# Projet de programmation C
# Morphing

L'objectif de ce projet est de développer une application graphique pouvant générer des morphings entre différentes images.

**Description macroscopique**

Le programme se lance avec pour arguments deux images, la première étant une image de départ et le second argument étant le chemin d'une autre image qui sera l'image d'arrivée. On va se restreindre à des images carrées de 512 pixels de cotés.

```
nborie@caradoc:$ ./morph img/moi.png img/gorille.png
```

Une interface graphique s'ouvre disposant les deux images devant l'utilisateur.

Les actions possibles (boutons cliquables en bas de l'interface) sont alors :

- L'ajout de points de contrainte au départ puis à l'arrivée : ce point doit arriver vers tel autre.
- Cacher/afficher les points de contraintes : pointe probablement vers un booléen activant ou pas l'affichage des triangulations au départ et à l'arrivée.
- Lancer le morphing : une fois cliqué, l'application fait la synthèse des frames puis les affiches pour visualiser la déformation, le morphing.
- Doubler ou diviser par deux le nombre de frames. Dans l'interface proposée au dessus, ce sont les boutons <<< et >>> qui assurent cette tache.
- Un bouton pour quitter le programme.

**Mise en place du morphing**

Le morphing est une déformation progressive et douce pour l'oeil humain qui prend pour départ et arrivée deux images différentes. Idéalement, les deux images montrent deux choses pas si éloignées (deux visages de personnes différentes, deux visages d'une même personne à des âges différents, etc...)

Pour mettre en place la déformation, on utilise un paramètre formel `t` qui sera une variable de type flottant qui démarre à 0 pour aller à 1 petit à petit. Il y a autant de valeurs possibles pour `t` que d'images voulues pour le morphing. Plus il y a d'images, plus c'est agréable à l'oeil mais c'est aussi plus long à calculer. Ainsi, pour 128 frames, `t` prendra les valeurs suivantes :

| *image* | 0 | 1 | 2 | 3 | 4 | ... | 64 | ... | 126 | 127 |
|---|---|---|---|---|---|---|---|---|---|---|
| `t` | 0 | $\frac{1}{128}$ | $\frac{2}{128}$ | $\frac{3}{128}$ | $\frac{4}{128}$ | ... | $\frac{64}{128}$ | ... | $\frac{126}{128}$ | $\frac{127}{128}$ |
| `float t` | 0.0 | 0.0078125 | 0.015625 | 0,0234375 | 0.03125 | ... | 0.5 | ... | 0.984375 | 0.9921875 |

Il s'agira par la suite d'utiliser chaque valeur possible de `t` pour générer une image déformée construite à partir des deux images départ et arrivée.

**Les points de contraintes**

Au début du morphing, on impose toujours 4 points de contraintes dans les coins de l'image de départ et d'arrivée. Parce que durant tout le morphing, toutes les frames de rendues graphiques doivent garder la même taille d'image (image carrée de 512 par 512 pixels). Ainsi le point de coordonnées $(0, 0)$ de l'image de départ doit se retrouver sur le point de coordonnées $(0, 0)$ dans l'image d'arrivée. Il en va de même pour les points de coordonnées $(0, 511)$, $(511, 0)$ et $(511, 511)$.

Juste avec ces quatre points, rien ne se passerait durant le morphing, on fera juste un dégradé pixel par pixel depuis la couleur $(r, g, b, a)$ au départ vers le pixel de même position dans l'image d'arrivée qui aurait la couleur $(r', g', b', a')$. Ainsi, pour chaque valeur du paramètre de déformation `t` et pour chaque pixel, on calculerait une couleur interpolée de la forme $((1-t)r + tr', (1-t)g + tg', (1-t)b + tb', (1-t)a + ta')$. Si on pousse l'expérience jusqu'au bout, on ne voit pas vraiment de morphing mais seulement une fondue au noir de l'image de départ superposée avec une fondue au blanc de l'image d'arrivée.

Ça devient marrant lorsque l'on force des éléments de l'image de départ à se diriger vers certains éléments de l'image d'arrivée, et on fait cela en définissant des points de contraintes. Il ne s'agit pas ici de faire de la reconnaissance faciale ou autre, c'est l'utilisateur qui choisira ces points en cliquant sur les images. Les points de contraintes sont donc des paires : l'un est sur l'image de départ et son correspondant doit être sur l'image d'arrivée (par exemple, le nez du zouave doit aller vers le nez du gorille petit à petit...). C'est l'utilisateur qui choisira le nombre de points de contrainte et leur placements, l'interface doit juste le forcer à construire les paires une par une.

**Détails techniques**

Une fois les deux images choisies et des paires de points placées, il faut passer à l'action. En fait, il se passe déjà des choses au fur et à mesure que l'utilisateur rajoute des couples de points : il faut recalculer à chaque ajout la triangulation.

**Triangulations de Delaunay**

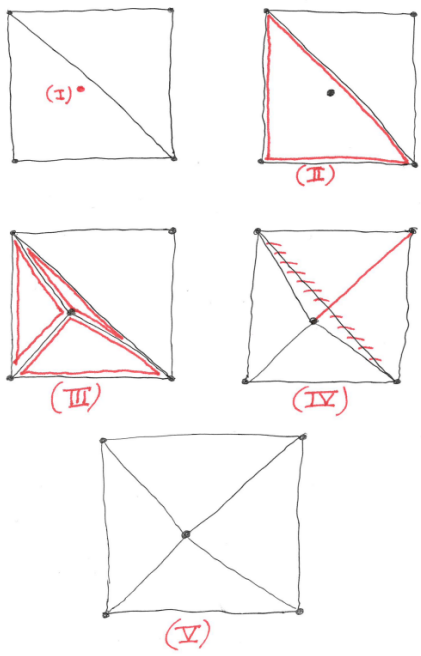
Il est vivement conseillé de consulter l'article de Wikipedia sur ce sujet.

La vidéo Youtube : https://www.youtube.com/watch?v=GUnuSYUXpwo intitulé Delaunay Triangulation donne aussi une bonne image de ce que sont les triangulations de Delaunay.

L'idée est de couvrir avec des triangles l'image de départ et l'image d'arrivée. Les triangles fonctionnent encore par paire : chaque triangle au départ doit se déformer progressivement vers le triangle à l'arrivée qui lui correspond. Toutefois, même s'il est relativement facile de construire des triangles à partir d'un nuage de points, certaines triangulations pourraient donner de mauvais morphings. Parmi toutes les triangulations, certaines sont dites de Delaunay. Ces triangulations maximisent la mesure de l'angle le plus petit. En gros, cela signifie globalement que ces triangulations évitent les triangles aplatis (les triangles aplatis donnent de mauvaises déformations). Au contraire, on a l'impression que les triangles occupent bien l'espace.

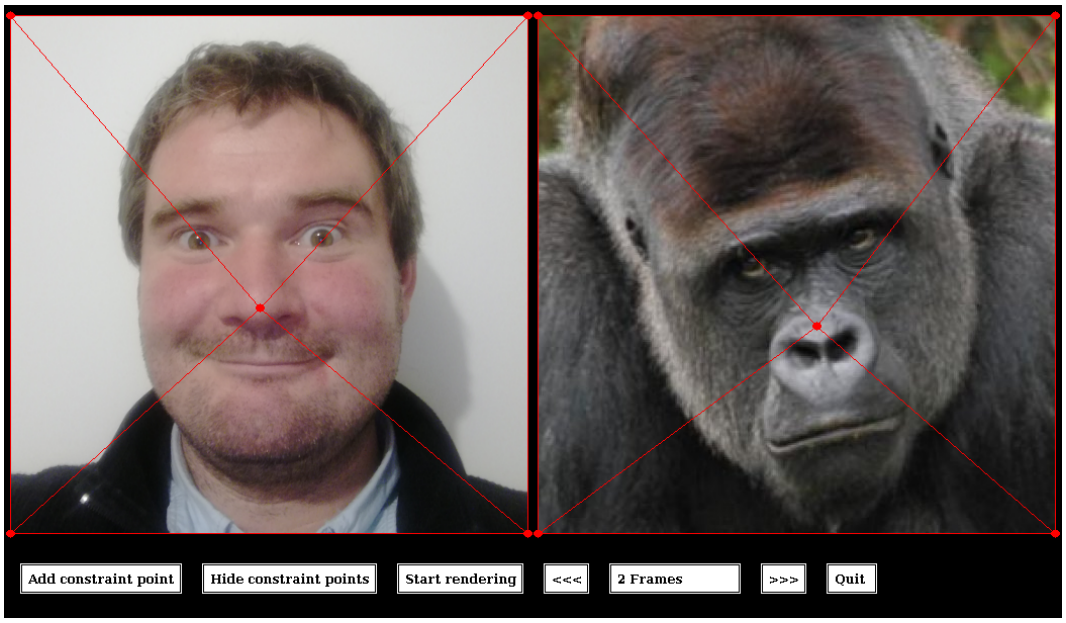
Pour qu'une triangulation soit de Delaunay, il suffit de correctement flipper des diagonales dans des quadrilatères convexes bien choisis. Voici une description de l'algorithme d'insertion.

- (I) Récupération des coordonnées d'un clic sur l'image.
- (II) Identification du triangle dans lequel se situe ce point.
- (III) Rajout de trois arêtes dans la triangulation : cela divise un ancien triangle en trois nouveaux produisant une suppression puis trois insertions dans la liste des triangles.
- (IV) Pour les nouveaux triangles et chacun de leurs voisins (ça forme des quadrilatères à chaque fois...), on regarde s'il y a besoin de flipper la diagonale.
- (V) À chaque flip d'arêtes, on vérifie récursivement si les nouveaux triangles produits ne déclenchent pas de nouveaux flips sur leurs voisins. Le traitement s'arrête quand il n'y a plus rien à flipper et des mathématiciens géomètres ont montré que cela arrive toujours fatalement.

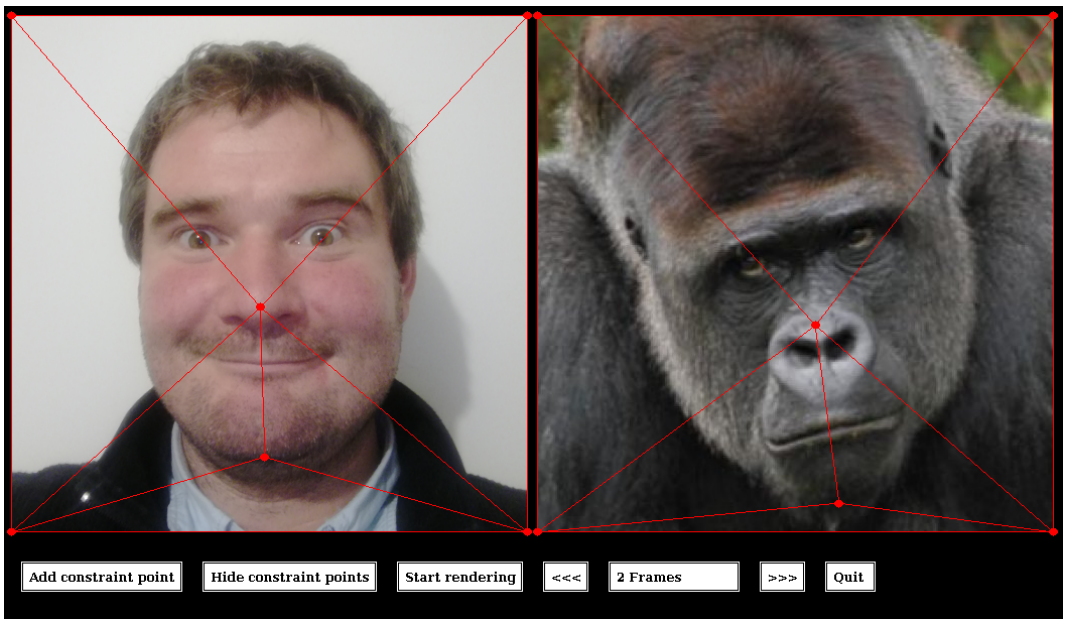


On essaiera d'avoir une triangulation de Delaunay sur l'image de départ seulement. Sur l'image d'arrivée, on n'a pas le choix, on doit garder la même liste de triangles qui doivent, comme les points, fonctionner par paire.

Voici la triangulation obtenue en rajoutant une paire de points de contrainte sur les truffes de ces deux animaux.

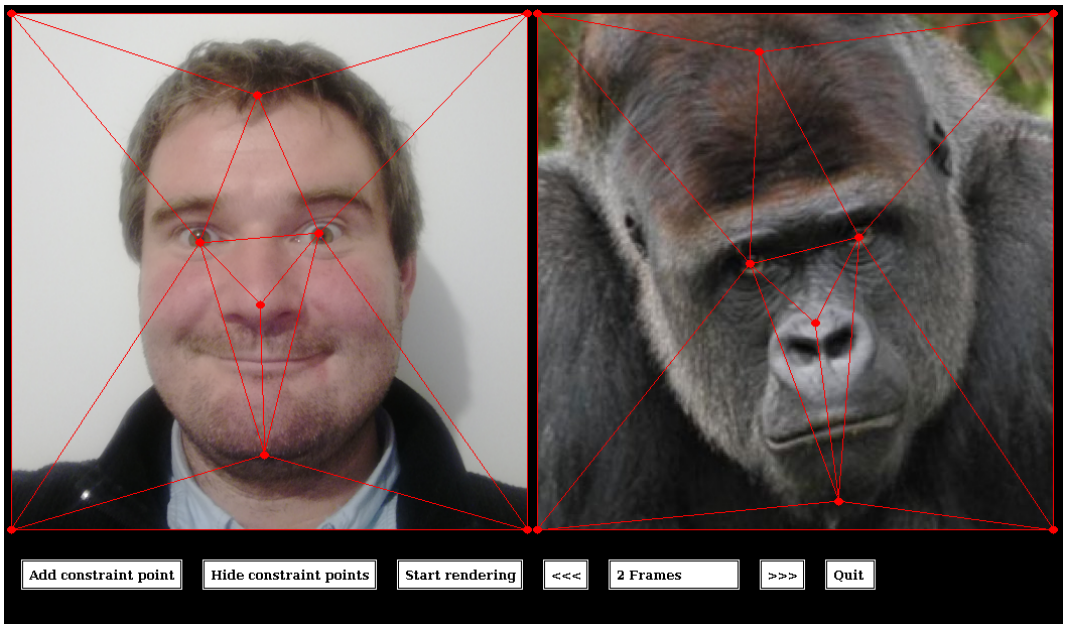



Ajouter une paire de points sur les mentons rajoute trois arêtes au départ ainsi qu'à l'arrivée mais aucun flip n'est effectué.




On remarque que la triangulation au départ est de Delaunay mais la triangulation déformée à l'arrivée ne l'est plus. On ne peut pas tout avoir...

La triangulation suivante comporte cinq paires de points de contraintes internes à l'image. De nombreux flip ont été appliqués durant sa construction.

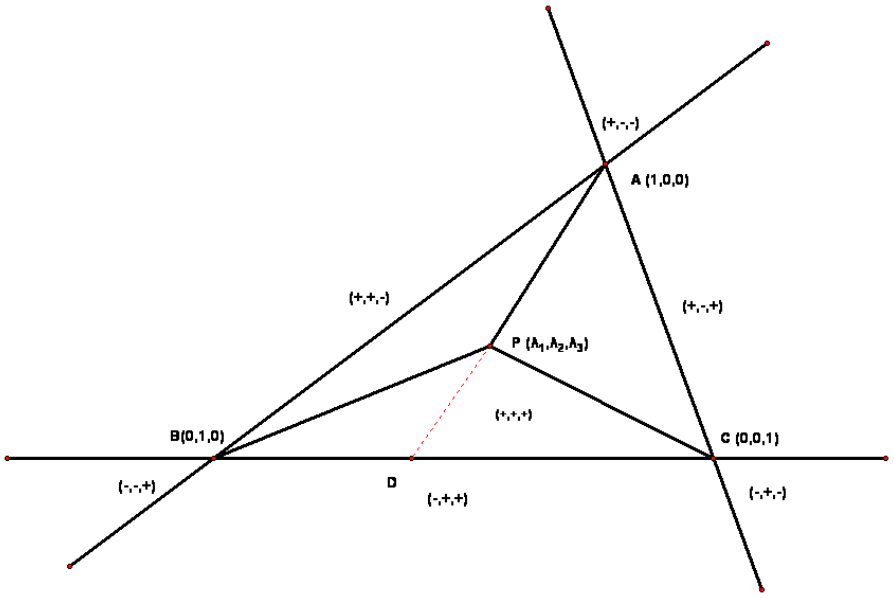



**ATTENTION :** La détermination du critère de Delaunay dans un quadrilatère nécessite le calcul d'un déterminant matriciel de taille 4. Si cela vous semble trop difficile, on peut dans un premier temps se contenter de flipper les arrêtes en sélectionnant la diagonale la plus courte dans les quadrilatères convexes (C'est pas aussi beau que Delaunay mais c'est juste deux calculs de distances Euclidiennes puis une détermination de minimum). Questionnez vos enseignants pour obtenir de l'aide (et pas à la dernière semaine !!!!!!!!).

## Coordonnées barycentriques

Il est vivement conseillé de consulter l'article de Wikipedia sur ce sujet.

Les coordonnées barycentriques permettent de décrire les points dans l'intérieur des triangles non aplatis avec un triplé de composantes positives. Ce système de coordonnés est particulièrement adapté car lorsque l'on déforme le triangle (les sommets bougent un peu), les points intérieurs gardent des coordonnées barycentriques semblables durant la déformation (Il faut imaginer un triangle en chewing-gum que l'on étirait...).




Les coordonnées barycentriques d'un point relativement à triangle ABC ne sont pas difficile

à calculer : il s'agit juste de trois déterminant de taille 2 (soit 3 produits en croix...). Toutefois, attention, les angles ont un sens dans le plan et parmi les deux triangles ABC et ACB, un seul des deux est orienté positivement. L'orientation étant un problème complexe, on peut se débrouiller pour l'éviter.

Sur le dessin (moyennement lisible), l'intérieur du triangle est la seule zone où les trois coordonnées barycentriques sont positives. Ainsi, un point est contenu dans un triangle si et seulement si les trois coordonnées barycentriques de ce point relativement à ce triangle sont positives. Cela est vrai si le triangle est bien orienté positivement (on doit donner la liste des trois sommets dans un bon ordre). Mais si on s'est trompé, alors l'orientation du triangle est négative et toutes les coordonnées barycentriques sont inversées (moins devient plus et inversement). Au final, quelque soit l'orientation : un point est dans un triangle si et seulement si ses trois coordonnées barycentriques sont de même signe (trois positives ou bien trois négatives).

On a donc un test pour détecter dans quel triangle se trouveront les clics des utilisateurs.

Pour avoir les vraies coordonnées barycentriques bien orientées (nécessaire dans le rendu graphique), lorsque les points sont intérieurs, on fera l'action suivante. Soit les trois coordonnées barycentriques $(\alpha, \beta, \gamma)$ sont toutes trois positives et c'est donc bon, soit elles sont alors forcément toutes trois négatives, on les remplace alors par leurs opposés $(-\alpha, -\beta, -\gamma)$.

**Génération des frames**

Et paf, nous voilà maintenant dans les calculs de rendus graphiques. Il va falloir maintenant générer une image 512 par 512 pour chaque frame en utilisant les deux images correctement triangulées.

Cela se fait frame par frame puis pixel par pixel... Aoutch, c'est pas gratuit le graphisme !

```
for (index_frame = 0 ; index_frame < nb_frame ; index_frame++){
  t = ((float)index) / nb_frame;
  ...
  /* Calcul ici de la triangulation déformée pour t */
  ...
  for (abs = 0 ; abs < 512 ; abs++){
    for (ord = 0 ; ord < 512 ; ord++){
      ...
      /* Calcul de la couleur du pixel (abs, ord)
         dans la frame courrante */
      ...
    }
  }
}
```

Pour toute valeur de `t`, il faudra calculer la triangulation déformée associée à la valeur de `t`. Pour cela, on ne touche pas à la liste des triangles (celle ci ne bouge que durant la construction des points de contraintes par l'utilisateur) ; on calcule juste une liste de points déformés. Si un couple de point départ $\rightarrow$ arrivée vaut $(x, y) \rightarrow (x', y')$, alors à la frame de paramètre `t`, ce point doit se trouver aux coordonnées $((1-t)x + tx', (1-t)y + ty')$. On remarque que les coins des frames ne bougent jamais... Si on fait ce calcul pour chaque paires de points, en gardant la même liste de triangles, on obtient un mélange pondéré par `t` des triangulations de départ et d'arrivée.

Une fois de calcul fait pour chaque la frame de paramètre `t`, on peut passer au pixels. À ce niveau d'avancement, on a donc en pré-requis :

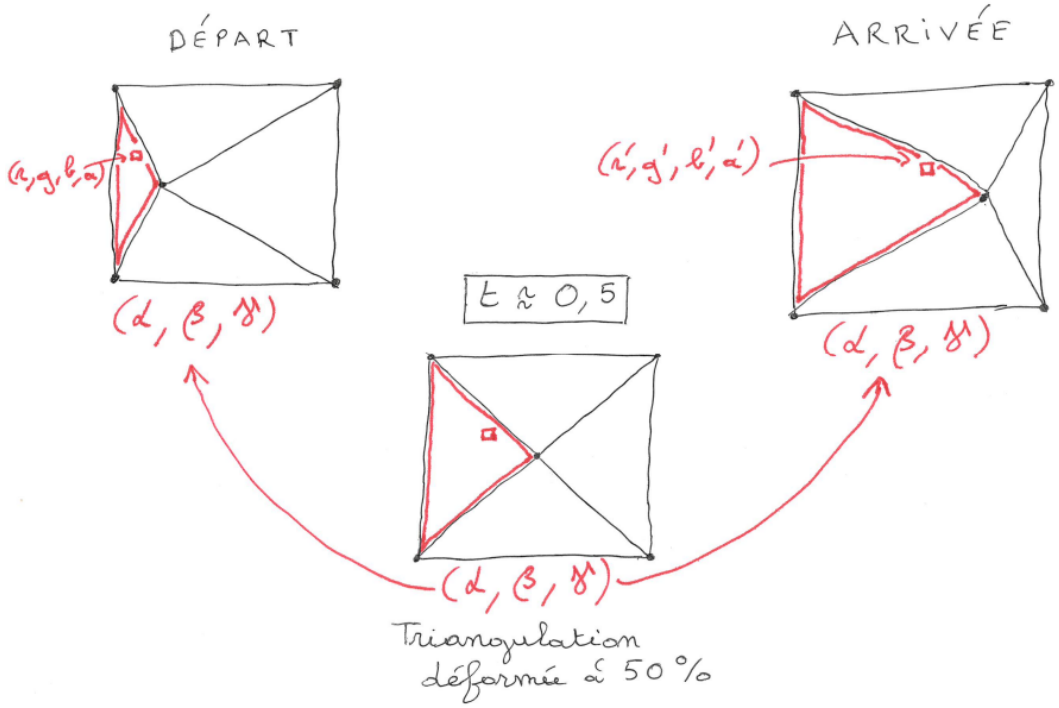
- Les images de départ et fin
- Une liste de pairs de points
- Une liste de triangles
- Une valeur spéciale de `t`
- Une triangulation déformée adaptée pour `t`

Pour chaque pixel du de la frame, il faudra alors opérer dans l'ordre :

- Trouver le triangle de la triangulation déformée dans lequel se trouve le pixel.
- Calculer les coordonnées barycentriques de ce pixel relativement au triangle trouvé.
- Utiliser LES MÊMES coordonnées barycentriques (c'est le chewing-gum) mais avec les sommets du triangle associé de l'image de départ pour trouver d'où vient ce pixel dans l'image de départ, on gardera alors en mémoire le $(r, g, b, a)$ de ce pixel.
- Utiliser LES MÊMES coordonnées barycentriques mais avec les sommets du triangle associé de l'image d'arrivée pour trouver où se dirige ce pixel dans l'image d'arrivée, on gardera alors en mémoire le $(r', g', b', a')$ de ce pixel.
- La couleur de notre pixel dans la frame `t` devra alors avoir la couleur interpolée : $((1-t)r + tr', (1-t)g + tg', (1-t)b + tb', (1-t)a + ta')$.

Attention : on travaille parfois avec des entiers et parfois avec des flottants. Il faudra faire les transtypages correctement et aux bons endroits pour ne pas voir apparaître des déformations bizarre à l'écran.

Voici un léger dessin récapitulatif de ce procédé :



**Suggestions d'amélioration**

Si vous arrivé déjà à faire tourner quelques chose qui fait un rendu en temps réel agréable, ce sera bien. Pour les plus énervés d'entre-vous, voici quelques pistes d'approfondissement :

- Gérer des images de tailles différentes.
- Générer des vidéos intéressantes de vos morphings.
- Trouver des exemples intéressants (une personne à différents âges, un homme avant/après s'être rasé, le produit minceur miracle, etc...).
- Tout ce qui vous paraîtrait croustillant... N'hésitez pas à demander conseil car les correcteurs ne sont pas toujours de votre avis.

**Modularité**

Votre projet doit organiser ses sources sémantiquement. Votre code devra être organisé en modules rassemblant les fonctionnalités traitant d'un même domaine. La partie graphique devrait être relativement importante et on peut même l'imaginer subdivisée au besoin.

Votre projet devra être compilable via un `Makefile` qui exploite la compilation séparée. Chaque module sera compilé indépendamment, et seulement à la fin, une dernière règle de compilation devra assembler votre exécutable à partir des fichiers objets en faisant appel au linker. Le flags -lMLV est normalement réservé aux modules graphiques ainsi qu'a l'assemblage de l'exécutable (sinon, c'est que votre interface graphique dégouline de partout et que vos sources sont mal modulées).

**Conditions de développement**

Le but de ce projet est moins de pondre du code que de développer le plus proprement possible. C'est pourquoi vous développerez ce projet en utilisant un système de gestion de versions git. Le serveur à disposition des étudiants de l'Université est disponible à cette adresse : `https://forge-etud.u-pem.fr/`. Comme nous attendons de vous que vous le fassiez sérieusement, votre rendu devra contenir un dump du fichier de logs des opérations effectuées sur votre projet, extrait depuis le serveur de gestion de versions ; afin que nous puissions nous assurer que vous avez bien développé par petites touches successives et propres (commits bien commentés), et non pas avec un seul commit du résultat la veille du rendu. L'évaluation tient compte de la capacité du groupe à se diviser équitablement le travail.

**Remarques importantes :**

- L'intégralité de votre application doit être développée exclusivement en langage C (la documentation technique dynamique fait évidemment exception). Toute utilisation de code dans un autre langage (y compris C++) vaudra 0 pour l'intégralité du projet concerné.
- En dehors des bibliothèques standards du langage C et de la libMLV, il est interdit d'utiliser du code externe : vous devrez tout coder vous-même.
  Toute utilisation de code non développé par vous-même vaudra 0 pour l'intégralité du projet concerné.

- Tout code commun à plusieurs projets vaudra 0 pour l'intégralité des projets concernés.

**Conditions de rendu :**

Vous travaillerez en binôme et vous lirez avec attention la Charte des Projets. Il faudra rendre au final une archive `tar.gz` de tout votre projet (tout le contenu de votre projet `git`), les sources de votre application et ses moyens de compilation. Vous devrez aussi donner des droits d'accès à votre chargé de TD et de cours à votre projet via l'interface `redmine`.

Un exécutable devra alors être produit à partir de vos sources à l'aide d'un `Makefile`. Naturellement, toutes les options que vous proposerez (ne serait-ce que –help) devront être gérées avec `getopt` et getopt_long.

La cible `clean` doit fonctionner correctement. Les sources doivent être propres, dans la langue de votre choix, et commentées. C'est bien de se mettre un peu à l'anglais si possible.

Votre archive devra aussi contenir :

- Un fichier `log_dev` correspondant au dump des logs de votre projet (nom des commits, qui? et quand?), extrait depuis le serveur de gestion de versions que vous aurez utilisé.
- Un fichier `makefile` contenant les règles de compilation pour votre application ainsi que tout autre petit bout de code nécessitant compilation (comme les tests par exemple).
- Un dossier `doc` contenant la documentation technique de votre projet ainsi qu'un fichier `rapport.pdf` contenant votre rapport qui devra décrire votre travail. Si votre projet ne fonctionne pas complètement, vous devrez en décrire les bugs.
- Un dossier `src` contenant les sources de votre application.
- Un dossier `include` contenant tous les headers de vos différents modules.
- Un dossier `bin` contenant à la fois les fichiers objets générés par la compilation séparée.
- Aucun fichier polluant du type `bla.c~` ou `.%smurf.h%` généré par les éditeurs. Votre dossier doit être propre !
- Si possible, la partie html générée par l'utilitaire `doxygen` à partir de votre application de visualisation. Ceci est optionnel mais tellement plus propre. Et encore plus propre, votre makefile appelle doxygen pour générer la documentation, ainsi votre rendu sera plus petit en mémoire.

Sachant que de nombreux vilains robots vont analyser et corriger votre rendu avant l'oeil humain, le non respect des précédentes règles peuvent rapidement avorter la correction de votre projet.

Mon père m'a toujours dit : "J'ai l'impression que tu descends du singe plus vite que les autres toi !"