MTGLDebug

MTGLDebug使用手册

- MTHawkeye提供对外接口，和加载时机
- 对外唯一接口设置
    - MTGLDebugDisable 【需要重启App】
      完全关闭MTGLDebug不做hook
      对原本性能完全无影响
    - MTGLDebugOptionsOnlyStatisticalGLObject【默认】
      进行统计不做逻辑判断，不会有异常抛出
      【addObject不做历史记录操作，check逻辑return，removeObject不做历史记录操作】
      【优化点：仅对内存相关的GL函数进行hook】

      对原本性能有稍微影响，如果只做性能检测，不做内存检测和错误检测建议用Disable

  关闭MTGLDebugOptionsOnlyStatisticalGLObject执行

    - MTGLDebugOptionCheckContextStates【上下文资源逻辑检测】
      一般是图像会出错
    - MTGLDebugOptionCheckAPIUsageStates【glAPI检测】
      一般是崩溃在某个gl函数

案例演示

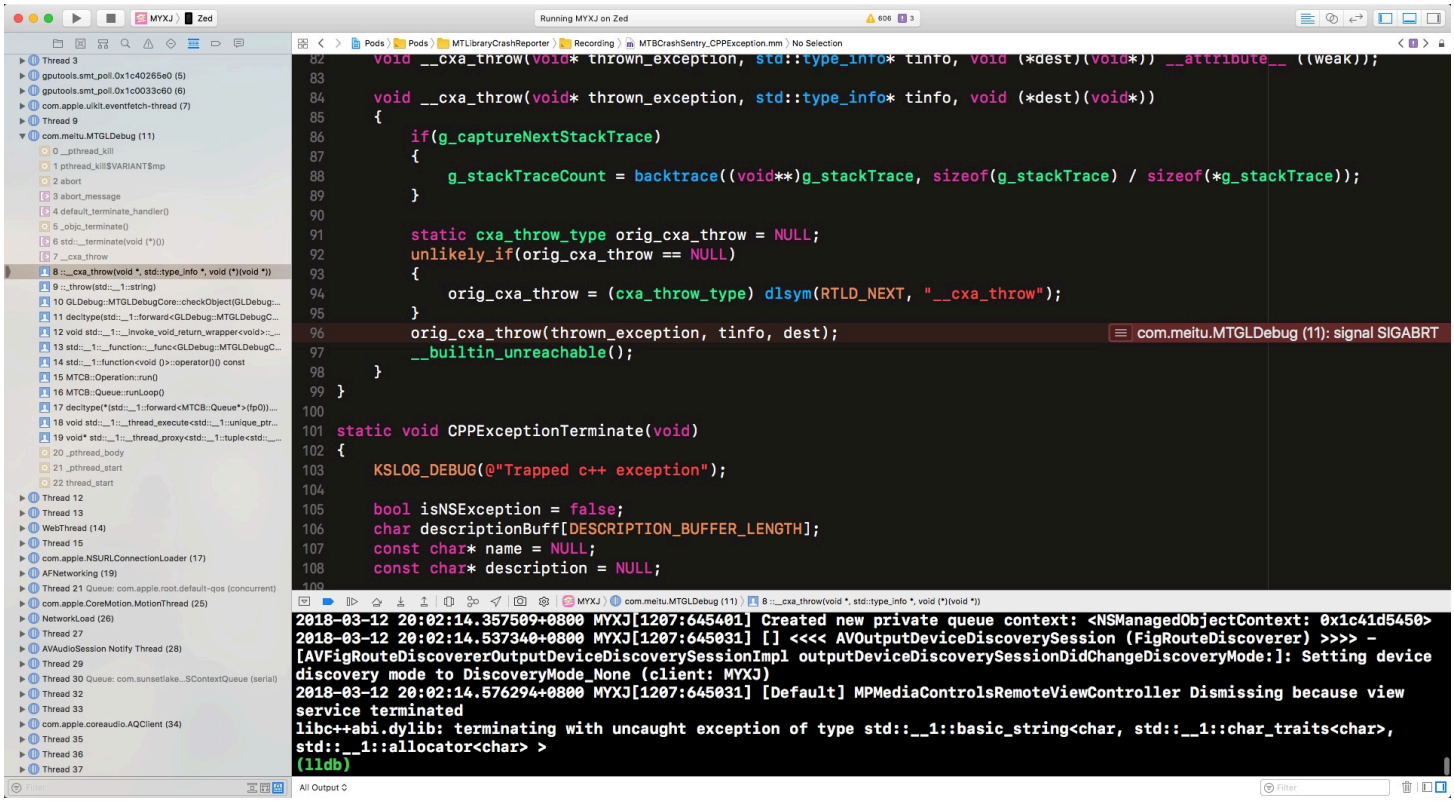
0x1 一些测试案例

0x2 美颜相机重复进出相机,拍摄界面卡死

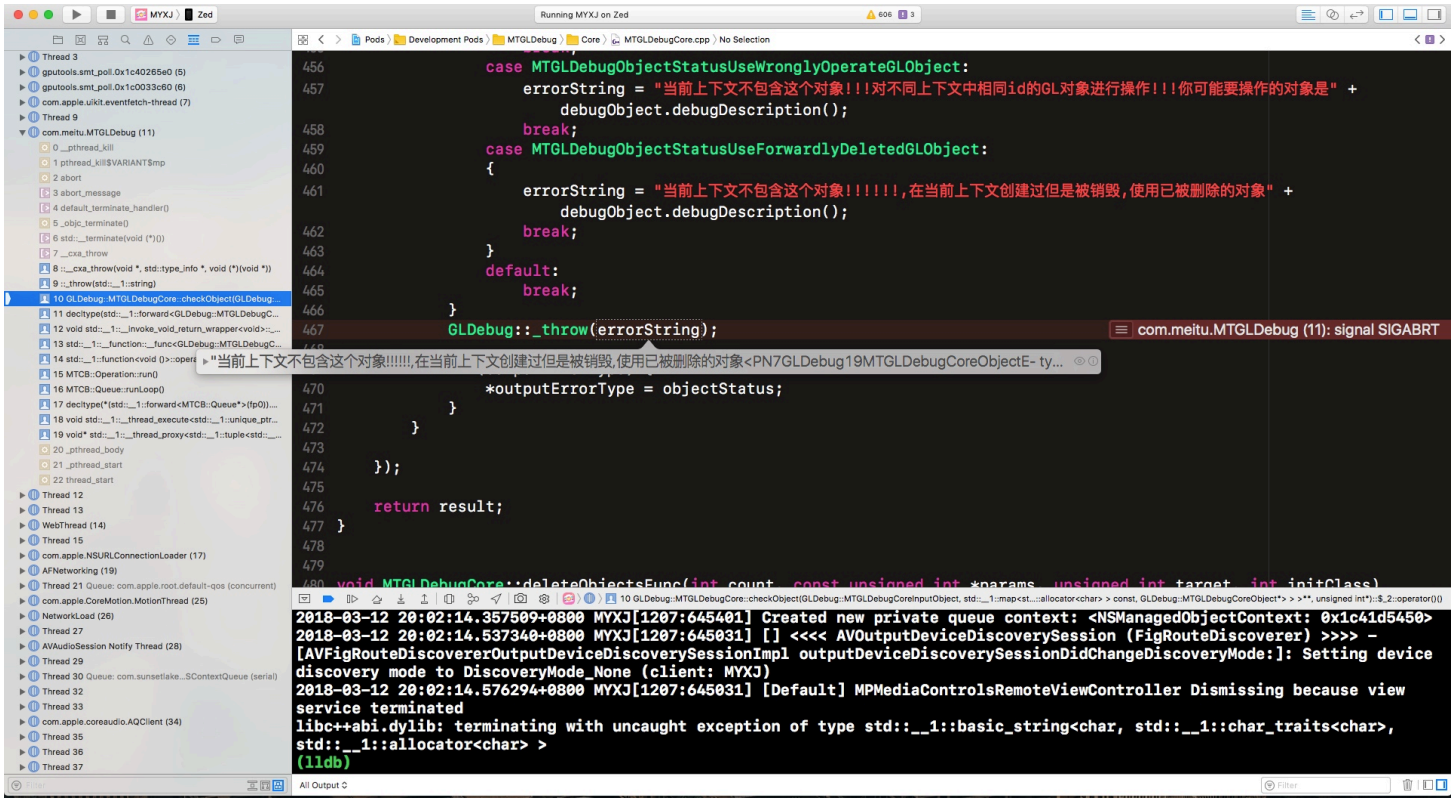
卡死后控制台log

```
<[mlab] ERROR:> glerror = 506
<[mlab] ERROR:> glerror = 506
<[mlab] ERROR:> glerror = 506
<[mlab] ERROR:> glerror = 506
<[mlab] ERROR:> glerror = 506
<[mlab] ERROR:> glerror = 506
<[mlab] ERROR:> glerror = 506
<[mlab] ERROR:> glerror = 506
<[mlab] ERROR:> glerror = 506
<[mlab] ERROR:> glerror = 506
<[mlab] ERROR:> glerror = 506
```

开启GLDebug，重复刚才的操作



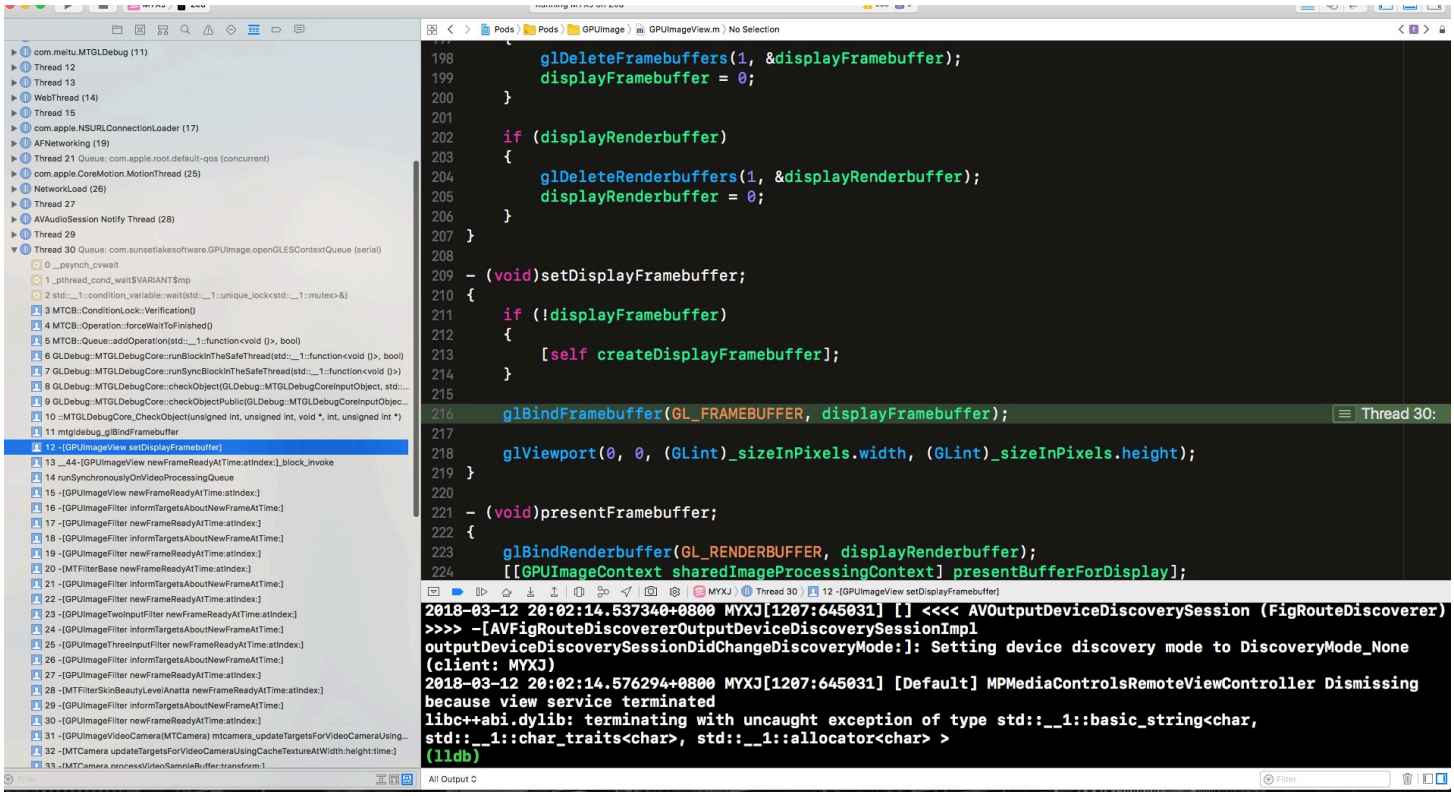
我们就挂在GLDebug抛出的异常上



左侧的堆栈往上找我们会看到com.meitu.MTGLDebug线程给出的错误提示

⚠️当前上下文不包含这个对象，使用了被删除的对象

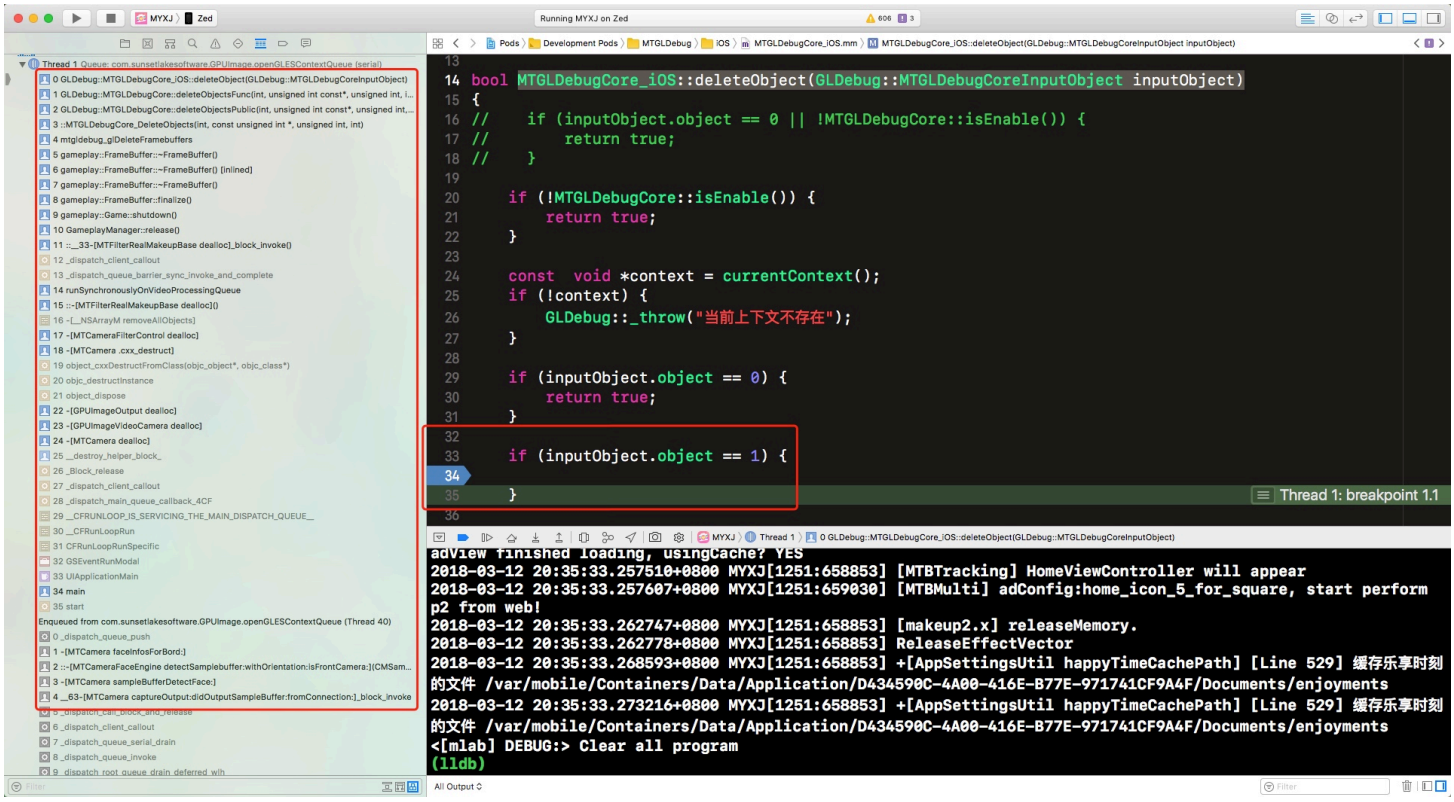
紧接着继续定位问题所在，查看这时刻的其他线程



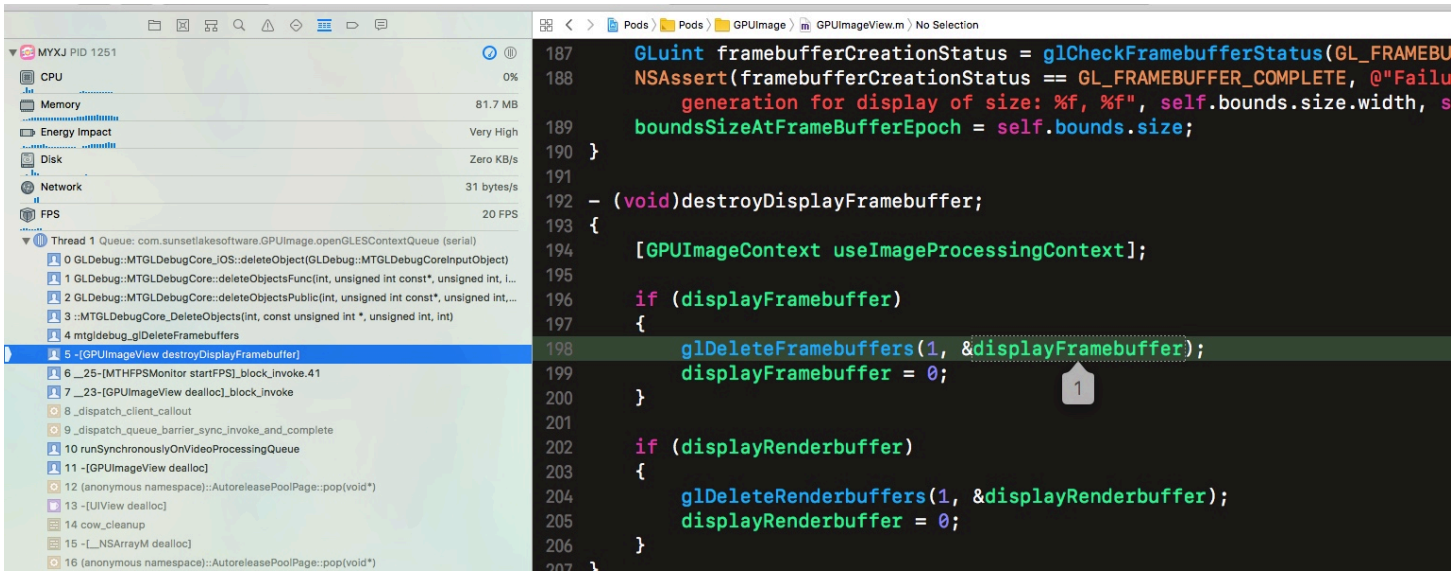
从图中我们可以看出是运行glBindFramebuffer这个函数检测GLDebugObject的时候出的问题。

然后，从上面抛出的异常信息可以判断，setDisplayFramebuffer目前绑定的displayFramebuffer其实已经被删除，所以导致拍摄画面不会被更新，显示出画面卡在上次退出的画面上。

接着验证，如果多删除了一次，在deleteFramebuffer这个函数删除两次句柄为displayFramebuffer的 Fbo。由上面得出displayFramebuffer为1，所以在delete函数加上inputObject==1 的判断。这里 MTFilterRealMakeupBase里面的 gamePlay::Frambuffer 删除了这个displayFramebuffer 有异常



继续执行断点，发现GPUImageView又删除了一次



MTGLDebug设计说明

- 创建DebugObject对象
    - addObject逻辑

        - 取当前的上下文检测是否存在
        - 以sharegroup地址为key值，共享的GL资源为value 【创建sharegroups字典】
        - 一个记录当前，一个记录历史
        - 如果sharegroup不在sharegroups中则以地址为key值加入sharegroup为value加入到sharegroups
        - 根据gl对象的target和object组合的字符串作为key值在加载入sharegroup共享资源字典里
    - CheckObject逻辑
        -

当前上下文	其他上下文	当前上下文的历史记录	产生原因
❎	❎	❎	这个GL对象从未被创建过
❎	❎	✅	在当前上下文创建过但是被(主动)销毁,使用已被删除的对象
❎	✅	✅	当前上下文不包含这个对象,在其他上下文中存在同id同类型的对象,当前上下文B的历史记录也存在,举例说明。上下文A存在id为1的纹理texture, 然后在上下文B中删除操作id为1的texture(本意是要删除A中的texture)B导致当前上下文B不存在该对象,历史记录中上下文B存在该对象。对不同上下文相同id(因误删除的)GL对象操作。
❎	✅	❎	当前上下文不包含这个对象,在其他上下文中存在同id同类型的对象,当前上下文B的历史记录不存在,举例说明。上下文A存在id为1的纹理texture, 然后在上下文B中操作id为1的texture(本意是要操作A中的texture),导致当前上下文B不存在该对象,上下文B历史记录中不存在该对象。对不同上下文相同id的GL对象操作,绑定错上下文。

    - DeletelObject逻辑
        - 删除当前，历史记录不删

附录：

用同一个shareGroup创建Context，才能资源共享

```
_movieWriterContext = [[GPUImageContext alloc] init];
[_movieWriterContext useSharegroup:[[[GPUImageContext sharedImageProcessingContext] context] sharegroup]];
```

```
- (void)useSharegroup:(EAGLSharegroup *)sharegroup;
{
    NSAssert(_context == nil, @"Unable to use a share group when the context has already been created. Call this method before you use the context for the first time.");

    _sharegroup = sharegroup;
}

- (EAGLContext *)createContext;
{
    EAGLContext *context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES2 sharegroup:_sharegroup];
    NSAssert(context != nil, @"Unable to create an OpenGL ES 2.0 context. The GPUImage framework requires OpenGL ES 2.0 support to work.");
    return context;
}
```

如果是要另一份不共享shareGroup就填nil去创建上下文

```objc
- (EAGLSharegroup *)sharegroup {
    Ivar audioIvar = class_getInstanceVariable([self class], "_sharegroup");
    EAGLSharegroup *outut = object_getIvar(self, audioIvar);
    return outut;
}


- (EAGLContext *)createContextPriorityOpenGLES3;
{
    EAGLContext *context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES3 sharegroup:[self sharegroup]];
    if (!context) {
        context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES2 sharegroup:[self sharegroup]];
        NSAssert(context != nil, @"Unable to create an OpenGL ES 2.0 context. The GPUImage framework requires OpenGL ES 2.
            support to work.");
    }
    return context;
}
```

Thread 1: step ove