0$ à $l - 1$ **faire**
 | | | **if**(isInside($P_i^*, (x, y)$)) $P_i^*[x][y] \leftarrow color$
 | | **fin**
 | **fin**
fin
retourner P^*
Algorithme 2 : morphingNaif

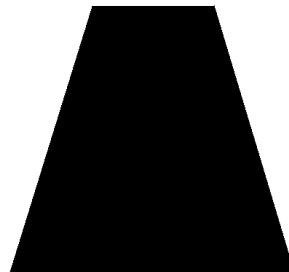
Résultats Considérons une exécution de l'algorithme sur deux instances du problème.



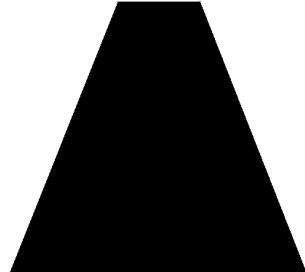
(a) Image initiale



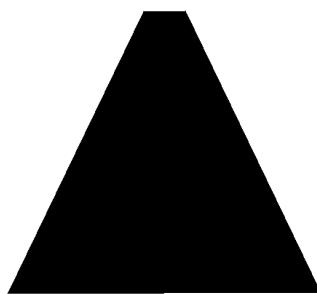
(b) Image intermédiaire 1



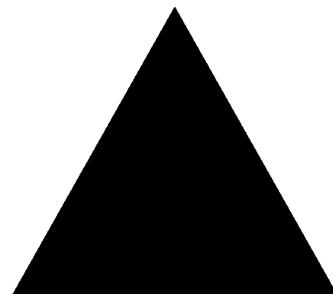
(c) Image intermédiaire 2



(d) Image intermédiaire 3



(e) Image intermédiaire 4



(f) Image finale

FIGURE 2 – Résultats de l'algorithme de morphing

Dans le cas de polygones simples, relativement semblables dans leur géométrie, l'algorithme de morphing naïf donne des résultats satisfaisants.

En pratique Toutefois, il apparaît qu'une telle approche ne préserve pas les propriétés géométriques du polygone. En effet, les images intermédiaires peuvent contenir des segments qui se croisent, ou des sommets qui ne sont pas partagés par exactement deux segments. Dans un tel cas, la transformation ne paraît pas naturelle.



FIGURE 3 – Interpolation linéaire, observez les auto-intersections. [2]

Ainsi, il est nécessaire de trouver une approche plus rigoureuse pour le morphing de polygones simples.

3.3 Morphing par interpolation longeur-angle

3.3.1 Principe et notation

Principe L'idée de cette méthode, issue des travaux de Sederberg [5], est de déformer progressivement le polygone P en un autre polygone Q , en interpolant linéairement la mesure des angles entre chaque arêtes consécutives, et leurs normes respectives. Ce faisant, l'avantage est de préserver les propriétés géométriques souhaitées pour un rendu visuel naturel. On parle d'*invariance par mouvement rigide*. de morphing améliorée est.

Notation Soit $n, N > 0$ et $(P_k)_{0 \leq k \leq N}$ une suite de polygones de $(\mathbb{P}_n)^{\mathbb{N}}$. On pose, pour $k \in \{0, \dots, N\}$:

- $l_i^{(k)}$ la longueur de l'arête $p_{i+}^{(k)}$,
- $\theta_i^{(k)}$ l'angle entre les arêtes $p_{i-}^{(k)}$ et $p_{i+}^{(k)}$.

Subséquemment, pour une image intermédiaire $k \in \{0, \dots, N\}$, avec $t = \frac{k}{N}$ on a :

$$\begin{aligned} l_i^{(k)} &= (1-t)l_i + tq_i \\ \theta_i^{(k)} &= (1-t)\theta_i + t\theta_i \end{aligned}$$

Ce faisant, pour le calcul des images intermédiaires on donne l'algorithme suivant :

Données : $P, Q \in \mathbb{P}_n$ deux polygones, $N > 0$ le nombre de frames

Résultat : Une suite de polygones (P_k)

$P^* \leftarrow (P_1, \dots, P_N)$

$P_0 \leftarrow P, P_N \leftarrow Q$

pour $k \leftarrow 0$ à N faire

$t \leftarrow \frac{k}{N}$
 $P_k = ((l_1^{(k)}, \theta_1^{(k)}), \dots, (l_n^{(k)}, \theta_n^{(k)}))$
pour $i \leftarrow 1$ à n faire
 $l_i^{(k)} \leftarrow (1-t)l_i + tq_i$
 $\theta_i^{(k)} \leftarrow (1-t)\theta_i + t\theta_i$

fin

fin

retourner P^*

Algorithme 3 : générationFramesLA

3.3.2 Cas des images matricielles

En l'état, et de par des contraintes de temps, nous n'avons pas implémenté ce second algorithme. On donne cependant une comparaison des résultats obtenus par *Sederberg et al* [5] avec les deux méthodes.

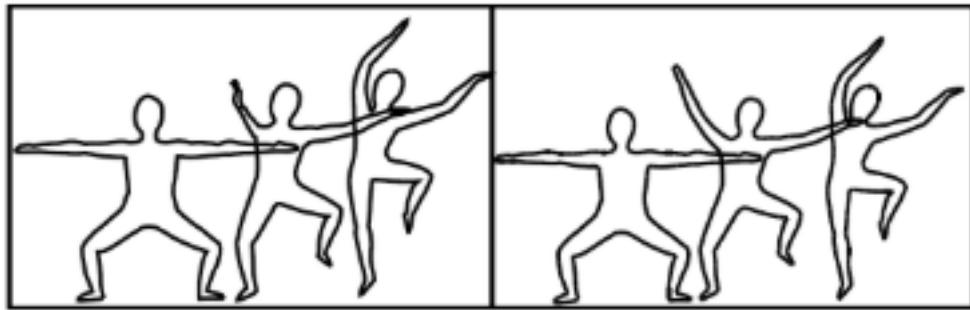


FIGURE 4 – Morphing par interpolation longueur-angle à droite, naïve à gauche. [5]

4 Morphing de formes courbes

Les polylignes à courbure nulle ne permettent pas une représentation précise de formes courbes. Pour pallier ce problème, nous allons nous appuyer sur les *courbes B-splines*, couramment utilisées en CAO pour représenter des formes courbes [4].

4.1 Propédeutique au morphing de courbes splines

4.1.1 Splines

Définition 1.4. Considérons des réels $u_0 < u_1 < \dots < u_m$, et $p \in \mathbb{N}$. On définit les *fonction B-Splines* $B_{i,p}$ par récurrence sur p et i dans \mathbb{N} comme suit :

$$\begin{cases} \text{Pour } 0 \leq i \leq m-1 \\ B_{i,0}(u) = 1 \text{ si } u \in [u_i, u_{i+1}[, \quad B_{i,0}(u) = 0 \text{ sinon} \end{cases} \quad (6)$$

$$\begin{cases} \text{Pour } 0 \leq i \leq m-1 \\ B_{i,p}(u) = \frac{u-u_i}{t_{i+p}-t_i} B_{i,p-1}(u) + \frac{u_{i+p+1}-u}{u_{i+p+1}-u_{i+1}} B_{i+1,p-1}(u) \end{cases} \quad (7)$$

Notation. Soit $j = 1, \dots, m+1-i$, on note : $\begin{cases} \omega_{i,j}(u) = \frac{u-u_i}{u_{i+j}-u_i} & \text{si } u_i < u_{i+1}, \\ \omega_{i,j} = 0 & \text{sinon.} \end{cases}$

Convention. Pour la suite, toute fonction dont le dénominateur est nul sera considérée comme nulle.

Définition 1.5. Ainsi, on a pour $i \in \{0, \dots, m-p-1\}$ et $p \in \mathbb{N}$:

$$\begin{aligned} B_{i,0}(u) &= 1 \quad \text{pour } t \in [u_i, t_{i+1}[= 0, \\ B_{i,p}(u) &= \sum_{j=1}^{m+1-i} \omega_{i,j}(u) B_{i,p-1}(u) \quad \text{pour } p > 0. \end{aligned}$$

4.1.2 Courbes B-Splines

Définition 1.6. Soit $m \in \mathbb{N}$. On appelle $(u_i)_{0 \leq i \leq m}$, *vecteur de noeuds*, et p , *degré de la B-spline*. On considère aussi des *points de contrôle* $\mathbf{P}_1, \dots, \mathbf{P}_m$ de \mathbb{R}^n . De fait, $(\mathbf{P}_i)_{0 \leq i \leq m}$ forme un *polygone de contrôle*. La *courbe B-Spline* d'ordre p associée à ces données est définie par :

$$u \longmapsto C(u) = \sum_{i=0}^m B_{i,p}(u) \mathbf{P}_i. \quad (8)$$

Propriété 1.3. Supposons que $C(u)$ soit une courbe B-spline de degré p définie comme suit :

$$C(u) = \sum_{i=0}^n B_{i,p}(u) \mathbf{P}_i$$

Soit le point de contrôle \mathbf{P}_i déplacé vers une nouvelle position $\mathbf{P}_i + \mathbf{v}$. Alors, la nouvelle courbe B-spline $D(u)$ de degré p est la suivante [4] :

$$D(u) = C(u) + N_{i,p}(u) \mathbf{v} \quad (9)$$

DÉMONSTRATION :

$$\begin{aligned}
 D(u) &= \sum_{i=0}^n B_{i,p}(u)(\mathbf{P}_i + \mathbf{v}) \\
 &= \sum_{i=0}^n B_{i,p}(u)\mathbf{P}_i + \sum_{i=0}^n B_{i,p}(u)\mathbf{v} \\
 &= C(u) + \sum_{i=0}^n B_{i,p}(u)\mathbf{v} \\
 &= C(u) + N_{i,p}(u)\mathbf{v}
 \end{aligned}
 \quad \text{o.e.d.}$$

En pratique Desormais, il nous est possible, sur la base de points de contrôle, d'un vecteur de noeud et du degré de la B-spline, de générer une courbe B-spline comme ci-dessous :

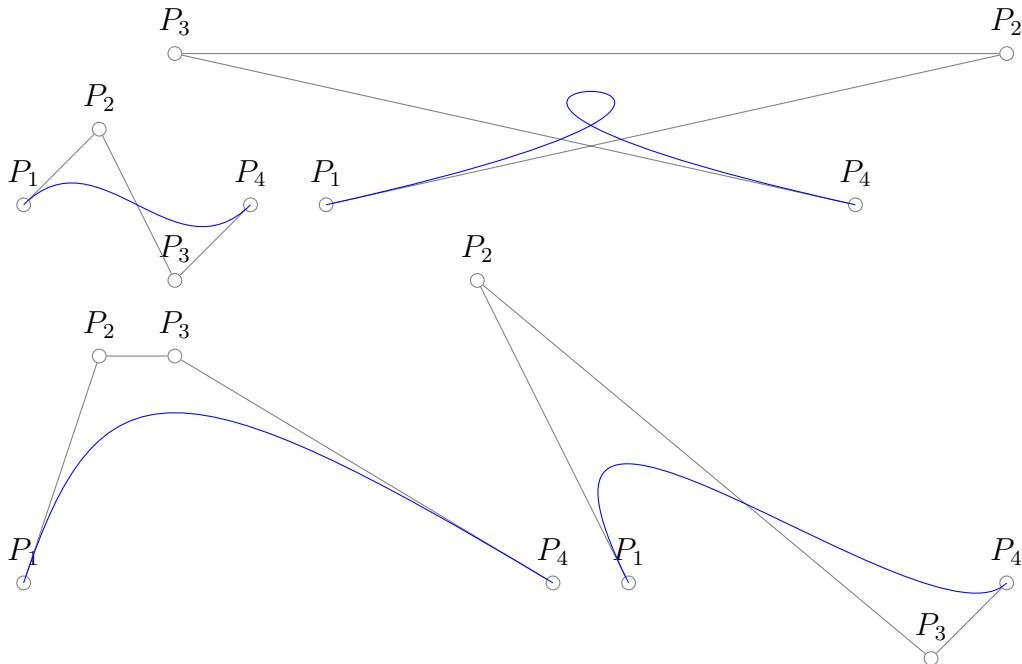


FIGURE 5 – Courbes B-splines

Lemme 2. — Soit $\mathcal{C} = (U, P, d)$ une courbe B-Spline. Alors, \mathcal{C} est une courbe fermée si et seulement si :

1. $\mathbf{P}_0 = \mathbf{P}_m$, où $m = \text{ord}(P)$,
2. U est un vecteur de noeuds uniforme.

Corollaire 3. — Soient $m, N > 0$ et $v = (\mathbf{v}_{i,k})_{(i,k) \in \llbracket 0, m \rrbracket \times \llbracket 0, N \rrbracket} \in \mathcal{M}_{m+1, N+1}(\mathbb{R}^2)$. Soit $\mathcal{C}_0 = (U, P^{(0)}, d)$ et $\mathcal{C}_k = (U, P^{(k)}, d)$, $k \in \mathbb{N}$, deux courbes B-Splines telles que :

1. $\forall k \in \llbracket 0, N \rrbracket, \forall i \in \llbracket 0, m \rrbracket, \mathbf{P}_i^{(k)} = \mathbf{P}_i^{(0)} + \mathbf{v}_{i,k}$,
2. U est un vecteur de noeuds uniforme.

4.2 Morphing de courbes B-splines

Principe Notre approche se distingue en deux phases. Premièrement, laisser à l'utilisateur le soin de réaliser l'ajout de points de contrôles, et subséquemment, l'ajustement de la courbe qui modélise la forme souhaitée. Ensuite, nous procérons au calcul des fonctions de base $B_{i,p}(u)$ ainsi qu'aux points de la courbes $C(u)$ pour l'image source. Ce faisant, en réalisant une interpolation linéaire des points de contrôles entre les polygones de contrôles de l'image source et de l'image destination, nous sommes en mesure de déterminer pour chaque image intermédiaire $0 \leq k \leq N$ la position $C^{(k)}(u)$ de la courbe en utilisant le corollaire 3.

4.2.1 Calcul d'une courbe B-spline

Calcul de la courbe Dans le sous-domaine mathématique de l'analyse numérique, l'algorithme de de Boor [6] est un algorithme polynomial et numériquement stable pour évaluer les courbes splines sous forme de B-splines. Il s'agit d'une généralisation de l'algorithme de de Casteljau pour les courbes de Bézier. Cet algorithme a été conçu par le mathématicien germano-américain Carl R. de Boor. Des variantes simplifiées et potentiellement plus rapides de l'algorithme de de Boor ont été créées, mais elles souffrent d'une stabilité comparativement plus faible.

Fonction `calculerPoint(u, controlPoints, nodeVector) :`

```


k ← trouverIntervalle(u, nodeVector);
d ← controlPoints.subList(k - deg, k + 1);

// Algorithme de De Boor
pour r ← 1 à deg faire
    pour j ← k à k - r pas -1 faire
        i ← j - k + deg;
        a ←  $\frac{u - nodeVector.get(j)}{nodeVector.get(j+deg-r+1) - nodeVector.get(j)}$ ;
        interpolated ← d.get(i - 1).nextPoint(d.get(i), a);
        d.set(i, interpolated);

    retourner d.get(deg);


```

Algorithme 4 : Calcul courbes B-splines

Remarque. On donne le pseudo-code de la fonction `trouverIntervalle` dans l'algorithme 9 en annexe.

Algorithme On donne ci-après l'algorithme principal pour le morphing de courbes B-splines, basé sur l'approche évoquée plus haut. Malheureusement, faute d'appréciation, nous n'avons pas pu tester l'algorithme. Nous en donnons une implémentation en Java complète dans les sources du projet. Par ailleurs, la javadoc est disponible pour une meilleure compréhension de l'implémentation.

Entrée : Les points de contrôle de l'image source $sourcePoints$, les points de contrôle de l'image destination $destinationPoints$, le nombre d'images intermédiaires à générer $numFrames$

Sortie : Une liste d'images intermédiaires représentant le morphing de courbes B-splines

```

// Vérifier que les listes de points de contrôle ont la même
taille
si  $sourcePoints.size() \neq destinationPoints.size()$  alors
    retourner null;

// Calculer les vecteurs de noeuds pour les courbes source et
destination
 $sourceNodeVector \leftarrow calculateNodeVector(sourcePoints.size(), deg);$ 
 $destinationNodeVector \leftarrow$ 
     $calculateNodeVector(destinationPoints.size(), deg);$ 

// Générer les images intermédiaires
 $intermediateImages \leftarrow$  une liste vide;
pour  $i \leftarrow 0$  à  $numFrames$  faire
     $t \leftarrow \frac{i}{numFrames};$ 
    // Calculer les points de contrôle intermédiaires par
    // interpolation linéaire
     $intermediatePoints \leftarrow$  une liste vide;
    pour  $j \leftarrow 0$  à  $sourcePoints.size()$  faire
         $sourcePoint \leftarrow sourcePoints.get(j);$ 
         $destinationPoint \leftarrow destinationPoints.get(j);$ 
         $x \leftarrow (1 - t) \cdot sourcePoint.getX() + t \cdot destinationPoint.getX();$ 
         $y \leftarrow (1 - t) \cdot sourcePoint.getY() + t \cdot destinationPoint.getY();$ 
         $intermediatePoints.add(newPoint(x, y));$ 

    // Calculer les points de la courbe B-spline intermédiaire
     $intermediateCurve \leftarrow$  une liste vide;
    pour  $u \leftarrow 0$  à  $1$  pas  $0.01$  faire
         $point \leftarrow calculatePoint(u, intermediatePoints, sourceNodeVector);$ 
         $intermediateCurve.add(point);$ 

    // Ajouter l'image intermédiaire à la liste
     $intermediateImage \leftarrow generateImage(intermediateCurve);$ 
     $intermediateImages.add(intermediateImage);$ 

retourner  $intermediateImages;$ 

```

Algorithme 5 : Morphing de courbes B-splines

5 Morphing d'images quelconques

Après avoir abordé la morphose dans des cas simples, notamment celui de formes unies simples et courbes, nous allons désormais nous intéresser à la morphose d'images quelconques. De ce fait, l'objectif à atteindre est de pouvoir passer d'une image à une autre de manière fluide, en conservant les caractéristiques de chacune des images, tout en **éitant l'effet juxtaposition**.



(a) Image de départ

(b) Image d'arrivée

FIGURE 6 – Paires d'images à morpher [1]

Pour ce faire, nous allons nous appuyer sur les travaux de Beier-Nelly [1], qui proposent une méthode de morphing basée sur la paramétrisation de l'image par des vecteurs de contrôle. Ainsi, chaque pixel de l'image est repéré par une association de positions relatives à ces vecteurs de contrôle. En conséquence de quoi, pour deux images différentes ayant le même nombre de vecteurs de contrôle, il est possible d'associer chaque pixel de l'une, à un pixel de l'autre (ceci ne signifie pas qu'il existe une bijection entre les pixels des deux images).

Principe Le principe de la morphose entre deux images repose sur deux étapes majeures. La première consiste à déformer les images de départ et d'arrivée, et la seconde à interpoler les images résultantes. Ce faisant, il nous est possible d'éviter l'effet de superposition des images. Naturellement, plus le nombre de vecteurs de contrôle est élevé, plus la morphose sera précise. De même, un grand nombre d'images intermédiaires permettra d'obtenir une animation plus naturelle.

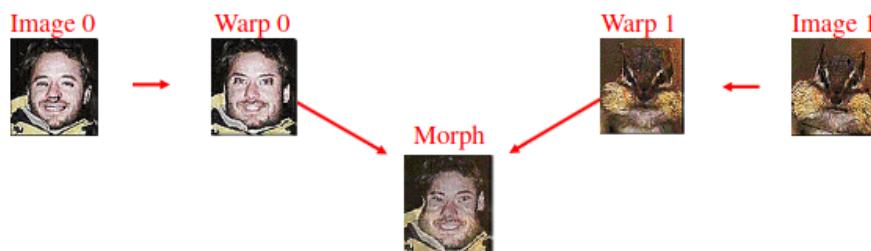


FIGURE 7 – Calcul d'un image intermédiaire [3]

Sur la figure 7, on peut observer le calcul d'une image intermédiaire *Morph*. Explicitement, on a : $Morph = (1 - \alpha) \times Wrap_0 + \alpha \times Wrap_1$, où α est un paramètre variant de 0 à 1.

5.1 Déformation des images

5.1.1 Préliminaires

Définition 1.7. Soient P une image. On note w sa largeur et h sa hauteur, ainsi que $\mathcal{D}(P) = [0, w] \times [0, h]$. On appelle **vecteur de contrôle** de P tout couple de points du plan $c = (c_1, c_2)$ tel que $c \in \mathcal{D}(P)$.

Définition 1.8. Soient P et Q deux images, ainsi que p et q deux vecteurs de contrôle de respectivement P et Q . On définit la relation \sim telle que $p \sim Q$ si et seulement si p et q sont appariés. Id est, p et q sont une seule et même entité, mais dans deux contextes (*images*) différents.

Conditions. Considérons deux images P et Q à morpher en $N > 0$ étapes. Alors, chacune possède $n + 1$ vecteurs de contrôle p_0, \dots, p_n et q_0, \dots, q_n tels que pour tout $i \in \{0, \dots, n\}$, $p_i \sim q_i$.

5.1.2 Cas simple : un seul vecteur de contrôle

Notations. Soit v un vecteur de contrôle et x un point du plan

- On note v_1 et v_2 les composantes de v ,
- On note $\hat{v} = v_2 - v_1 \in \mathbb{R}^2$,
- On note $v^\perp = \frac{\hat{v}}{\|\hat{v}\|} \times \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$,
- On note $d(X, v)$ la distance entre x et la droite passant par la première composante de v_1 et portée par v .

Propriété 1.4. Soit $c = (P, Q)$ un vecteur de contrôle, X un point du plan, alors :

$$u = \frac{c \cdot (P, X)}{\|c\|^2} \quad (10)$$

$$v = d(X, c).v^\perp \quad (11)$$

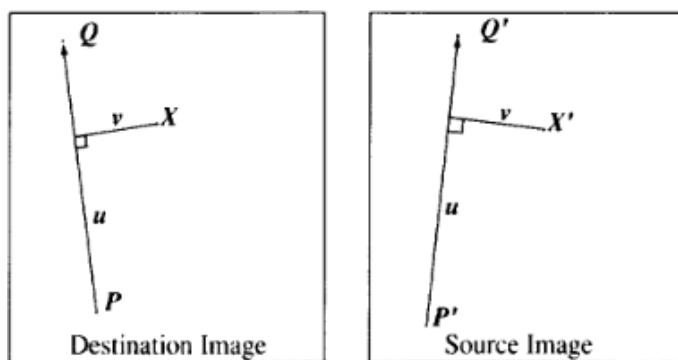


FIGURE 8 – Apparairement à un seul vecteur [1]

Algorithm. L'algorithme de déformation d'une image avec un seul vecteur de contrôle est donné par l'algorithme 6 suivant.

Données : Deux images D et S à morpher, $(P, Q) \sim (P', Q')$ des vecteurs de contrôle de D et S resp.

Résultat : Une image S déformée

pour chaque $X \in \mathcal{D}(D)$ faire

| déterminer (u, v) ;
| déduire $X' \in \mathcal{D}(S)$ pour ce couple (u, v) ;

Couleur(X) dans D \leftarrow Couleur(X') dans S;

Algorithm 6 : Déformation d'une image avec un seul vecteur de contrôle [1]

5.1.3 Cas général : plusieurs vecteurs de contrôle

Principe L'idée est de déformer l'image d'arrivée en fonction des vecteurs de contrôle de l'image de départ. Pour ce faire, - et à fortiori, bénéficier de l'ensemble des vecteurs de contrôle - on détermine le barycentre des positions X' calculées dans l'image source pour chaque vecteur de contrôle. Dans la logique des chose, le poids attribué à chaque ligne doit être le plus fort lorsque le pixel est exactement sur la ligne, et diminuer au fur et à mesure que le pixel s'en éloigne.

Poids. Le poids attribué à chaque ligne est donné par l'équation suivante :

$$\text{weight} = \left(\frac{\text{length } p}{(a + \text{dist})} \right)^b \quad (12)$$

où **length** est la longueur d'une ligne. **dist** est la distance du pixel à la ligne, et **a**, **b**, et **p** sont des constantes qui peuvent être utilisées pour modifier l'effet relatif des lignes.

En pratique, nous avons utilisé $a = 0.1$, $b = 0.5$ et $p = 1$ qui ont démontré donné des résultats très satisfaisants. Toutefois, il est essentiel de garder à l'esprit que ces valeurs peuvent être ajustées en fonction des images à morpher et de la sensibilité voulues.

Vecteurs de contrôle. Pour chaque déformation intermédiaire d'ordre k , les vecteurs de contrôles sont déterminés par interpolation linéaire des vecteurs de contrôle de départ et d'arrivée.

$$\begin{aligned} P_{i,k} &= (1 - u)P_{i,0} + uP_{i,N} \\ Q_{i,k} &= (1 - u)Q_{i,0} + uQ_{i,N} \end{aligned} \quad (13)$$

En notant N le nombre d'étapes de la morphose, et $u = \frac{k}{N}$.

Algorithm. L'algorithme de déformation d'une image avec plusieurs vecteurs de contrôle est donné par l'algorithme 7 suivant.

Données : Deux images D et S à morpher, $(P_i, Q_i) \sim (P'_i, Q'_i)$ des vecteurs de contrôle de D et S resp.

Résultat : Une image S déformée

pour chaque $X \in \mathcal{D}(D)$ faire

$DSUM \leftarrow (0, 0);$

$sommeDesPoids \leftarrow 0;$

pour chaque ligne (P_i, Q_i) faire

déterminer $(u, v);$

déduire $X'_i \in \mathcal{D}(S)$ pour ce couple $(u, v);$

déterminer $D_i = X'_i - X_i;$

déterminer la distance la plus courte de X à $(P_i, Q_i);$

déterminer le poids associé à cette distance;

$DSUM \leftarrow DSUM + D_i \times poids;$

$sommeDesPoids \leftarrow sommeDesPoids + poids;$

$X' \leftarrow X + \frac{DSUM}{sommeDesPoids};$

Couleur(X) dans $D \leftarrow$ Couleur(X') dans $S;$

Algorithme 7 : Déformation d'une image avec deux vecteurs de contrôle [1]

Exemple. La figure 9 illustre la déformation de l'image source en image destination, mais surtout la détermination de la position X' calculée comme une combinaison linéaire des vecteurs (X, X'_1) et (X, X'_2) [1].

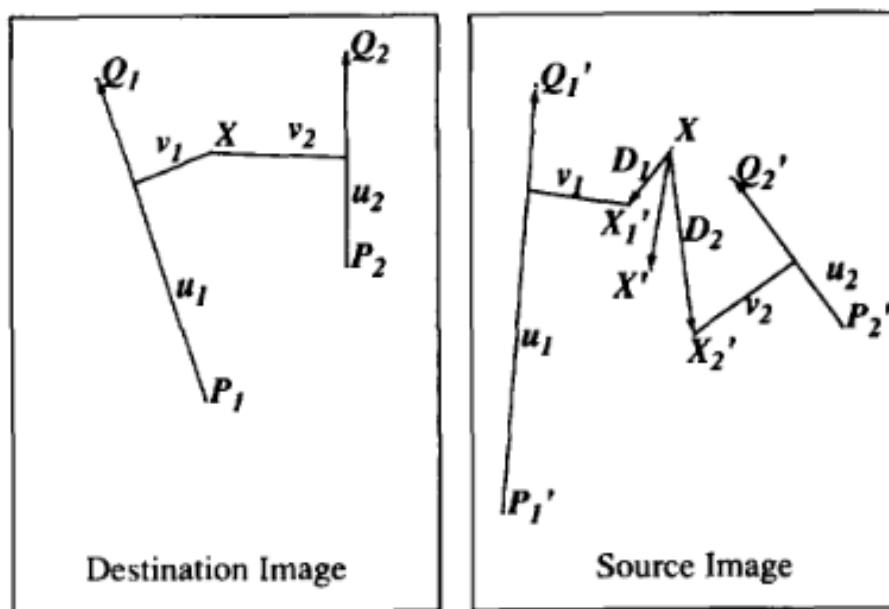


FIGURE 9 – Multiple lignes [1]

5.2 Interpolation des images

Principe L'interpolation des images consiste à déterminer une image intermédiaire entre l'image de départ et l'image d'arrivée. Pour ce faire, nous avons pré-calculé des images temporaires, étant pour chacune une déformation judicieuse de l'image de départ et d'arrivée. Desormais, nous pouvons déterminer une image intermédiaire en interpolant les images temporaires, *i.e.*, en combinant les couleurs des pixels de ces images.

Algorithm. L'algorithme d'interpolation des images est donné par l'algorithme 8 suivant.

Données : Deux images D et S à morpher, N le nombre d'étapes de la morphose

Résultat : Une image intermédiaire

pour chaque étape k de la morphose faire

```

wrapSrc ← wrapImage( $D, S, k$ );
wrapDst ← wrapImage( $S, D, k$ );
frame ← Image vide de taille convenable;
pour chaque  $X \in \mathcal{D}(frame)$  faire
    Couleurframe( $X$ ) ←
        ←  $(1 - \alpha) \times \text{Couleur}_{wrapSrc}(X) + \alpha \times \text{Couleur}_{wrapDst}(X)$   $S$ ;
    
```

Algorithm 8 : Interpolation des images [1]

6 Analyse et conception

On fait suivre sur les pages suivantes les différents diagrammes.

6.1 Diagramme de cas d'utilisation

Avant tout projet, un diagramme de cas d'utilisation permet de bien se rendre compte des besoins utilisateurs pour ne pas oublier certaines fonctionnalités. Ce diagramme est général, il convient donc aux trois exercices du sujet.

6.2 Diagramme de classe

Pour pouvoir implémenter nos classes en Java, il est plus que nécessaire de concevoir un diagramme de classe.

6.3 Diagramme d'activité

Un diagramme d'activité est aussi intéressant pour comprendre le fonctionnement de l'application.

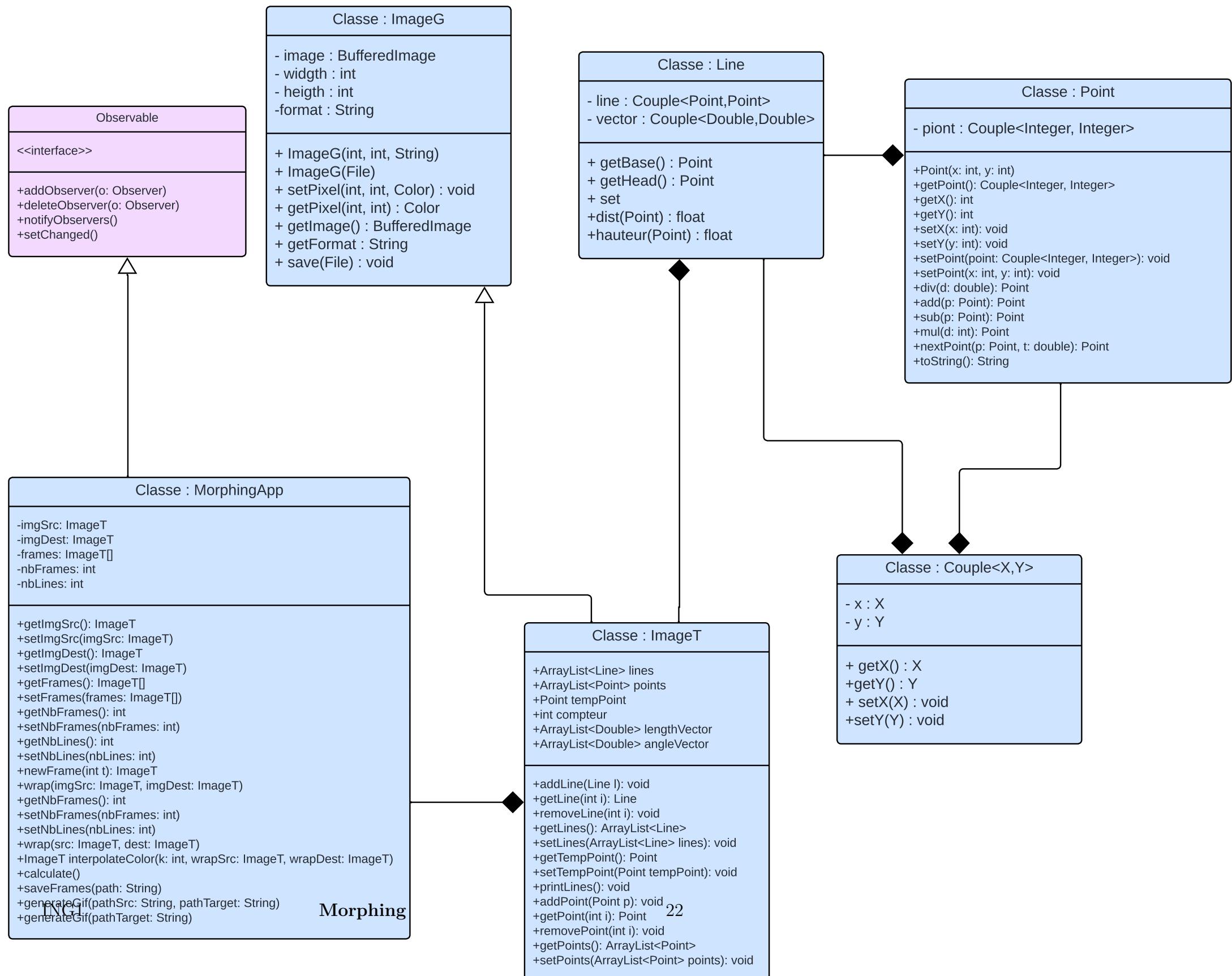


Diagramme de classe *MorphingSpline*

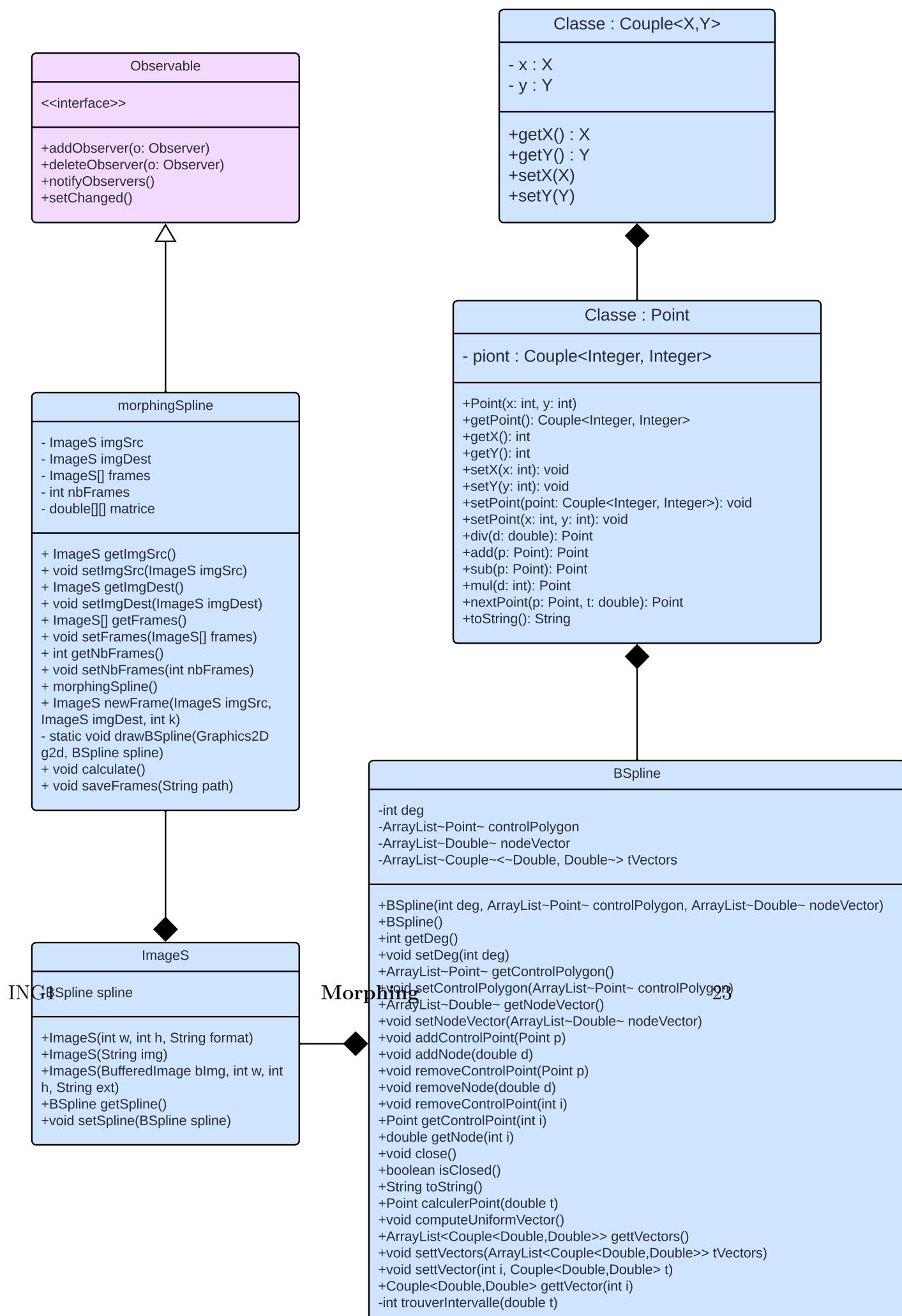


Diagramme d'activité

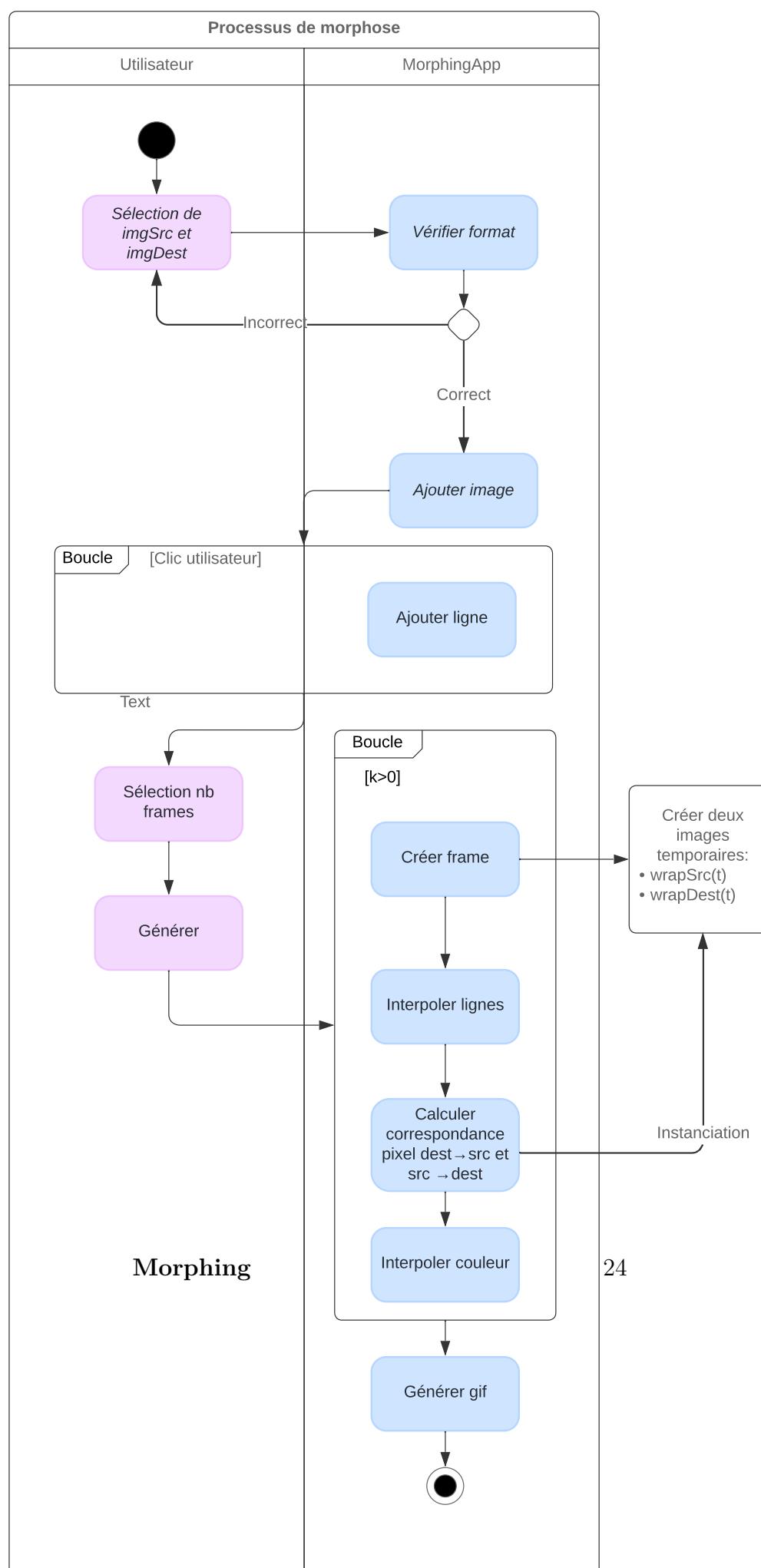
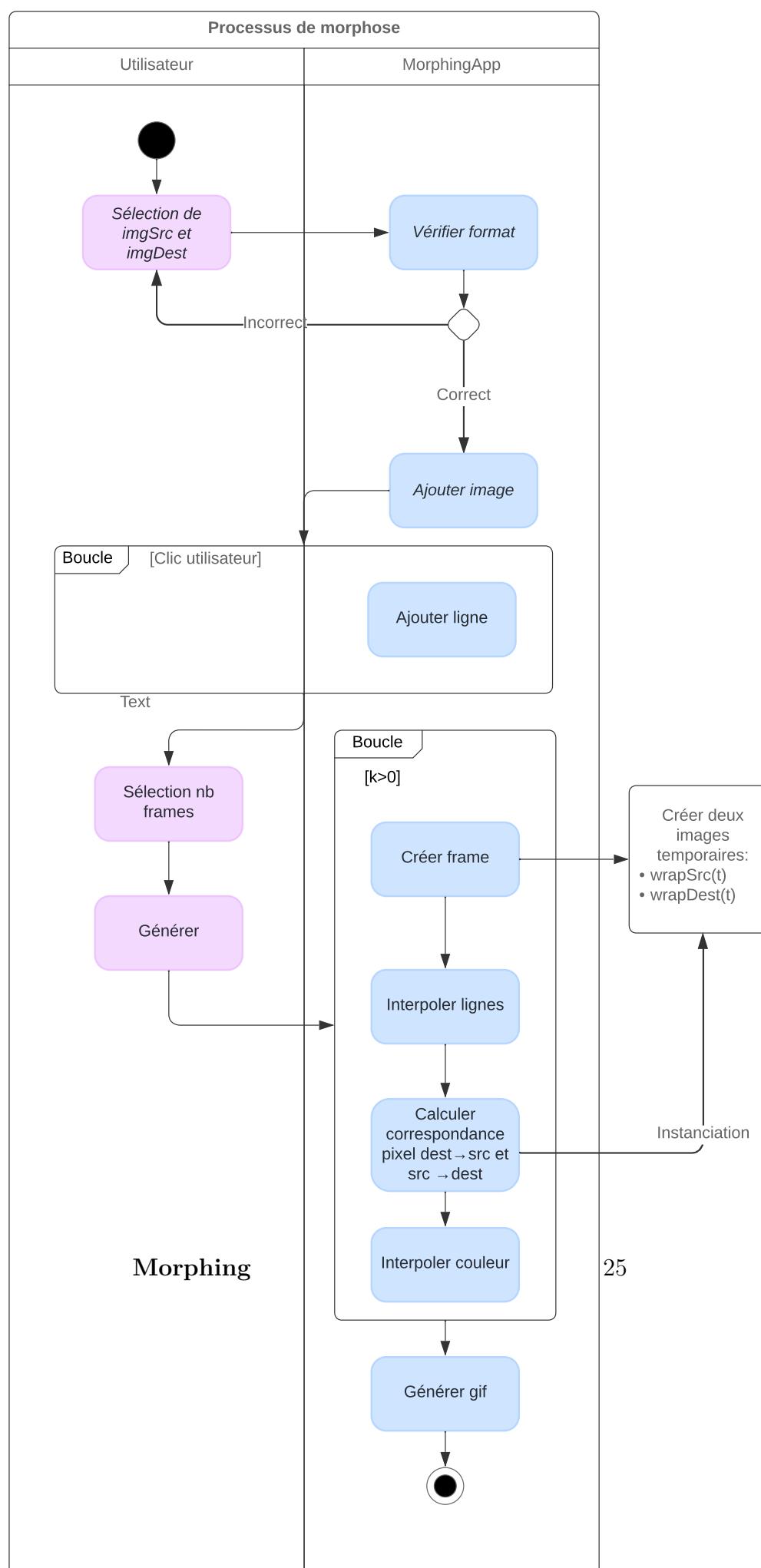


Diagramme d'activité



7 Testing

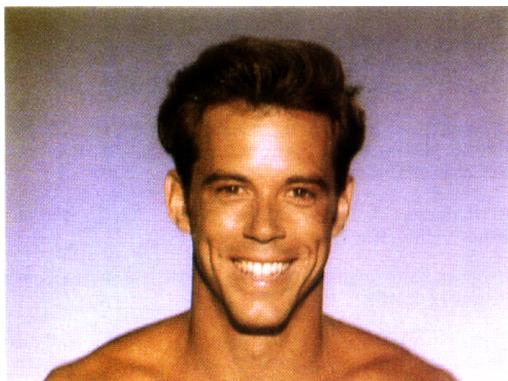
Le testing a été une partie cruciale de notre projet de morphing. Assurer la fiabilité et la précision des algorithmes que nous avons développés nécessitait une approche rigoureuse et méthodique du test. Nous avons mené une série de tests approfondis pour vérifier la robustesse de notre code et garantir que chaque composant fonctionnait comme prévu.

Approche et Méthodologie Pour tester efficacement notre projet, nous avons adopté une approche en plusieurs étapes :

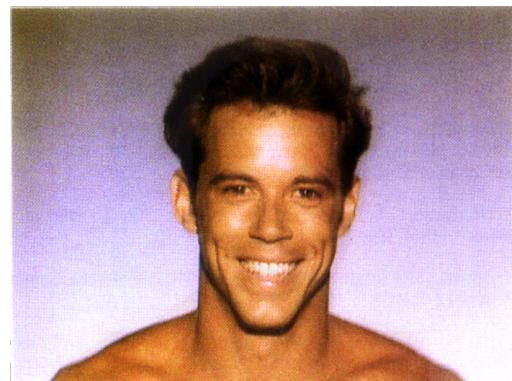
- Tests Unitaires : Nous avons testé chaque fonction individuelle pour s'assurer qu'elle renvoyait les résultats attendus pour un ensemble donné d'entrées.
- Tests d'Intégration : Nous avons vérifié que les différentes classes interagissaient correctement lorsqu'elles étaient combinées.
- Tests de Régression : Nous avons effectué des tests de régression pour s'assurer que les nouvelles modifications n'altéraient pas le comportement existant de l'application.
- Validation Visuelle : Nous avons utilisé des tests visuels pour valider les résultats

Tests Unitaires :

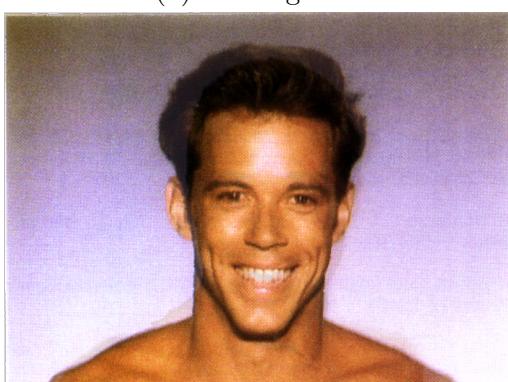
Remarque. Les classes des tests peuvent être trouvées en annexe.



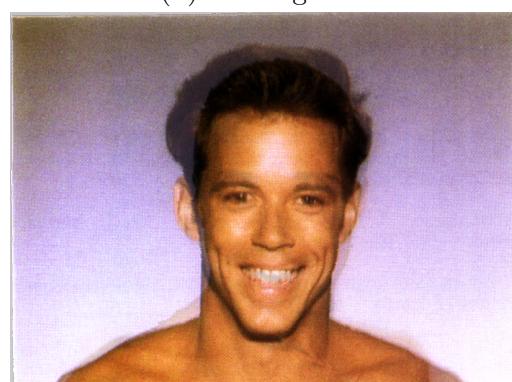
(a) Sous-figure 1



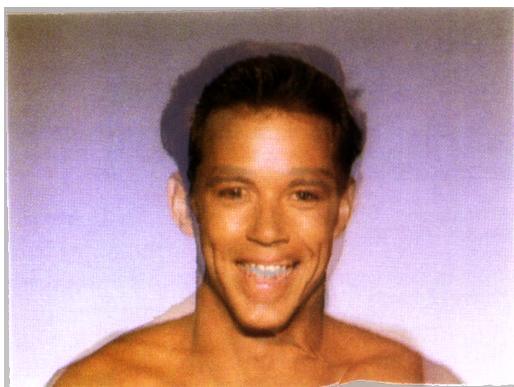
(b) Sous-figure 2



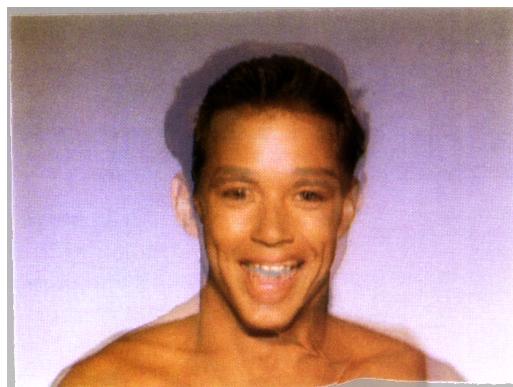
(c) Sous-figure 3



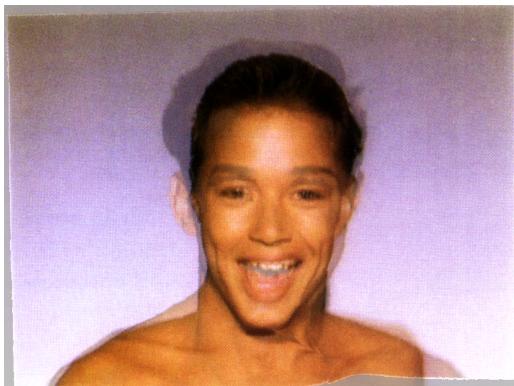
(d) Sous-figure 4



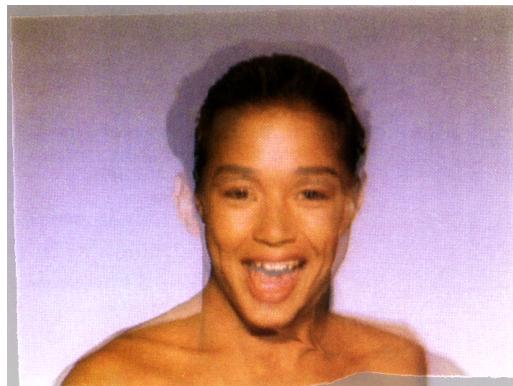
(a) Sous-figure 5



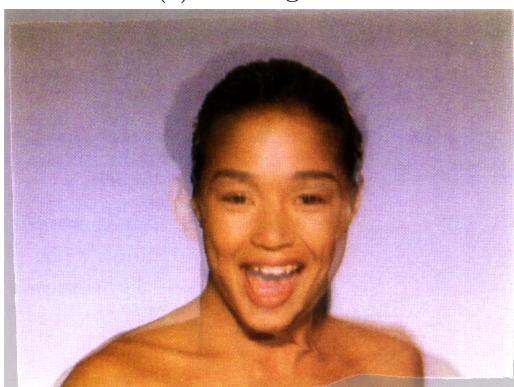
(b) Sous-figure 6



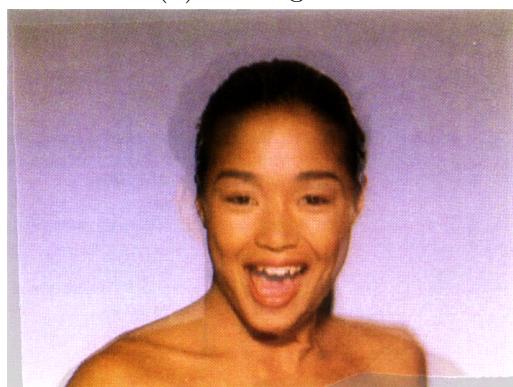
(c) Sous-figure 7



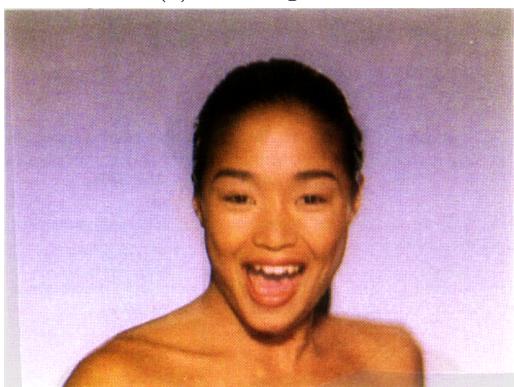
(d) Sous-figure 8



(e) Sous-figure 9



(f) Sous-figure 10



(g) Sous-figure 11

FIGURE 11 – Résultats des tests visuels

8 Bilans personnels

8.1 Bilan Rubens

Ce projet m'a permis de beaucoup gagner en expérience sur de nombreux aspects, notamment concernant la création d'application JavaFx, ainsi que l'arrangement du code avec la méthode PAC. J'ai eu l'impression de bien plus en apprendre sur Java en général avec un réel projet qui permettait de mettre en application toutes les connaissances (mêlé avec des mathématiques), plutôt qu'avec les différents TPs. Également, travailler avec des gens que l'on connaît peu est une expérience dont j'avais peu l'habitude et qui sera importante plus tard, donc je pense que c'était une bonne idée, bien que déplaisante au départ. Finalement, je trouve que ce projet s'est révélé très intéressant, enrichissant et amusant.

8.2 Bilan Alex

Ce projet fut l'occasion de mettre en pratique nos compétences acquises tout au long de cette première année en cycle ingénieur. J'ai pu comprendre profondément l'utilité de l'étape de conception d'un projet avec les différents diagrammes à concevoir pour ne pas se perdre dans le code. De plus, j'ai appris à utiliser GitHub qui est un outil dont on m'avait déjà beaucoup parlé mais que je n'ai jamais utilisé. Sa prise en main a été assez difficile pour moi au début, car j'aime comprendre exactement ce que je fais et ne connaissant pas cette application, j'ai dû me lancer dans un projet sans connaître parfaitement la prise en main de GitHub.

Néanmoins, la partie la plus importante et intéressante selon moi est l'organisation d'un groupe de travail. En effet, durant mes deux années de classe préparatoire aux grandes écoles, le travail de groupe n'était pas du tout mis en avant. Ainsi, avec ce projet, j'ai pu voir la difficulté de travailler avec des personnes qui n'ont pas les mêmes idées, les mêmes manières de penser, les mêmes manières de coder... J'ai donc beaucoup appris sur cet aspect là et j'en garde une très belle expérience.

8.3 Bilan Paul

Le projet de morphing a été une expérience extrêmement enrichissante pour moi. J'ai pu approfondir mes connaissances en algorithmes, notamment en travaillant sur l'algorithme de Beier-Neely pour faire du morphing d'images et des transformations géométriques, tout en mettant en pratique des concepts théoriques complexes. Travailler en groupe a également été très bénéfique ; j'ai appris différentes méthodes de programmation grâce aux contributions variées de mes coéquipiers, ce qui a enrichi mes compétences et ma perspective sur la résolution de problèmes.

8.4 Bilan Romain

J'ai trouvé ce projet très intéressant. Cela m'a permis de mieux comprendre le langage Java ainsi que la partie JavaFx. En effet, nous avons eu des tp sur ceci, mais là nous avons eu la possibilité de voir comment utiliser toutes les fonctionnalités vues en tp dans un projet concret. L'utilisation de la méthode PAC a été plus claire grâce à ce projet. Effectivement, je n'avais pas très bien compris comment celle-ci

fonctionnait et grâce à notre travail j'ai finalement compris pourquoi l'utiliser et comment cela fonctionnait. Enfin, le fait de ne pas avoir pu choisir les personnes avec qui nous voulions travailler fut une belle expérience. En effet, je n'en avais pas l'habitude jusque-là et j'ai dû faire face à d'autres manières de penser, d'autres manières de travailler. Cette expérience me sera, j'en suis sur, très bénéfique pour la suite de ma vie étudiante et professionnelle.

8.5 Bilan Ryan

Ce projet a été l'occasion pour moi de constater une énième fois les nécessités impérieuses auxquelles je dois faire face, tant dans la gestion d'un groupe, que dans la gestion de mon temps. En effet, j'ai pu constater que la gestion d'un groupe de travail est une tâche ardue, qui nécessite une communication constante, adapté à l'auditoire et, malheureusement, assez répétitive. Autrement, j'ai pu constater que la gestion de mon temps est un aspect que je dois améliorer, car j'ai souvent été attentiste dans les situations requérant la finalisation du travail d'autrui, ce qui ne m'a pas permis d'aboutir le projet en totalité.

Liste des Algorithmes

0	générationFramesNaif	7
1	isInside	8
2	morphingNaif	8
3	générationFramesLA	10
4	Calcul courbes B-splines	14
5	Morphing de courbes B-splines	15
6	Déformation d'une image avec un seul vecteur de contrôle [1]	18
7	Déformation d'une image avec deux vecteurs de contrôle [1]	19
8	Interpolation des images [1]	20
9	Trouver intervalle	32

Table des figures

2	Résultats de l'algorithme de morphing	9
3	Interpolation linéaire, observez les auto-intersections. [2]	10
4	Morphing par interpolation longueur-angle à droite, naïve à gauche. [5]	11
5	Courbes B-splines	13
6	Paires d'images à morpher [1]	16
7	Calcul d'un image intermédiaire [3]	16
8	Apparaïement à un seul vecteur [1]	17
9	Multiple lignes [1]	19
11	Résultats des tests visuels	27
12	Résultat des tests unitaires	33
13	Résultat des tests unitaires	34
14	Résultat des tests unitaires	35
15	Résultat des tests unitaires	36
16	Résultat des tests unitaires	37
17	Résultat des tests unitaires	38

Annexes

A Morphing de courbes splines

```

// Trouve l'intervalle dans le vecteur de noeuds qui contient
le paramètre u
Fonction trouverIntervalle(u, nodeVector) :
    n ← nodeVector.size() – 1;
    si u ≥ nodeVector.get(n) alors
        ↘ retourner n – 1;
    pour i ← 0 à n – 1 faire
        si u ≥ nodeVector.get(i) and u < nodeVector.get(i + 1) alors
            ↘ retourner i;
    retourner –1 // Interval not found

```

Algorithme 9 : Trouver intervalle

B Tests unitaires

```

package morphing;
public class MorphingTest {
    public static void main(String[] args) {
        testCouple();
        testPoint();
        testLine();
        testBezier();
    }

    public static void testCouple() {
        System.out.println("Testing Couple Class");
        Couple<Integer, Integer> couple = new Couple<>(5, 10);
        System.out.println("Expected: 5, Actual: " + couple.getX());
        System.out.println("Expected: 10, Actual: " + couple.getY());

        couple.setX(15);
        couple.setY(20);
        System.out.println("Expected: 15, Actual: " + couple.getX());
        System.out.println("Expected: 20, Actual: " + couple.getY());

        System.out.println("Expected: Couple{x=15, y=20}, Actual: " + couple.toString());

        Couple<Integer, Integer> coupleCopy = couple.copy();
        System.out.println("Expected: true, Actual: " + couple.equals(coupleCopy));
    }

    public static void testPoint() {
        System.out.println("Testing Point Class");
        Point point = new Point(3, 4);
        System.out.println("Expected: Couple{x=3, y=4}, Actual: " + point.getPoint());

        point.setPoint(5, 6);
        System.out.println("Expected: Couple{x=5, y=6}, Actual: " + point.getPoint());

        Point pointAdded = point.add(new Point(1, 1));
        System.out.println("Expected: Couple{x=6, y=7}, Actual: " + pointAdded.getPoint());

        Point pointSubbed = point.sub(new Point(1, 1));
        System.out.println("Expected: Couple{x=4, y=5}, Actual: " + pointSubbed.getPoint());
    }
}

```

FIGURE 12 – Résultat des tests unitaires

```

package morphing;
public class MorphingTest {
    public static void main(String[] args) {
        testCouple();
        testPoint();
        testLine();
        testBezier();
    }

    public static void testCouple() {
        System.out.println("Testing Couple Class");
        Couple<Integer, Integer> couple = new Couple<>(5, 10);
        System.out.println("Expected: 5, Actual: " + couple.getX());
        System.out.println("Expected: 10, Actual: " + couple.getY());

        couple.setX(15);
        couple.setY(20);
        System.out.println("Expected: 15, Actual: " + couple.getX());
        System.out.println("Expected: 20, Actual: " + couple.getY());

        System.out.println("Expected: Couple{x=15, y=20}, Actual: " + couple.toString());

        Couple<Integer, Integer> coupleCopy = couple.copy();
        System.out.println("Expected: true, Actual: " + couple.equals(coupleCopy));
    }

    public static void testPoint() {
        System.out.println("Testing Point Class");
        Point point = new Point(3, 4);
        System.out.println("Expected: Couple{x=3, y=4}, Actual: " + point.getPoint());

        point.setPoint(5, 6);
        System.out.println("Expected: Couple{x=5, y=6}, Actual: " + point.getPoint());

        Point pointAdded = point.add(new Point(1, 1));
        System.out.println("Expected: Couple{x=6, y=7}, Actual: " + pointAdded.getPoint());

        Point pointSubbed = point.sub(new Point(1, 1));
        System.out.println("Expected: Couple{x=4, y=5}, Actual: " + pointSubbed.getPoint());

        Point pointMul = point.mul(2);
        System.out.println("Expected: Couple{x=10, y=12}, Actual: " + pointMul.getPoint());

        Point pointDiv = point.div(2);
        System.out.println("Expected: Couple{x=2, y=3}, Actual: " + pointDiv.getPoint());
    }
}

```

FIGURE 13 – Résultat des tests unitaires

```

3   public static void testLine() {
4       System.out.println("Testing Line Class");
5       Point start = new Point(0, 0);
6       Point end = new Point(3, 4);
7       Line line = new Line(start, end);
8
9       System.out.println("Expected: Couple{x=Point{x=0, y=0}, y=Point{x=3, y=4}}, Actual: "
10      + line.getStart().getPoint().toString() + ", " + line.getEnd().getPoint().toString());
11      System.out.println("Expected: 5.0, Actual: " + line.norme());
12
13      Couple<Double, Double> vector = line.getVector();
14      System.out.println("Expected: Couple{x=3.0, y=4.0}, Actual: " + vector);
15
16      Couple<Double, Double> normalVector = line.vectorNormal();
17      System.out.println("Expected: Couple{x=-4.0, y=3.0}, Actual: " + normalVector);
18
19      Double dotProduct = line.scalaire(new Line(new Point(0, 0), new Point(1, 1)));
20      System.out.println("Expected: 7.0, Actual: " + dotProduct);
21
22      double relativeHeight = line.hauteurRelative(new Point(5, 4));
23      System.out.println("Expected (approx.): 1.24, Actual: " + relativeHeight);
24
25      double dist = line.dist(new Point(5, 4));
26      System.out.println("Expected (approx.): 1.6, Actual: " + dist);
27  }
28
29  public static void testBezier()
30  {
31     CourbesBezier curve = new CourbesBezier();
32     curve.ajouterPointControle(0, 0);
33     curve.ajouterPointControle(10, 10);
34     curve.ajouterPointControle(20, 0);
35
36     System.out.println("Point at t=0.0 Expected (0, 0): " + curve.calculerPoint(0.0).getPoint());
37     System.out.println("Point at t=0.5 Expected (10,5): " + curve.calculerPoint(0.5).getPoint());
38     System.out.println("Point at t=1.0 Expected (20,0): " + curve.calculerPoint(1.0).getPoint());
39  }
40
41

```

FIGURE 14 – Résultat des tests unitaires

```

public class LineTest {

    public static void main(String[] args) {
        // Create some points to be used for lines
        Point start = new Point(0, 0);
        Point end = new Point(4, 4);
        Point anotherPoint = new Point(1, 2);

        // Create a line using two points
        Line line = new Line(start, end);

        // Test getLength of line
        System.out.println("Length of line: " + line.norme());

        // Test perpendicular line
        Line perpLine = line.perpendicular();
        System.out.println("Perpendicular line starts at: (" + perpLine.getStart().getX() + ", " + perpLine.getStart().getY() +
                           ") and ends at: (" + perpLine.getEnd().getX() + ", " + perpLine.getEnd().getY() + ")");

        // Test distance from a point to the line
        double distance = line.dist(anotherPoint);
        System.out.println("Distance from point (" + anotherPoint.getX() + ", " + anotherPoint.getY() +
                           ") to line is: " + distance);

        // Test hauteur (scalar projection) from a point to the line
        double height = line.hauteur(anotherPoint);
        System.out.println("Scalar projection (hauteur) from point (" + anotherPoint.getX() + ", " + anotherPoint.getY() +
                           ") onto line is: " + height);
    }
}
    
```

FIGURE 15 – Résultat des tests unitaires

```

public static void main(String[] args) {
    ArrayList<Point> controlPolygon = new ArrayList<>();
    controlPolygon.add(new Point(0, 0)); // Starting point
    controlPolygon.add(new Point(100, 200)); // Influence point
    controlPolygon.add(new Point(300, -100)); // Influence point
    controlPolygon.add(new Point(400, 300)); // Influence point
    controlPolygon.add(new Point(700, 0)); // Ending point

    ArrayList<Double> knotVector = new ArrayList<>();
    int deg = 2;
    int n = controlPolygon.size();
    int m = n + deg;
    for (int i = 0; i <= m; i++) {
        if (i < deg + 1) {
            knotVector.add(0.0);
        } else if (i >= m - deg) {
            knotVector.add(1.0);
        } else {
            knotVector.add((double) (i - deg) / (n - deg + 1));
        }
    }

    BSpline spline = new BSpline(deg, controlPolygon, knotVector);

    System.out.println("Initial Points on the B-Spline:");
    for (double t = 0.0; t <= 1.0; t += 0.1) {
        try {
            Point p = spline.calculerPoint(t, false);
            if (p != null) {
                System.out.printf("t = %.1f -> Point(%d, %d)\n", t, p.getX(), p.getY());
            }
        } catch (Exception e) {
            System.err.println("Error computing point at t=" + t + ": " + e.getMessage());
        }
    }

    System.out.println("Modifying control points and recomputing:");
    spline.setcontrolPolygon(new ArrayList<>(controlPolygon));
    spline.getcontrolPolygon().remove(2); // Remove one point
    spline.addControlPoint(new Point(350, 150)); // Add a new control point
}

```

FIGURE 16 – Résultat des tests unitaires

```

        }

        BSpline spline = new BSpline(deg, controlPolygon, knotVector);

        System.out.println("Initial Points on the B-Spline:");
        for (double t = 0.0; t <= 1.0; t += 0.1) {
            try {
                Point p = spline.calculerPoint(t, false);
                if (p != null) {
                    System.out.printf("t = %.1f -> Point(%d, %d)\n", t, p.getX(), p.getY());
                }
            } catch (Exception e) {
                System.err.println("Error computing point at t=" + t + ": " + e.getMessage());
            }
        }

        System.out.println("Modifying control points and recomputing:");
        spline.setcontrolPolygon(new ArrayList<>(controlPolygon));
        spline.getcontrolPolygon().remove(2); // Remove one point
        spline.addControlPoint(new Point(350, 150)); // Add a new control point

        for (double t = 0.0; t <= 1.0; t += 0.1) {
            try {
                Point p = spline.calculerPoint(t, false);
                if (p != null) {
                    System.out.printf("After modification - t = %.1f -> Point(%d, %d)\n", t, p.getX(), p.getY());
                }
            } catch (Exception e) {
                System.err.println("Error computing point at t=" + t + ": " + e.getMessage());
            }
        }

        try {
            spline.initMatrice();
            System.out.println("Matrix and additional structures initialized successfully.");
        } catch (Exception e) {
            System.err.println("Error initializing matrix: " + e.getMessage());
        }
    }
}

```

FIGURE 17 – Résultat des tests unitaires

Références

- [1] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. *Computer Graphics*, 26(2) :35–42, 1992.
- [2] Mélanie Cornillac. *Morphing multirésolution de courbes*. Modélisation et simulation, Université de Grenoble, 2010. NNT : ff, tel-00581474f.
- [3] Department of Computer Science, University of Toronto and Flores-Mangas, Fernando. Csc320w : Introduction to visual computing. Course Material, 2021. Course syllabus, lecture slides, and other materials may be available on the course website or through the department.
- [4] Pierre Pansu. Courbes b-splines, 2004. Accessed : 2024-05-30.
- [5] T.W. Sederberg, P. Gao, G. Wang, and H. Mu. 2-d shape blending : An intrinsic solution to the vertex path problem. In *Computer Graphics (SIGGRAPH 93 Proceedings)*, volume 27, pages 15–18, 1993.
- [6] C.-K. Shene. Cs3621 introduction to computing with geometry notes, Year Not Specified. Professor, Department of Computer Science, Michigan Technological University.