RAPPORT DE PROJET
MORPHING

Sommaire

Manuel utilisateur

Pour compiler : $ make

Pour lancer le programme :

$ ./Morphing media/image-départ.jpg media/image-d'arrivée.jpg

Une fenêtre apparaît avec deux images et divers boutons :

- Ajouter points : cliquer une fois sur l'image de gauche puis une fois sur l'image de droite.

- Cacher point : n'affiche plus les points de contraintes et les lignes.

-Rendu : lancer le morphing

-<<< : divise par deux le nombre de frames.

-n frames : nombre de frames (d'images générées) pour le rendu.

->>> : multiplie par deux le nombre de frames.

-Quitter : quitte le programme et ferme la fenêtre.

A savoir sur l'ajout de points de contraintes :

L'utilisateur a très peu de contraintes pour placer ses points : il peut les placer n’importe où dans l'image de gauche puis n’importe où dans l'image de droite.

Si l'utilisateur demande des déformations étranges, le rendu peut ne pas être esthétique.

Modularisation

Modules

Point : le module point contient une structure point représentée par deux entiers : abscisse une et une ordonnée.

La structure point est aussi appelée vecteur car un vecteur est représenté par deux entiers, tout comme un point.

Point gère aussi les comparaisons de points, leur affichage et les vecteurs.

Triangle : le module triangle contient deux structures : triangle qui est représenté par 3 points.

ListeTriangle : représente une liste de triangles avec une cellule

Chaque cellule est caractérisé par un triangle, 3 voisins(un voisin est un pointeur vers une autre cellule de la liste), un suivant pour accéder à la prochaine cellule et un autre pointeur vers sa paire (son triangle équivalent sur l'image de droite).

Triangle contient toutes les fonctions basiques sur les triangles et les cellules : initialisation de la liste, des cellules, rechercher un triangle dans la liste, mise à jour des voisins, affichage etc...

Matrice : contient une structure permettant de représenter une matrice via un tableau à deux dimensions d'entiers, contient toutes les fonctions permettant de calculer le déterminant d'une matrice de taille n (découpage d'une matrice de taillen en n matrices de taille n -1, initialisation, affichage, etc...)

Frame : permet le rendu, une structure pixel est utilisée : il s'agit d'un point p pour sa position et de 4 entiers pour la couleur (r,g,b,a)

Une structure Image est aussi utilisée, il s'agit d'un tableau à deux dimensions de pixels.

Morphing – BIGUENET Denis – GOSSET Séverin

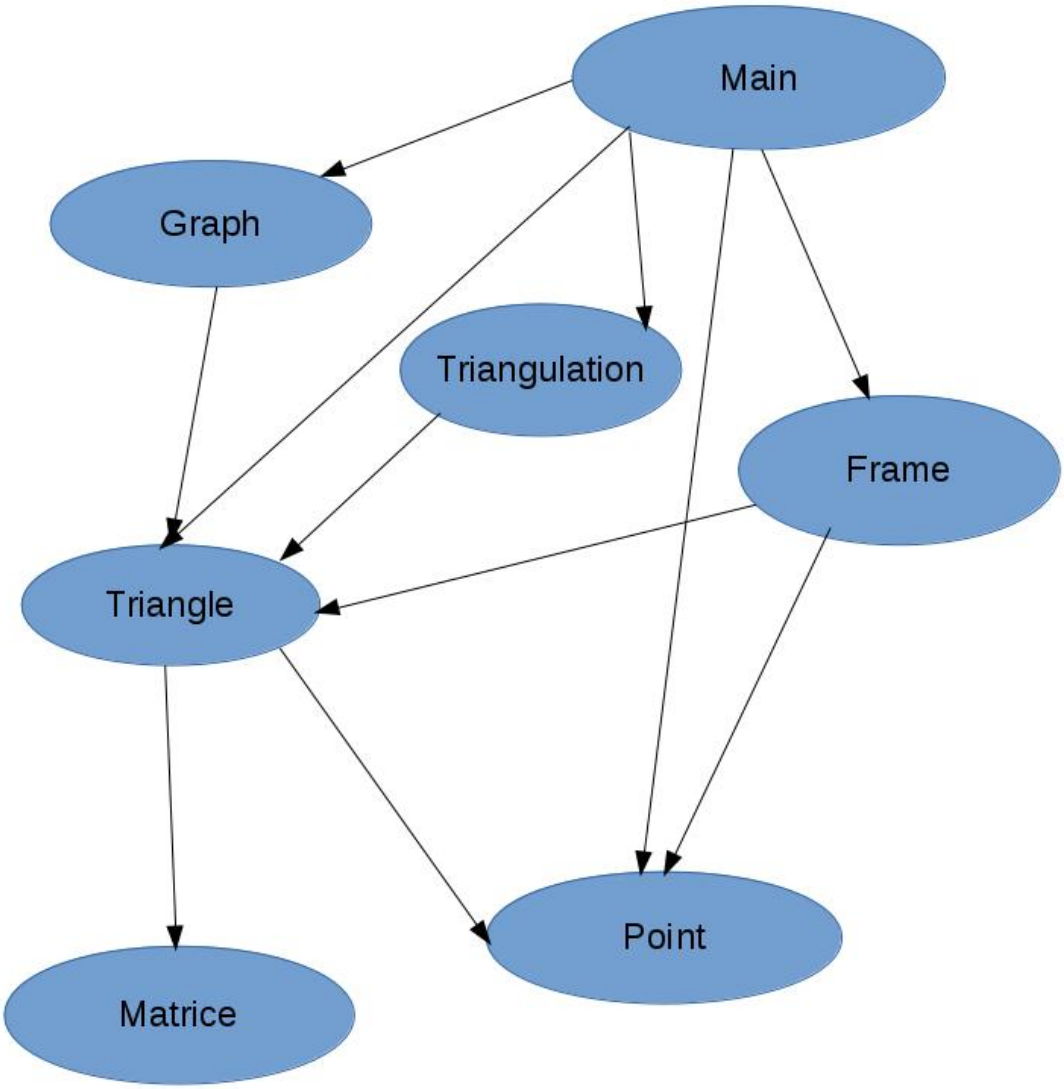
Les fonctions du module frame permettent l'affichage du rendu

(afficher une image, sauvegarder une image, retrouver un pixel dans un triangle via alpha, beta et gamma…)

Graph : module qui permet l'affichage graphique à l'aide le la bibliothèque graphique MLV. Contient toutes les fonctions qui affichent chaque élément du projet, l'interface, les images de départ, le rendu, etc. Deux structures sont utilisées ici : une structure bouton qui contient toutes les informations d'un bouton : le texte et les dimensions ; et une structure Interface est utilisée pour rassembler toutes les variables en rapport avec l'affichage : les boutons utilisés, les dimensions des images, et les images elles-même.

Triangulation : module qui rassemble les fonctions permettent la triangulation (notamment les fonctions de flip, calcul de delaunay).

Graphe d'inclusion



Fonctionnement et fonctions principales

1) Défauts du programme et choix techniques

Pour mener à bien ce projet nous avons eus plusieurs choix :

-Lorsque l'on veut vérifier si un triangle (t) est bien de delaunay il faut vérifier si les points des trois triangles adjacents sont en dehors du cercle circonscrit de t.

Pour cela il y avait deux moyens : soit parcourir la liste à chaque fois et vérifier si le triangle courant est bien voisin de t. Ou alors d'ajouter trois pointeurs vers les triangles adjacents de t. Ces deux méthodes ont chacune des avantages et des défauts : la deuxième solution permet d'éviter des parcours de liste inutiles mais d'avoir accès directement au voisin d'un triangle donné. Cependant la gestion des voisins nous a posé problème et l'ajout de pointeur augmente la mémoire nécessaire pour une cellule.

-De même dans la fonction genere_frame, quand on genere la liste de triangles déformés, on a voulu faire en sorte que l'on puisse accéder au triangle équivalent (dans l'image de gauche) d'un triangle déformé via le champ « paire », mais la fonction recherche_triangle qui permet de savoir dans quel triangle se trouve un point p ne permet pas cela : si un point p n'est pas trouvé dans un triangle t de la liste de triangles déformés, la fonction cherche dans son triangle paire, comme le triangle déformé est différent de son équivalent sur l'image de gauche, le point peut être trouvé dessus, et renvoyer un résultat faux qui provoquera une erreur de segmentation.

On a donc résolu ce problème en stockant l'équivalent d'un triangle t déformé dans le champ « voisin_AB » (les voisins n’ayant aucune importance une fois le rendu commencé).

2) fonctions principales

void genere_frame(ListeTriangle lt, Image img_gauche, int nb_frame) :

Cette fonction permet d'afficher le rendu avec un nombre nb_frame de frames générés et affichées.

On commence par calculer t : la déformation (commence à 0 et finit à 1)

Ensuite on crée une liste de triangle déformés à partir de la liste de triangles, et l'on calcule chaque déformation de triangle, on stocke dans « voisin_AB » le triangle de l'image de gauche dont est issu le triangle déformé (on peut accéder au triangle de l 'image de droite équivalent en accédant au champ « paire » de ce triangle).

Ensuite pour chaque pixel de la frame, on cherche dans quel triangle déformé est-il, on calcule alpha, beta et gamma, et on retrouve sa position dans l'image de gauche et de droite. On calcule la couleur du pixel selon t, et on met le pixel dans le tableau de pixels.

int triangle_est_delaunay(CelTriangle* t) :

Prend une cellule en argument, teste si le triangle est bien de delaunay : on calcule pour chaque voisin tv de t, le point qui est dans tv mais pas dans t.

On fait 3 calculs de déterminants pour savoir si l'un de ces points est ou non dans le cercle circonscrit de t.

On renvoie une valeur négative pour indiquer le coté du triangle t qui n'est pas delaunay.

void triangule(ListeTriangle* l, CelTriangle* cel_t, CelPoint* p) :

Principale fonction du module Triangulation.

Divise le triangle en 3 sous triangles après que l'utilisateur ait cliqué, ajoute ces triangles dans la liste et met à jour les voisins. On vérifie que chacun de ces triangles soit de delaunay.

void flip(CelTriangle * t1, CelTriangle * t2, int cote) :

Si un triangle n'est pas delaunay on flippe une arête : on garde en mémoire les voisins

des triangles t1 et t2 (pour ne pas perdre l'accès à ces triangles lorsque l'on met à jour leurs voisins , sinon on peut écraser un triangle et en perdre l'accès dans la fonction) et on modifie les coordonnées du triangle pour flipper l'arête. On met à jour les voisins grâce aux cellules gardées en mémoires.

On vérifie récursivement si les triangles modifiés sont bien delaunay.

void retrouve_pixel(CelTriangle* t, Image img_gauche, Pixel* pixel, int a, int b, int c) :

Lorsque l'on a calculé la position d'un pixel dans le triangle déformé, on obtient des valeurs alpha beta et gamma, on réutilise ces données pour retrouver la position du pixel dans les triangles équivalents (à gauche et à droite). On retrouve la position de ce pixel grâce à ces valeurs et aux coordonnées des sommets.

On stocke dans la variable pixel, la position du pixel dans le triangle équivalent

La fonction prend en argument l'image de gauche, car à chaque frame on affiche la nouvelle frame sur l'image de gauche, il faut donc prendre la couleur de l'image de gauche de base, et non pas de l'image de gauche actuelle.

long determinant(Matrice * m) :

Calcule le déterminant d'une matrice de taille n. Un déterminant de taille n n’est que n déterminants de taille n – 1, donc la fonction agit récursivement.

Difficultés et problèmes rencontrés

Nous avons rencontrés divers problèmes lors de ce projet

-La mise à jour des voisins dans le flip (on perdait l'accès quand on les mettait à jour, il fallait donc les garder en mémoire).

-Le déterminant pour savoir si un triangle est delaunay ou non, mais sur une fenêtre MLV l'ordonnée est inversée par rapport à un repère orthonormé classique : on avait donc un déterminant bon, mais une matrice fausse, ce qui empêchait le flip et la triangulation.

-Notre projet git ne fonctionnait plus, à partir de début décembre plus rien ne

fonctionnait dans les branches crées : impossible de push ni de pull, seule la branche principale (master) fonctionnait.

-

Répartition du travail

D'une manière générale on a réfléchit ensemble aux gros problèmes de ce projet (flip, triangulation, structures…), on réfléchissait d'abord chacun de notre côté puis on discutait de nos résultats.

Pour le code, Séverin a davantage fait la partie graphique et les matrices, Denis, davantage la triangulation et les frames.

Comme on publiait régulièrement notre avancée sur git il fallait se coordonner sur le travail à faire pour éviter des conflit sur git.

Compilation/Exécution

Taper « $ make » pour compiler et pour exécuter : « $ ./Morphing media/nom_de_l'image_depart.jpg media/nom_de_l'image_arrivee.jpg