滤镜功能学习记录

|  |  |  |  |
| --- | --- | --- | --- |
| **Document Number:** |  | **Document Version:** |  |
| **Owner:** | Jeff Wang | **Date:** | 2018-02-01 |
| **Document Type:** |  | | |
| **NOTE:** | ALL MATERIALS INCLUDED HEREIN ARE COPYRIGHTED AND CONFIDENTIAL UNLESS OTHERWISE INDICATED. The information is intended only for the person or entity to which it is addressed and may contain confidential and/or privileged material. Any review, retransmission, dissemination, or other use of or taking of any action in reliance upon this information by persons or entities other than the intended recipient is prohibited.  This document is subject to change without notice. Please verify that your company has the most recent specification.  Copyright © 2014-2017 Spreadtrum Communications Inc. | | |



[www.spreadtrum.com](http://www.spreadtrum.com)

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | Author | Description |
| 0.1 | 2018-11-9 | jeff wang | Draft |
| 0.2 | 2018-12-3 | jeff wang | Modify … |
| 0.3 |  |  | Second review |
| 1.0 |  |  | Applicability described. |

Table of Contents

1. Introduction 4

1.1. Request & Purpose 4

1.2. Definitions &Abbreviations 4

1.3. Reference 4

2. Overview 4

3.学习记录 5

3.1. Demo 工程结构 5

3.2.调试方法 5

3.3.单幅图像的处理流程 6

3.4.重点函数的解读 8

3.5.查找表 9

3.6.多维查找表在滤镜功能中的应用 11

3.7.查找表怎么来的 13

3.8.对现有代码的优化方案 13

3.9.优化评价方法及优化结果 15

3.10.预留待补充 16

4.一些备忘 17

4.1.一些关键log 17

4.2.开关配置 18

4.3.预留待补充 19

# Introduction

学习滤镜算法时做的记录和总结。

## Request & Purpose

记录自己在学习中遇到的、看明白的问题和新学到的知识。

## Definitions &Abbreviations

|  |  |
| --- | --- |
| **API** | Application Programming Interface |
|  |  |
|  |  |
|  |  |
|  |  |

## Reference

[1] code

# Overview

学习滤镜算法时做的记录和总结。

# 学习记录

## Demo 工程结构

从jni/Android.mk中可以看到，这个工程编译出的内容有两部分：

第一部分是算法库，LOCAL\_MODULE := libSprdImageFilter

第二部分是调用该算法库的可执行程序，LOCAL\_MODULE := test\_imagefilter

这句指明了该程序要调用的库：LOCAL\_SHARED\_LIBRARIES += libSprdImageFilter

安装并配好NDK后，cd到工程的jni中，用"ndk-build"命令进行编译

结果生成在工程的/obj/local/目录中，有4组结果，对应不同的CPU架构，分别为：

"arm64-v8a" , "armeabi-v7a" , "x86" , "x86\_64"

这是在/jni/Application.mk中配置的：

APP\_ABI := armeabi-v7a arm64-v8a x86\_64 x86

对于它的解释，可参考

https://blog.csdn.net/qq\_31387043/article/details/50899589

目前Android系统支持以下七种不同的CPU架构，所以编出来的东西只能在android上跑，电脑上不行。

## 调试方法

在9.0 SharkLE上调试，用"armeabi-v7a"目录中的程序和so

编译成功后，先将test\_imagefilter push到手机的data目录下，并adb shell进到data目录去chmod 777 test\_imagefilter，让它可被执行。

它的入口是代码中的main函数，代码中可以看到，它执行时要去读data目录下的1.bmp文件，所以还要将1.bmp也放到手机的data目录中

这时候并没有push库到手机里，目的是想看需要调用到库里的函数却没有库，会报什么错误

例如：main中会调用ImageFilterCreateEngine()函数，它的实现在Imagefilter\_interface.cpp

它在LOCAL\_MODULE := libSprdImageFilter的LOCAL\_SRC\_FILES列表中，所以这个函数会被封装到libSprdImageFilter.so中，main调用到它时如果找不到库，是应该报错的

但实际运行发现它并没有报错，ImageFilterCreateEngine()函数能正常地执行过去。

为了查清这个问题，可以在main中调用ImageFilterCreateEngine()以后，写一个死循环，目的是让main不退出。

push , chmod(每新push一次imagefilter都要进行chmod) , 运行...

再开一个terminal，adb shell进去，ps -A | grep "test\_imagefilter"，可以找到该程序的进程号(例如：29630)

cd proc/29630

cat maps

在列出的内容中，搜索库名"libSprdImageFilter",可以搜到：

ed4ca000-ed4d1000 r-xp 00000000 103:18 864 /vendor/lib/libSprdImageFilter.so

ed4d1000-ed4d2000 r--p 00006000 103:18 864 /vendor/lib/libSprdImageFilter.so

ed4d2000-edb93000 rw-p 00007000 103:18 864 /vendor/lib/libSprdImageFilter.so

**也就是说，它调用到了/vendor/lib/libSprdImageFilter.so**

因为我们手机中本身就有这个库，DreamCamera的滤镜功能用的就是这个库。

为了既能进行这个工程的调试学习，又不影响camera的滤镜功能，可以在工程的mk中将库换一个名字，例如：

libSprdImageFilterXX，上面提到的两处都要改

修改后，会编译出libSprdImageFilterXX.so,将它和test\_imagefilter一起push到手机data目录中并运行，报错：

CANNOT LINK EXECUTABLE "./test\_imagefilter": library "libSprdImageFilterXX.so" not found

这次是真找不到了。换个地方试试，把库放到system/lib，可以正常运行了。（放vendor/lib也可以）

至此，工程的程序和库都可以正常工作了，可以开始加log或其他修改的尝试了。（记得把刚加的死循环去掉）。

## 单幅图像的处理流程

从main()中看，调用的库函数有3个：

ImageFilterCreateEngine

ImageFilterRun

ImageFilterDestroyEngine

这三个函数是从hal代码中调用的,位置在sprd/modules/libcamera/oemxvx/src/sprd\_filter.c中，分别被sprd\_filter\_create\_engine(…) , sprd\_filter\_doeffect(…) , sprd\_filter\_distroy\_engine(…)调用，对于调用前的参数准备工作，可以参考hal的代码。

一般来说，图像处理算法都是这个套路，初始化、处理、释放，逐个看：

ImageFilterCreateEngine

返回值类型是IFENGINE，它的定义：typedef void \* IFENGINE，就是个指针。看代码中实际返回对象的类型是：

typedef struct

{

IFInitParam initParam;

// engin的初始化参数，和这个方法的输入参数是同样的类型；

unsigned char \*buffer[MAX\_BUFFER\_NUM]; // 用于存放图像数据的buffer，NUM为2

ImageFilterParam defParam;

// 滤镜参数的union，用于颜色变换的查找表等

CThread\_pool \*pool;

// 为优化执行速度，将全图的处理分为多线程来做，线程数定义为4；这个参数是线程相关的参数

} ImageFilterEngine;

参数：IFInitParam\* , 它的定义：

typedef struct

{

IFImageType imageType; // 对应一个枚举，表示数据格式，NV21或RGB888

int imageWidth;

// 图像的宽高

int imageHeight;

char \*paramPath;

// 一个路径，存放的是滤镜查找表

} IFInitParam;

这个函数做的3件事：

1. 为要返回的engine申请内存；

2. 将入参的内容赋值给出参的initParam；

3. 按照图像尺寸申请buffer，所需字节数为：宽x高x3x2

4. 初始化线程池，4线程

只有 defParam 参数暂时未作处理

ImageFilterDestroyEngine

参数：IFENGINE ， (void)  , 实际用的就是ImageFilterCreateEngine()中创建出的ImageFilterEngine对象

这个函数做的几件事：

1. 将初始化时申请的buffer释放掉

2. 将线程池占的内存释放掉

3. 将engine占用的内存释放掉。

## 重点函数的解读

线程相关的内容之后再看

先看滤镜处理部分：ImageFilterRun

参数：

IFENGINE engine,   // 前面初始化好的engine

IFImageData \*input, // 输入图像

IFImageData \*output,  // 输出图像

IFFilterType filterType,  // 滤镜编号，每种效果一个编号

IFFilterParam \*filterParam // 参数，这个参数可缺省，实际调用时也没传这个参数

它只做了一件事：check各参数合法之后，按入参原封不动地将任务dispatch出去：ImageFilterRunDispatch(...)

看ImageFilterRunDispatch做了什么：

ImageFilterGetParam(...)

创建4个元素的数组，每个元素是一组算法参数，对应一个线程；

将图像数据按高度4等分，4个元素之间只有.startRow不同(对应的.endRow也不同，但各子线程处理的高度相同)；

将线程用 pool\_add\_worker(...)方法添加到线程池，这个方法的参数包括：

pool: 创建并初始化好的线程池；

process: 线程要执行的操作；

arg: process要用到的参数；

用循环分别对4个线程进行设参数、pool\_add\_worker操作；

在循环外等4个线程都结束： sync\_thread\_done(pool , 4);

\*\*\* 最关键的process参数，对应的是ImageFilterSlice(...)函数 \*\*\*

看看它里面做了什么：

1. 算法用RGB数据计算，如果不是RGB，会先进行格式转换；

2. 根据filterType，去走对应的IFxxxFilter()方法；  \*\*\*

3. 和1对应的，如果不是RGB，再将处理后的图像数据转回原来的格式；

4. 对本线程做一个完成的标记 (计数++)；

2是关键，目前demo中（也是当前我们手机上用到的9种效果）走到的都是 IFLookupFilter(...)

函数中用到的lut(查找表，Look Up Table)是一个 512 x 512 x 3 的 unsigned char数组

// B-Quad G-Row R-Col

## 查找表

插播广告：什么是查找表？怎么用查找表？为什么用查找表？

是什么？

查找表(LUT: Look Up Table)，顾名思义它是个表，通过下标去得到对应的元素值。

怎么用？

在图像处理中，通常将"像素值"当作下标，其在表中对应的元素，是这一像素值经过某种计算后得到的结果。

为什么用？

对于过程较为复杂且输入有特定范围的算法，查找表可以离线生成，在线过程只关心输入和输出，根据输入直接寻址得到输出，速度更快。

以单通道灰度图像举个实例：

目的是提高图像的亮度，映射关系为：，X的取值范围是[0 , 255]的整数。

给定一个x，求对应的y的计算量为：

一次浮点除法，一次开平方，一次浮点乘法，一次取整，记为A

考虑到全图的像素数(width \* height)要远大于256，所以逐像素进行上述计算的过程中一定会产生大量的重复计算，即：同一个x值被多次重复计算。

这样的重复计算对硬件资源和时间都造成了浪费，所以这种情况可以用查找表来进行优化。

先计算出一个256个元素的数组lut，作为查找表，其元素按照上面的公式进行计算得到。

这一过程是离线进行的，不占任何时间，空间上增加256单位的内存消耗。

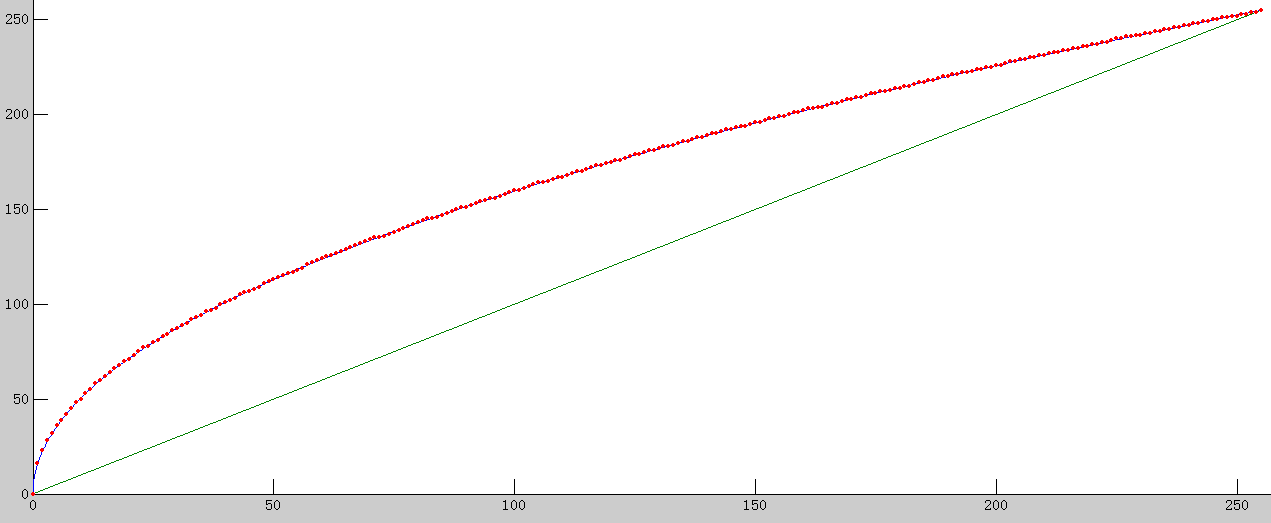
计算过程中将像素p值作为下标，则lut[p]即为其对应的处理结果。

其过程相对A来说避免了一系列的耗时操作，从而实现了速度优化，并且结果不变。

下图中绿色直线为y=x；

蓝色曲线为：的函数图像，其中y>=0;

红色离散点为按计算得到的查找表中的对应元素



下图为算法处理前后的对比图像：



Lut demo的Matlab代码请见：

## 多维查找表在滤镜功能中的应用

滤镜功能的IFLookupFilter()函数中用到的lut是一个 512 x 512 x 3 的 unsigned char数组

作为rgb数据的查找表，理论上(1对1，即：1组rgb输入对应1组rgb输出)的规模应该是 256 x 256 x 256 x 3，其占用内存为48M，目前DreamCamera中支持9种滤镜效果，理论上查找表占用的内存就是48x9=432M，这样的内存消耗是不可能在产品上落地的。

当前滤镜的优化方案是：将256降采样为64，一个查找表的大小变为 64 x 64 x 64 x 3 = 768K，9个效果为768K x 9 = 6.75M , (编出的so大小为7.1M，大部分用于存查找表)

对算法的优化，实质上是时间、空间、效果3者之间的权衡。

举例：将查找表的RGB3个维度理解成为一个3维坐标，那么：

查找表lut规模为256 x 256 x 256 x 3，输入像素值为R\_in , G\_in , B\_in时：

输出应该为：

R\_out = lut[R\_in][G\_in][B\_in][0]

G\_out = lut[R\_in][G\_in][B\_in][1]

B\_out = lut[R\_in][G\_in][B\_in][2]

为了便于叙述和理解，先将RGB3个分量视作一个整体，做如下想象：

将R和G的256x256理解为一张纸上写了一个256x256的矩阵 (矩阵的每个元素对应一组rgb)；

将B的256理解为有256张这样写着矩阵的纸摞起来放；

这一摞纸就是查找表，我们拿着像素值去查表，要做的就是找到对应的那一张纸上的对应的那一行那一列的元素；

将表的规模降为 64 x 64 x 64 x 3 后，输入像素值仍为R\_in , G\_in , B\_in时：

输出应该是在一个范围内做插值得到，这个范围是一个三位坐标系中的立方体区域，8个定点分别为：

lut[R\_in / 4 + x][G\_in / 4 + x][B\_in / 4 + x]，其中x为0，1两个值在3个维度上的共8种组合。

再对应到"一摞纸"的描述，相当于：

1.每4张纸为一组，留下第1张，扔掉其他3张; // B从256降到了64

2.每张纸上的元素，每4行为一组，留下第1行，删掉其他3行; // G从256降到了64

3.每张纸上的元素，每4列为一组，留下第1列，删掉其他3列; // R从256降到了64

因为我们丢掉了很多数据，留下的数据相较完整的查找表，数据量是原来的1/64，所以在查表时，只有1/64的概率能直接查到准确值，剩下63/64的概率都不能，必须做插值。

插值可以理解为：在不能直接查表得到准确值的情况下，查出来本来应该对应位置结果附近的几个值，再进行计算得到近似结果。

(插值，既和一对一查找表的效果有差异，又要额外花时间计算。这就体现出了优化中空间和效果、空间和时间之间的权衡)

目前滤镜功能中，对R和G用的是最近邻插值；对B用的是线性插值。举例说明两种插值方法：（查找表规模为64，下标为[0~63]）

最近邻插值：假设r=203，203除以4 = 50.75，在参考50和51两个值的情况下他离51更近，所以对203直接以51为下标去查表。其实也就是所谓的"保留整数，小数点后四舍五入"。对于202，50.5同样按51算。

在实际计算过程中常用的四舍五入技巧为：index = (value + 2) / 4; // 加的数为除数的一半

线性插值：假设b=203，除以4=50.75，参考50和51两个值的结果，以理论下标到两个参考下标的距离作为权重，做加权平均。即：f(203) = lut[50] \* 0.25 + lut[51] \* 0.75;

实际计算过程中常用的加权平均技巧(以按除以4采样为例)为：

index = value / 4;

w1 = value % 4;

w2 = 4 - w1;

f(value) = (lut[index + 1] \* w2 + lut[index] \* w1) / 4;

在当前滤镜的代码中，查找表除了被降为了 64 x 64 x 64 x 3 之外，在读取时还做了个形变，变为了：512 x 512 x 3

typedef struct

{

// B-Quad G-Row R-Col

unsigned char val[IFLUT64\_SIZE][IFLUT64\_SIZE][3];// 512 x 512 RGB texture

} IFLUT64;

这个变换还可以用前面那"一摞纸"来理解：

查找表本来是"一摞纸"，现在我们将他们一张挨一张地放到桌面上。1，2，...，7，8，放满8张，开始另起一行再放；

原来的64张纸正好放满8行8列；

原来每张纸上是一个64x64的矩阵，现在就变成了一个大的 (64x8) x (64x8) 矩阵，也就是512 x 512。每个矩阵元素是一组rgb，也就是 512 x 512 x 3 了。

至于为什么要做这个变换，通过咨询了解到的一个原因，是因为为了和预览统一：

预览时用的是openGL ES的那套体系，512 x 512 x 3 的数据可以当作一个纹理(图)作为参数，所以拍照也就随着改了。至于3维数组是否比4维数组寻址快，没有做具体研究。

(反正优化方案里干脆连数组都不用了，直接改成了指针，几维根本不care)

以上述内容为辅助，应该就能完全理解目前代码中的IFLookupFilter()函数了。

## 查找表怎么来的

可以自己写图像处理算法，也可以借助工具(如PS)通过可视化界面调节并导出。当前版本上的9种效果都是用后者做的。

用PS CC及以上的版本可以导出查找表，后缀为.cube的文本文件。导出时可以选择查找表的"Grid Points"，包括256、64等，我们的项目中用64。



PS相关操作可以参考：

http://www.filmaker.cn/thread-55207-1-1.html

http://www.16xx8.com/photoshop/jiaocheng/2016/140982.html

http://bbs.fengniao.com/forum/3631945.html

导出的文件以文本形式，记录了查找表的数据，以64为例，共262144组数据，每一组数据是3个float值，取值范围从0.0~1.0。实际应用的时候需要乘以255并取整。

我们的代码中用到的是整数，并且是512 x 512 x 3的组织方式，所以需要将.cube处理成我们需要的形式。CubeConvert.py这个python脚本来做这件事。

 , 

由于PS版本问题，目前还没有用创建出来的lut成功地处理照片，结果上有色块。当前是cc 2014，正在下载2019版本。​

## 对现有代码的优化方案

主要的优化都在函数IFLookupFilter()中。

1. 对2的整数次幂的数进行乘除计算，可以用左右移位代替乘除。(编译器可能会自动优化，也可能不会，所以还是自己改吧)
2. 循环中的判断尽量去掉，例如代码中的：

int r\_round = (r\_in + 2) / 4;

if (r\_round == 64)

r\_round = 63;

这种计算r\_round的方法，符合前面提到的查找表适用的“输入有特定范围”的特征，应该先将r\_in从0~255对应的r\_round结果都计算出来作为一个查找表int rRound[256]存起来，那么上述过程可以被简化为：

int r\_round = rRound[r\_in];

或者是指针形式：

int r\_round = \*(pRRound + r\_in); // pRRound是指向rRound数组的指针；

1. 重复的计算尽量去掉，例如代码中的：



循环中，相同的乘法计算重复做；

对于这种问题，通常的方法是将重复的部分赋值给一个变量，目的是只计算一次，不做无用功。



循环中，相同数组下表的寻址重复做；

数组按下表取值，其实就是寻址操作，其偏移量是根据下表和数组的该维度在内存中的跨度计算得到，所以维数越多取值时计算量越大。

上面的代码，rgb其实是相邻的3个内存单元，完全可以代替为：r寻址，然后+1得到g，再+1得到b；以达到减少重复计算的目的，从而实现优化。

1. 用指针代替多维数组

这个算法中的多维数组是个三维查找表，因为我们清楚地知道表的结构为512x512x3，所以完全可以将这个表用一维数组表示，举例：lut[y][x][2]，如果简化为一维数组，其下表应该为：index = y \* (512 \* 3) + x \* 512 + 2;而一维数组的另一种表现形式是指针，

\*(pLut + index);

并且在index的计算过程中，也要注意避免重复计算。

1. 二重循环简化为一重

这个算法中是逐像素进行处理，并且处理过程中不参考邻域像素，也就是说，这个算法完全就是一个在内存中按顺序逐字节处理的过程。

根据这一特性，可以发现用二重循环来标识行和列的做法，可以被简化为一重循环。

循环体中的末尾，对指针进行后移一单元的操作，可完全忽略所谓“下一行”的问题；

1. 对加权平均的优化

这个算法中是对两个值做加权平均，(v1 \* w1 + v2 \* w2) / m , 所以其计算量为：

2次乘法，1次加法，1次除法；记为A；

具体分析，其实w1w2的组合只有4种：04，13，22，31

其中只有13和31两种情况，计算量为A；

04时，因为w1为0，所以其结果就是v2，不用计算；记为B；

22时，因为w1和w2相等，所以其结果是 (v1 + v2)/ 2 ; 计算量为1次加法，1次除法；记为C；

也就是说BC的计算量都比A小，一个很容易宣想到的方法是：对w1(或w2也可以)做switch，针对不同的case用不同的方法去计算。

这种方法是没问题的，但需要说明一个情况：

第一：每个像素都要经历这么一个过程，也就是说每个像素都会比原来多一次判断，这会带来一个必然的时间成本的增加；

第二：如果判断后的结果是B或C的计算量，那么在时间上有可能是“赚”的；但如果结果还是要走A，那么就是稳“赔”。

这个判断条件是跟像素值有关的，而图像内容是不可预知且随机的，所以这个优化不好预估其效果，只能做理论上的分析。

上述的几个优化点，可参考新添加的IFLookupFilterOpt()函数。

还有一个要注意的问题是，上面的几个优化点都是时间与空间的权衡，于效果无关。换句话说，所做的优化是不允许影响处理后图像的效果的。也就是说，用相同的输入去跑优化前后的两个函数，所输出的图像数据逐字节做比对，必须完全相同。这一检查机制通常称为bit match。所以在算法优化coding的过程中，每做一步优化修改，除了关注耗时外，还必须进行bit match的检查，以确保优化的修改没有影响算法的图像处理效果。

## 优化评价方法及优化结果

实验用的是9.0 SharkLE (2G RAM)，通过DreamCamera的Filter模式拍照。

过滤的log为：

adb logcat -c ; adb logcat | grep -e " Filter: " -e "ImageFilterRun:"

每次拍照会打出如下log：

01-15 07:59:30.800   339  5024 D imagefilter-v0.3.1: ImageFilterSlice 2, Filter: 137.23 ms

01-15 07:59:30.814   339  5023 D imagefilter-v0.3.1: ImageFilterSlice 0, Filter: 149.79 ms

01-15 07:59:30.891   339  5022 D imagefilter-v0.3.1: ImageFilterSlice 1, Filter: 211.13 ms

01-15 07:59:30.900   339  5021 D imagefilter-v0.3.1: ImageFilterSlice 3, Filter: 207.29 ms

01-15 07:59:30.912   339  4954 I imagefilter-v0.3.1: ImageFilterRun: 261.09 ms(21)

前面4行为执行lut查表的4个线程各自的耗时；

最后一行是ImageFilterRun函数执行的耗时

分别用未经优化和优化过后的两个so文件进行测试，filter模式下拍照20次。

不包含进入filter模式后的第一次拍照，那次拍照有初始化操作，相较之后的拍照耗时较长。

测试结果的详细数据在time\_cost.txt和time\_compare.xlsx中

，

结论为：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 优化前 | 优化后 | 优化幅度 |
| 20x4线程平均耗时 | 165.8218 | 116.8423 | 29.54% |
| 20次ImageFilterRun平均耗时 | 261.2105 | 216.5935 | 17.08% |

## 预留待补充

# 一些备忘

## 一些关键log

判断每帧处理时间的关键log：

01-01 08:50:47.752 5467 5467 V ArcGlSurfaceView: onFrameAvailable

01-01 08:50:47.754 5467 5675 V ArcFilterRenderer: onDrawFrame(): Enter

01-01 08:50:47.765 5467 5675 I ArcFilterRenderer: onDrawFrame(): finished, spendTime = 2 Exit with res = 0

ArcFilterRenderer相关log：

14764 14764 I ArcGlSurfaceView: new ArcFilterRenderer

14764 14764 I ArcFilterRenderer: Init(): Enter

14764 14764 I ArcFilterRenderer: Init(): Exit

14764 22943 V ArcFilterRenderer: onSurfaceCreated(): Enter

14764 22943 V ArcFilterRenderer: createOESTextureObject() finish

14764 22943 V ArcFilterRenderer: onSurfaceCreated(): Exit

14764 22943 V ArcFilterRenderer: onSurfaceChanged(): Enter, width = 720 height = 1280

14764 22943 V ArcFilterRenderer: m\_ArcFilterEng.prepareEngine() res =0

14764 22943 V ArcFilterRenderer: onSurfaceChanged(): Exit

14764 22943 V ArcFilterRenderer: onDrawFrame(): Enter

14764 22943 V ArcFilterRenderer: wait preview start

14764 22943 V ArcFilterRenderer: onDrawFrame(): Enter

14764 22943 V ArcFilterRenderer: wait preview start

14764 22943 V ArcFilterRenderer: onSurfaceChanged(): Enter, width = 720 height = 960

14764 22943 V ArcFilterRenderer: onSurfaceChanged(): Exit

14764 22943 V ArcFilterRenderer: onDrawFrame(): Enter

14764 22943 I ArcFilterRenderer: onDrawFrame(): finished, spendTime = 23 Exit with res = 0

14764 22943 V ArcFilterRenderer: onDrawFrame(): Enter

14764 22943 I ArcFilterRenderer: onDrawFrame(): finished, spendTime = 27 Exit with res = 0

14764 22943 V ArcFilterRenderer: onDrawFrame(): Enter

14764 22943 I ArcFilterRenderer: onDrawFrame(): finished, spendTime = 4 Exit with res = 0

14764 22943 V ArcFilterRenderer: onDrawFrame(): Enter

14764 22943 I ArcFilterRenderer: onDrawFrame(): finished, spendTime = 8 Exit with res = 0

滤镜拍照过程中的几个明显log：

01-03 05:22:47.698 15035 15035 I CAM\_PhotoModule: onShutterButtonClick mPaused:false,mCameraState:1 isShutterEnabled=trueisFreezeFrameDisplay=false mCaptureCount = 0

01-03 05:22:47.699 15035 15035 I CAM\_PhotoModule: onShutterButtonClick: mCameraState=1 mContinuousCaptureCount=1

01-03 05:22:47.702 15035 15035 I CAM\_PhotoModule: focusAndCapture

01-03 05:22:47.703 15035 15035 I CAM\_PhotoModule: capture

01-03 05:22:47.703 15035 15035 I CAM\_FilterModuleAbs: updateFilterType filterType = 268435458 mPPEffectType = 31

01-03 05:22:47.705 15035 15035 I CAM\_PhotoModule: sensorOrientation = 90 ,deviceOrientation = 0 isFrontCamera = false

01-03 05:22:48.932 15035 15035 I ArcsoftSmallAdvancedFilter: updateFilterType = 268435458

## 开关配置

版本切换：

adb shell setprop persist.sys.cam.filter.version X (X可选0，1，2，分别代表0关，1虹软，2自研)

BoardConfig中：

PRODUCT\_USE\_CAM\_FILTER:=false (false代表关闭，true代表打开)

## 预留待补充