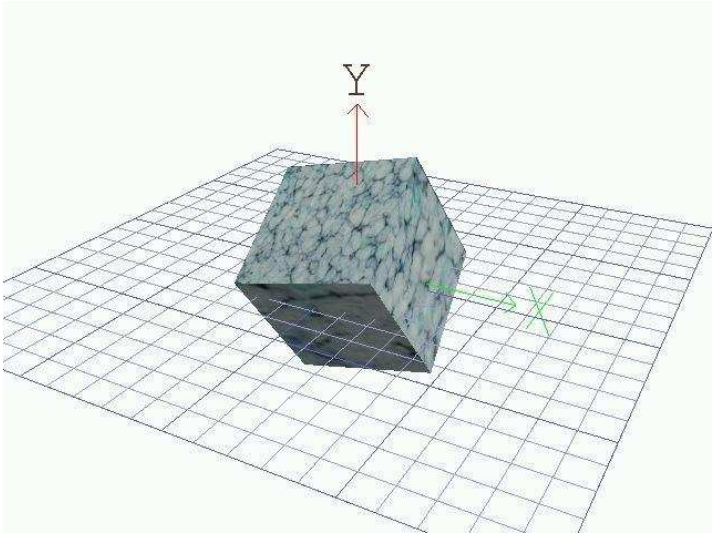
Benjamin VARIOT
Anthony CROS

Année 2003/2004
Vendredi 11 juin 2004
Université Claude Bernard Lyon 1

Rapport de TER

Licence informatique

Comparaison d'images et Optimisation



Encadreur : Saida BOUAKAZ

Tables des matières

Introduction

Le TER (Travail Encadré de Recherche) dont nous nous sommes occupés a pour intitulé exact : Comparaison d'Images et Optimisation. Nous nous sommes tournés vers ce TER tout d'abord parce qu'il offrait la possibilité de toucher une partie de l'informatique sur laquelle nous avions peu de connaissances : les images. De plus, nous avons pu reprendre un langage de programmation qui nous était familier mais pour lequel nous manquions de travaux pratique : le C++.

Le but du projet est donc de générer des images de synthèses les plus ressemblantes possibles à des images réelles fournies. Ces images de synthèse sont réalisées à partir d'un modèle générique fourni lui aussi.

Le sujet en lui-même est divisé en deux grandes parties de programmation. La première consiste à implémenter un calcul de différence entre deux images. Ceci nous offre la possibilité de nous tourner vers la structure même des images. La deuxième concerne plus l'algorithmique, il s'agit de l'optimisation. L'algorithme de base dont on s'est servi est le simplexe, qui, appliqué à deux images, permet d'optimiser leur ressemblance à partir d'un modèle. Nous expliquerons ce que sont les mutations et comment elles permettent un meilleur fonctionnement du simplexe. Enfin le but final du projet étant d'appliquer notre programme sur une vidéo, nous présenterons un exemple simple de l'utilisation.

Mais avant de parler de la partie pratique, nous nous attacherons à expliquer le plus clairement possible les bases théorique qui sont en jeu en vue de l'établissement du programme.

I. Etude théorique

I.1. Objectif du simplexe

La méthode du simplexe fait partie des algorithmes de descente. Cet algorithme présente un intérêt non négligeable pour la programmation : celui de ne pas utiliser de calcul de dérivées, difficiles à réaliser sans expression mathématique de la formule (comme c'est le cas dans notre étude). Comme son nom l'indique l'algorithme de descente a pour but de trouver une estimation d'un minimum (dans notre travail, un minimum de la différences entre 2 images) ceci en partant d'une certaine valeur initiale et en tentant de la faire décroître pour chaque itération d'où le terme de descente.

Le simplexe appliqué aux images correspond donc parfaitement au travail à réaliser c'est-à-dire comparer puis optimiser des images pour les faire ressembler le plus possible au modèle d'origine.

Pour cela on doit disposer d'un modèle possédant des paramètres que l'on puisse faire évoluer (ex : couleur, taille, rotation…). C'est-à-dire qu'il faudra que l'on ait une fonction permettant à partir d'un vecteur de ces paramètres de générer l'image correspondante avec la position, la couleur ou la taille souhaitée.

Puis à partir de ce modèle, nous devrons appliquer ce fameux algorithme qui nous permettra de minimiser la valeur de la fonction de comparaison des deux images.

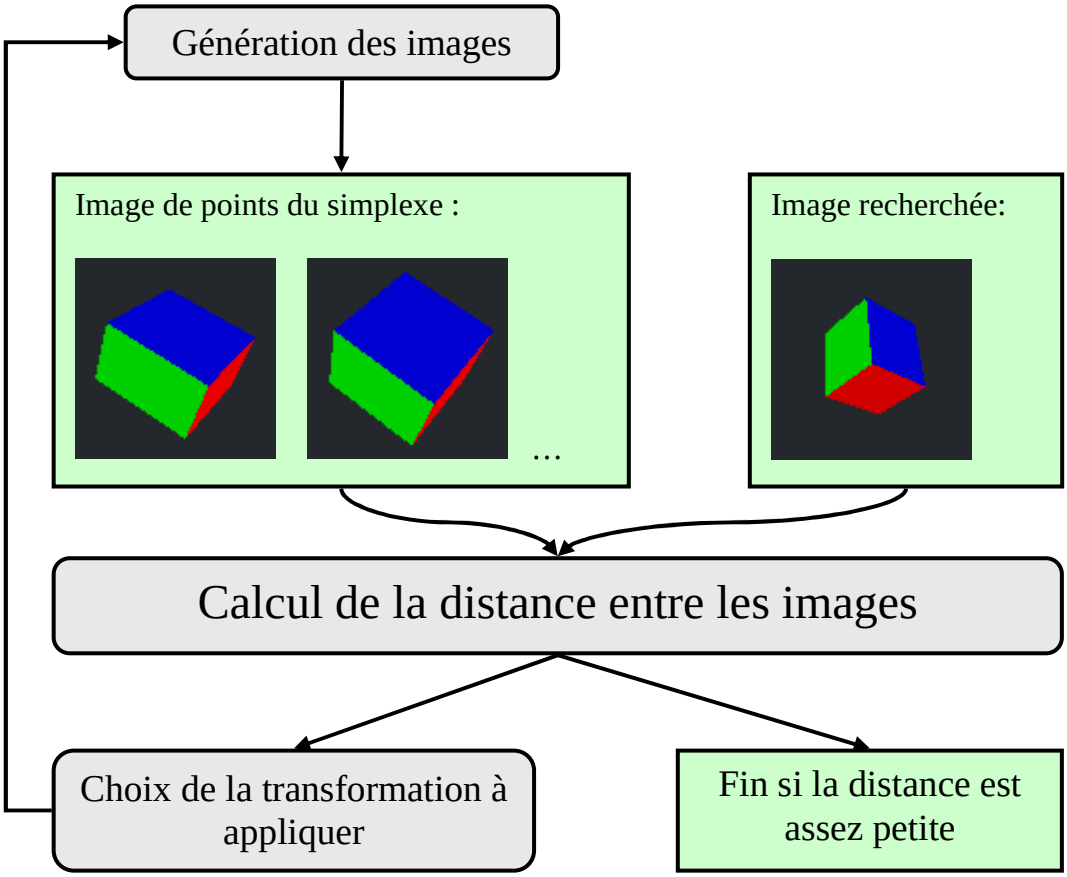
I.2. Principe du simplexe

Avant de pouvoir utiliser ce principe on doit s'assurer de posséder une fonction efficace de comparaison entre images. Cette fonction que l'on nomme distance() permet d'obtenir une valeur réelle positive correspondant à une mesure de différence entre les pixels, examinés un à un sur chacune des deux images.
Ensuite cette méthode est appliquée à un ensemble de n+1 vecteurs (avec n le nombre de paramètres variables du modèle) que l'on appelle un simplexe. Il faudra donc pouvoir générer une image à partir d'un vecteur.

Puis parmi toutes les images/vecteurs de l'ensemble on devra repérer celle pour laquelle la valeur retournée par distance() est le maximum. Une fois cela fait, on recherchera à l'aide de diverses transformations (que l'on explicitera plus tard), une image plus proche (c'est-à-dire avec une distance inférieure) que le minimum de l'ensemble. On remplacera alors le maximum par un nouveau point.

Enfin on réitérera cette opération jusqu'à obtenir une valeur de la distance suffisamment proche de l'image souhaitée (à une constante de distance près).

Voici le schéma explicatif d'un cycle du simplexe :



Nous verrons plus tard que cet algorithme seul pose plusieurs problèmes dont la solution sera d'effectuer des mutations.

I.3. Quelles sont les transformations ?

Les transformations sont une partie essentielle du programme, contribuant notamment à l'efficacité de la méthode du simplexe.

Leur objectif est de permettre de déplacer un point (en réalité une image appartenant à l'ensemble du simplexe), ou éventuellement plusieurs points, selon le type de transformations nécessaires pour obtenir un nouvel ensemble plus satisfaisant.

Pour cela on aura besoin du centre de gravité de l'ensemble des points privé du point maximum.

En se basant sur la thèse [2], nous avons utilisé 4 types de transformations :

- Réflexion : a pour effet de faire avancer le simplexe dans la direction dans lequel il se trouve déjà.

- Réflexion + Expansion : permet de faire avancer le simplexe plus rapidement si la direction de la réflexion est bonne. Grossièrement on peut dire qu'elle va doubler l'effet de la réflexion.

- Réflexion + Contraction1 : l'effet de contraction1 est de ramener le maximum dans la direction du groupe de points.

- Réflexion + Contraction1 + Contraction2 : est celle qui rapproche le plus tous les points puisqu'elle fait évoluer tous les points dans la direction du minimum.

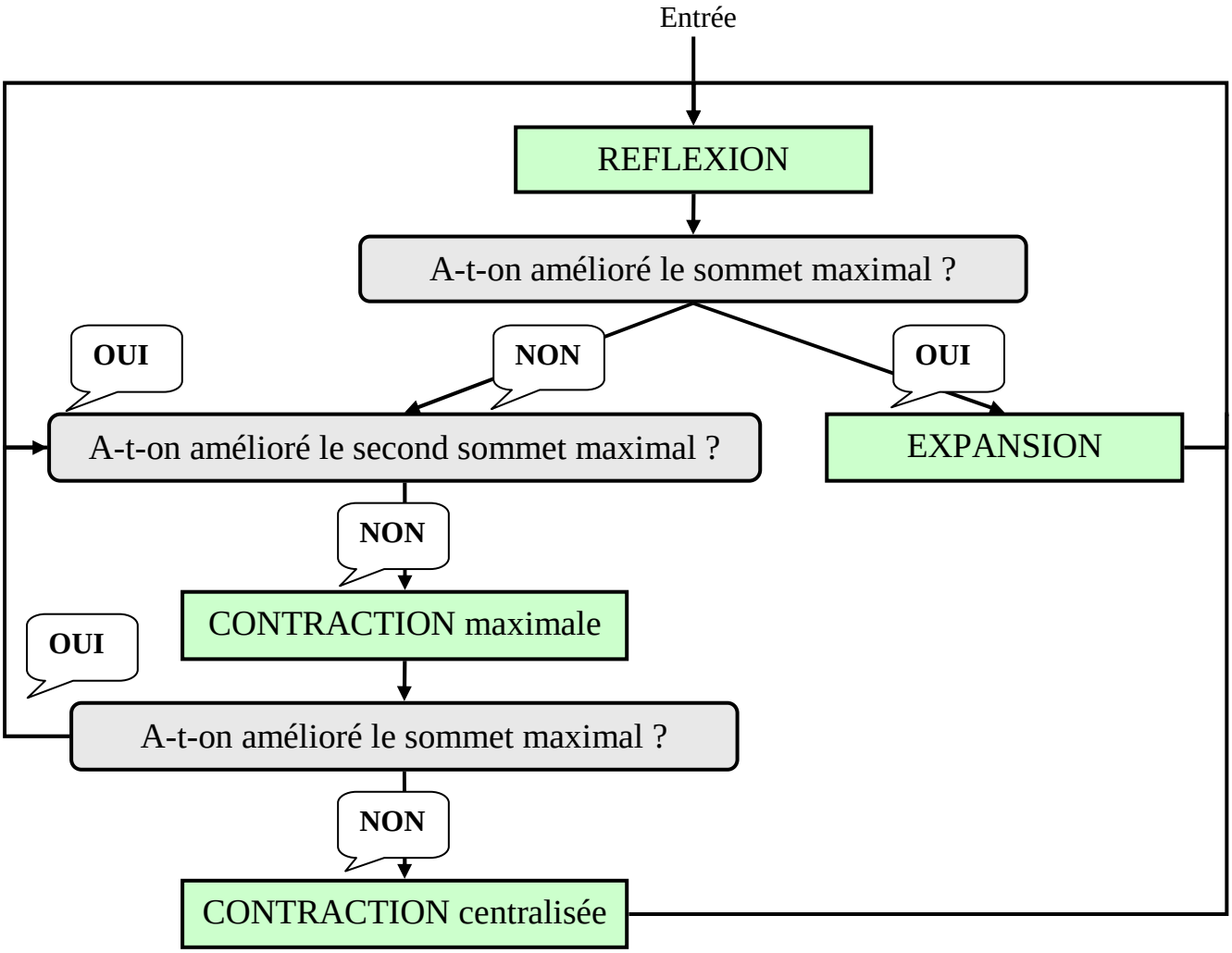
Nous détaillerons les conséquences exactes de ces transformations dans la partie pratique sur des exemples simples.

I.4. Cas d'utilisation des transformations

Nous allons maintenant étudier les différents cas d'application des transformations.
En effet, les 4 transformations citées précédemment doivent chacune être déclenchées dans un cas précis.
Pour gérer ces cas, nous avons eu besoin de conserver en mémoire plusieurs informations importantes :

- La valeur et l'indice du maximum dans le simplexe
- La valeur du second sommet maximal
- La valeur et l'indice du minimum

Voici le schéma récapitulatif des conditions posées avant chaque transformation :



II. Réalisation pratique

II.1. Distance entre images

Comme nous l'avons expliqué auparavant l'algorithme du simplexe utilise la notion de distance entre 2 images. Nous avons donc implanté une fonction qui compare la chrominance et la luminance entre chaque pixel des 2 images.

La fonction distance() renvoie la moyenne des distances entre les pixels qui compose l'ensemble des deux images. La distance entre 2 pixels est calculée en sommant et en pondérant d'une part, la distance du point de vue de la chrominance, et d'autre part la distance du point de vue de la luminance. Dans notre programme, nous avons privilégié la luminance en lui affectant un coefficient plus élevé (poids_luminance : [constantes.h]).

Avec deux images I_1et I_2 on obtient la formule suivante pour le calcul complet de la distance :

$$\varepsilon(I_1, I_2) = \frac{1}{Card(D)} \times \sum_{(x,y) \in D} d_{couleur}(I_1[x,y], I_2[x,y])$$

où D est le tableau de pixels des images.

Avec la fonction distance couleur qui vaut:

$$d_{couleur}(C_1, C_2) = \alpha \times d_{chrominance}(C_1, C_2) + (1-\alpha) \times d_{luminance}(C_1, C_2)$$

où C_1 et C_2 sont des pixels.

Pour déterminer la distance entre 2 pixels du point de vue de la chrominance, nous avons utilisé la formule suivante :

$$d_{chrominance}(C_1, C_2) = \sqrt{(C_{r1} - C_{r2})^2 + (C_{b1} - C_{b2})^2}$$

où C_r et C_b sont les deux composantes de la chrominance.

Pour déterminer la distance entre 2 pixels du point de vue de la luminance, nous avons utilisé la formule suivante :

$$d_{luminance}(C_1, C_2) = |Y_1 - Y_2|$$

où Y est la composante de la luminance.

Mais pour effectuer tous ces calculs, il faut récupérer le vecteur (Y, Cr, Cb) correspondant respectivement à la luminance et aux deux composantes de la chrominance (permettant de faire les calculs précédents) à partir de la norme RGB - red, green, blue - dont nous pouvions récupérer les valeurs à l'aide de notre classe [C_Pixel]) et en effectuant la multiplication suivante (en se basant sur la thèse [1]):

$$\begin{pmatrix} Y \\ C_r \\ C_b \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 & 0 \\ -0.1687 & -0.3313 & 0.5 & 128 \\ 0.5 & -0.4187 & -0.0813 & 128 \end{pmatrix} \times \begin{pmatrix} R \\ G \\ B \\ 1 \end{pmatrix}$$

Par la suite, c'est la valeur renvoyée par la fonction distance() que l'on va chercher à minimiser. Dans notre programme, on considère que cette valeur est suffisamment petite lorsqu'elle vaut la valeur epsilon ou moins (epsilon : [constantes.h]).

Le programme traite des images au format TGA. La récupération des pixels est donc basée sur ce format d'images. Par conséquent pour pouvoir traiter des images d'un autre format, il serait nécessaire de modifier la seule fonction distance() dans le fichier [distance.cpp], les autres fonctions du fichier étant complètement générique et donc indépendantes du format.

II.2. Ecartement initial du simplexe

Revenons maintenant à l'application du simplexe dans notre cas. On veut optimiser la ressemblance entre 2 images, l'une étant fixe, l'autre étant celle modifiable via un vecteur de paramètres. Partant d'un vecteur générant une image relativement proche de celle que l'on veut retrouver, on crée le groupe de points appelé simplexe. Ce groupe est constitué de nb_variables + 1 points, avec la contrainte de ne pas avoir 3 points ou plus alignés (sinon le simplexe ne peux plus évoluer dans toutes les dimensions et reste « bloqué » dans un sous-espace).

Il est donc nécessaire d'établir un écartement initial autour du vecteur initial. La fonction ecartement() [simplexe.cpp] permet d'ajouter au simplexe, en plus du vecteur initial, autant de points que de dimensions. Chacun de ces points est calculé en ajoutant ou en soustrayant un certain décalage à une seule des variables. Ces variables sont toutes définies par un domaine de définition propre à chacune (plus une pondération cf. partie amélioration). Par défaut, le décalage est ajouté. Si la valeur de la variable auquel on ajoute le décalage sort du domaine de définition de la variable, alors on soustraie ce décalage plutôt que l'ajouter.

Le décalage est calculé pour une variable donnée en divisant son domaine de définition par une certaine constante (inverse_ecartement_initial_simplexe : [constantes.h])

Exemple :

- On part du vecteur suivant : (1,1,1,1) avec des paramètres pouvant tous évolués dans l'intervalle [0,10]
- inverse_ecartement_initial_simplexe vaut 5

On obtient donc le simplexe suivant :

(3,1,1,1) → 1+(10/5) = 1 + 2 = 3
(1,3,1,1)
(1,1,3,1)
(1,1,1,3)
(1,1,1,1)

- Avec un vecteur initial (1,5,7,9), on obtiendrait le simplexe :

(3,5,7,9)
(1,7,7,9) → 5+(10/5) = 5 + 2 = 7
(1,5,9,9)
(1,5,7,7) → 9+(10/5) = 9 + 2 = 11 → 11 n'appartient pas à [0,10] → 9-(10/5) = 9 - 2 = 7
(1,5,7,9)

II.3. Comment les transformations sont-elles mises en œuvre ?

On a vu dans la première partie que l'algorithme du simplexe s'appuie sur un certain nombre de transformations. Ces transformations permettent au simplexe de converger vers une valeur minimale de la fonction distance(). Notre programme utilise 4 types de transformations introduites dans la première partie du rapport.

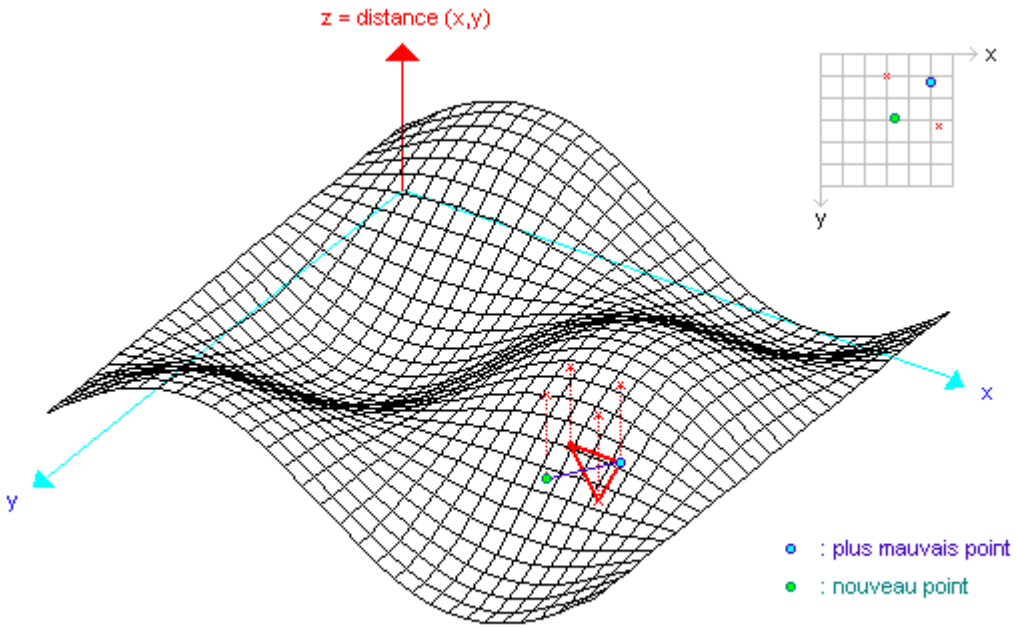
Nous allons explicités les transformations à l'aide de schémas représentants la valeur de la distance pour un couple de valeurs de paramètres. On prend donc l'hypothèse qu'on a un modèle paramétrable selon 2 variables x et y. Le simplexe associé est donc composé de 3 points.

Nous avons utilisé une seule et même formule pour trois des quatre transformations à un coefficient près. Il s'agit de la Réflexion, la Réflexion + Expansion et la Réflexion + Contraction par rapport au maximum. De plus comme nous l'avions introduit celle-ci utilise le centre de gravité des points sans le maximum. Voici cette formule :

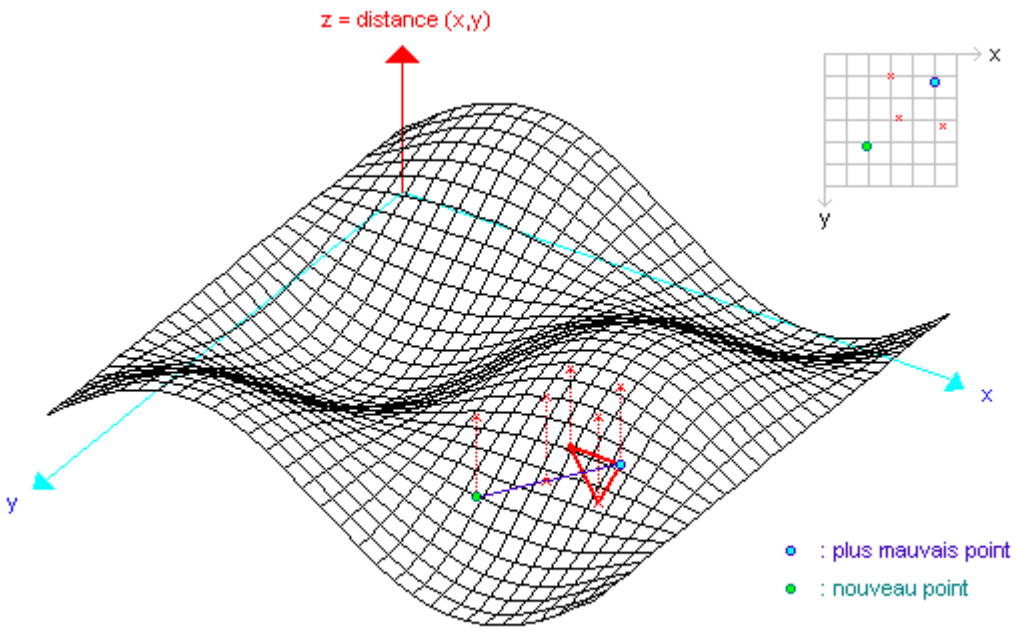
$$h' = [C \times (1+\alpha)] - [h \times \alpha]$$

où h est le vecteur max, h' le nouveau vecteur et C est le centre de gravité (sans max).
Alpha est la constante que l'on fait varier selon le type de transformation auquel on a à faire.

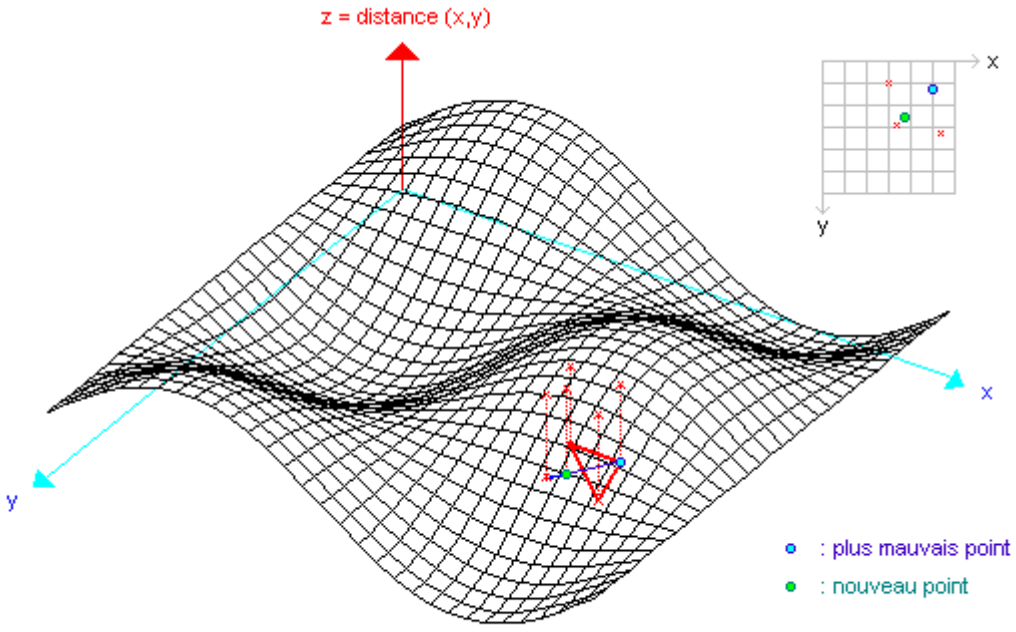
Observons la réflexion : celle-ci s'effectue avec la formule présenté ci-dessus et une valeur de alpha = 1. Il faut imaginer que le triangle se trouve sur le plan formé par les axes x et y, plan d'ailleurs reproduit sur la grille en haut à droite des schémas.



Puis la Réflexion + Expansion : elle s'effectue en appliquant 2 fois la formule, avec alpha = 1 puis alpha = -2.



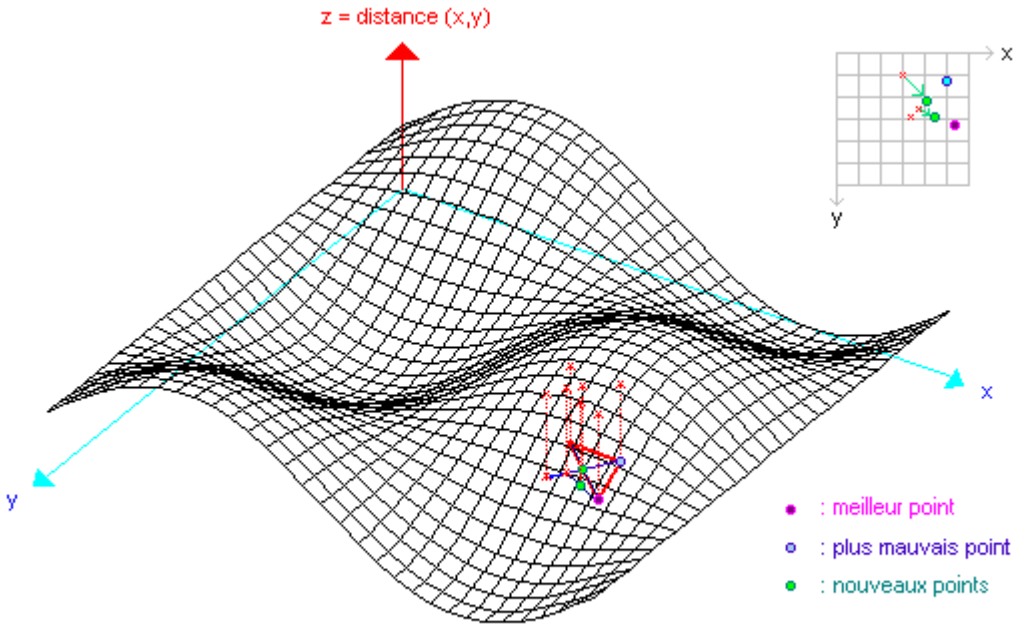
Puis la Réflexion + Contraction par rapport au maximum : c'est toujours la même formule qui est utilisée avec alpha = 1 tout d'abord, puis alpha = -½.



Enfin la Réflexion + Contraction par rapport au maximum + Contraction centralisée : on ajoute à la précédente transformation l'opération de contraction centralisée suivante :

$$h' = \frac{(h+l)}{\beta}$$

où l est le minimum des points, h' l'image finale, h tous les points hormis l et bêta vaut 2.

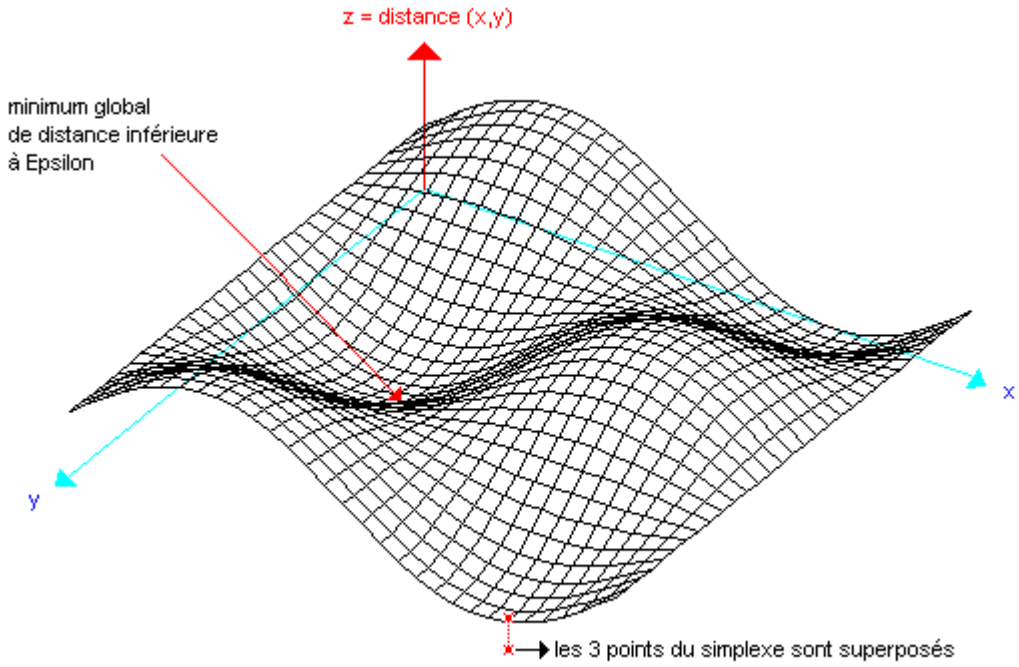


Remarque : les 3 valeurs de alpha 1, -2 et -½ vues dans ces transformation correspondent respectivement aux constantes controle_distance_reflexion, controle_distance_expansion et controle_distance_contraction du fichier [constantes.h].
De même, le paramètre bêta de la contraction centralisée correspond à la constante controle_distance_contraction_centralisee de ce même fichier.

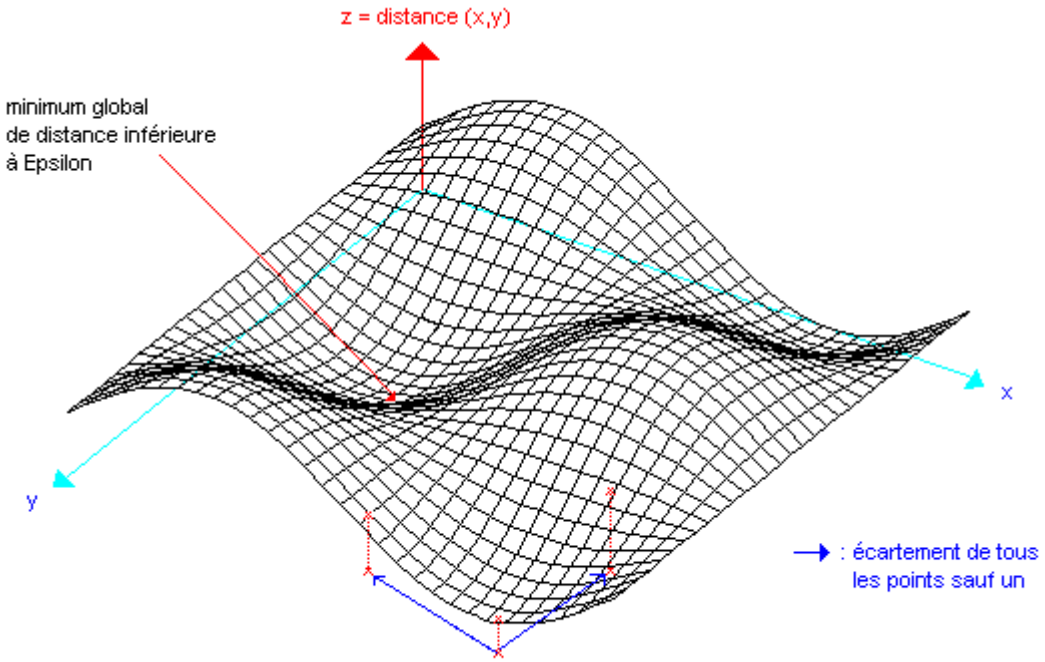
II.4. Mutations

Nous en arrivons donc au principal défaut de l'algorithme du simplexe. Celui-ci vient du fait qu'il ne distingue pas les minima locaux du minimum global. En effet, il est possible que le simplexe converge vers un minimum local qui ne soit pas suffisamment petit pour convenir aux exigences de la ressemblance souhaitée (inférieur à la constante epsilon). Lorsque l'un des points du simplexe a atteint ce minimum local, le simplexe se contracte et finit par n'être plus représenté que par un seul point, puisqu'il lui est impossible de descendre d'avantage dans cette zone.

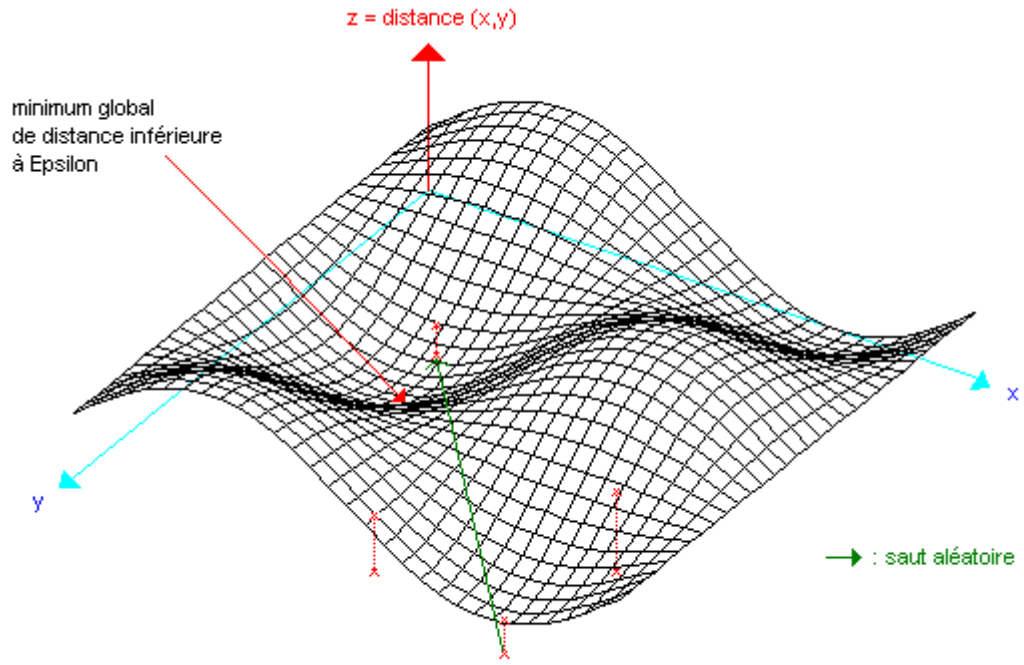
Cas de lancement de la mutation :



Pour effectuer la mutation, on effectue dans un premier temps une opération qui écarte le simplexe afin entre autre de récupérer toutes les dimensions. On se sert de la même fonction que pour l'écartement initial du simplexe autour du point sur lequel s'est contracté le simplexe, mais avec une constante différente pour le calcul du décalage (inverse_ecartement_mutation_simplexe : [constantes.h]). Ceci permet également de faire remonter la valeur de la distance pour tous les points du simplexe à l'exception du point considéré comme le minimum.



Dans un deuxième temps, on permet au simplexe de sortir du minimum local en donnant au point minimum une valeur aléatoire respectant les contraintes des domaines de définitions. Avoir effectué l'écartement des points avant permet donc également d'augmenter la probabilité pour le simplexe de changer de « puit ». L'opération d'écartement permet donc alors d'éviter que la nouvelle valeur aléatoire soit immédiatement ramenée dans le minimum local, car il est peu probable de trouver aléatoirement un vecteur dont la distance soit plus petite que celle du minimum local.



Concrètement, notre programme déclenche une mutation lorsque les points du simplexe sont confondus. Dans le cas contraire, soit le minimum a une valeur inférieure à epsilon et on renvoie le vecteur correspondant (fin de l'algorithme), soit on effectue des transformations (Cf. partie précédente).

II.5. Choix du modèle paramétrable

Le modèle que nous utilisons nous a été donné par Benoît VIGUIER, qui dans le cadre de son TER (encadré par Jean-Claude IEHL) a entre autre implémenté un module de génération de formes géométriques. Ce module nous permet de générer un parallélépipède au centre de l'image dont la couleur des faces est rouge, verte et bleue, et qu'il est possible de voir sous tous les angles. Dans notre programme, on peut faire varier 6 paramètres, qui sont respectivement les 3 rotations suivant les axes et les 3 longueurs des arêtes du parallélépipède. Nous avons choisit de faire varier 6 paramètres, mais il possible d'en faire varier plus, notamment la couleur des 3 faces et la position des 3 lumières.

Pour augmenter le nombre de variables dont le modèle dispose, il faut effectuer différents changements dans notre programme. Par exemple, pour faire varier la couleur d'une des faces et donc passer à 9 variables, il faut effectuer les changements suivants :

- Modifier la fonction generation_image() [generation_image.cpp] de manière à transformer les constantes voulues en variables.

```
Color CSuperGeom1( 0 , 0 , 250 );
```

```
Color CSuperGeom1 ( vect.get_param(8) , vect.get_param(7) , vect.get_param(6) );
```

- Changer la valeur de nb_variables [constantes.h] de 6 à 9

- Modifier le programme principal [main.cpp] :

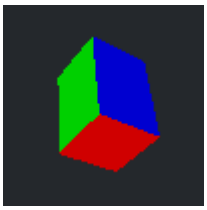
```
type_vecteur tab1[ nb_variables ] = { Pi/5 , Pi/5 , Pi/5 , 1 , 1 , 1 , 0 , 0 , 250 };
type_vecteur tab2[ nb_variables ] = { Pi/5 , Pi/5 , Pi/5 , 1.1 , 1 , 1 , 2 , 5 , 220 };

type_vecteur min_params [ nb_variables ] = { 0 , 0 , 0 , 0.1 , 0.1 , 0.1 , 0 , 0 , 0 };
type_vecteur max_params [ nb_variables ] = { Pi , Pi , Pi , 2 , 2 , 2 , 255 , 255 , 255 };
type_vecteur poids_params [ nb_variables ] = { 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 };
```

II.6. Reconstitution d'une image

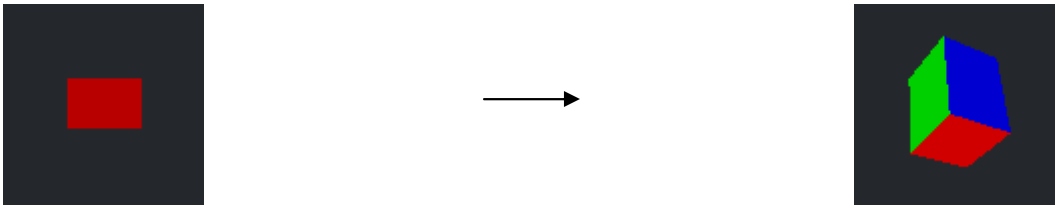
Les différentes phases du simplexe utilisé par notre programme sont expliquées précédemment. Voici maintenant un exemple d'application. Nous allons observer le comportement du vecteur minimum du simplexe à différentes étapes du processus d'optimisation (avant mutation, transformation ou acceptation).

Pour ce test nous avons choisi une valeur pour epsilon de 1 ce qui d'après nos tests précédents donnent une image proche de celle d'origine.

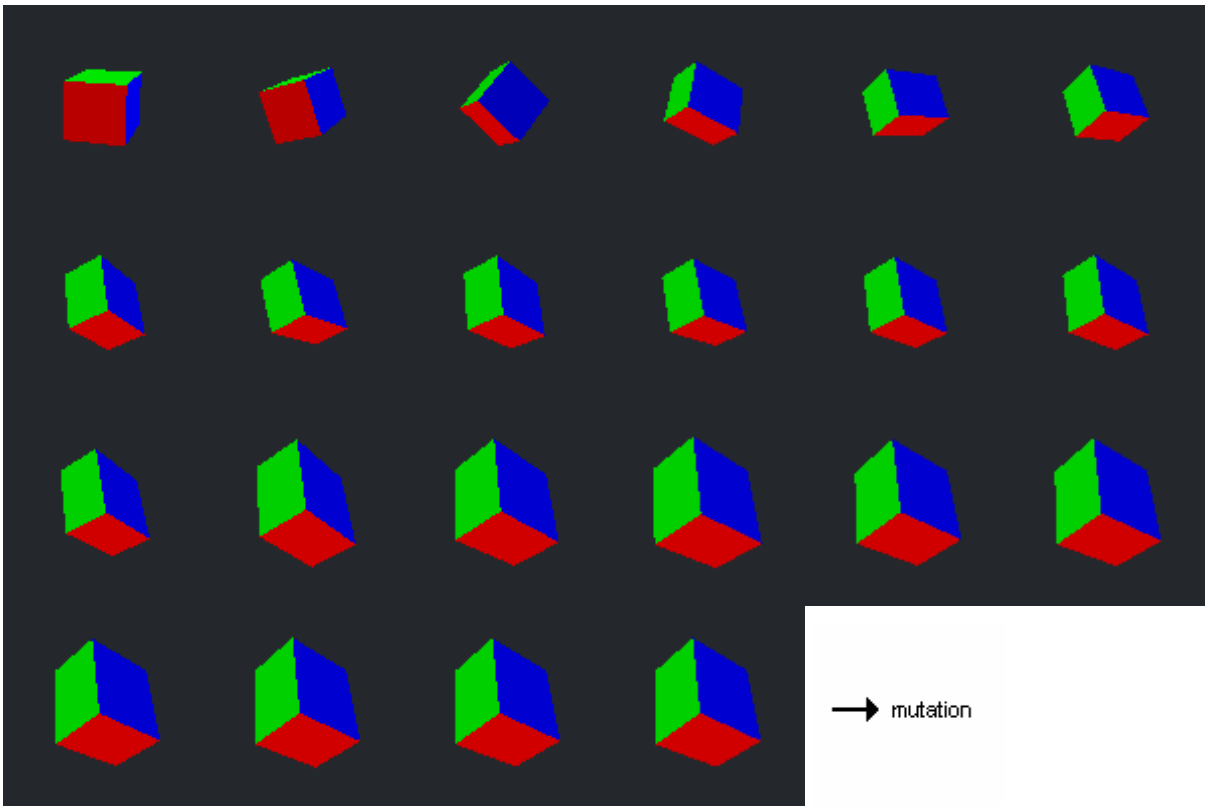
Voici l’image que nous cherchons à retrouver :

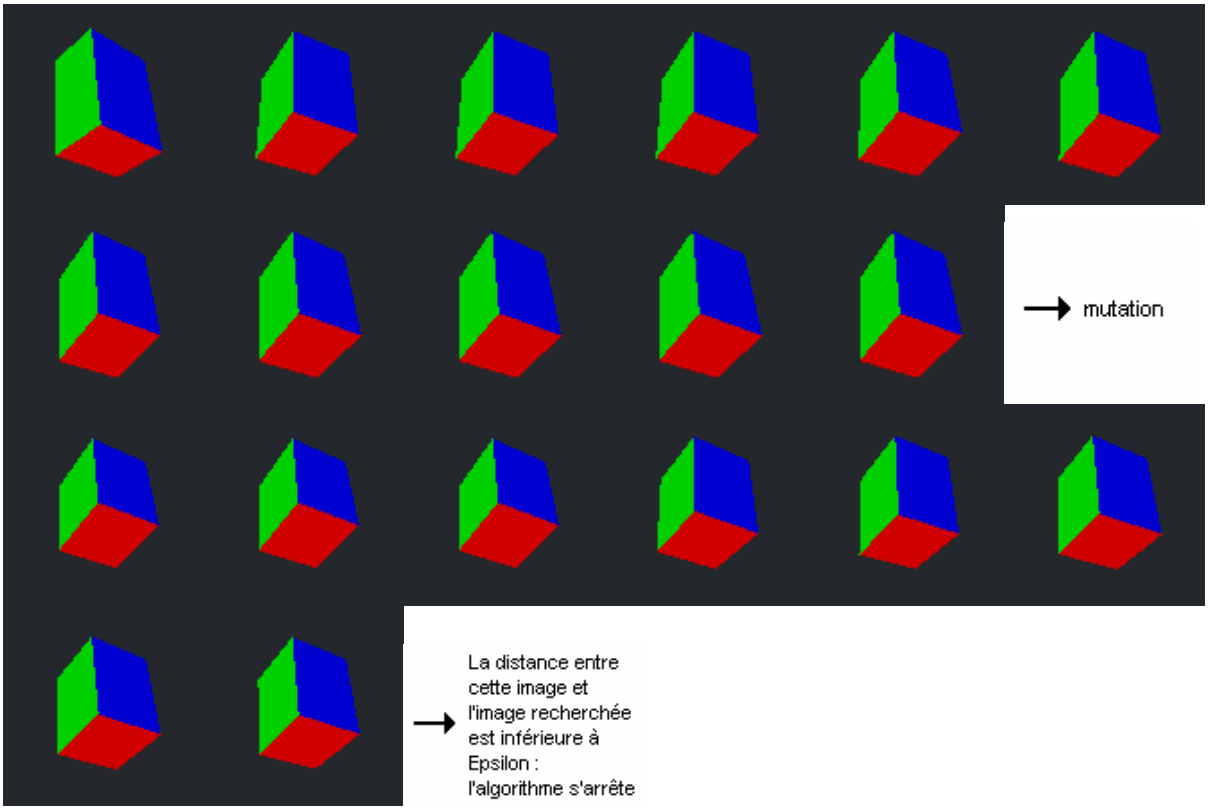


Voici l’image de départ et l’image obtenue par optimisation :



L’évolution présentée ci-dessous ne donne qu’une image générée après chaque transformation sur dix et ce pour rendre plus claire le mouvement du cube.
Voici donc l’évolution du cube jusqu’à obtention d’une image considérée comme ressemblante :





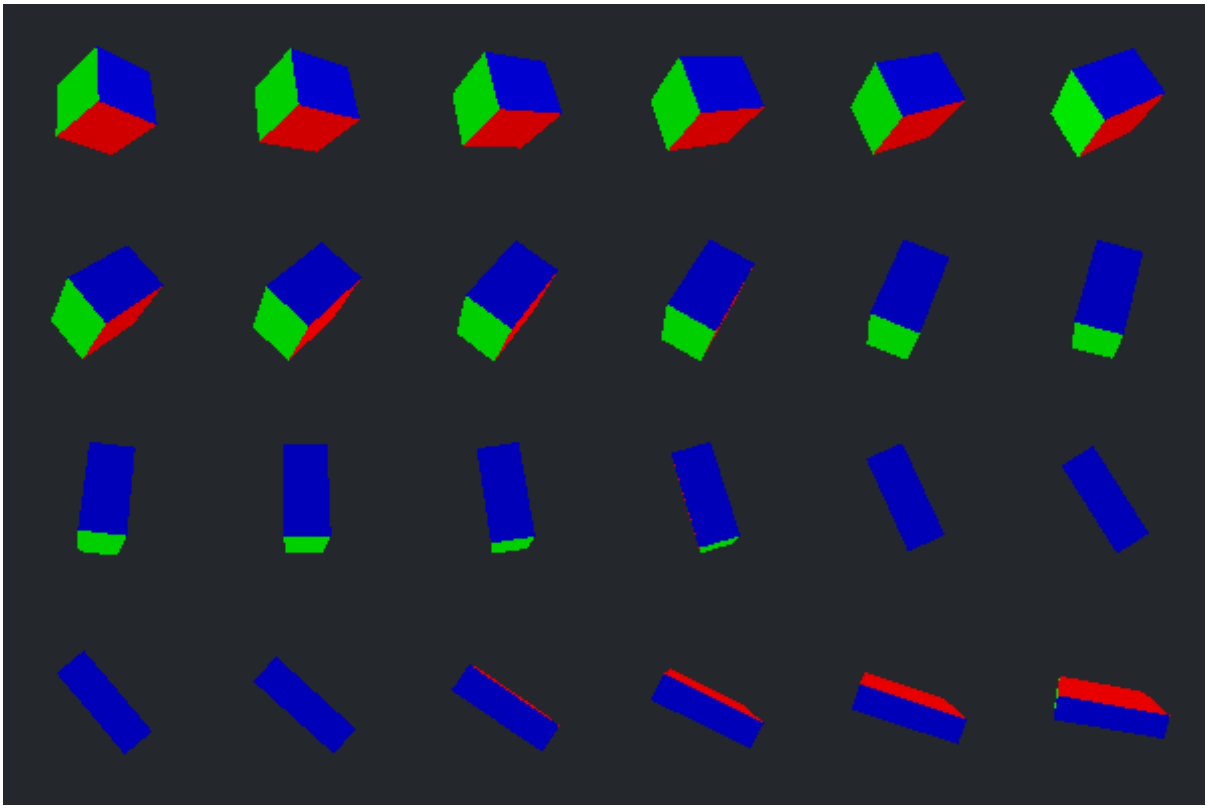
II.7. Reconstitution d'une vidéo

L'objectif final de ce TER était de pouvoir, à partir d'une vidéo composée d'images réelles d'un homme, reconstituer cette même vidéo à partir d'un modèle humain articulé obtenu à l'aide du logiciel MAYA. Malheureusement, ce modèle n'ayant pas pu être terminé à temps, nous présentons une séquence d'images que nous avons nous-même crées, utilisant le même module de dessin que celui utilisé pour reconstituer la vidéo.

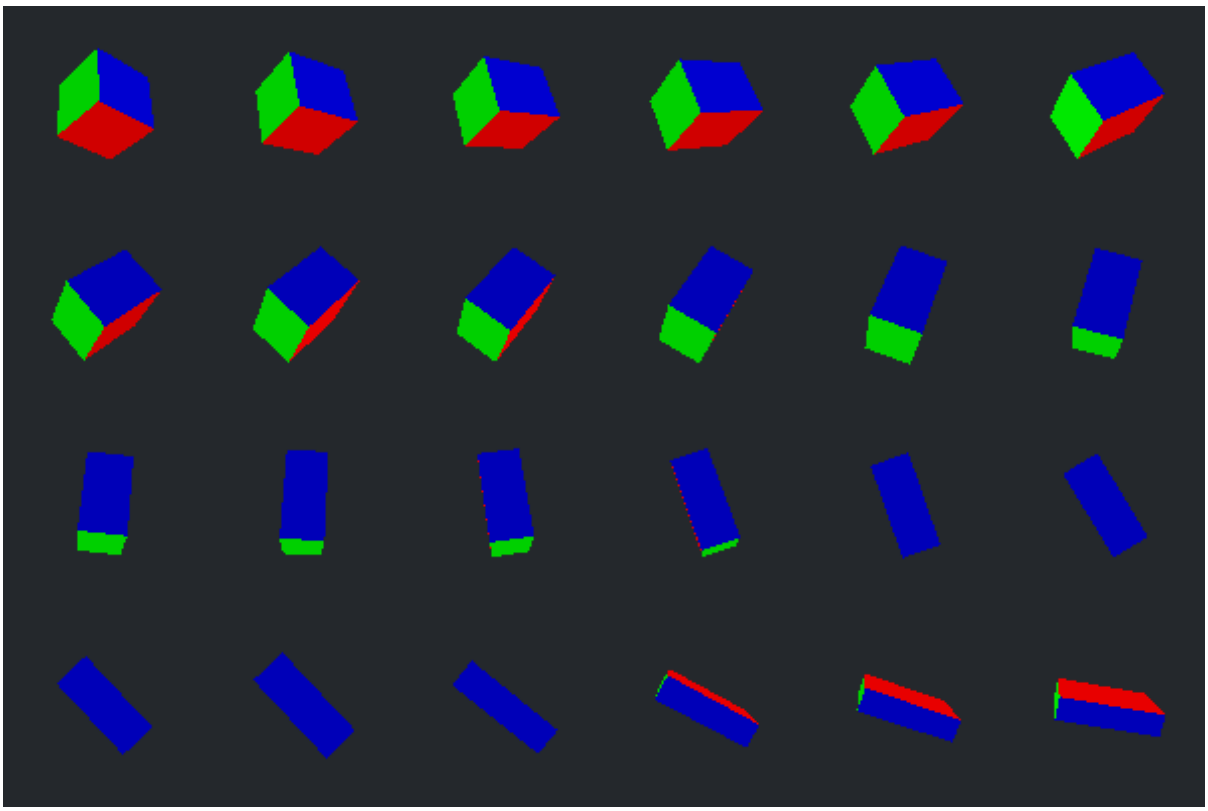
Le principe reste tout à fait le même que lorsqu'on lance le programme sur une seule image. Simplement maintenant ce dernier est lancé sur chacune des images jusqu'à obtenir pour chacune d'entre elles une distance satisfaisante. Comme l'évolution des images sur une vidéo est généralement «continue » c'est-à-dire que l'image suivante est très proche de celle que l'on traite, on ne se servira de l'image de base (celle que nous générons nous même en lui affectant les paramètres) que pour la première application du simplexe. Ensuite nous nous servons de l'image précédente qui est une bonne base puisqu'elle est censée être relativement proche de la suivante.

Comme lors du test précédent la valeur de epsilon est de 1.

Voici tout d’abord la vidéo qui nous a servie de base :



Et voici la vidéo reconstituée :



Remarque : On peut observer de petites différences, ceci est dû au choix de epsilon ("grand").

III. Améliorations possibles

De nombreuses améliorations sont envisageables pour ce programme. Nous ne les avons pas mises en place faute de temps. En voici néanmoins une liste, ainsi que les explications associées :

II.1. Différence de taille entre deux images

Notre programme traite des images de type TGA d'une taille fixe (taille_image : [constantes.h]). Comme nous l'avons dit précédemment, l'utilisation du module de forme géométriques devait à l'origine n'être que provisoire. Comme nous utilisons ce même module pour créer la vidéo de base et la vidéo reconstituée, toutes nos images sont de même format et de même taille.

Pour utiliser notre programme sur des images réelles d'un Homme en mouvement et tenter de les reconstituer avec un modèle en 3D, il faudra alors considérer deux cas possibles :
- soit il est possible de générer les images du modèle 3D dans la taille et le format voulu
- soit il faudra copier chacune des images réelles, les convertir au format qu'utilisent le modèle 3D et les redimensionner.

II.2. Utiliser la pondération des variables

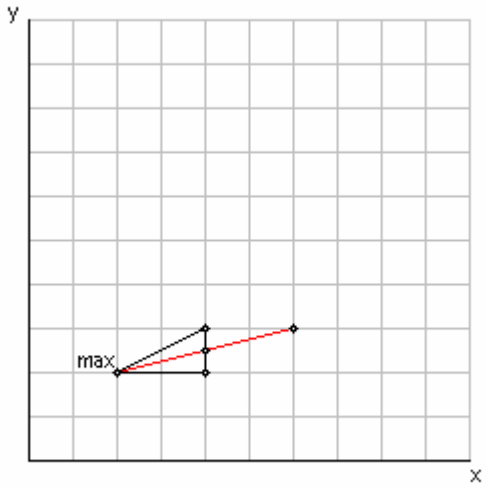
Notre programme permet d'attribuer un poids à chaque variable du vecteur de génération d'image. Elles valent toutes 1 par défaut, mais nous n'avons pas pris en compte la pondération des variables dans la suite du programme.

Si l'on prend l'exemple du parallélépipède qu'utilise notre programme, il semble que les trois variables de rotations soient un peu plus importantes que les variables concernant la taille des arêtes (en effet, selon les rotations, on verra ou non le nombre correcte de faces en couleurs).

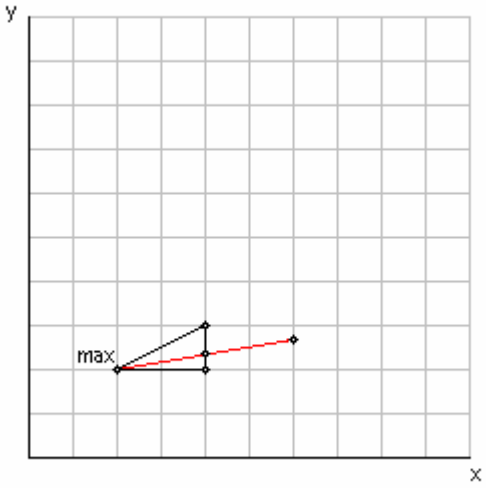
L'idée serait de permettre un plus petit mouvement durant les transformations aux variables ayant un poids plus grand. Les variables ayant un poids plus petit sont celles qui peuvent le plus varier sans engendrer trop de changements

Exemple d'une simple réflexion avec deux variables x et y :

Avec pondération(x) = pondération(y) = 1 :



Avec pondération(x) = 1 et pondération(y) = 1.5 :



Ainsi le mouvement de la variable y est moins grand que celui de la variable x, puisque le centre de gravité a une ordonnée qui se situe au 1/1.5 = 2/3 de celle du centre de gravité dans le cas où les deux pondérations sont égales.

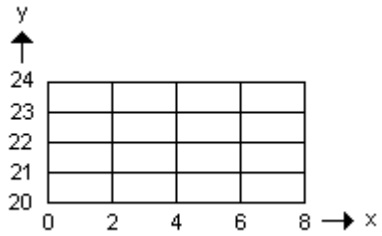
II.3. Retenir les minima locaux

Dans l'étude des cas qui nous intéressent, il y a relativement peu de minima locaux. Une idée pour « se souvenir » des ces minima, lorsque le simplexe s'y trouve coincé, serait de partager le domaine de définition des variables afin de créer un tableau à n dimensions (si n est le nombre de paramètres pour le vecteur de génération d'images).

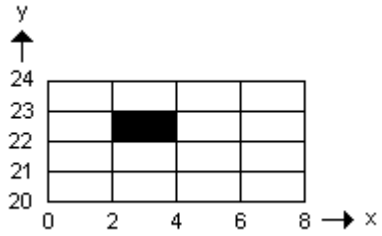
Ainsi, lorsque le simplexe déclencherait une mutation, il marquerait la case du tableau correspondant à la position du minimum, de manière à ce que les sauts aléatoires ne se fassent plus dans cette zone. Si toutes les cases du tableau venaient à être marquées, il faudrait alors recommencer en effectuant une division plus grande de domaine de chaque variable.

Exemple avec 2 variables x et y, telles que x peut varier de 0 à 8 et y de 20 à 24 (en choisissant de diviser par 4 les domaines des 2 variables) :

Au départ, on obtient le tableau à 2 dimensions suivant :



Admettons que l'algorithme déclenche une mutation alors que le simplexe s'est contracté sur lui-même au point (2.7 , 22.4), alors on « coche » la case correspondante dans le tableau :



On condamne ainsi cette zone à moins que toutes les autres cases soient également cochées.

II.4. Anticiper le prochain mouvement

En excluant les deux premières images de la vidéo, il pourrait être intéressant pour retrouver la $n^{ième}$ image de se baser sur l'évolution entre la $(n-2)^{ième}$ et la $(n-1)^{ième}$ image.

Dans notre programme actuel, pour retrouver la $n^{ième}$ image, on part de la $(n-1)^{ième}$. Or on peut imaginer que le mouvement, quel qu'il soit, est à peu près continu. D'autant plus qu'une vidéo est une succession rapide d'images, ce qui implique que la différence entre deux images ne peut être très grande.

On pourrait donc calculer le vecteur de départ en choisissant le vecteur symétrique de celui trouvé pour la $(n-2)^{ième}$ image par rapport à celui trouvé pour la $(n-1)^{ième}$ image.

Exemple (avec seulement 3 variables) :

- Vecteur trouvé pour la $(n-2)^{ième}$ image : (1 , 1 , 1)
- Vecteur trouvé pour la $(n-1)^{ième}$ image : (1.1 , 0.9 , 0.8)

- Vecteur de départ pour la $n^{ième}$ image :
  (1+2*(1.1-1) , 1+2*(0.9-1) , 1+2*(0.8-1)) = (1.2 , 0.8 , 0.6)

Dans le pire des cas, le mouvement a totalement changé de direction, or on a vu qu'entre deux images d'une vidéo, la différence ne peut pas être énorme. Il y a par conséquent peu de chance que le vecteur ainsi calculé soit beaucoup plus loin du vecteur recherché que ne l'est celui trouvé à l'image précédente. Dans la plupart des cas, le vecteur sera par contre plus près du vecteur recherché.

Conclusion

La méthode du simplexe que nous avons utilisé pour mettre en place ce programme était bien adaptée au problème posé. Il aurait été intéressant de développer une autre méthode d'optimisation afin de pouvoir comparer les performances entre les méthodes.

De plus, bien que nous n'ayons pu utiliser un modèle humain articulé pour observer le résultat de l'optimisation de la ressemblance, le modèle de formes géométriques nous a tout de même permis de nous rendre compte que le programme est efficace et le résultat satisfaisant. En effet, nous avons pu reconstituer une vidéo entière en un temps correct avec un programme qui s'adapte au nombre de variables nécessaire à l'utilisation du modèle.

Nous sommes contents d'avoir atteint les objectifs fixés par le sujet du TER, même s'il aurait été plus intéressant de générer des images de synthèse d'un humain. Nous avons beaucoup aimé travailler sur ce projet car, d'une part il s'agit du premier que l'on réalise de bout en bout, et d'autre part, le thème de celui-ci était très intéressant.

Bibliographie

[1] Thèse de Yannick PERRET
[2] Thèse de Rami KANHOUCHE
[3] www.stat.ucl.ac.be/ISpersonnel/govaerts/stat2520/transparents/optimize.pdf

Annexes

Voici la liste des fichiers que nous avons écrit :

.cpp :

- C_PIXEL.cpp
- C_SIMPLEXE.cpp
- C_VECTEUR.cpp
- distance.cpp
- generation_image.cpp (de Benoit VIGUIER -> modifié)
- generation_video.cpp
- mutation.cpp
- simplexe.cpp
- transformation.cpp

- main.cpp

.h :

- C_PIXEL.h
- C_SIMPLEXE.h
- C_VECTEUR.h
- distance.h
- generation_image.h
- generation_video.h
- mutation.h
- simplexe.h
- transformation.h

- constantes.h

Les autres fichiers proviennent soit du TER de Benoit VIGUIER, soit des fichiers de gestion d'images de Jean-Claude IEHL.

Nous avons joint les fichiers les plus importants dans les pages qui suivent.

[CONSTANTES.H]

```
#ifndef _CONSTANTES_H
#define _CONSTANTES_H

//---------------------- RELATIF A LA GENERATION DES IMAGES ----------------------------------------------------------

/* nombre de variables du vecteur pour générer des images */
#define nb_variables 6

/* largeur des images générées (en pixels) */
#define taille_largeur_image 50

/* hauteur des images générées (en pixels) */
#define taille_hauteur_image taille_largeur_image


//---------------------- RELATIF A LA DISTANCE ET A L'ALGORITHME DU SIMPLEXE ------------------------------------------

/* valeur de la distance entre deux images en dessous de laquelle on considère la ressemblance comme satisfaisante */
#define epsilon 1

/* les points du simplexe sont ecartes autour du vecteur de base de (1/inverse_ecartement_initial_simplexe) de la taille du
domaine de definition de la variable
   lors de l'initialisation */
#define inverse_ecartement_initial_simplexe 15

/* les points du simplexe sont ecartes autour du vecteur minimal de (1/inverse_ecartement_mutation_simplexe) de la taille du
domaine de definition de la variable
   lors d'une mutation */
#define inverse_ecartement_mutation_simplexe 15

/* coefficient accordé à la chrominance */
#define poids_chrominance 0.3

/* coefficient accordé à la luminance */
#define poids_luminance (1-poids_chrominance)


//---------------------- RELATIF AUX TRANSFORMATIONS DE L'ALGORITHME DU SIMPLEXE -------------------------------------

/* coefficient appliqué à l'homothétie de sorte à obtenir une réflexion */
#define controle_distance_reflexion   1

/* coefficient appliqué à l'homothétie de sorte à obtenir une expansion */
#define controle_distance_expansion  -2

/* coefficient appliqué à l'homothétie de sorte à obtenir une contraction */
#define controle_distance_contraction -0.5

/* coefficient appliqué lors du calcul  de la contraction centralisée autour du minimum */
#define controle_distance_contraction_centralisee 2


//---------------------- RELATIF A LA GENERATION DE LA VIDEO DE BASE --------------------------------------------------

/* nombre d'images par secondes de la video de base */
#define nb_images_sec 6

/* nombre de secondes de la video de base */
#define nb_secs 4

/* nombre total d'images de la video de base */
#define nb_images_video_base (nb_images_sec * nb_secs)


//---------------------- AUTRE ----------------------------------------------------

/* taille maximale des noms de fichiers */
#define taille_nom 30

/* valeur approximative de PI */
#define Pi 3.1415

//-------------------------------------------------------------------------------

#endif
```

[MAIN.CPP]

```
#include <iostream.h>
#include <stdio.h>

#include "constantes.h"
#include "simplexe.h"
#include "C_VECTEUR.h"
#include "C_SIMPLEXE.h"
#include "generation_image.h"
#include "generation_video.h"



//-------------------------------------------------------------------------------
```

```
int main()
{
	/* valeurs du vecteur a partir duquel sera generé la video de base */
	type_vecteur tab1[ nb_variables ] = { Pi/5 , Pi/5 , Pi/5 , 1 , 1 , 1 };

	/* valeurs du vecteur de depart pour la reconstitution de la video */
	type_vecteur tab2[ nb_variables ] = { Pi/5 , Pi/5 , Pi/5 , 1 , 1 , 1 };

	C_VECTEUR vect1( tab1 );
	C_VECTEUR vect2( tab2 );

	/* on fixe les valeurs minimales et maximales pour chaque paramètre (depend du modele), ainsi que le poids accordé a
	chacun */
	type_vecteur min_params [ nb_variables ] = { 0 , 0 , 0 , 0.1 , 0.1 , 0.1 };
	type_vecteur max_params [ nb_variables ] = { Pi , Pi , Pi , 2 , 2 , 2 };
	type_vecteur poids_params [ nb_variables ] = { 1 , 1 , 1 , 1 , 1 , 1 };

	/* declaration de la variable simplexe utilisee dans tout le programme */
	C_SIMPLEXE simplexe ( min_params , max_params , poids_params );

	/* tableaux contenant les valeurs de vecteurs permettant de generer les images */
	C_VECTEUR video_base [ nb_images_video_base ];
	C_VECTEUR reconstitution_video [ nb_images_video_base ];

	//---------------------------------------------

	// Generation de la video de reference
	generation_video_base ( "video_base_" , video_base , vect1 , min_params , max_params );

	//---------------------------------------------

	// Reconstitution de la video
	reconstitution ( video_base , "reconstitution_video_" , reconstitution_video , vect2 , simplexe );

	//---------------------------------------------

	return 0;
}
```

[GENERATION_VIDEO.CPP]

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

#include "C_VECTEUR.h"
#include "C_SIMPLEXE.h"
#include "constantes.h"
#include "generation_image.h"
#include "simplexe.h"

//--------------------------------------------------------------------------------
/* Cette fonction genere nb_images_video_base images (constante inclue dans constantes.h) qui, une fois concatenées, forme
une video qui sera celle de base */

void generation_video_base ( char* nom_video_base , C_VECTEUR video_base [ nb_images_video_base ] , C_VECTEUR vect ,
type_vecteur min_params[ nb_variables ] , type_vecteur max_params[ nb_variables ] )
{
	char nom[taille_nom];

	int i , j , k ;
	type_vecteur vars[nb_variables] , variation[nb_variables] ;

	video_base [0] = vect ;

	for ( k=0 ; k<(nb_variables/2) ; k++ ) variation[k] = ( ((float)((rand()%1600) - 800 )) / 10000);
	for ( k=(nb_variables/2) ; k<nb_variables ; k++ ) variation[k] = ( ((float)((rand()%800) - 400 )) / 10000);

	for ( i = 0 ; i < nb_images_video_base ; i+=nb_images_sec )
	{
		for ( j = 0 ; j < nb_images_sec ; j++ )
		{
			for ( k=0 ; k<nb_variables ; k++ )
			{
				vars[k] = vect.get_param ( k );

				vars[k] += variation[k];

				if ( vars[k] < min_params [ k ] ) vars[k] = min_params [ k ];
				else if (vars[k] > max_params [ k ] ) vars[k] = max_params [ k ];

				vect.set_param ( vars[k]  , k );
			}

			video_base [ i+j ] = vect;

			generation_image ( nom , nom_video_base , video_base [ i+j ] , (i+j) );
		}
	}
}

//--------------------------------------------------------------------------------
/* fonction qui reconstitue la video de base (generee avec la fonction au-dessus) à partir d'un vecteur dont l'image est
proche de la premiere image de la video de base */
```

```
void reconstitution ( C_VECTEUR video_base [ nb_images_video_base ] , char* nom_reconstitution_video , C_VECTEUR
reconstitution_video [ nb_images_video_base ] , C_VECTEUR vect_depart , C_SIMPLEXE simplexe )
{
	char nom_image1[taille_nom];
	char nom_image2[taille_nom];

	for ( int i=0 ; i < nb_images_video_base ; i++ )
	{
		sprintf( nom_image1 , "video_base_%4.0d.tga" , i );

		int j = 0 ;
		while ( nom_image1[j] != '\0' )
		{
			if ( nom_image1[j] == ' ' ) nom_image1[j] = '0' ;
			j++;
		}

		if ( i == 0 ) reconstitution_video [ i ] = methode_simplexe ( simplexe , nom_image1 , vect_depart );
		else reconstitution_video [ i ]=methode_simplexe ( simplexe , nom_image1 , reconstitution_video [ i-1 ] );

		generation_image ( nom_image2 , nom_reconstitution_video , reconstitution_video [ i ] , i );

		cout << "-------------------------------------------------------------------" << endl;
		cout << "Image\t" << i+1 << "/" << nb_images_video_base << "\tcree" << endl ;
		cout << "-------------------------------------------------------------------" << endl;
	}

	cout << "Fin de la reconstitution" << endl;
}
```

[SIMPLEXE.CPP]

```
#include <iostream.h>
#include <stdio.h>

#include "constantes.h"
#include "simplexe.h"
#include "C_VECTEUR.h"
#include "C_SIMPLEXE.h"
#include "distance.h"
#include "transformation.h"
#include "generation_image.h"
#include "mutation.h"

#define mutation_ecart_min_max 0.1
#define mutation_changements_min 0.01
#define declenchement_mutation 5

//---------------------------------------------------------------------------

type_vecteur calcul_decalage ( type_vecteur donnee , type_vecteur min , type_vecteur max , int inverse_ecartement )
{
	type_vecteur offset;

	offset = (max-min) / inverse_ecartement;

	if ( (donnee+offset) < max ) return (donnee+offset);
	else return (donnee-offset);
}

//---------------------------------------------------------------------------

C_SIMPLEXE ecartement ( C_SIMPLEXE simplexe , C_VECTEUR vect , int place , int inverse_ecartement )
{
	int i , j ;

	type_vecteur min , max , decalage ;

	j=0;
	for ( i=0 ; i< nb_points_simplexe ; i++ )
	{
		simplexe.set_vecteur ( i , vect );

		if ( i != place )
		{
			min = simplexe.get_min_param ( j ) ;
			max = simplexe.get_max_param ( j ) ;

			// calcul_decalage renvoie la valeur decalee, il peut s'agir d'un decalage positif ou negatif
			decalage = calcul_decalage ( vect.get_param( j ) , min , max , inverse_ecartement );

			simplexe.set_param_simplexe ( decalage , i , j ) ;

			j++;
		}
	}

	return simplexe;
}
```

```
//--------------------------------------------------------------------------------

void min_max_distance ( C_SIMPLEXE simplexe , C_VECTEUR vect , char nom_image1[] , float* ad_min , float* ad_max , float*
ad_second_max , int* ad_indice_min , int* ad_indice_max )
{
          int i ;
          float tmp;

          char nom_image_tmp[taille_nom];

          generation_image ( nom_image_tmp , "A_Image_Tmp_" , simplexe.get_vecteur ( 0 ) , 0 );

          tmp = distance ( nom_image1 , nom_image_tmp );
          (*ad_min)=tmp;
          (*ad_indice_min) = 0 ;
          (*ad_max)=tmp;
          (*ad_indice_max) = 0 ;
          (*ad_second_max) = tmp;

          for ( i=1 ; i< nb_points_simplexe  ; i++ )
          {
                    generation_image ( nom_image_tmp , "A_Image_Tmp_" , simplexe.get_vecteur ( i ) , 0 );

                    tmp = distance ( nom_image1 , nom_image_tmp );

                    if ( tmp > (*ad_max) )
                    {
                              (*ad_max) = tmp ;
                              (*ad_indice_max) = i ;
                    }
                    else
                    {
                              if ( tmp > (*ad_second_max) )
                                        (*ad_second_max) = tmp;
                    }
                    if ( tmp < (*ad_min) )
                    {
                              (*ad_min) = tmp ;
                              (*ad_indice_min) = i ;
                    }
          }
}

//--------------------------------------------------------------------------------
/* affiche le compte rendu de la dernière transformation ou mutation */

void resultats_min_max_distance ( type_vecteur min , type_vecteur max , int indice_min , int indice_max , type_vecteur
second_max )
{
                    cout<< endl << "Le vecteur le plus eloigne vaut : \t\t" << max << "\t\tet se trouve a l'indice :\t\t"<<
                    indice_max << endl ;
                    cout<<"Le 2eme vecteur le plus eloigne vaut : \t\t"<< second_max << endl ;
                    cout<<"Le vecteur le moins eloigne vaut : \t\t" << min  << "\t\tet se trouve a l'indice :\t\t"<<
                    indice_min << endl << endl ;

                    cout<<"\tEpsilon = " << epsilon << "\tMin = " << min << endl << endl;
}

//--------------------------------------------------------------------------------

C_VECTEUR methode_simplexe ( C_SIMPLEXE simplexe , char* nom_image1 , C_VECTEUR vect )
{
          int indice_max , indice_min , nb_mutations = 0 , nb_transformations = 0 ;
          float max , min , second_max ;

          // Remplissage du simplexe : il s'agit de l'initialisation
          simplexe = ecartement ( simplexe , vect , nb_variables , inverse_ecartement_initial_simplexe );

          while ( 1 )
          {
                    // on recupere la valeur du min, du max, du second max ainsi que leur indice dans le tableau du simplexe
                    min_max_distance ( simplexe , vect , nom_image1 , &min , &max , &second_max , &indice_min , &indice_max );

                    // on affiche les resultats obtenus
                    resultats_min_max_distance ( min , max , indice_min , indice_max , second_max );


                    // si le min est suffisamment petit, on renvoie le vecteur associé
                    if ( min <= epsilon )
                    {
                              cout << endl << "Nombre de transformations : " << nb_transformations << endl << "Nombre de
                              mutations : " << nb_mutations << endl <<endl;

                              if ( remove ( "A_Image_Tmp_0000.tga" ) != 0 ) cout << "Impossible de supprimer le fichier
                              temporaire" << endl ;
                              return simplexe.get_vecteur ( indice_min );
                    }
                    // sinon il faut effectuer des changements
                    else
                    {
                              // si le simplexe s'est contracté sur lui meme et ne forme plus qu'un seul point, il faut lancer
                              une mutation
                              if ( min == max && simplexe.vecteurs_tous_egaux() )
                              {
                                        simplexe = mutation ( simplexe , indice_min , indice_max );
                                        nb_mutations++;
                              }
                              // sinon il faut effectuer une transformation pour diminuer la distance

                              else
                              {
                                        if ( min == max ) cout<< "\t\t\t\t----- SIMPLEXE REGROUPE -----" << endl ;
```

```
                                        simplexe = transformation ( nom_image1 , simplexe , min , max , second_max ,
                                        indice_min , indice_max );
                                        nb_transformations++;
                                }
                        }
                }
}
```

[SIMPLEXE.CPP]

```
#include <iostream.h>
#include <math.h>

#include "constantes.h"
#include "C_PIXEL.h"
#include "tga.h"


//------------------------------------------------------------------------------
/* Obtention de la luminance a partir d'un pixel */

float luminance ( C_PIXEL pixel )
{
        float result;
        result = ( 0.299 * pixel.get_R() ) + ( 0.587 * pixel.get_V() ) + ( 0.114 * pixel.get_B() ) + ( 0 * 1 ) ;
        return result;
}

//------------------------------------------------------------------------------
/* Obtention de la composante R de la chrominance a partir d'un pixel */

float chrominance_r ( C_PIXEL pixel )
{
        float result;
        result = ( -0.1687 * pixel.get_R() ) + ( -0.3313 * pixel.get_V() ) + ( 0.5 * pixel.get_B() ) + ( 128 * 1 ) ;
        return result;
}

//------------------------------------------------------------------------------
/* Obtention de la composante B de la chrominance a partir d'un pixel */

float chrominance_b ( C_PIXEL pixel )
{
        float result;
        result = ( 0.5 * pixel.get_R() ) + ( -0.4187 * pixel.get_V() ) + ( -0.0813 * pixel.get_B() ) + ( 128 * 1 ) ;
        return result;
}

//------------------------------------------------------------------------------
/* Obtention de la distance du point de vue de la chrominance entre 2 pixels */

float distance_chrom ( C_PIXEL pixel1 , C_PIXEL pixel2 )
{
        float tmp1 , tmp2 ;

        tmp1 = (float)pow ( ( chrominance_r ( pixel1 ) - chrominance_r ( pixel2) ) , 2 );
        tmp2 = (float)pow ( ( chrominance_b ( pixel1 ) - chrominance_b ( pixel2) ) , 2 );

        return (float)sqrt( tmp1 + tmp2 );
}

//------------------------------------------------------------------------------
/* Obtention de la distance du point de vue de la luminance entre 2 pixels */

float distance_lum ( C_PIXEL pixel1 , C_PIXEL pixel2 )
{
        return (float)fabs ( luminance ( pixel1 ) - luminance ( pixel2 ) );
}

//------------------------------------------------------------------------------
/* Obtention de la distance entre 2 pixels */

float distance_pixel ( unsigned char* p1 , unsigned char* p2 )
{
        C_PIXEL pixel1 ( p1[0] , p1[1] , p1[2] ) , pixel2 ( p2[0] , p2[1] , p2[2] );

        return ( (float)poids_chrominance * distance_chrom ( pixel1 , pixel2 )) + ( poids_luminance * distance_lum ( pixel1
        , pixel2 ));
}

//------------------------------------------------------------------------------
/* Obtention de la distance entre 2 images */

float distance ( void* nom_image1 , void* nom_image2 )
{
        IMG* image1 , * image2 ;
        image1 = new IMG;
        image2 = new IMG;


        /* récuperation des données des 2 images */
        image1 = lire_tga( (char*)nom_image1 );
        image2 = lire_tga( (char*)nom_image2 );
```

```
	/* Les 2 images doivent avoir les memes dimensions */
	if ( image1->largeur != image2->largeur || image1->hauteur != image2->hauteur ||
	image1->alignement != image2->alignement )
	{
		cout<< "ERREUR - Les 2 images n'ont pas le meme format" << endl;
		return -1;
	}

	// Les 2 images ont les memes hauteurs largeurs et alignement
	int largeur_image = image1->largeur , hauteur_image = image1->hauteur ;

	float result , tmp ;

	/* pour le calcul de la moyenne des distances entre tous les pixels */
	result = ( 1 / ((float)largeur_image * (float)hauteur_image));

	/* somme de toutes les distances entre les pixels */
	int i , j ;
	tmp=0;
	for ( i = 0 ; i < hauteur_image ; i++ )
	{
		for ( j = 0 ; j < largeur_image ; j++ )
		{
			tmp += distance_pixel ( &( image1->data [ (i * 3 * largeur_image) + (3 * j) ] ) ,
			&( image2->data [ (i * 3 * largeur_image) + (3 * j) ] ) );
		}
	}

	result *= tmp ;

	//cout<<"Resultat de la fonction distance : "<< result<<endl ;
	return result ;
}
```

[TRANSFORMATION.CPP]

```
#include <iostream.h>
#include <math.h>

#include "constantes.h"
#include "C_VECTEUR.h"
#include "C_SIMPLEXE.h"
#include "distance.h"
#include "generation_image.h"

//-------------------------------------------------------------------------
/* fonction qui renvoie l'indice du minimum d'un tableau dont on connait la taille */

int minimum ( float tableau[] , int taille )
{
	float min = tableau[0] ;
	int indice_min = 0 ;

	for ( int i=1 ; i<taille ; i++ )
	{
		if ( tableau[i] < min )
		{
			min = tableau[i];
			indice_min = i;
		}
	}
	return indice_min ;
}

//-------------------------------------------------------------------------
/* fonction qui renvoie le centre de gravité du groupe de point formé par le simplexe privé du point max (d'un point de vue
distance par rapport à l'image de base) */

C_VECTEUR centre_gravite_sans_max ( C_SIMPLEXE simplexe , int indice_max )
{
	int i , j ;

	C_VECTEUR coord_g_sans_max ;

	type_vecteur tmp1;

	for ( i=0 ; i< nb_points_simplexe  ; i++ )
	{
		if ( i != indice_max )
		{
			for ( j=0 ; j< nb_variables  ; j++ )
			{
				tmp1 = coord_g_sans_max.get_param ( j ) + simplexe.get_param_simplexe ( i , j );
				coord_g_sans_max.set_param ( tmp1 , j );
			}
		}
	}

	coord_g_sans_max.mult_scalaire ( 1 / ((float) (nb_points_simplexe-1) ) );

	return coord_g_sans_max;
}
```

```
//-------------------------------------------------------------------------------
/* fonction qui effectue une homothétie du point max par rapport au centre de gravité partiel calculé par la
fonction au-dessus
Le controle correspond au coefficient de l'homothétie :
s'il vaut 1 --> on a une symétrie (réflexion)
Voir le fichier [constantes.h] pour les autres transformations
*/

C_SIMPLEXE homothetie ( C_SIMPLEXE simplexe , int indice_max , C_VECTEUR coord_g_sans_max , float controle )
{
	int i ;

	type_vecteur tmp1 , tmp2 , tmp3 ;

	for ( i=0 ; i< nb_variables  ; i++ )
	{
		tmp1 = coord_g_sans_max.get_param ( i ) ;
		tmp1 *= ( 1 + controle );

		tmp2 = simplexe.get_param_simplexe ( indice_max , i );
		tmp2 *= controle ;

		tmp3 =  tmp1 - tmp2 ;

		// Si le resultat d'une transformation fait sortir une variable de son domaine, on doit la coller à la
		bordure
		type_vecteur min = simplexe.get_min_param ( i ) , max = simplexe.get_max_param ( i ) ;
		if ( tmp3 < min ) tmp3 = min ;
		else if ( tmp3 > max ) tmp3 = max;

		simplexe.set_param_simplexe ( tmp3 , indice_max , i );
	}

	return simplexe;
}

//-------------------------------------------------------------------------------
/* fonction qui effectue une contraction centralisée autour du point min
La distance de tout point par rapport au min est divisée par controle_distance_contraction_centralisee : [constantes.h]
*/

C_SIMPLEXE contraction_centralisee ( C_SIMPLEXE simplexe , int indice_min )
{
	int i , j ;

	type_vecteur tmp1 , tmp2 ;

	for ( i=0 ; i< nb_variables  ; i++ )
	{
		for ( j=0 ; j < nb_points_simplexe  ; j++ )
		{
			if ( j != indice_min )
			{
				tmp1 = simplexe.get_param_simplexe ( indice_min , i );

				tmp2 = simplexe.get_param_simplexe ( j , i );

				tmp2 += tmp1;
				tmp2 /= controle_distance_contraction_centralisee;

				simplexe.set_param_simplexe ( tmp2 , j , i );
			}
		}
	}
	return simplexe;
}

//-------------------------------------------------------------------------------
/* Cette fonction effectue la ou les transformations adéquates selon la valeur des points du simplexe */

C_SIMPLEXE transformation ( char nom_image1[] , C_SIMPLEXE simplexe , float min , float max , float second_max , int
indice_min , int indice_max )
{
	C_VECTEUR coord_g_sans_max ;
	coord_g_sans_max = centre_gravite_sans_max ( simplexe , indice_max );

	// Operation de Reflexion
	simplexe = homothetie ( simplexe , indice_max , coord_g_sans_max , controle_distance_reflexion );

	char nom_image_tmp[taille_nom];

	generation_image ( nom_image_tmp , "A_Image_Tmp_" , simplexe.get_vecteur( indice_max ) , 0 );
	float result_ref ;
	result_ref = distance ( nom_image1 , nom_image_tmp );

	if ( result_ref < min )
	{
		// On stocke le vecteur max dans le cas pour le cas ou on annule l'expansion
		C_VECTEUR tmp;
		tmp = simplexe.get_vecteur ( indice_max ) ;

		// Operation d'Expansion
		simplexe = homothetie ( simplexe , indice_max , coord_g_sans_max , controle_distance_expansion );

		generation_image ( nom_image_tmp , "A_Image_Tmp_" , simplexe.get_vecteur ( indice_max ) , 0 );
		float result_exp ;
		result_exp = distance ( nom_image1 , nom_image_tmp );

		if ( result_exp > result_ref )
		{
			simplexe.set_vecteur ( indice_max , tmp );
			cout<< "\t\t\t\tReflexion (Expansion annulee)" << endl ;
```

```
                }
                else cout<< "\t\t\t\tReflexion\t\tExpansion" << endl ;
        }
        else
        {
                 if ( result_ref >= second_max )
                 {
                        // Operation de Contraction du Sommet Maximal
                        simplexe=homothetie (simplexe , indice_max , coord_g_sans_max , controle_distance_contraction );

                        generation_image ( nom_image_tmp , "A_Image_Tmp_" , simplexe.get_vecteur ( indice_max ) , 0 );
                        float result_contr_som_max ;
                        result_contr_som_max = distance ( nom_image1 , nom_image_tmp );

                        if ( result_contr_som_max >= result_ref )
                        {
                                // Operation de Contraction Centralisee
                                simplexe = contraction_centralisee ( simplexe , indice_min );
                                cout<< "\t\t\t\tReflexion\t\tContraction1\t\tContraction2" << endl ;

                        }
                        else cout<< "\t\t\t\tReflexion\t\tContraction1" << endl ;

                }
                else cout<< "\t\t\t\tReflexion" << endl ;
        }

        return simplexe;
}
```

[MUTATION.CPP]

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#include "constantes.h"
#include "C_SIMPLEXE.h"
#include "simplexe.h"

//---------------------------------------------------------------------------
/* attribue des valeurs aléatoires au vecteur dont l'image est la plus proche de celle recherchee */

C_SIMPLEXE saut_aleatoire ( C_SIMPLEXE simplexe , int indice_min )
{
        srand ( time (NULL) );

        type_vecteur tmp1 , tmp2 , tmp3 ;

        for ( int i=0 ; i< nb_variables ; i++ )
        {
                tmp1 = (type_vecteur)rand();

                tmp2 = simplexe.get_max_param ( i ) - simplexe.get_min_param ( i ) ;

                // pour obtenir un nombre entre 0 et tmp2 (l'operateur % ne marche qu'avec les entiers)
                tmp3 = tmp1 - ( tmp2 * (floor)( tmp1/tmp2 ) );

                tmp3 += simplexe.get_min_param ( i ) ;

                simplexe.set_param_simplexe ( tmp3 , indice_min , i );
        }

        return simplexe;
}

//---------------------------------------------------------------------------
/* effectue une mutation, c'est a dire, un ecartement autour du vecteur min, puis l'attribution d'une valeur aleatoire pour
ce vecteur */

C_SIMPLEXE mutation ( C_SIMPLEXE simplexe , int indice_min , int indice_max )
{
        cout<< "\t\t\t\t----- MUTATION -----" << endl ;

        // On écarte le simplexe
        simplexe = ecartement ( simplexe , simplexe.get_vecteur ( indice_min ) , indice_min ,
        inverse_ecartement_mutation_simplexe );

        simplexe = saut_aleatoire ( simplexe , indice_min );

        return simplexe;
}
```

[GENERATION_IMAGE.CPP

```
#include <iostream.h>
#include <stdio.h>


#include "C_VECTEUR.h"
#include "constantes.h"
```

```
/* Ce fichier provient du TER de Benoit VIGUIER à quelques modifications pres */

// ============================================================================================================

/*A inclure obligatoirement dans la fonction de generation d'image*/
#include "img.h"
#include "tga.h"
#include "Camera.h"
#include "Ray.h"
#include "Vector3D.h"
#include "Matrix3D.h"
#include "Trigo.h"
#include "SuperGeom.h"
#include "Inter.h"
#include "Light.h"
#include "Scene.h"


/*Voici un exemple d'utilisation, cette version ne contient que des lumieres
Ponctuelles, et des objets de Type "SuperGeom" ) , ce sont des cubes a 3couleurs
Il n'y a pas de reflets de lumiere, mais ca a l'avantage d'etre rapide.

Important: Orientation des axes
          Y
          ^
          \
          \
          \_____> X
        /
       /
     <
          Z
*/

// ============================================================================================================


/* Cette fonction genere l'image d'un parallélépipède selon les paramètres contenus dans le vecteur vect */
void generation_image ( char nom_image[] , char* nom_fichier , C_VECTEUR vect , int numero_image )
{
          Scene *s=new Scene();
          /*Rien a rajouter, c'est l'espace qui contient les objets...*/

          SuperGeom *supergeom;
          /*attention aux majuscules... toutes les classes commencent par une majuscule*/
          Vector3D VSuperGeom(0,0,0);

          //-------------------------------------------------

          /*Vecteur de translation*/
          Color CSuperGeom1( 0 , 0 , 250 );      /*Couleur de droite et Gauche*/
          Color CSuperGeom2( 0 , 250 , 0 );      /*Couleur Dessus Dessous*/
          Color CSuperGeom3( 250 , 0 , 0 );      /*Couleur Devant et Derriere*/
          /*Couleur en RGB*/

          supergeom = new SuperGeom( vect.get_param(5) , vect.get_param(4) , vect.get_param(3) , vect.get_param(2) ,
          vect.get_param(1) , vect.get_param(0) , VSuperGeom , &CSuperGeom1 , &CSuperGeom2 , &CSuperGeom3 );

          /*Creation reelle de l'objet,respectivement les parametres correspondent a:
                    1,1,1 => Taille suivant les axes X-Y-Z
                    PI/4,PI/4,PI/5 => Rotation (en Radian) autour des axes X-Y-Z (Attention au sens)
                    VSuperGeom => Translation
                    &CSuperGeom1,&CSuperGeom2,&CSuperGeom3 => ADRESSES des couleurs
          ATTENTION!! Les Transformations dans l'espace ne sont pas commutatives!
          l'ordre choisi est d'abord les Homothetie, les Rotations PUIS la Translation*/

          //-------------------------------------------------

          s->AddObject(supergeom);
          /*Ajout de l'objet a la Scene*/

          Light *lumiere1,*lumiere2,*lumiere3;
          /*Creation des pointeurs sur 3 lumieres*/
          Color Clumiere(250,250,250);
          /*meme couleur (blanche) pr les 3*/
          lumiere1=new Light(0,-2,2,Clumiere);
          lumiere2=new Light(1.3,1.3,2,Clumiere);
          lumiere3=new Light(-1.3,1.3,2,Clumiere);
          /*on cree les objets,
                    1,1,1 => Coordonnees  X-Y-Z
                    Clumiere => Couleur */
          s->AddLight(lumiere1);
          s->AddLight(lumiere2);
          s->AddLight(lumiere3);
          /* Ajout des lumieres*/

          Camera *cam;
          Vector3D Vcamera(0,0,2);
          /*Vecteur de translation de la camera (centre de l'ecran)*/
          cam=new Camera(2,2,4,0,0,0,Vcamera, taille_largeur_image , taille_hauteur_image );
          /*Creation
                    2,2,4 => Largeur, Hauteur, et profondeur de champ(zoom) de la Camera
                    0,0,0 => Rotations autour des axes X-Y-Z
                    Vcamera => Translation
                    500,500 => Taille de l'image en Pixel*/

          s->AddCamera(cam);
          /*Ajout de la camera a la Scene*/

          //s->Affiche();
          /*Facultatif, c'est un recapitultif*/

          IMG *img=s->Rendu();
```

```
        /*Lance le calcul de l'image, et renvoie une structure (de JC Iehl)*/

        delete s;

        sprintf( nom_image , "%s%4.0d.tga" , nom_fichier , numero_image );

        int i =0;
        while ( nom_image[i] != '\0' )
        {
                if ( nom_image[i] == ' ' ) nom_image[i] = '0' ;
                i++;
        }

        ecrire_tga( nom_image , img);

        /*Conversion en tga, avec un nom precis pour l'image*/

        free_img(img);
}
```
