LVGL: Light and Versatile Graphics Library

Table of Contents

Introduction

LVGL (**L**ight and **V**ersatile **G**raphics **L**ibrary) est une bibliothèque graphique open source fournissant tout ce dont on a besoin pour créer une interface graphique intégrée avec des éléments graphiques faciles à utiliser, de beaux effets visuels et une faible empreinte mémoire. **LVGL** est une bibliothèque graphique ciblant les microcontrôleurs aux ressources limitées. Cependant, il est possible de l'utiliser pour créer des interfaces graphiques intégrées avec des microprocesseurs et des cartes haut de gamme exécutant le système d'exploitation Linux. Il existe actuellement deux manières de procéder :

- simulateur PC avec bibliothèque multiplateforme SDL 2
- en utilisant simplement le périphérique de tampon de trame de Linux (généralement /dev/fb0).

Dans cet article Wiki, on utilisera un exemple de simulateur de PC avec SDL2 et on le modifiera légèrement pour afficher l'interface utilisateur en plein écran plutôt que dans une fenêtre.

Caractéristiques

- Blocs de construction puissants : boutons, graphiques, listes, curseurs, images, etc.
- Moteur graphique avancé : animations, anti-aliasing, opacité, défilement fluide, modes de fusion, etc.
- Prend en charge divers périphériques d'entrée : écran tactile, souris, clavier, encodeur, boutons, etc.
- Prend en charge plusieurs écrans
- Indépendant du matériel, peut être utilisé avec n'importe quel microcontrôleur et écran
- Évolutif pour fonctionner avec peu de mémoire (64 Ko Flash, 16 Ko RAM)

- Prise en charge multilingue avec gestion UTF-8, prise en charge des scripts CJK, bidirectionnels et arabes
- Éléments graphiques entièrement personnalisables via des styles de type CSS
- Des mises en page puissantes inspirées des CSS : Flexbox et Grid
- Le système d'exploitation, la mémoire externe et le GPU sont pris en charge mais non requis. (prise en charge intégrée de STM32 DMA2D et NXP PXP et VGLite)
- Rendu fluide même avec un seul frame buffer
- Écrit en C et compatible avec C++
- Micropython Binding expose l'API LVGL dans Micropython
- Simulateur pour développer sur PC sans matériel embarqué
- Plus de 100 exemples simples
- Documentation et références API en ligne et en PDF

Préparer l'environnement de développement

Sur reTerminal

Sur Rasperry Pi OS, on peut facilement installer SDL2 à l'aide d'un terminal : `sudo apt-get update && sudo apt-get install -y build-essential libsdl2-dev`

Cloner ensuite le projet de simulateur et les sous-modules associés :

```
git clone --clone git récursif https://github.com/littlevgl/pc_simulator.git
```

Sur PC hôte

EdgeLine est un éditeur WYSIWYG pour LVGL, qui permet aux utilisateurs de créer une interface, puis d'exporter du code C/Micropython pour une utilisation sur l'appareil cible. Il est actuellement en phase bêta avec des fonctionnalités limitées et disponible pour Windows et Linux.

Pour la version Linux, rendre **Edgeline.x86_64** et **server/micropython** exécutables. (`chmod +x nom de fichier`)

Après cela, **EdgeLine** peut être exécuté par `./Edgeline.x86_64`

Le code exporté ne charge aucun des écrans par défaut, il faut donc appeler `lv_scr_load(scr_name)` manuellement sur l'écran souhaité.

Exécuter des démos

Les étapes suivantes peuvent être utilisées avec **CMake** sur un système d'exploitation Raspberry Pi.

CMake doit être installé (la commande cmake fonctionne sur le terminal).

```
cd pc_simulator/
```

```
mkdir build
cd build.
cmake ..
make -j4
```

Le binaire sera dans `../bin/main`, et peut être exécuté en tapant cette commande : `DISPLAY=:0 ./../bin/main`

Cela afficherait la démo du widget dans un mode fenêtré - pour le changer en plein écran, ouvrir `pc_simulator/lv_drivers/display/monitor.c` et changer #L344 en

```
static void window_create(monitor_t * m)
{
    m->window = SDL_CreateWindow("TFT Simulator",
                                 SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED,
MONITOR_HOR_RES * MONITOR_ZOOM, MONITOR_VER_RES * MONITOR_ZOOM,
SDL_WINDOW_FULLSCREEN); /*dernier param. SDL_WINDOW_BORDERLESS pour masquer
les bordures*/
```

De plus, modifier la résolution de l'écran dans `pc_simulator/lv_drv_conf.h` #L90

```
/*-------------------
 *  Monitor of PC
 *-------------------*/
#ifndef USE_MONITOR
#  define USE_MONITOR         1
#endif


#if USE_MONITOR
#  define MONITOR_HOR_RES     1280
#  define MONITOR_VER_RES     720
```

Exécuter à nouveau la commande make et exécuter le binaire pour voir l'application de démonstration en plein écran !

Méthode de création d'une application

https://github.com/AIWintermuteAI/Seeed_reTerminal_LVGL_UI_Demo fournit un exemple de code. La méthode pour créer de nouvelles applications est similaire à ce qui se passe à l'intérieur de la fonction (assistant_create()). :

- Initialiser les widgets sur le(s) écran(s)
- Créer un rappel basé sur une minuterie ou sur un événement pour obtenir les données des capteurs/système
- Modifier le contenu des widgets en fonction des données - cela se fait normalement à l'aide de

variables globales déclarées au haut du code

Dans **assistant_create** on créer un objet panneau pour l'onglet et on définit sa hauteur.

```
    lv_obj_t * panel1 = lv_obj_create(parent);
    lv_obj_set_height(panel1, lv_pct(100));
```

Ensuite, on créer un objet bouton image à partir du tableau C situé dans le dossier assets, obtenu avec l'outil de conversion d'image LVGL. On initialise et affecte également la transformation de style de pression de bouton à l'objet bouton image (le bouton devient vert à la pression). De plus, un rappel d'événement **speech_event_cb** est affecté à la pression d'un bouton - puisqu'il ne s'agit que d'un exemple fictif, qui n'imprimera qu'un texte dans le terminal. Mais dans une application réelle, il peut être utilisé pour démarrer **Intelligent Assistant**.

```
    LV_IMG_DECLARE(speech_btn_img);

    /*Créer une animation de transition lors de la transformation de la
largeur et recolorer.*/
    static lv_style_prop_t tr_prop[] = {LV_STYLE_IMG_RECOLOR_OPA, 0};
    static lv_style_transition_dsc_t tr;
    lv_style_transition_dsc_init(&tr, tr_prop, lv_anim_path_linear, 500, 0,
NULL);

    static lv_style_t style_def;
    lv_style_init(&style_def);
    lv_style_set_text_color(&style_def, lv_color_white());
    lv_style_set_transition(&style_def, &tr);

    /*Assombrir le bouton lorsqu'on appuie dessus et l'agrandir*/
    static lv_style_t style_pr;
    lv_style_init(&style_pr);
    lv_style_set_img_recolor_opa(&style_pr, LV_OPA_70);
    lv_style_set_img_recolor(&style_pr, lv_palette_main(LV_PALETTE_GREEN));

    /*Créer un bouton image*/
    lv_obj_t * speech_btn = lv_imgbtn_create(panel1);
    lv_imgbtn_set_src(speech_btn, LV_IMGBTN_STATE_RELEASED, NULL,
&speech_btn_img, NULL);
    //lv_img_set_zoom(speech_btn, 128);
    lv_obj_set_size(speech_btn, 300, 300);
    lv_obj_add_event_cb(speech_btn, speech_event_cb, LV_EVENT_ALL, NULL);
    lv_obj_add_style(speech_btn, &style_def, 0);
    lv_obj_add_style(speech_btn, &style_pr, LV_STATE_PRESSED);
```

Dans le bloc de code suivant, on créer des étiquettes de texte pour l'heure, la date, le message d'accueil de l'utilisateur. Ceux-ci sont initialisés avec le texte par défaut, qui sera modifié dans le rappel **time_timer** chaque seconde.

```
```

```
    lv_obj_t * name = lv_label_create(panel1);
    lv_label_set_text(name, "Hi there, Username");
    lv_obj_add_style(name, &style_large, 0);

    clock_label = lv_label_create(panel1);
    lv_obj_add_style(clock_label, &style_clock, 0);
    lv_label_set_text(clock_label, timeString);
    lv_label_set_long_mode(clock_label, LV_LABEL_LONG_WRAP);

    lv_obj_t * time_icn = lv_label_create(panel1);
    lv_obj_add_style(time_icn, &style_large, 0);
    lv_label_set_text(time_icn, LV_SYMBOL_BELL);

    date_label = lv_label_create(panel1);
    lv_label_set_text(date_label, dateString);
    lv_obj_add_style(date_label, &style_large, 0);
```

Enfin, on structure les widgets qu'on a placé dans cet onglet en utilisant la disposition **Grid**. La disposition **Grid** est un sous-ensemble de **CSS Flexbox**.

Il peut organiser des éléments dans une “table” 2D comportant des lignes ou des colonnes (pistes). L'élément peut s'étendre sur plusieurs colonnes ou lignes. La taille de la piste peut être définie en pixel, au plus grand élément (`LV_GRID_CONTENT`) ou en “Free unit” (`FR`) pour répartir l'espace libre proportionnellement.

Pour faire d'un objet un conteneur de grille, appeler `lv_obj_set_layout(obj, LV_LAYOUT_GRID)`.

La fonctionnalité de disposition de grille de LVGL doit être globalement activée avec `LV_USE_GRID` dans `lv_conf.h`.

```
    static lv_coord_t grid_main_col_dsc[] = {LV_GRID_FR(1), LV_GRID_FR(1),
LV_GRID_TEMPLATE_LAST};
    static lv_coord_t grid_main_row_dsc[] = {LV_GRID_CONTENT,
LV_GRID_CONTENT, LV_GRID_TEMPLATE_LAST};

    /*Create the top panel*/
    static lv_coord_t grid_1_col_dsc[] = {400, 50, LV_GRID_CONTENT,
LV_GRID_FR(2), LV_GRID_FR(1), LV_GRID_FR(1), LV_GRID_TEMPLATE_LAST};
    static lv_coord_t grid_1_row_dsc[] = {200, 100, 100, LV_GRID_CONTENT,
10, LV_GRID_CONTENT, LV_GRID_CONTENT, LV_GRID_TEMPLATE_LAST};

    lv_obj_set_grid_dsc_array(parent, grid_main_col_dsc, grid_main_row_dsc);
```

```
    lv_obj_set_grid_cell(panel1, LV_GRID_ALIGN_STRETCH, 0, 2,
LV_GRID_ALIGN_CENTER, 0, 1);

    lv_obj_set_grid_dsc_array(panel1, grid_1_col_dsc, grid_1_row_dsc);
    lv_obj_set_grid_cell(speech_btn, LV_GRID_ALIGN_CENTER, 0, 1,
LV_GRID_ALIGN_CENTER, 0, 5);
    lv_obj_set_grid_cell(name, LV_GRID_ALIGN_START, 2, 2,
LV_GRID_ALIGN_CENTER, 0, 1);
    lv_obj_set_grid_cell(clock_label, LV_GRID_ALIGN_STRETCH, 2, 4,
LV_GRID_ALIGN_START, 1, 1);
    lv_obj_set_grid_cell(time_icn, LV_GRID_ALIGN_CENTER, 2, 1,
LV_GRID_ALIGN_CENTER, 3, 1);
    lv_obj_set_grid_cell(date_label, LV_GRID_ALIGN_START, 3, 1,
LV_GRID_ALIGN_CENTER, 3, 1);
```

Les autres onglets ont des widgets différents, mais le flux de travail global est le même.

Pour compiler l'application, depuis le dossier de projet (contenant le fichier source main.c)

```
mkdir build
cd build.
cmake ..
make -j4
```

Le binaire sera dans `../bin/main`, et peut être exécuté en tapant cette commande : `DISPLAY=:0 ./../bin/main`

Si on ajoute d'autres dossiers au projet, il faut modifier `CMakeLists.txt` en conséquence et réexécuter cmake .. à partir du répertoire de construction, sinon on rencontrera des erreurs de liaison.

Traitement des fichiers images

Pour traiter les fichiers images, il faut ajouter un lecteur à LVGL. En bref, un lecteur est un ensemble de fonctions (ouvrir, lire, fermer, etc.) enregistrées dans LVGL pour effectuer des opérations sur les fichiers. On peut ajouter une interface à un système de fichiers standard (FAT32 sur carte SD) ou créer un système de fichiers simple pour lire les données d'une mémoire Flash SPI. Dans tous les cas, un lecteur n'est qu'une abstraction pour lire et/ou écrire des données en mémoire.

Les images stockées sous forme de fichiers ne sont pas liées à l'exécutable résultant et doivent être lues dans la RAM avant d'être dessinées. En conséquence, elles ne sont pas aussi respectueuses des ressources que les images variables. Cependant, ils sont plus faciles à remplacer sans avoir besoin de recompiler le programme principal.

Système de fichiers

LVGL dispose d'un module d'abstraction 'File system' qui vous permet d'attacher n'importe quel type de système de fichiers.Le système de fichiers est identifié par une lettre de lecteur. Par exemple, si la carte SD est associée à la lettre 'S', un fichier peut être atteint comme “S:chemin/vers/fichier.txt”.

Le référentiel **lv_fs_if** contient des pilotes prêts à l'emploi utilisant POSIX, la norme C et l'API FATFS.

Pour ajouter un lecteur, **lv_fs_drv_t** doit être initialisé comme ci-dessous. **lv_fs_drv_t** doit être statique, alloué globalement ou dynamiquement et non une variable locale.

```
statique lv_fs_drv_t drv; /* Doit être statique ou global */
lv_fs_drv_init (& drv); /* Initialisation de base */

drv.lettre='S'; /* Une lettre majuscule pour identifier le lecteur */
drv.ready_cb=my_ready_cb; /* appel pour dire si le lecteur est prêt à être
utilisé */
drv.open_cb=my_open_cb; /* appel pour ouvrir un fichier */
drv.close_cb=my_close_cb; /* appel pour fermer un fichier */
drv.read_cb=my_read_cb; /* appel pour lire un fichier */
drv.write_cb=my_write_cb; /* appel pour écrire un fichier */
drv.seek_cb=my_seek_cb; /* appel pour rechercher dans un fichier (Déplacer
le curseur) */
drv.tell_cb=mon_tell_cb; /* appel pour indiquer la position du curseur */

drv.dir_open_cb=my_dir_open_cb; /* Rappel pour ouvrir le répertoire pour
lire son contenu */
drv.dir_read_cb=my_dir_read_cb; /* Rappel pour lire le contenu d'un
répertoire */
drv.dir_close_cb=my_dir_close_cb; /* Rappel pour fermer un répertoire */

drv.user_data=my_user_data; /* Toutes les données personnalisées si
nécessaire */

lv_fs_drv_register(&drv); /* Enfin enregistrer le lecteur */
```

N'importe lequel des appels peut être NULL pour indiquer que l'opération n'est pas prise en charge.

Implémentation des appels

Appel open

Le prototype d'`open_cb` ressemble à ceci :

```
void * (*open_cb)(lv_fs_drv_t * drv, const char * path, lv_fs_mode_t mode);
```

path est le chemin après la lettre du pilote (par exemple `S:chemin/vers/fichier.txt" ->`

`"chemin/vers/fichier.txt`). le mode peut être **LV_FS_MODE_WR** ou **LV_FS_MODE_RD** pour ouvrir en écriture ou en lecture.

La valeur de retour est un pointeur sur l'objet fichier qui décrit le fichier ouvert ou NULL s'il y a eu des problèmes (par exemple, le fichier n'a pas été trouvé). L'objet fichier renvoyé sera transmis à d'autres rappels liés au système de fichiers. (voir ci-dessous)

Autres appels

Les autres appels sont assez similaires. Par exemple, `write_cb` ressemble à ceci :

```
lv_fs_res_t (*write_cb)(lv_fs_drv_t * drv, void * file_p, const void * buf,
uint32_t btw, uint32_t * bw);
```

LVGL passe la valeur de retour de `open_cb` à **file_p**, **buf** est les données à écrire, **btw** est les octets à écrire, **bw** est les octets réellement écrits.

Pour un modèle pour les rappels, voir lv*fs*template.c.

Exemple d'utilisation

L'exemple ci-dessous montre comment lire à partir d'un fichier :

```
lv_fs_file_t f;
lv_fs_res_t res;
res = lv_fs_open(&f, "S:folder/file.txt", LV_FS_MODE_RD);
if(res != LV_FS_RES_OK) my_error_handling();

uint32_t read_num;
uint8_t buf[8];
res = lv_fs_read(&f, buf, 8, &read_num);
if(res != LV_FS_RES_OK || read_num != 8) my_error_handling();

lv_fs_close(&f);
```

Le mode dans lv*fs*open peut être **LV_FS_MODE_WR** pour ouvrir en écriture ou **LV_FS_MODE_RD | LV_FS_MODE_WR** pour les deux

Cet exemple montre comment lire le contenu d'un répertoire. C'est au pilote comment marquer les répertoires, mais cela peut être une bonne pratique d'insérer un '/' devant le nom du répertoire.

```
lv_fs_dir_t dir;
lv_fs_res_t res;
res = lv_fs_dir_open(&dir, "S:/folder");
if(res != LV_FS_RES_OK) my_error_handling();
```

```
char fn[256];
while(1) {
    res = lv_fs_dir_read(&dir, fn);
    if(res != LV_FS_RES_OK) {
        my_error_handling();
        break;
    }

    /*fn is empty, if not more files to read*/
    if(strlen(fn) == 0) {
        break;
    }

    printf("%s\n", fn);
}

lv_fs_dir_close(&dir);
```

Décodeur d'images

Les images manipulées dans LVGL doivent être décodées en pixel, l'exemple suivant montre l'élaboration d'un décodeur d'image et l'utilisation de certaines fonctions pour ouvrir/fermer les fichiers PNG. Cela devrait ressembler à ceci :

```
/*Créer un nouveau décodeur et enregistrer des fonctions */
lv_img_decoder_t * dec=lv_img_decoder_create();
lv_img_decoder_set_info_cb(dec, decoder_info);
lv_img_decoder_set_open_cb(dec, decoder_open);
lv_img_decoder_set_close_cb(dec, decoder_close);

/**
 * Obtenir des informations sur une image PNG
 * @param decoder pointeur vers le décodeur auquel appartient cette fonction
 * @param src peut être un nom de fichier ou un pointeur vers un tableau C
 * @param header stocke les informations ici
 * @return LV_RES_OK : pas d'erreur ; LV_RES_INV : impossible d'obtenir les
informations
 */
static lv_res_t decoder_info(lv_img_decoder_t * decoder, const void * src,
lv_img_header_t * header)
{
  /*Vérifier si le type `src` est connu du décodeur*/
  if(is_png(src) == false) return LV_RES_INV;

  /* Lire l'en-tête PNG et trouver `width` et `height` */
  ...

  header->cf=LV_IMG_CF_RAW_ALPHA;
  header->w=width;
```

```
  header->h=height;
}

/**
 * Ouvre une image PNG et retourne l'image décidée
 * @param decoder pointeur vers le décodeur auquel appartient cette fonction
 * @param dsc pointeur vers un descripteur qui décrit cette session de
décodage
 * @return LV_RES_OK : pas d'erreur ; LV_RES_INV : impossible d'obtenir les
informations
 */
static lv_res_t decoder_open(lv_img_decoder_t * decoder,
lv_img_decoder_dsc_t * dsc)
{

  /*Vérifier si le type `src` est connu du décodeur*/
  if(is_png(src) == false) return LV_RES_INV;

  /*Décode et stocke l'image. Si `dsc->img_data` est `NULL`, la fonction
`read_line` sera appelée pour obtenir les données de l'image ligne par
ligne*/
  dsc->img_data=my_png_decoder(src);

  /*Modifie le format de couleur si nécessaire. Pour PNG, généralement 'Raw'
convient */
  dsc->header.cf=LV_IMG_CF_...

  /*Appelle une fonction de décodeur intégrée si nécessaire. Ce n'est pas
obligatoire si `my_png_decoder` a ouvert l'image au format True Color.*/
  lv_res_t res=lv_img_decoder_built_in_open(decoder, dsc);

  return res;
}

/**
 * Décode les pixels `len` à partir des coordonnées `x`, `y` données et les
stocke dans `buf`.
 * Obligatoire uniquement si la fonction "open" ne peut pas ouvrir
l'ensemble du tableau de pixels décodé. (dsc->img_data == NULL)
 * @param decoder pointeur vers le décodeur la fonction associée à
 * @param dsc pointeur vers le descripteur de décodeur
 * @param x début x coordonnée
 * @param y début y coordonnée
 * @param len nombre de pixels à décoder
 * @param buf un buffer pour stocker les pixels décodés
 * @return LV_RES_OK : ok ; LV_RES_INV : échec
 */
lv_res_t decoder_built_in_read_line(lv_img_decoder_t * decoder,
lv_img_decoder_dsc_t * dsc, lv_coord_t x,
                                                lv_coord_t y, lv_coord_t
len, uint8_t * buf)
```

```
{
   /*Avec PNG, ce n'est généralement pas nécessaire*/

   /*Copier les pixels `len` des coordonnées `x` et `y` au format True color
vers `buf` */
}

/**
 * Libére les ressources allouées
 * @param decoder pointeur vers le décodeur auquel appartient cette fonction
 * @param dsc pointeur vers un descripteur qui décrit cette session de
décodage
 */
static void decoder_close(lv_img_decoder_t * decoder, lv_img_decoder_dsc_t *
dsc)
{
  /*Libére toutes les données allouées*/

  /*Appelle la fonction de fermeture intégrée si la ligne open/read_line
intégrée a été utilisée*/
  lv_img_decoder_built_in_close(decoder, dsc);

}
```

En résumé :

- **decoder_info** collecte des informations de base sur l'image et les stocke dans l'en-tête.
- **decoder_open** essaye d'ouvrir la source de l'image pointée par `dsc->src`. Son type est déjà dans `dsc->src_type == LV_IMG_SRC_FILE/VARIABLE`. Si ce format/type n'est pas pris en charge par le décodeur, retourne **LV_RES_INV**. Cependant, s'il peut ouvrir l'image, un pointeur vers l'image en couleurs vraies décodée doit être défini dans `dsc->img_data`. Si le format est connu mais qu'on ne veut pas décoder l'image (par exemple, on a pas assez de mémoire pour elle), définir `dsc->img_data=NULL` pour appeler `read_line` pour obtenir les pixels.
- **decoder_close**, libére toutes les ressources allouées.
- **decoder_read** est facultatif. Le décodage de l'image entière nécessite de la mémoire supplémentaire et une surcharge de calcul. Cependant, si on décode une ligne de l'image sans décoder toute l'image, on peut économiser de la mémoire et du temps. Pour indiquer cela, la fonction de lecture de ligne doit être utilisée, définir `dsc->img_data=NULL` dans la fonction open.