Rockchip LVGL用户指南

文件标识：RK-KF-YF-A22

发布版本：V1.1.0

日期：2024-03-25

文件密级：□绝密 □秘密 □内部资料 ■公开

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址： 福建省福州市铜盘路软件园A区18号

网址： www.rock-chips.com

客户服务电话： +86-4007-700-590

客户服务传真： +86-591-83951833

客户服务邮箱： fae@rock-chips.com

前言

概述

本文档主要介绍 LVGL编译和测试方法。

支持的系统和版本

系统	版本
Buildroot	8.3.x

读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

修订记录

日期	版本	作者	修改说明
2023-11-08	V1.0.0	Jair Wu	初始版本
2024-03-25	V1.1.0	Jair Wu	新增OpenGL渲染支持

目录

1. LVGL 简介

LVGL 是一个较为流行的免费和开源的嵌入式图形库，可以在MCU、MPU以及各种屏幕上绘制美观的UI。

LVGL 是完全开源的，除了部分扩展功能外没有外部依赖，这使得它的移植非常简单。它可与任何现代 MCU 或 MPU 配合使用，并可与任何 (RT)OS 或裸机设置一起使用，以驱动电子纸、单色、OLED 或 TFT 显示器，甚至监视器。因此，甚至可以在Windows、Linux上调试UI界面，有一些基础的平台无关的UI调试，反复编译烧录固件是十分繁琐的，这时就可以在PC上搭建环境调试，再拷贝到嵌入式设备上编译运行。

2. LVGL 源码

在通用Linux SDK中，LVGL有三个源码仓库，分别为lvgl，lv_drivers，lvgl_demo。其中lvgl仓库为LVGL框架源码，lv_drivers为一些比较常用的显示接口，比如drm，sdl，wayland等，lvgl_demo则是由RK编写的一些应用示例，比如如何调用一些基本的初始化，如何将官方的DEMO运行起来等。

其中lvgl与lv_drivers源码均通过官方github仓库下载，并打上RK提供的一些补丁，详见`buildroot/package/lvgl/lvgl/`与`buildroot/package/lvgl/lv_drivers/`。lvgl_demo源码则在同步SDK时下载，路径为`<SDK>/app/lvgl_demo`。

3. LVGL 配置

LVGL的配置主要区别在于渲染后端的选择，目前可选DRM直接送显以及通过SDL送显。其中DRM配置使用在一些没有GPU的平台如RK3308等，SDL配置则使用在一些有GPU的平台如RK3568等。

3.1 DRM 配置

基础的配置已保存在`<SDK>/buildroot/configs/rockchip/lvgl.config`。

```
# 开启LVGL
BR2_PACKAGE_LVGL=y
# 使用32bit位深，即ARGB8888。也可以使用16bit位深，即RGB565
BR2_PACKAGE_LVGL_COLOR_DEPTH=32
# 使用DRM送显，则绘制渲染由CPU完成
BR2_PACKAGE_LVGL_USE_DRM=y
# 开启LV_DRIVERS
BR2_PACKAGE_LV_DRIVERS=y
# 开启LV_DRIVERS中DRM相关代码
BR2_PACKAGE_LV_DRIVERS_USE_DRM=y
# 开启LVGL_DEMO
BR2_PACKAGE_LVGL_DEMO=y
# 开启官方默认的播放器DEMO
```

```
BR2_PACKAGE_LVGL_DEMO_MUSIC=y
```

3.2 SDL 配置

部分平台有GPU，则可以开启SDL，实现GPU硬件加速。

```
# LVGL 相关配置
BR2_PACKAGE_LVGL=y
BR2_PACKAGE_LVGL_COLOR_DEPTH=32
BR2_PACKAGE_LVGL_USE_SDL=y
BR2_PACKAGE_LV_DRIVERS=y
BR2_PACKAGE_LV_DRIVERS_USE_SDL_GPU=y
BR2_PACKAGE_LVGL_DEMO=y
BR2_PACKAGE_LVGL_DEMO_WIDGETS=y
# SDL 相关配置，使用wayland作为后端，可依据需求修改
BR2_PACKAGE_SDL2=y
BR2_PACKAGE_SDL2_OPENGLES=y
BR2_PACKAGE_SDL2_WAYLAND=y
```

如果需要额外的OpenGL(ES)接口支持，则可以开启：

```
BR2_PACKAGE_LV_DRIVERS_USE_OPENGL=y
```

4. LVGL 编译

```
cd buildroot
source envsetup.sh
# 选择板子对应配置
make lvgl lv_drivers lvgl_demo -j20
```

需要注意，如果进入buildroot menuconfig修改了LVGL相关的配置，建议将每个仓库都重新编译一遍，避免配置不一致导致链接或运行出错，如：

```
cd buildroot
make lvgl-reconfigure lv_drivers-reconfigure lvgl_demo-reconfigure -j20
```

5. LVGL_DEMO 代码说明

源码路径为`<SDK>/app/lvgl_demo/`。

5.1 目录结构

```
app/lvgl_demo/
├── cJSON    # cJSON源码
├── hal      # drm，sdl，触摸，按键相关适配
├── lv_demo  # 基础示例程序，运行官方DEMO
├── lvgl     # 文件系统、触摸接口注册
├── rk_demo  # RK显控DEMO，包含智能家居、家电显控、楼宇对讲、系统设置等DEMO
│   ├── furniture_control # 家电显控
│   ├── home              # DEMO首页
│   ├── include           # rkwifibt相关头文件
│   ├── intercom_homepage # 楼宇对讲
│   │   ├── intercom_call # 对讲呼叫
│   │   └── video_monitor # RTSP码流播放
│   ├── resource          # 图片、字体资源
│   ├── rockit            # rockit相关适配
│   ├── setting           # 系统设置
│   ├── smart_home        # 智能家居
│   └── wifibt            # rkwifibt相关适配
└── sys                   # 时间戳，trace debug等，已弃用
```

5.2 lv_demo 代码说明

源码路径为`<SDK>/app/lvgl_demo/lv_demo`，主要作为一个示例程序，演示如何将官方的DEMO运行起来。以下说明略过一些无关的代码，仅挑选需要关注的代码进行说明。

```c
/* <SDK>/app/lvgl_demo/lv_demo/main.c */
...
/* 触摸旋转 */
static int g_indev_rotation = 0;
/* 显示旋转，仅SDL支持 */
static int g_disp_rotation = LV_DISP_ROT_NONE;
...
static void lvgl_init(void)
{
    /* 一切LVGL应用的开始 */
    lv_init();

    /* 根据配置选择SDL或DRM初始化，注册送显接口 */
#ifdef USE_SDL_GPU
    hal_sdl_init(0, 0, g_disp_rotation);
#else
    hal_drm_init(0, 0, g_disp_rotation);
#endif
    /* 文件系统初始化 */
    lv_port_fs_init();
    /* 触摸设备初始化 */
    lv_port_indev_init(g_indev_rotation);
}

int main(int argc, char **argv)
{
    lvgl_init();
```

```
    /* 根据配置选择对应的DEMO初始化，绘制对应UI */
#if LV_USE_DEMO_WIDGETS
    lv_demo_widgets();
#elif LV_USE_DEMO_KEYPAD_AND_ENCODER
    lv_demo_keypad_encoder();
#elif LV_USE_DEMO_BENCHMARK
    lv_demo_benchmark();
#elif LV_USE_DEMO_STRESS
    lv_demo_stress();
#elif LV_USE_DEMO_MUSIC
    lv_demo_music();
#endif
    while (!quit)
    {
        ...
        /* 调用LVGL任务处理函数，LVGL所有的事件、绘制、送显等都在该接口内完成 */
        lv_task_handler();
        ...
    }

    return 0;
}
```

6. OpenGL(ES)接口说明

由于LVGL本身不支持OpenGL(ES)接口，且SDL不支持3D对象，因此针对一些需要使用到相关功能的应用，封装出一套lv_gl接口，使LVGL应用可以在原有的UI应用上直接叠加上3D图像效果。

6.1 文件说明

```
lv_drivers/sdl/gl/
├── gl.c                // 接口实现
├── gl.h                // 接口及外部结构体定义
├── mat.c               // 矩阵函数实现
├── mat.h               // 矩阵函数定义
└── shaders             // OpenGL着色器
    ├── bgra_cube.frag  // BGRA格式立方体片段着色器
    ├── bgra.frag       // BGRA格式图片片段着色器
    ├── common.vert     // 通用顶点着色器，包含旋转矩阵、缩放矩阵等
    ├── cube.vert       // 立方体顶点着色器
    ├── default.vert    // 图片顶点着色器
    ├── rgba_cube.frag  // RGBA格式立方体片段着色器
    └── rgba.frag       // RGBA格式图片片段着色器
```

6.2 枚举说明

6.2.1 颜色格式

目前支持的是LV_GL_FMT_RGBA和LV_GL_FMT_BGRA，后续考虑加入YUV格式支持。

```
enum {
    LV_GL_FMT_ALPHA,
    LV_GL_FMT_LUMINANCE,
    LV_GL_FMT_LUMINANCE_ALPHA,
    LV_GL_FMT_INTENSITY,
    LV_GL_FMT_RGB,
    LV_GL_FMT_RGBA,
    LV_GL_FMT_BGRA,
};
```

6.2.2 纹理类型

用于区分平面图形和立方体图形，使用不同的渲染方式

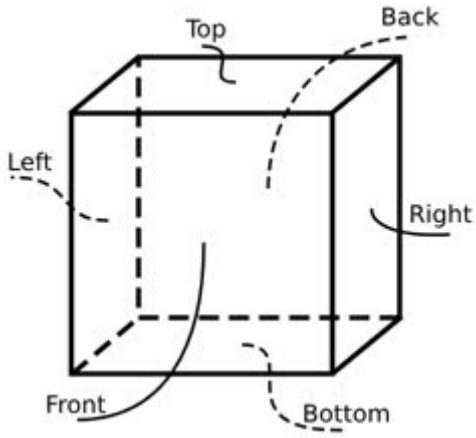
```
enum {
    GL_TEX_TYPE_2D,
    GL_TEX_TYPE_CUBE,
};
```

6.2.3 立方体平面

用于选择立方体平面，在导入图像，或使用立方体对象作为渲染目标时选择目标面。

```
enum {
    CUBE_LEFT,
    CUBE_RIGHT,
    CUBE_TOP,
    CUBE_BOTTOM,
    CUBE_FRONT,
    CUBE_BACK,
};
```

在OpenGL的坐标中，屏幕中间为原点(0, 0)，X坐标向右为正，Y坐标向上为正，Z坐标垂直于屏幕向外为正，因此，当一个立方体被创建，无旋转的情况下，面向用户的方向为正（CUBE_FRONT），其余方向类推，如下图所示（虚线表示不可见部分）：



6.3 结构体说明

应用上使用，主要关注lv_gl_img_t，lv_gl_tex_t，lv_gl_obj_t三个结构体。

- lv_gl_img_t
- 用于将外部图片导入为GPU纹理对象

```
typedef struct {
    /* 图片像素数据地址 */
    const void *pixels;
    /* 图片格式 */
    int format;
    /* 宽 */
    int w;
    /* 高 */
    int h;
} lv_gl_img_t;
```

- lv_gl_tex_t
- 储存GPU纹理对象及相关信息

```
typedef struct {
    /* 宽 */
    int w;
    /* 高 */
    int h;
} lv_gl_size_t;

typedef struct {
    /* GPU纹理对象 */
    GLuint gl_tex;
    /* GPU FBO帧缓冲对象，用于将该纹理作为FB时使用 */
    GLuint FBO;
    /* 纹理大小，普通2D纹理仅有第一组size有效，立方体有6个面则有6组size信息 */
    lv_gl_size_t size[6];
    /* 纹理格式 */
    int format;
    /* 纹理类型，用于区分2D和立方体 */
    int type;
    /* 引用计数 */
```

```
    int ref_cnt;
} lv_gl_tex_t;
```

- lv_gl_obj_t
- 储存GPU渲染对象及相关信息，如顶点、变换参数等

```
typedef struct {
    float x, y, z;
} lv_gl_vec_t;

typedef struct {
    /* 宽 */
    int w;
    /* 高 */
    int h;
    /* Y方向纹理翻转，应用可以忽略，按默认0即可 */
    int reverse_y;
    /* 绑定的纹理 */
    lv_gl_tex_t *tex;
    /* 渲染区域（位置及大小） */
    SDL_Rect r;
    /* 视图区域（位置及大小） */
    SDL_Rect view;
    /* 裁剪区域（位置及大小） */
    SDL_Rect crop;
    /* 视图区域需要更新，内部使用，应用可忽略 */
    int view_dirty;
    /* 裁剪区域需要更新，内部使用，应用可忽略 */
    int crop_dirty;
    /* 裁剪区域使能 */
    int crop_en;
} lv_gl_base_t;

typedef struct {
    lv_gl_base_t base;
    /* GPU VAO，顶点数组对象，会将VBO及一些属性配置缓存，方便快速切换 */
    GLuint VAO;
    /* GPU VBO，顶点缓冲对象，存储坐标、纹理顶点信息 */
    GLuint VBO;
    /* 输出目标，默认GL_TEXTURE_2D，如果需要在立方体上渲染，则通过该变量指定渲染的面 */
    int out_type;
    /*
     * 手动指定2D对象的四个顶点坐标
     * 默认顶点为全屏（注意是framebuffer的视图大小，而不是屏幕的全屏，两者可以不相等）
     * 再通过设置scale, offset等属性进行矩阵变换。也可以通过该属性自由指定顶点坐标
     * 但不建议两种方式共同使用，涉及的变换比较复杂，人力计算不好估算最终的效果
     * x, y, z范围均为[-1, 1]
     */
    lv_gl_vec_t p[4];
    /* 手动指定纹理的引用范围，在只需要使用到纹理的部分切片时使用 */
    SDL_Rect tp;

    /* 全局透明度 */
    float alpha;
    /*
     * 以下变换的顺序为：
     * 默认顶点坐标pos * offset * scale * self_rot * view_rot * perspective * move
```

```
     * 从左到右依次计算，其中perspective为视图转换，实现近大远小效果
     */
    /* 三轴方向的缩放 */
    lv_gl_vec_t scale;
    /* 三轴方向的偏移 */
    lv_gl_vec_t offset;
    /* 三轴方向的旋转 */
    lv_gl_vec_t self_rot;
    /* 三轴方向的视角旋转 */
    lv_gl_vec_t view_rot;
    /* 三轴方向的位移 */
    lv_gl_vec_t move;
} lv_gl_obj_t;
```

6.4 函数说明

- void lv_gl_set_render_cb(lv_gl_render_cb_t cb);

  注册应用层渲染回调，对象的渲染只能在回调里完成

  cb：渲染回调函数指针

- void lv_gl_set_fb(lv_gl_obj_t *obj);

  设置framebuffer

  obj：不为空时则后续的渲染会将图像合成在该对象的纹理内存中，为空则表示恢复到默认的framebuffer，直接渲染到屏幕上

- void lv_gl_read_pixels(void *ptr, SDL_Rect *r, int type);

  将当前的framebuffer的指定区域拷贝到内存中

  ptr：目标内存

  r：目标区域，起始点及大小

  type：目标类型，传0则默认GL_TEXTURE_2D，如果需要读取立方体的面，则通过该变量指定

- lv_gl_tex_t *lv_gl_tex_create(int type, int w, int h, lv_gl_img_t *img);

  创建纹理对象

  type：纹理类型，GL_TEX_TYPE_2D或GL_TEX_TYPE_CUBE

  w：纹理宽度，img不为空时会使用img中的宽度值，该值被忽略

  h：纹理高度，img不为空时会使用img中的高度值，该值被忽略

  img：导入的图像，为空时则会只会在GPU内申请对应空间（默认值全为0.0），不为空时则会将对应图像也导入到GPU内。如果纹理类型为GL_TEX_TYPE_CUBE，则该参数需要传入一个大小至少为6的数组

  返回值：lv_gl_tex_t *，纹理对象指针

- void lv_gl_tex_del(lv_gl_tex_t *tex);

  删除纹理对象

  tex：纹理对象指针

- void lv_gl_tex_import_img(lv_gl_tex_t *tex, lv_gl_img_t *img);

  纹理对象导入外部图像，如果纹理对象在创建时已经传入过图像指针，则该接口不需要再次被调用。如果需要更换图像，或者创建时仅创建空内存，则使用该接口导入图像

  tex：纹理对象指针

img：导入的图像地址。如果纹理类型为GL_TEX_TYPE_CUBE，则该参数需要传入一个大小至少为6的数组

- void lv_gl_tex_clear(lv_gl_tex_t *tex, float r, float g, float b, float a);

  将纹理对象使用指定颜色填充

  tex：纹理对象指针

  r：红色通道

  g：绿色通道

  b：蓝色通道

  a：透明度通道

- lv_gl_obj_t *lv_gl_obj_create(int w, int h);

  创建渲染对象。创建后的对象无法直接渲染，需要绑定纹理后才可渲染

  w：宽度，用于缩放计算

  h：高度，用于缩放计算

  返回值：lv_gl_obj_t *，渲染对象指针

- void lv_gl_obj_del(lv_gl_obj_t *obj);

  删除渲染对象

  obj：渲染对象

- void lv_gl_obj_resize(lv_gl_obj_t *obj, lv_gl_obj_t *parent);

  根据渲染目标重设渲染缩放大小。默认创建的渲染对象为全屏显示，因此需要使用该接口重置scale参数。需要注意的是，该接口使用obj的宽高和parent的视图宽高进行计算，而非和parent的宽高进行计算

  obj：渲染对象

  parent：渲染目标。为空时则选为默认值，即屏幕

- void lv_gl_obj_move(lv_gl_obj_t *obj, lv_gl_obj_t *parent);

  根据渲染目标重设渲染位置。根据scale和base.r的值与parent的视图宽高进行计算，由于使用了scale的值，该接口建议在resize之后调用，且scale有手动修改时，均建议重新调用下该接口

  obj：渲染对象

  parent：渲染目标。为空时则选为默认值，即屏幕

- void lv_gl_obj_reset_points(lv_gl_obj_t *obj);

  将渲染对象的顶点参数重置，即全屏显示

  obj：渲染对象

- void lv_gl_obj_reset_tex_points(lv_gl_obj_t *obj);

  将渲染对象的纹理顶点参数重置，即使用全图

  obj：渲染对象

- void lv_gl_obj_update_vao(lv_gl_obj_t *obj);

  根据当前的顶点参数和纹理顶点参数更新渲染对象的VAO和VBO。因此如果手动修改了顶点参数和纹理顶点参数后，需要调用该接口才会生效。如未创建过则会创建新的VAO和VBO。

  obj：渲染对象

- void lv_gl_obj_release_vao(lv_gl_obj_t *obj);

  释放VAO和VBO，使用默认的参数

obj：渲染对象

- void lv_gl_obj_bind_tex(lv_gl_obj_t *obj, lv_gl_tex_t *tex);

  绑定纹理对象，纹理对象引用计数会加1。如果渲染对象已经有绑定的纹理对象，则会对原有的纹理对象减引用，当引用小于等于0时会自动释放。因此需要注意，该接口可能触发内存释放，应用上需要避免外部引用纹理对象和重复释放

  obj：渲染对象

  tex：纹理对象

- void lv_gl_obj_set_crop(lv_gl_obj_t *obj, SDL_Rect *r, int en);

  设置裁剪范围，以屏幕左下角为原点坐标，向上为Y正方向，向右为X正方向，与LVGL默认的左上角为原点不同，需要特别注意

  obj：渲染对象，为空时表示屏幕

  r：裁剪范围，起始坐标和大小，为空时起始坐标重置为原点，大小重置为渲染对象大小

  en：为1使能裁剪，为0关闭裁剪

- void lv_gl_obj_get_crop(lv_gl_obj_t *obj, SDL_Rect *r);

  获取裁剪范围

  obj：渲染对象，为空时表示屏幕

  r：裁剪范围，起始坐标和大小

- void lv_gl_obj_set_viewport(lv_gl_obj_t *obj, SDL_Rect *r);

  设置视图范围，以屏幕左下角为原点坐标，向上为Y正方向，向右为X正方向，与LVGL默认的左上角为原点不同，需要特别注意

  obj：渲染对象，为空时表示屏幕

  r：视图范围，起始坐标和大小，为空时起始坐标重置为原点，大小重置为渲染对象大小

- void lv_gl_obj_get_viewport(lv_gl_obj_t *obj, SDL_Rect *r);

  获取视图范围

  obj：渲染对象，为空时表示屏幕

  r：视图范围，起始坐标和大小

- void lv_gl_obj_render(lv_gl_obj_t *obj);

  将渲染对象渲染在framebuffer上，只能在回调中使用

  obj：渲染对象

7. Gallery DEMO代码说明

针对相册，图库类应用，新增Gallery DEMO，包含多个动画效果，如滑入，淡入，立方体旋转，立方体翻转，立方体文字显示，滚筒图片预览，照片流等

7.1 目录结构

```
gallery/
├── anims                    // 动画实现
│   ├── cube_flip.c          // 立方体翻转动画
│   ├── cube_flip.h
│   ├── cube_lyric.c         // 立方体字符渲染
│   ├── cube_lyric.h
│   ├── cube_rotate.c        // 立方体旋转
│   ├── cube_rotate.h
│   ├── fade_out.c           // 淡出
│   ├── fade_out.h
│   ├── fade_slide_out.c     // 滑动淡出
│   ├── fade_slide_out.h
│   ├── fold.c               // 照片折叠
│   ├── fold.h
│   ├── photo_stream.c       // 照片流
│   ├── photo_stream.h
│   ├── roller.c             // 滚桶图片预览
│   ├── roller.h
│   ├── slide_out.c          // 滑动退出
│   ├── slide_out.h
│   ├── stiker.c             // 贴图测试
│   └── stiker.h
├── CMakeLists.txt
├── gallery.c                // UI入口。渲染对象，纹理初始化等
├── gallery.h
├── main.c                   // 应用入口。lvgl，屏幕，触摸初始化等
├── main.h
└── pics                     // 图片资源
```

7.2 UI入口函数

main.c内函数与[lv_demo 代码说明]一致，不做过多介绍，主要介绍UI相关代码。

```
// gallery.c

/* 定义在立方体字符渲染动画中用到的字符，每两行渲染到一个面上，共12行 */
static char *lyric[]...

/* 各个动画中用到的纹理对象 */
lv_gl_tex_t *tex_cube;
lv_gl_tex_t *tex_2d[6];
lv_gl_tex_t *tex_roller;
lv_gl_tex_t *tex_fb;

/* 各个动画中用到的渲染对象 */
lv_gl_obj_t *obj_fb;
lv_gl_obj_t *obj_img0;
lv_gl_obj_t *obj_img1;
lv_gl_obj_t *obj_cube;
lv_gl_obj_t *obj_fold[4];
lv_gl_obj_t *obj_roller_items[6];
```

```
lv_gl_obj_t *obj_roller;
/*
 * 立方体字符渲染动画中用到的结构体，共12行则会创建一个12个成员的lyric_row数组
 * 每个lyric_row内都包含了当前行的所有单个字符
 */
lyric_row *obj_lyrics;

/*
 * 视图大小，为屏幕大小减去按钮矩阵的大小
 * 即屏幕分为两部分，下半部分为按钮，用于交互触发动画，上半部分为动画渲染区域
 */
SDL_Rect view;

/* 屏幕完整大小 */
SDL_Rect screen;

/* 指示当前动画是否在播放中，一次只能等待一个动画播放完成 */
int animing = 0;

/* ttf文件导入的字体 */
lv_ft_info_t ttf_main;

/* lvgl对象，屏幕 */
lv_obj_t *scr;
/* lvgl对象，图片，用于图片切换动画 */
lv_obj_t *img1;
/* lvgl对象，图片，用于图片切换动画  */
lv_obj_t *img2;
/* lvgl对象，动画区域容器 */
lv_obj_t *anim_area;
/* lvgl对象，按钮矩阵 */
lv_obj_t *btn_mat;
/* lvgl对象，进度条，显示动画进度 */
lv_obj_t *slider;
/* lvgl对象，照片流容器 */
lv_obj_t *photo_box;
/* lvgl对象，照片流图片 */
lv_obj_t *photos[6];

/* 定义了所有的动画，各个动画都通过宏定义在各自的头文件中，并在此处引用 */
static lv_anim_t anims[]...

/* 定义按钮矩阵的文字，与动画顺序一一对应 */
static const char *btnm_map[]...

/* 按钮矩阵回调，用于触发动画 */
static void event_handler(lv_event_t * e)

/* 字体初始化，用于中文字符渲染 */
static void font_init(void)

/* 创建渲染画布，用于后续将文字渲染为图片 */
static label_canvas *create_canvas(lv_color_t color, lv_font_t *font)

/* 将文字渲染为图片，并创建GPU渲染对象，自动绑定纹理后返回 */
static lv_gl_obj_t *utf8_to_obj(lv_gl_obj_t *parent, label_canvas *lc, char
*text)
```

```c
/* 通用动画起始回调，用于重置一些动画参数 */
void common_anim_start(void)

/* 纹理对象初始化 */
static void tex_init(void)

/* UI入口 */
void gallery(void)
```

```c
// gallery.h
typedef struct
{
    /* 指定字体及渲染颜色 */
    lv_draw_label_dsc_t label_dsc;
    /* 画布buffer，用于反复利用，减少申请内存次数 */
    lv_img_dsc_t *img_dsc;
    /* 画布对象 */
    lv_obj_t *canvas;
} label_canvas;

typedef struct {
    /* 字符转出的GPU渲染对象 */
    lv_gl_obj_t **objs;
    /* 当前行字符长度 */
    int len;
} lyric_row;
```

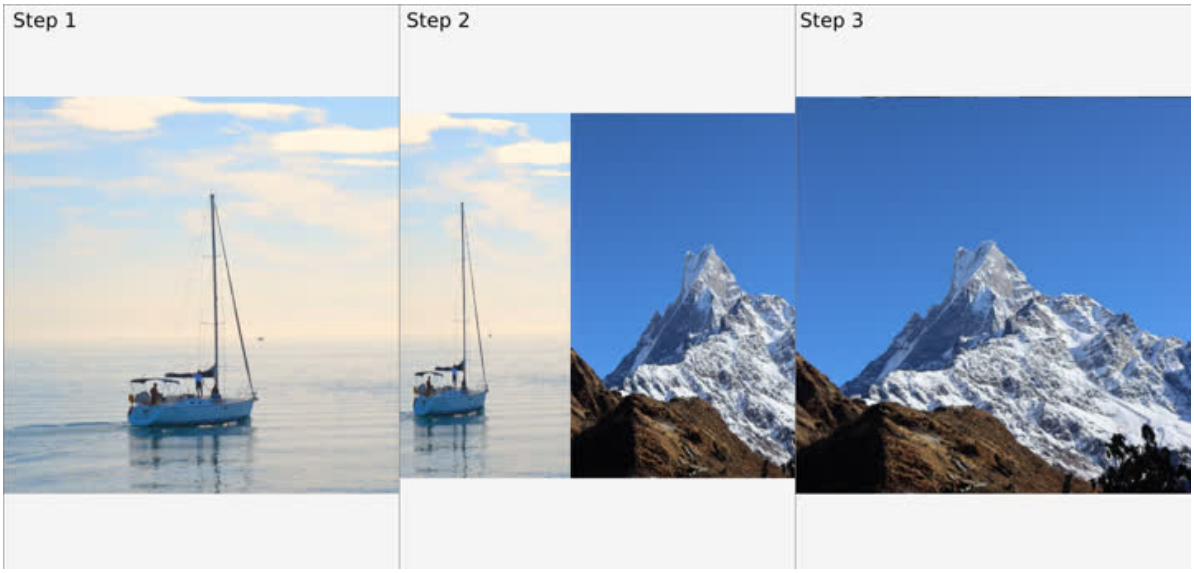
7.3 动画

由于lv_gl无法自动触发渲染，需要在动画回调中触发LVGL框架渲染流程，目前是在所有的动画回调中使用lv_obj_invalidate(lv_layer_top())触发渲染，同时由于top layer没有子对象，不会影响到正常的渲染性能。

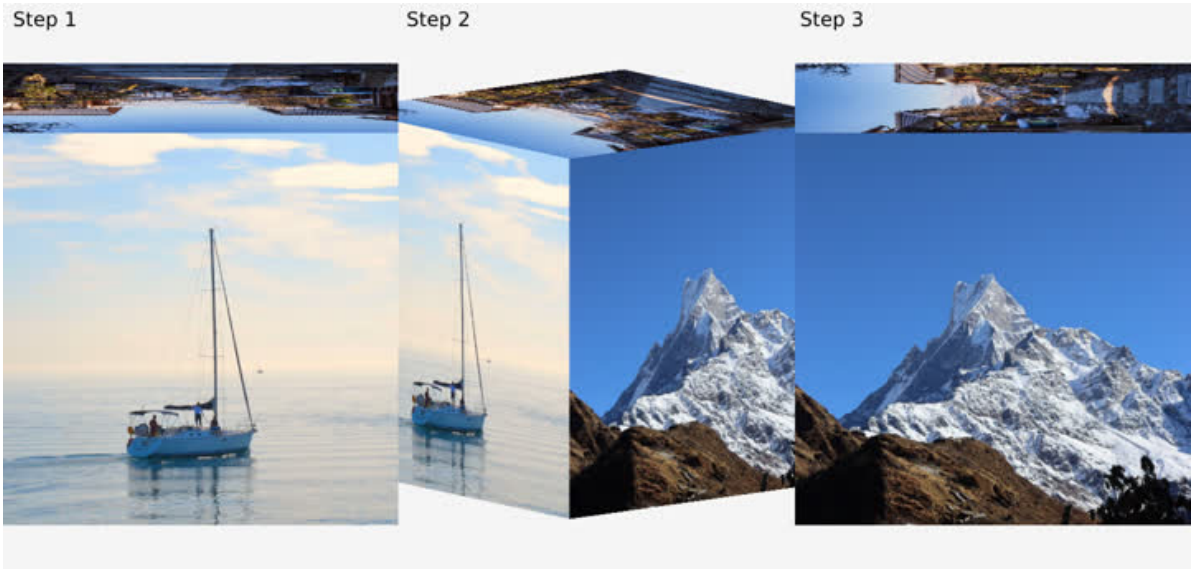
7.3.1 立方体翻转

- 效果说明

动画效果如图所示，通过立方体的旋转，实现两张照片的切换，旋转过程中立方体适当缩小



通过X轴旋转15度视角可以更好的观察这个动画



- 代码说明

```
/* anims/cube_flip.c */
/*
 * 定义额外的三个子动画。应用中将完整的动画分为4步，每步旋转90度
 * 第一个动画定义在anims/cube_flip.h中，从0度转到90度
 * 第一个子动画从90度旋转到180度
 * 第二个子动画从180度转到270度
 * 第三个子动画从270度转到360度
 * 每个子动画都设定了act_time = -1000，从而实现每个动作之间间隔1s
 */
static lv_anim_t sub_anims[]...
/* 设置立方体动画视图范围，所有立方体动画通用 */
SDL_Rect cube_view;
/* 根据当前的动画进度设置缩放效果 */
#define ZOOM_IN ...
/* 立方体动画渲染回调，所有立方体动画通用 */
void anim_cube_render(void)
/*
 * 立方体动画起始回调，所有立方体动画通用
 * 特别注意，在起始回调中将视图修改为一个特别的值
 * 首先取默认视图中宽高较小的一边（比如宽比高小，则取宽）
 * 并将该值除0.57 + 2作为新的视图的宽高值（视图需要为正方形，否则后续变换可能异常）
```

```
 * 由于正方体的中心到任意一角的距离d=sqrt(3)/2*a
 * a为边长，且等于2.0（坐标系为[-1，-1，-1]-[1，1，1]）
 * 因此为了保证无论如何旋转，立方体的任意角都不会超出坐标系
 * 即d应该始终小于1.0
 * 则选择缩放比例为0.57。sqrt(3)/2*a*0.57=0.9873
 * +2仅为经验值，保证立方体略微超出屏幕，不让左右留有缝隙，根据实际应用情况可去除
 */
void anim_cube_start(lv_anim_t *a)
/* 起始回调 */
void anim_cube_flip_start(lv_anim_t *a)
/* 动画回调，根据当前动画进度，设置Y轴旋转角度和缩放比例 */
void anim_cube_flip(void *var, int32_t v)
/* 结束回调，根据a->var的值判断当前是第几段动画，并执行下一段动画或结束 */
void anim_cube_flip_end(lv_anim_t *a)
```

7.3.2 立方体旋转

- 效果说明

通过简单的360度旋转展示不同面上的照片

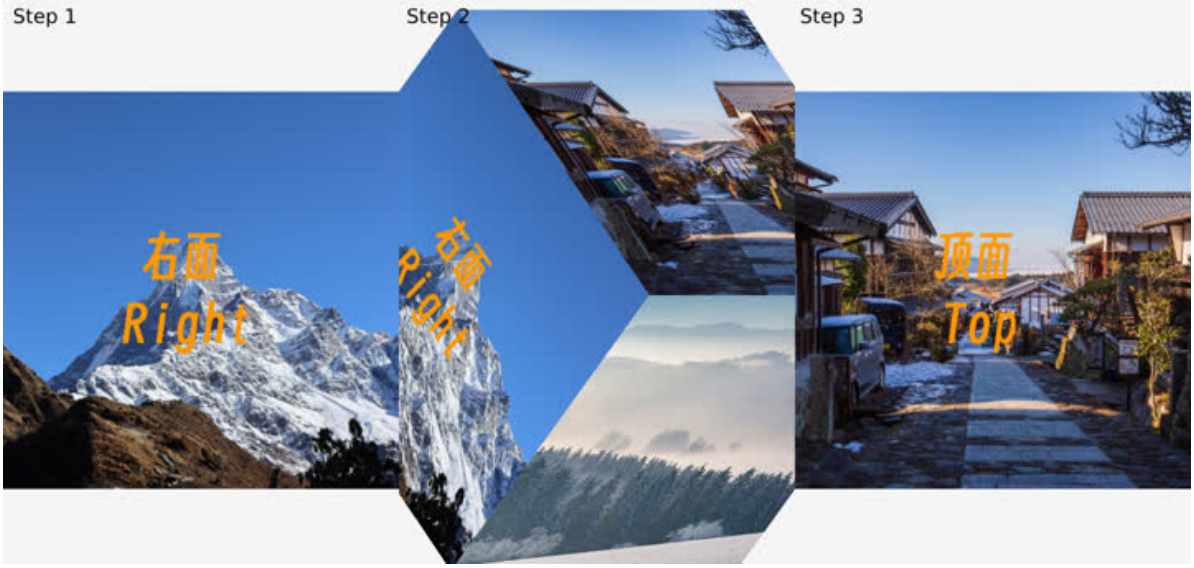
- 代码说明

```
/* anims/cube_rotate.c */
/* 起始回调，直接调用anim_cube_start，说明见前一小节 */
void anim_cube_rotate_start(lv_anim_t *a)
/* 动画回调，设置缩放为0.57(原因见前一节)，并根据动画进度设置X，Y轴的旋转角度 */
void anim_cube_rotate(void *var, int32_t v)
/* 结束回调，恢复动画标志位，恢复进度条 */
void anim_cube_rotate_end(lv_anim_t *a)
```

7.3.3 立方体字符渲染

- 效果说明

依次旋转立方体6个面，并在当前显示的面上绘制字符，将字符替换为歌词即可配合音乐实现3D歌词展示



- 代码说明

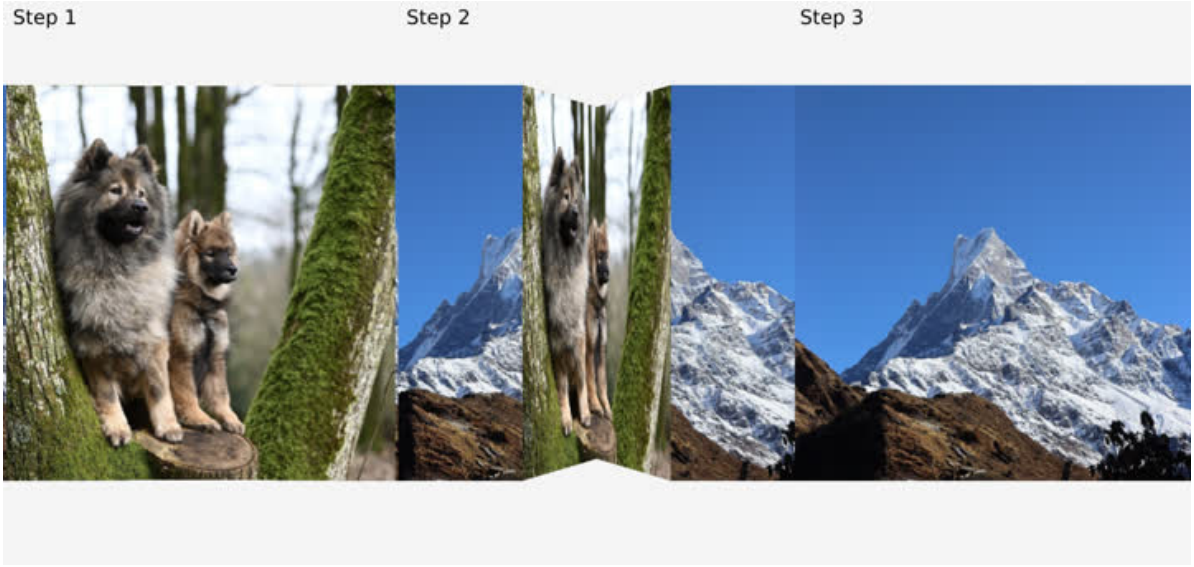
```
/* anims/cube_lyric.c */
/*
 * 定义额外的两个子动画。应用中将每次旋转，每行文字的渲染都分为子动画去完成
 * 动画从左面开始，主动画完成左面的两行绘制
 * 主动画完成后使能第一个旋转子动画，从左面旋转到右面
 * 旋转完成后使能第一个渲染子动画，完成右面的两行绘制
 * 绘制完成后使能第二个旋转子动画，如此循环直至6个面绘制完成
 */
static lv_anim_t sub_anims[]...
/* 表示当前绘制进行到第几行的第几个字 */
static int line;
static int row;
static int col;
/*
 * 渲染回调，和普通的立方体不同，多了一步在立方体上绘制，因此独立一个回调
 * 在这个回调中
 * 第一步先将立方体设置为framebuffer，表示后续的渲染都会绘制到立方体上
 * 第二步绘制当前字符
 * 第三步将framebuffer重置，表示后续的渲染直接显示到屏幕上
 * 第四步将立方体渲染到屏幕上
 */
void anim_cube_lyric_render(void)
/*
 * 起始回调，先调用立方体通用起始回调，并将渲染回调设置为anim_cube_lyric_render
 * 再将立方体旋转到左面，方便后续动画进行
 */
void anim_cube_lyric_start(lv_anim_t *a)
/* 绘制动画回调，根据当前进度，更新字符索引，触发渲染 */
void anim_cube_lyric(void *var, int32_t v)
/* 旋转动画回调，根据当前进度，旋转立方体到合适的面 */
void anim_cube_lyric_sub(void *var, int32_t v)
```

```
/* 旋转动画结束回调，根据下一行文字的长度，更新绘制动画的时长（按每字300ms计算），并启动绘制动画 */
void anim_cube_lyric_sub_end(lv_anim_t *a)
/*
 * 绘制动画结束回调，每次绘制完行数加一
 * 如果行数是奇数，说明当前面还有一行未显示，根据下一行文字的长度
 * 更新绘制动画的时长（按每字300ms计算），并启动绘制动画
 * 如果行数是偶数，说明当前面已绘制完成，启动旋转动画
 * 如果所有12行都绘制完成，则结束动画
 * DEMO中共12行，每面2行，如果是其他行数规划，则逻辑会有不同
 */
void anim_cube_lyric_end(lv_anim_t *a)
```

7.3.4 图片折叠

- 效果说明

图一向内折叠退出，图二分两部分从画面外向内位移



- 代码说明

```
/* anims/fold.c */
/* 渲染回调。应用将两张图分为四个部分（每张都分为左右两部分），因此在渲染回调中调用了四次渲染 */
static void anim_fold_render(void)
/*
 * 起始回调
 * 1 将视图设置为480x480（图片大小，可选，根据实际情况调整）
 * 2 根据新视图重置4个渲染对象的缩放参数
 * 3 将图片切分为4块，obj_fold[0]为图一左半，obj_fold[1]为图一右半，以此类推
 * 4 设置渲染回调
 */
void anim_fold_start(lv_anim_t *a)
/*
 * 动画回调
 * 由于该需求使用旋转变换相对复杂，因为旋转轴在不停发生变化，且需要根据偏移值反推旋转角度等
 * 因此通过直接设置顶点坐标来实现，只需要在动画回调中不停更新四个对象各个顶点的x轴坐标即可
 * 而obj_fold[0]的右边两点和obj_fold[1]的左边两点是固定向内折且不变的
 * 因此这4个点的x坐标不需要改变，仅需将z坐标设置为负值即可（目前是设定为0~-0.2，根据效果调整）
```

```
 * 顶点着色器内的透视变换会自动变换出近大远小的效果
 */
void anim_fold(void *var, int32_t v)
/* 结束回调 */
void anim_fold_end(lv_anim_t *a)
```

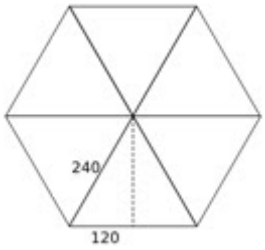
7.3.5 滚桶图片预览

- 效果说明

六张图片构成一个桶形，旋转展示每张图片

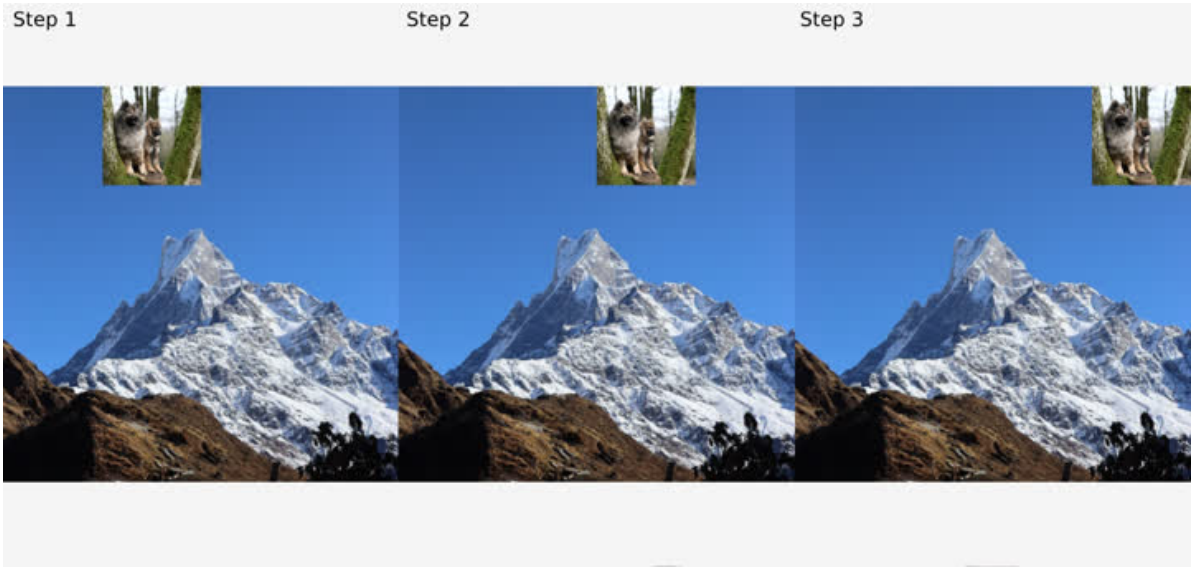
- 代码说明

```
/* anims/roller.c */
/* 渲染回调，将6张图片依次渲染 */
void anim_roller_render(void)
/*
 * 起始回调
 * 1 将视图设置为2000x2000，因为涉及到旋转，因此为了避免图片超出坐标造成裁切丢失画面，设为一个较大值
 * 2 将渲染对象根据新视图重新计算大小，并且将Z轴偏移设为0.20785，使图片距离中心有一定距离，
 *   从而形成桶形。Z轴计算为简单的勾股定理，sqrt(240^2 - 120^2) / 2000 * 2.0 = 0.20785
 *   图片宽240（见下图），由于正好是六边形，因此简单的sqrt(240^2 - 120^2) 即可计算出每个面到
 *   中心的距离，2000是视图大小，由此算出缩放比例，再乘上范围2.0（坐标-1.0~1.0）即可得到偏移值
 *   如果不是六边形，则需要通过其他方式计算距离，计算缩放比例和偏移值的方式不变
 * 3 将X轴旋转设为30，Z轴旋转设为15，为了有一个倾斜的效果，根据需要随意设置
 */
void anim_roller_start(lv_anim_t *a)
/* 动画回调。为每张图片Y轴加上旋转值，第一张图为0度加上旋转值，第二张图为60度加上旋转值，依次类推 */
void anim_roller(void *var, int32_t v)
/* 结束回调 */
void anim_roller_end(lv_anim_t *a)
```



7.3.6 贴图测试

- 在一个背景图上，每次按一小格步进一段距离，贴上一张小图



- 代码说明

```
/* anims/stiker.c */
/*
 * 渲染回调
 * 1 将obj_fb设为framebuffer
 * 2 将obj_fb清除为(0.0, 0.0, 0.0, 1.0)，因为混合方式中，最终的alpha为d.a，即底图的
alpha，
 *   因此这边需要设为1.0，否则图片将透明或不可见
 * 3 将obj_img1渲染到obj_fb上作为背景图
 * 4 将obj_img0渲染到obj_fb上作为贴图
 * 5 重置framebuffer，将obj_fb渲染到屏幕上
 * 实际使用中，可以将两张图片直接渲染到屏幕上，DEMO只是为了演示合成功能及framebuffer功能
 */
static void anim_stiker_render(void)
/* 起始回调。将贴图的坐标设为0,0，并调用lv_gl_obj_move更新参数值 */
void anim_stiker_start(lv_anim_t *a)
/* 动画回调。根据当前动画进度，设置贴图坐标，并调用lv_gl_obj_move更新参数值 */
void anim_stiker(void *var, int32_t v)
/* 结束回调 */
void anim_stiker_end(lv_anim_t *a)
```

7.3.7 滑动退出

- 效果说明

一个简单的滑动退出的DEMO，如图所示，图一向左逐渐滑出画面，同时图二向左滑入画面

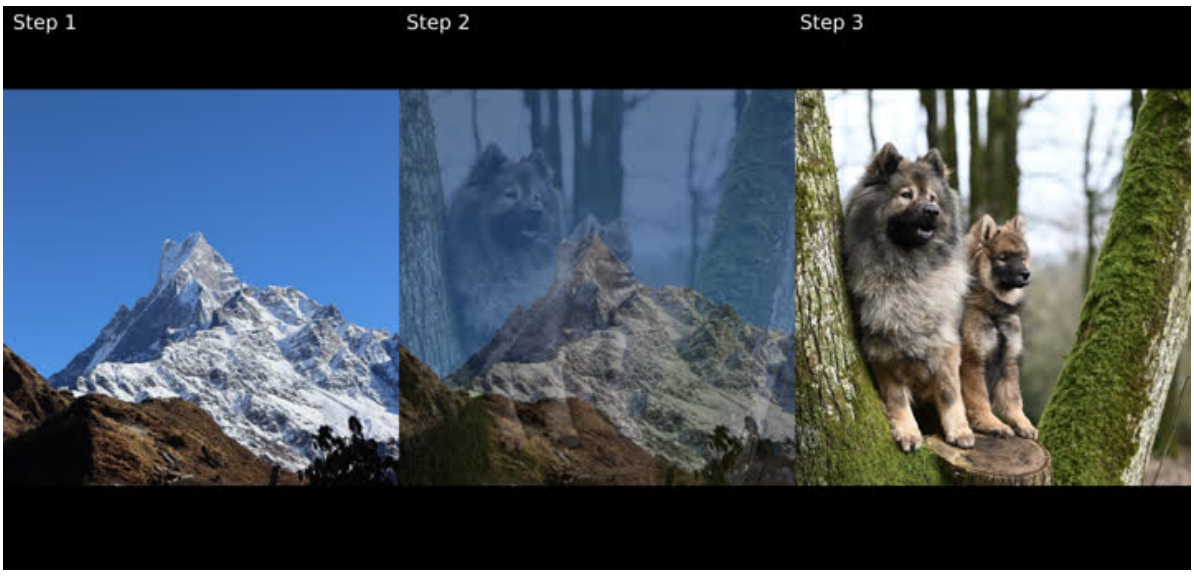
- 代码说明

```
/* anims/slide_out.c */
/* 调用通用的起始回调，并将动画区域设为可见 */
void anim_slide_out_start(lv_anim_t *a)
/* 动画回调，根据当前动画进度更新两张图片的x坐标 */
void anim_slide_out(void *var, int32_t v)
/* 结束回调 */
void anim_slide_out_end(lv_anim_t *a)
```

7.3.8 图片淡出

- 效果说明

一个简单的淡入淡出DEMO，如图所示，图一透明度降低逐渐淡出画面，同时图二增加透明度逐渐淡入画面



- 代码说明

```
/* anims/fade_out.c */
/* 调用通用的起始回调，并将动画区域设为可见 */
void anim_fade_out_start(lv_anim_t *a)
/* 动画回调，根据当前动画进度更新两张图片的透明度 */
void anim_fade_out(void *var, int32_t v)
/* 结束回调 */
void anim_fade_out_end(lv_anim_t *a)
```

7.3.9 滑动淡出

- 效果说明

将前两种动画结合，图一滑出的同时降低透明度，同时图二滑入画面

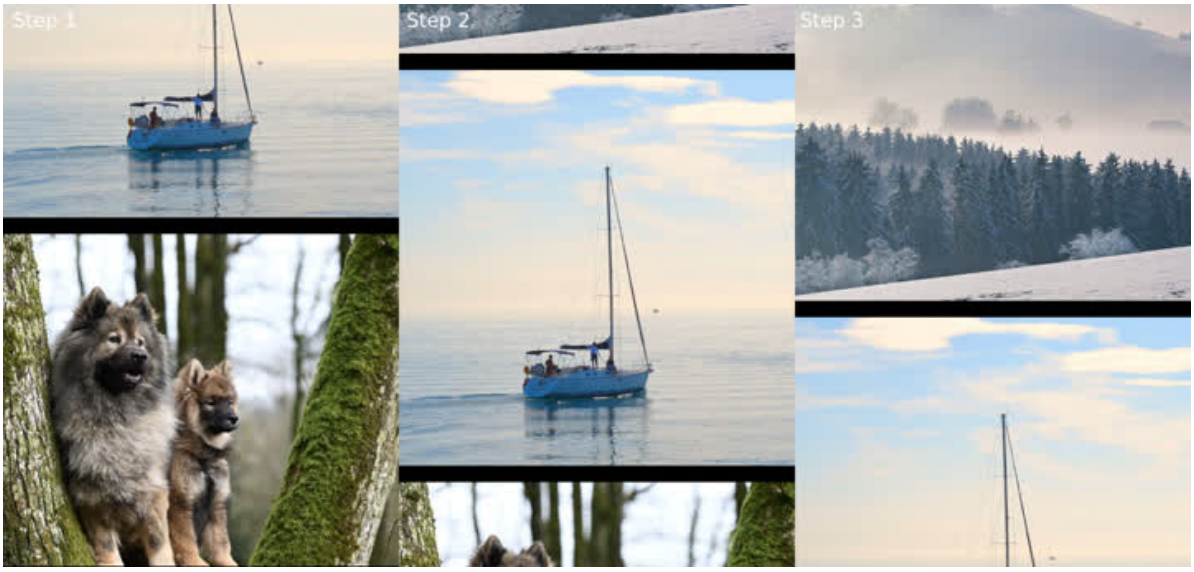
- 代码说明

```
/* anims/fade_slide_out.c */
/* 调用通用的起始回调，并将动画区域设为可见 */
void anim_fade_slide_out_start(lv_anim_t *a)
/* 动画回调，根据当前动画进度更新两张图片的x坐标及图一的透明度 */
void anim_fade_slide_out(void *var, int32_t v)
/* 结束回调 */
void anim_fade_slide_out_end(lv_anim_t *a)
```

7.3.10 照片流

- 效果说明

6张图片由上往下滚动播放，触摸或鼠标点按时暂停动画并随鼠标移动，放开后继续动画



- 代码说明

```
/* anims/photo_stream.c */
/* 图片高度 */
static int32_t img_h = 480;
/* 图片间隔 */
static int32_t gap_h = 20;
/*
 * 根据图片高度、图片间隔、图片数量计算出最大的Y坐标
 * max_y = n * img_h + (n - 1) * gap_h
 */
static int32_t max_y;
/*
 * 边界值，与max_y类似，用于在图片到达max_y后重置到照片流顶部继续循环
 * boundary = n * img_h + n * gap_h
 * 由于图片循环后需要回到顶部继续轮转，且与第一张图还需要有一个间隔，因此boundary比max_y多一个gap_h
 */
static int32_t boundary;
/* 定时更新图片坐标 */
static lv_timer_t *timer;
/* 定时器定时增加的坐标偏移 */
static int32_t timer_ofs = 0;
/* 上一次触摸增加的坐标偏移 */
static int32_t touch_ofs_s = 0;
/* 本次触摸增加的坐标偏移 */
static int32_t touch_ofs = 0;
/* 更新并检查图片Y坐标，实现动画效果 */
static void update_y(void)
/* 定时器回调，timer_ofs自增 */
static void lv_timer_cb(lv_timer_t *timer)
/*
 * 触摸回调
 * 在LV_EVENT_PRESSED回调中暂停定时器，并获取按下的坐标
 * 在LV_EVENT_PRESSING回调中获取当前触摸坐标，和按下坐标对比加上touch_ofs_s计算出偏移值并更新坐标
 * 在LV_EVENT_RELEASED回调中重新启用定时器，并将此次触摸偏移量记录到touch_ofs_s
 * 之所以分了三个偏移量，是为了保证触摸按下和触摸释放时的动画坐标连续，避免出现坐标瞬移
 */
static void touch_handler(lv_event_t * e)
/* 停止照片流 */
```

```
void anim_photo_stream_stop(void)
/* 照片流初始化，创建定时器，计算出max_y，boundary等 */
void anim_photo_stream_start(lv_anim_t *a)
/* 动画回调
 * 实际上照片流是使用定时器实现的，与动画无关，这边使用动画接口只是为了接口统一
 * 方便嵌入到DEMO框架中，实际应用开发使用定时器即可
 */
void anim_photo_stream(void *var, int32_t v)
void anim_photo_stream_end(lv_anim_t *a)
```