Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Documentation libgfl

version 02/2025

installation ☞ `install-libgfl.7a-25.pdf`

Ce document concerne principalement la version 2D *de la* `ligfl`. *Cette version masque presque complètement les fonctionalités* `OpenGL/freeglut`. *Elle est donc utilisable sans aucun pré-requis sur cette librairie.*

En revanche, la version 3D *ne propose que très peu de fonctionalités spécifiques pour le dessin. il s'agit juste d'une légère sur-couche facilitant la gestion des fenêtres, interactions, caméra, lumière... Elle s'adresse donc à des utilisateurs connaissant déjà* `OpenGl`.

Principes généraux et structure des codes

L'architecture de fonctionnement repose sur un découpage imposé des tâches semblable à une architecture "Modèle-Vue-Contrôleur". De nombreuses fonctionnalités, habituellement à la charge du programmeur, sont prédéfinies ici, en particulier pour tout ce qui touche à la gestion des fenêtres et des options de contrôles.

2D ou 3D ?

C'est la configuration du `Makefile` qui précise cela via les définitions des `PFLAGS` et `LDFLAGS` :

- 2D : `$(incGFL2) $(libGFL2)`
- 3D : `$(incGFL3) $(libGFL3)`

Ce choix active/inhibe certaines fonctionalités spécifiques et pointe vers la bonne version de la lib (`ligf2d.so` ou `ligf3d.so`)

Structure générale des codes

Un programme utilisant la `libgfl` sera structuré sur la base de quelques fonctions essentielles présentées dans l'exemple `{2|3}Ddemo/src/gfl{2|3}_squelette.c` et détaillées ci-dessous.

gfl_squelette.c

```
01|  #include <gfl.h>
02|
03|  /* dimension (entières) de la fenêtre (pixels) */
04|  #define WWIDTH  720
05|  #define WHEIGHT 640
06|
07|  /* les 6 fonctions essentielles */
08|  static void init(void) { /* la fonction d'initialisation          */ }
09|  static void ctrl(void) { /* la fonction de contrôle               */ }
10|  static void evts(void) { /* la fonction de gestion des événements */ }
11|  static void draw(void) { /* la fonction de dessin                 */ }
12|  static void anim(void) { /* la fonction d'animation               */ }
13|  static void quit(void) { /* la fonction de sortie                 */ }
14|
15|  int main(int argc, char *argv)
16|  { /* initialisation          */
17|    gfl_InitWindow(*argv,WWIDTH,WHEIGHT);
18|    gfl_SetWindowCoord(-5.,-5.,+5.,+5.);      /* spécifique 2D */
18|    //gfl_SetDrawZone(-5.,-5.,10.,0.);        /* alternative (1) */
18|    //gfl_SetCenteredDrawZone(0.,0.,10.,0.);  /* alternative (2) */
19|    /* les 6 handlers           */
20|    gfl_SetInitFunction(init);  /* fonction d'initialisation */
21|    gfl_SetCtrlFunction(ctrl);  /* fonction de contrôle      */
22|    gfl_SetEvtsFunction(evts);  /* fonction d'événements     */
23|    gfl_SetDrawFunction(draw);  /* fonction de dessin        */
24|    gfl_SetAnimFunction(anim);  /* fonction d'animation      */
25|    gfl_SetExitFunction(quit);  /* fonction de sortie        */
26|
27|    return gfl_MainStart();     /* la boucle d'éxécution     */
29|  }
```

Convention de nommage

- Les **modules** : de la forme `gfl_nom.[h,c]` - à l'exception du *header* principal `<gfl.h>`
- Les **types** : de la forme `GFLnom` - à l'exception des (re-)définitions du module `<gfl_types.h>` qui crée/renomme quelques types standards (`bool` , `uchar` , `uint` ....)
- Les **macros** : de la forme `GFLNOM` - à l'exception de quelques macros simples et classiques, définies dans `<gfl_types.h>` (`PI, ZERO, SQR(), MIN(), MAX()` ....)
- Les **fonctions** : de la forme `gfl_NomFonction()` - pas d'exception à cette règle

Les dépendances

`01| #include <gfl.h>` : seule dépendance essentielle dans la plupart des cas.

Cet appel contient déjà la plupart des `include` standards (`stdio.h` , `stdlib.h` , `math.h` ...)

La fenêtre graphique

`03| WWIDTH` `04| WHEIGHT` → `17| gfl_InitWindow(*argv,WWIDTH,WHEIGHT)`

Création de la fenêtre, avec ses dimensions initiales (en pixels), positionnée au coin supérieur gauche de l'écran (coord. `(0,0)`).

Dimensions et position de la fenêtre sont ajustables en cours d'exécution et un mode 'plein-écran' est disponible (`<Ctrl+'f'>` ou `<F11>`). Les dimensions courantes sont accessibles à tout moment via des fonctions `get` : `gfl_GetPixWidth(); gfl_GetPixHeight();`

→ C'est la 1ère fonction à appeler et elle est **indispensable** pour initialiser le système graphique.

cas particulier : pour les applications de type «traitement d'image», une autre fonction d'initialisation peut être utilisée : `GFLpixmap* gfl_InitImage(char *name, char **path, char **root, char **ext);`
☞ cf. exemple `work/src/gfl_image.c`

La zone de travail réelle

La fenêtre graphique en tant que 'matrice de pixels' n'est pas directement accessible à l'utilisateur (sauf manip. particulière, hors sujet ici)

La zone de travail associée est en réalité un espace réel infini dans lequel on peut naviguer.

En dimension 3

En `3D`, la zone d'espace visible dans la fenêtre est définie par la caméra et ses paramètres optiques, comme ce que l'on voit dans la visée d'un appareil photo. La `libgfl3d` disposant d'une caméra prédéfinie, ces paramètres ne seront pas plus développés ici. Précisons juste qu'ils renvoient directement à la fonction `gluLookAt` de freeglut.

En dimension 2

En `2D` il faut initialiser le positionnement et les dimensions de la fenêtre graphique sur cet espace. Cela se fait en associant des coordonnées réelles aux coins inf. gauche (x_{min}, y_{min}) et sup. droit (x_{max}, y_{max}).

☞ définition de la zone de travail (réelle – type `double`) associée à la fenêtre graphique :

`18| gfl_SetWindowCoord(xmin,ymin,xmax,ymax);`

attention : ces valeurs initiales doivent être compatibles avec les tailles `WWIDTH` et `WHEIGHT` pour assurer un rapport d'aspect cohérent (un cercle apparaît bien comme un cercle, et non une ellipse) :
`(xmax-xmin)/(ymax-ymin) = WWIDTH/WHEIGHT` .

☞ définitions alternatives : une solution souvent plus naturelle d'initialiser la zone réelle consiste à positionner le coin inférieur gauche (`xmin,ymin`) ou le centre de la fenêtre (`xctr,yctr`) et les dimensions (`width,height`) de cette zone :

```
18| gfl_SetDrawZone(xmin,ymin,width,height);
18| gfl_SetCenteredDrawZone(xctr,yctr,width,height);
```

Si l'un des deux paramètres (`width` ou `height`) est $\leq 0.$, la dimension correspondante est calculée par rapport à l'autre pour assurer un rapport d'aspect cohérent.

Dans l'exemple, les 3 versions de la ligne `18|` sont donc équivalentes.

Ces trois fonctions sont spécifiques à la version `2D` et n'ont pas d'équivalent en `3D`.

Si la zone réelle n'est pas initialisée, le système utilisera la zone réelle par défaut définie par les tailles de la fenêtre graphique : `(0.,0.,WWIDTH,WHEIGHT);`

Les coordonnées de la zone réelle s'adaptent automatiquement lors d'un redimensionnement de la fenêtre graphique ou lors d'un zoom, d'un *panscan*, en conservant le rapport d'apsect initial.

Les dimensions réelles courantes sont accessibles à tout moment via des fonctions d'accès dédiées :

`gfl_GetXMin(); gfl_GetXMax(); gfl_GetYMin(); gfl_GetYMax();`.

☞ Cela permet de contrôler/ajuster les comportements des programmes en cours d'éxécution.

note : on peut aussi connaître la taille 'réelle' d'un pixel écran : `gfl_Get{X|Y}PixSize();`

☞ Ca peut permettre de limiter la précision des calculs/dessins en coordonnées réelles (inutile d'aller au dela de la précision d'un pixel écran) ou d'ajuster certains tracés en fonction du rapport d'échelle (pixel/réel).

Background

La couleur de fond (`BackGround`) de l'écran (`2D|3D`) est limitée à une valeur de gris $\in [0,1]$) réglable et accessible via les fonctions `gfl_SetBkGdCol` et `gfl_GetBkGdCol`.

La commande prédéfinie `<Ctrl+'w'>` inverse ce niveau de gris. Le fond par défaut est blanc (1.)

Navigation & séquences prédéfinies

Les boutons/clics souris ainis qu'un certain nombre de séquences de touches prédéfinissent des fonctionalités de l'interface graphique :

- le `zoom` avant/arrière : associé à la molette (*wheel*) de la souris et aux touches `'+'/'-'` du clavier.
- le `panscan` (déplacements x/y) associé au clic-milieu de la souris
- la touche `'='` remet tout ça aux conditions initiales définies au lancement.
- les touches `<Ctrl+'f'>` ou `<F11>` basculent en mode plein écran
- les touches `'?'` ou `<F12>` affichent une aide qui résume les principales commandes prédéfinies. Les options de contrôle crées par l'utilisateur viendront s'ajouter automatiquement (ou presque) à cette liste (cf. `<gfl_REFERENCE.c>`)
- la touche `<ESPACE>` lance / stoppe l'animation (si une fonction d'animation est définie).
- la touche `<Ctrl+'r'>` affiche le repère général (Ω, x, y, z)
- les touches `<Ctrl+'p'>/<Ctrl+'j'>` font un *screenshot* de la fenêtre de dessin au format `png/jpg`
- le clic-droit de la souris donne un petit menu de fonctionalités (capture d'écran, vidéo, sortie)
- les touches `<ESC>` ou `<Ctrl+'q'>` quitte l'application

La structuration du reste du code, c'est-à-dire, l'interface utilisateur `libgfl`, passe par la définition de 6 fonctions spécifiques, de format imposé, chacune avec un rôle très précis.

A chacune de ces fonctions correspond un *handler* spécifique `gfl_Set***Function(void (*f)(void));`.

Cette structure, parfois un peu contraignante, à l'avantage de rendre les codes très facile à maintenir et à faire évoluer. Cela permet également de passer assez simplement de la `libgfl` à une autre lib. graphique (telle que la `libMLV`).

L'architecture qui en découle peut être assimilée à ce que l'on peut trouver dans un jeu vidéo, avec ses différents "moteurs" :

① `init();` → chargement

`ctrl();` → interface de dialogue

② Boucle de jeu

ⓐ `evts();` → capture des événements (clavier, souris, joystick....)

ⓑ → interprétation des événements (interactions)

ⓒ `anim();` → calculs divers ("moteurs physiques" ....)

ⓓ `draw();` → affichage ("moteur graphique")

③ `quit();` → sortie

les 6 fonctions spécifiques et leurs *handlers*

Le protoype imposé `void f(void)`, sans paramètre ni valeur de retour, implique que les différentes fonctions devront communiquer via des variables globales. Bien que ça ne fasse pas partie du *"manuel des bonnes pratiques en C"*, il n'y pas vraiment d'alternative, puisque les sous-couches (`Gl>X11`) fonctionnent ainsi.

1. procédure d'initialisation : `08| void init(void) {...}` → `20| gfl_SetInitFunction(init);`

C'est l'étape chargement des données. Appelée une seule fois, avant le lancement de la boucle principale, cette fonction crée et initialise les données globales.

☞ C'est ici[(1)] que doivent se faire les éventuelles allocations mémoire, chargement de fichier...

Cette fonction ne doit contenir aucun appel à des routines d'affichage (ils ne seraient pas pris en compte).

Elle peut ne pas être définie → `20| gfl_SetInitFunction(NULL);`

2. procédure de contrôle : `09| void ctrl(void) {...}` → `21| gfl_SetCtrlFunction(ctrl);`

Définit les outils de contrôle et d'interaction (création des `bouttons`, `scrollbars`...) .

Elle est appelée une seule fois, juste après la fonction d'initialisation et avant le lancement de la boucle principale.

Tout ce qu'il y a ici pourrait être directement écrit dans la fonction `init()`, mais c'est plus 'propre' et plus pratique de séparer.

Cette fonction ne doit contenir aucun appel à des routines d'affichage (ils ne seraient pas pris en compte)

Si elle n'est pas définie → `21| gfl_SetCtrlFunction(NULL);`

[(1)] et ici **uniquement**

3. procédure d'événements : `10| void evts(void) {...}` → `22| gfl_SetEvtsFunction(evts);`

Réception et traitement des interruptions clavier, souris.... Cette fonction est appelée, à chaque cycle, juste avant la fonction d'affichage `draw()`. Tout ce qu'il y a ici pourrait d'ailleurs être directement intégré à la fonction d'affichage mais c'est plus 'propre' et plus pratique de séparer.

Si elle n'est pas définie → `22| gfl_SetEvtsFunction(NULL);`

4. procédure d'affichage principal : `11| void draw(void) {...}` → `23| gfl_SetDrawFunction(draw);`

C'est la seule fonction (parmi les 6) qui soit **indispensable**. Sans elle, rien ne s'affichera.

Elle est appelée par la boucle principale qui gère également le rafraîchissement, les `zoom/panscan` et la synchro avec les fonctions de contrôle (`ctrl`), capture d'événement (`evts`) et d'animation (`anim`).

Etant appelée en **boucle infinie**, elle ne doit pas contenir d'appel de gestion de mémoire et ne doit pas modifier les données globales. Si celles-ci doivent être modifiées dynamiquement, c'est par l'une des 2 autres fonctions `evts` (clavier/souris) ou `anim` (changement d'état).

5. procédure d'animation : `12| void anim(void) {...}` → `24| gfl_SetAnimFunction(anim);`

Réalise les calculs de **changement d'état** des données. Comme son nom l'indique, son usage est réservé aux applications produisant une animation, c'est à dire une séquence d'images (produites par la fonction d'affichage draw) entre lesquelles les "données" (points, vecteurs, couleurs...) changent.

C'est elle qui pilote cette modification des données.

Elle ne doit contenir aucun appel graphique ni aucun appel de gestion de mémoire. Elle est appellée en boucle, en synchronisation, entre la fonction de gestion des événements `evts` et la fonction d'affichage `draw`.

Si elle n'est pas définie → `24| gfl_SetAnimFunction(NULL);`

6. procédure de sortie : `13| void quit(void) {...}` → `25| gfl_SetExitFunction(quit);`

A priori lorsque l'application se termine (sortie de la boucle infinie) elle ne revient pas dans le programme source (c'est `X11` ou `freeglut` qui envoie le signal d'arrêt `exit()`).

Les tâches à effectuer en fin de programme (affichage terminal, libération de mémoire, sauvegarde sur fichier....) sont à placer dans cette fonction `quit()` qui sera passée à `atexit()` à l'arrêt du programme (cf. `$>man atexit`).

Si il n'y a rien à faire en sortie → `25| gfl_SetExitFunction(NULL);`

la boucle d'exécution : `27| return gfl_MainStart();`

L'appel à cette fonction démarre tout le processus : branchements, exécution et synchronisation des fonctions spécialisées et surtout lancement de la boucle infinie (`<ESC>` ou `<Ctrl+'q'>` pour quitter proprement).

Cette boucle met en séquence les 3 étapes que sont, dans l'ordre, la récupération des événements (`evts`) les calculs d'évolution des données (`anim`) et l'affichage (`draw`).

Au final, la fonction principale `int main(int argc, char* argv[ ]);` ne changera (presque) jamais, mis à part pour mettre à `NULL` un des *handlers* (ou supprimer la ligne d'appel) lorsqu'une des fonctions d'interface est inutile (jamais `draw` , rarement `init` , souvent `anim`).

Rapide survol des modules

La documentation complète et précise de la `libgfl` est encore en cours (`doxygen`).

A ce stade, elle se résume essentiellement aux commentaires présents dans les fichiers d'en-tête `<gfl_*.h>`
☞ il faut aller les consulter directement.

Les basiques

- `<gfl_types.h>` : contient des (re-)définition (raccourcis) de types standards (par exemple le type `uchar` pour `unsigned char`, le type énuméré `bool` ...) et la définition de quelques constantes (`PI, RAC2...`) et macro (`SQR, MIN, MAX ...`) classiques.

  On trouve également ici la constante `ZERO` et la macro `GFLISZERO` (ou `IS_ZERO(x)`), qui servent à remplacer les tests d'égalité sur les réels. Par exemple, avec 2 réels simples `double a=1., b=0.1;`, le test `if (10.*b==a)` donne pour résultat `false` à cause de l'imprécision des réels au format IEEE.

  On utilisera plutôt le test `if (IS_ZERO(10*b-a)) ....`

  De manière générale, les tests d'égalité sur les réels sont à proscrire.

- `<gfl_geom{2|3}d.h>` : contient les définitions des types `GFLpoint` et `GFLvector`, les deux entités élémentaires de la géométrie, ainsi que de nombreuses fonctions associées à ces deux types : constructeurs, opérateurs scalaires et vectoriels, fonctionnalités de normes et distances ...

  Les 2 types sont identiques, mais géométriquement les deux objets sont bien différents. Il est donc utile de pouvoir les distinguer.

  Ce même type possède un 3° nom (`GFLcoord`) qui peut être utilisé lorsque l'objet peut prendre les deux formes (point et/ou vecteur).

  ```
  typedef struct { double x,y(,z); } GFLcoord, GFLpoint, GFLvector;
  ```

  Ce module contient également de nombreux constructeurs et opérateurs sur ces objets (produits scalaires, vectoriels, normes et normalisation .....) ☞ la plupart des outils essentiels y sont !

  attention : il existe deux modules différents (2D et 3D). Les noms des types sont identiques, de même que de nombreuses fonctions, mais bien sûr leurs implémentations diffèrent.

- `<gfl_colors.h>` : contient les types, constantes et fonctions associées à la gestion des couleurs. Les couleurs sont définies par défaut en mode `RGBA` (structure de 4 `float`, dans l'intervalle `[0,1]`, pour les composantes `Rouge`, `Vert`, `Bleu` et le canal de 'transparence' `Alpha`)

  ```
  typedef struct { float r,g,b,a; } GFLcolor;
  ```

  attention : contrairement à de nombreux systèmes (`OpenGl, SDL` ...), le champ `a` (composante `alpha`) représente bien la **transparence** et non l'opacité : la couleur est pleinement visible pour `a=0.` et disparaît pour `a=1.`

  Ce module propose également quelques fonctions de conversion / représentation des couleurs en mode `HSVA (Hue, Saturation, Value, Alpha)` ainsi que des fonctions de création de cartes de couleurs de type "arc-en-ciel" (dégradé de teintes)

  Enfin il propose de nombreuses couleurs prédéfinies sous forme de macros, telles que `GFLr`, `GFLo`, `GFLy`, `GFLg` .... pour les couleurs rouge, orange, jaune, vert....

  compatibilité `libMLV` / traitement d'image :

  - la `libMLV` gère les couleurs en mode `RGBA-32bits` via 4 `unsigned char` (ou 1 `unsigned int`).
  - il en va de même de l'immense majorité des méthodes de traitement d'image.

  ☞ le type `GFLcolor;` a un 'alias' `GFLfcolor;`, pour bien indiquer qu'il s'agit de réels
  ☞ le module `<gfl_colors.h>` propose également un type `typedef uint GFLucolor;` définissant les couleurs en `0xRRGGBBAA`, ainsi que quelques fonctionalités de conversion `GFLfcolor`↔`GFLucolor`.

- `<gfl_tools.h>` : contient quelques outils périphériques comme des générateurs aléatoires (dans un intervalle réel, ou centré sur une valeur) et quelques outils de mesure du temps (`GFLclock`).

Les "moteurs"

- `<gfl_control.h>` : contient l'ensemble des objets et fonctionnalités de contrôle associées à l'interface (`button` , `switch` , `popup` , `scrollbar`), les fonctionnalités de gestion des interruptions (`keyboard` , `mouse`) ainsi que la gestion des points de contrôle (`GFLctrlpt` : point "cliquables", déplaçable en "drag & drop").

- `<gfl_window.h>` : contient les fonctions de gestion de la fenêtre `gfl_InitWindow(), gfl_SetWindowCoord()` , les 6 *handlers* de fonctions spécifiques `gfl_Set***Function(void (*f)(void))` et la fonction de démarrage `gfl_MainStart()`

  On y trouve également les fonctions `gfl_Set*` et `gfl_Get*` d'attribution/récupération de nombreuses variables d'environnement, des fonctions d'affichage formaté de chaînes de caratères, des fonctions de tracé d'axes et grilles de graduation, ainsi que diverses fonctions utilitaires, plus ou moins marginales

Les "haut-niveau"

- `<gfl_draw2d.h>` (spécifique `2D`) : contient les fonctions de dessin de quelques primitives simples telles que points, droites, cercle & ellipses, triangles, rectangles, quadrilatères quelconques...

- `<gfl_geoalgo2d.h>` (spécifique `2D`) : ce module, beaucoup plus évolué, contient des fonctions réalisant des algorithmes simples mais d'usage assez courant sur les primitives géométriques (intersction de segments, de cercles)

- `<gfl_polygon2d.h>` (spécifique `2D`) : un module de gestion de polygône quelconque à base de listes doublement chaînées circulaires.

- `<gfl_transfo{2|3}.h>` : contiennent les fonctions de transformations géométriques en coordonnées homogènes 2D et 3D (translation, homothétie, rotation), le type `GFLhmat` (matrice 3x3 ou 4x4) associé et les opérateurs matriciels simples (produits `Matrice×Point` , `Matrice× Vecteur` et `Matrice×Matrice`)

Les périphériques

fonctionnement non garanti sous système autre que `linux`

- `<gfl_pixmap.h>` : un module permettant de manipuler des images au format brut `PNM` [(2)].

  Ce module (lecture/écriture d'images) utilise la boîte à outils `Netpbm` (spécifique `linux`). Supporte les formats `pnm | bmp | gif | jpeg | png | tiff | raster | eps` .

- `<gfl_colsyst.h>` : plus marginal, ce module contient des fonctions de conversion entre différents systèmes colorimétriques (`RGB|HSV|YIQ|YCbCr`), utiles essentiellement pour la manipulation d'images en couleurs.

- `<gfl_capture.h>` : module pour les captures d'écran, les vidéo. Ces éléments sont tous activés automatiquement, il n'y a donc aucune raison de l'utiliser.

  Ce module utilise la suite `Netpbm` pour l'image et `mencoder` (et donc `ffmpeg`) pour la vidéo (formats disponibles `mp4 | mkv | flv | avi | mpg`)

[(2)] `PNM: Portable aNy Map`