Android动态模糊实现方案分享

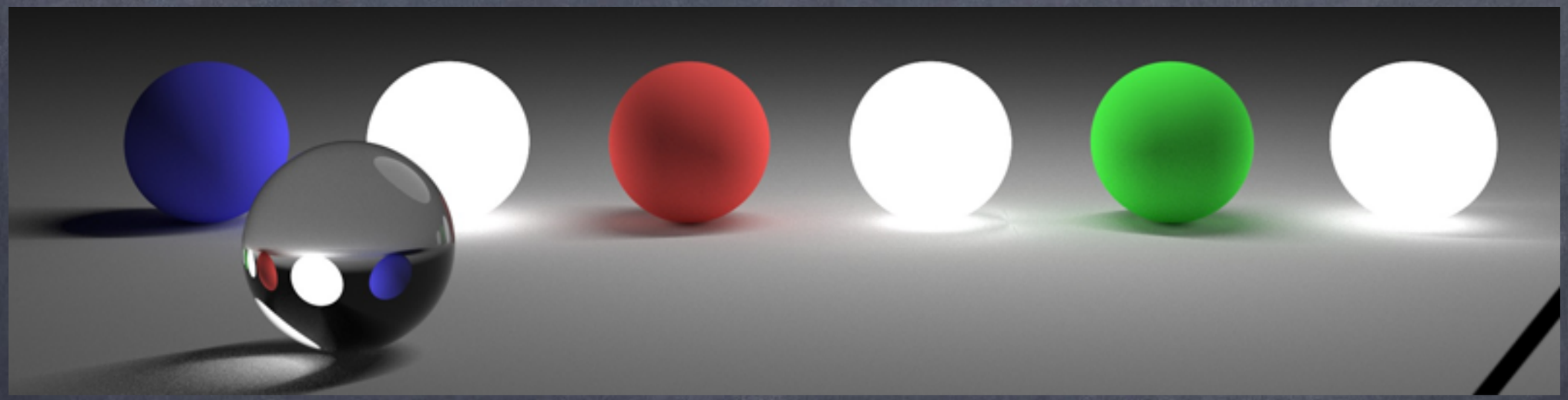
by 橡皮

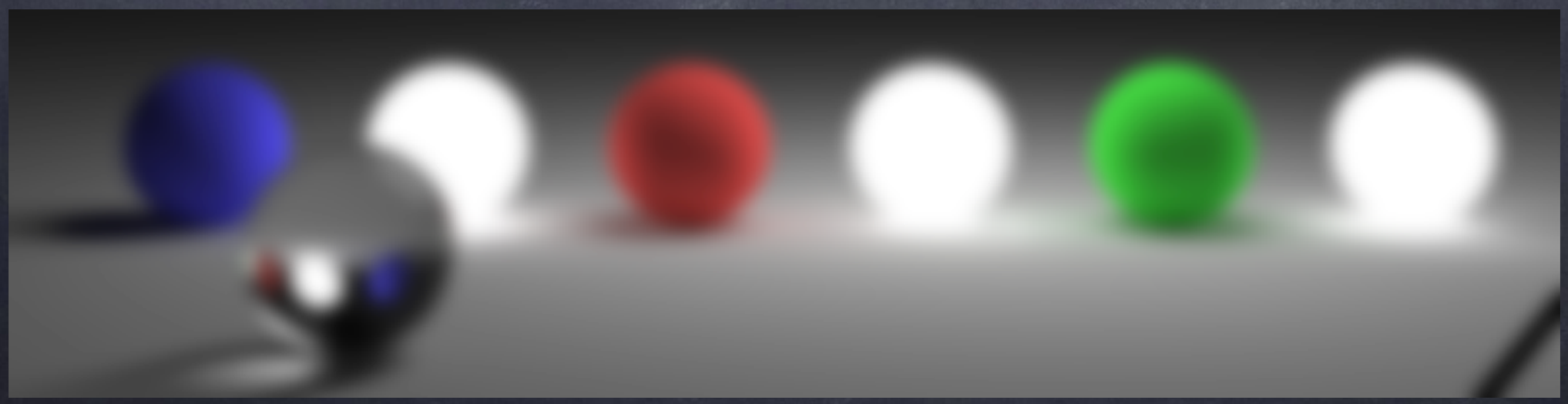
目录

- 模糊的概念和类型
- 模糊的实现方案
- 性能对比和分析
- 演示和总结

模糊

模糊效果实际上是由于图像的相邻像素具有相同或相似的像素特征引起的。模糊图像是原高分辨率图像的像素信息与相邻像素信息混合叠加的结果。





模糊核

这种叠加关系由模糊核表示，代表当前像素与周围像素点之间的加权关系。如下是一个核半径为1的模糊核。模糊核大小和半径关系为 size = 2 * radius + 1 。

a	b	c
d	e	f
g	h	i

模糊效果作用于图像之上，在数学上可看作图像与模糊核进行卷积（Convolution）运算，可看作加权和的结果

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}\right)[2,2] = (i*1) + (h*2) + (g*3) + (f*4) + (e*5) + (d*6) + (c*7) + (b*8) + (a*9).$$

均值模糊

均值模糊（Box Blur）顾名思义，即模糊图像的每一个像素为周围相邻像素的平均取值。

1/9	**1/9**	**1/9**
1/9	1/9	1/9
1/9	1/9	1/9

高斯模糊

称为高斯模糊（Gaussian Blur）是由于利用高斯函数计算权重值。

一维模糊核，权重值计算公式为：

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

二维模糊核，权重值计算公式为：

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

σ=1.5的高斯模糊核

0.0453542	**0.0566406**	**0.0453542**
0.0566406	0.0707355	0.0566406
0.0453542	0.0566406	0.0453542

归一化

0.0947416	**0.118318**	**0.0947416**
0.118318	0.147761	0.118318
0.0947416	0.118318	0.0947416

Stack模糊

Stack Blur是对高斯模糊的一种近似。

模糊核权重的生成方式上更简单一些，每个相邻像素权重值与其中心像素间距离相关。 对于半径为r的一维模糊核，模糊核可以表示为:

1	2	...	r	r+1	r	...	2	1

半径为1的一维模糊核

1	2	1

二维模糊核，模糊图像可以当做横向模糊，再纵向模糊处理的结果斯模糊核

1	2	1
1	2	1
1	2	1

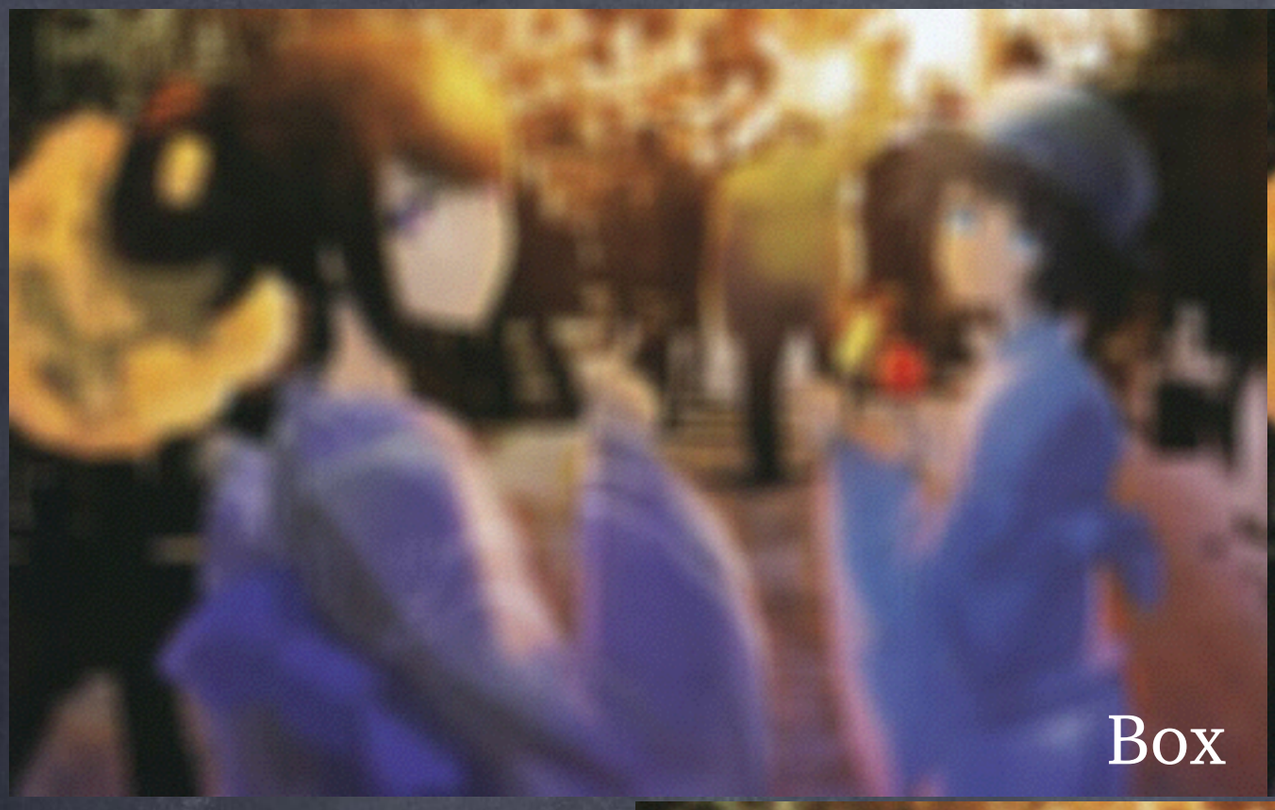
纵向模糊 →

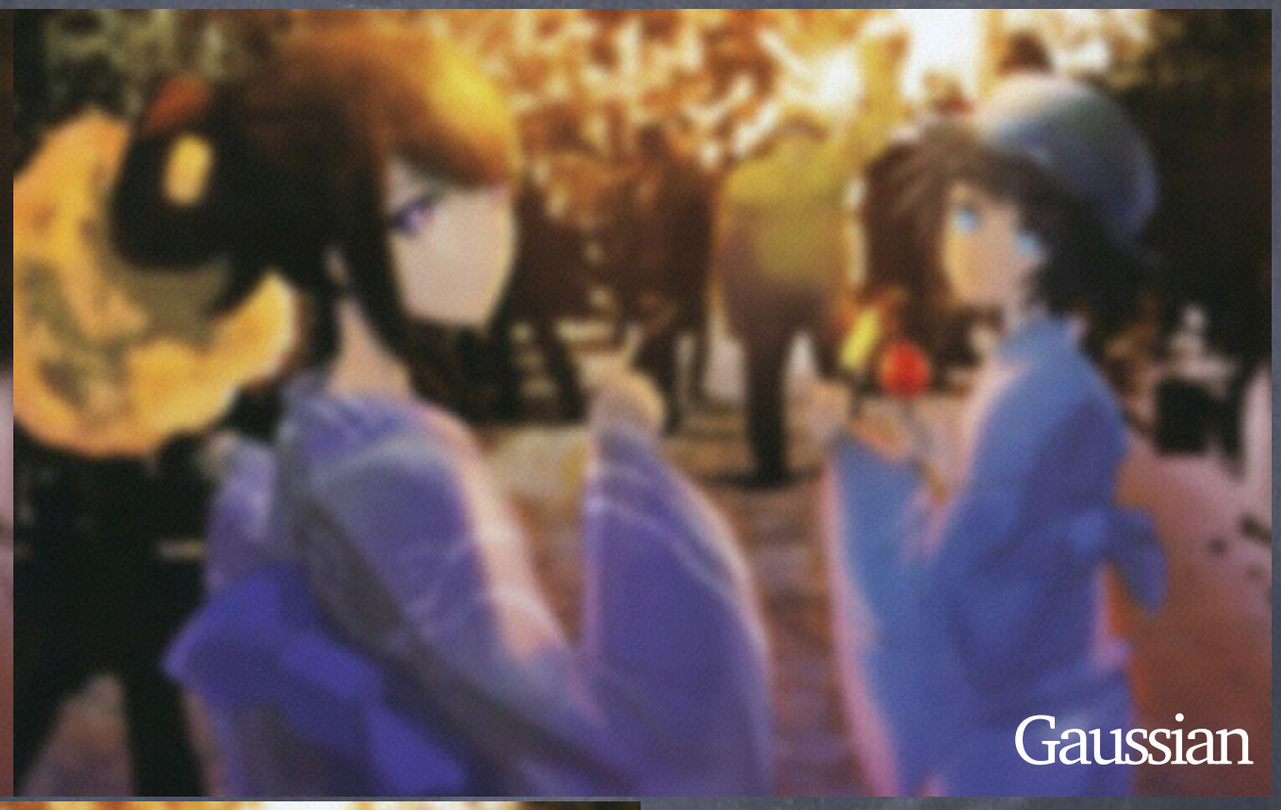
1	2	1
2	4	2
1	2	1

归一化 →

1/16	2/16	1/16
2/16	4/16	2/16
1/16	2/16	1/16

效果示例





模糊的实现方案

- Java的实现方案
- Native（C/C++）的实现方案
- RenderScript的实现方案
- OpenGL的实现方案

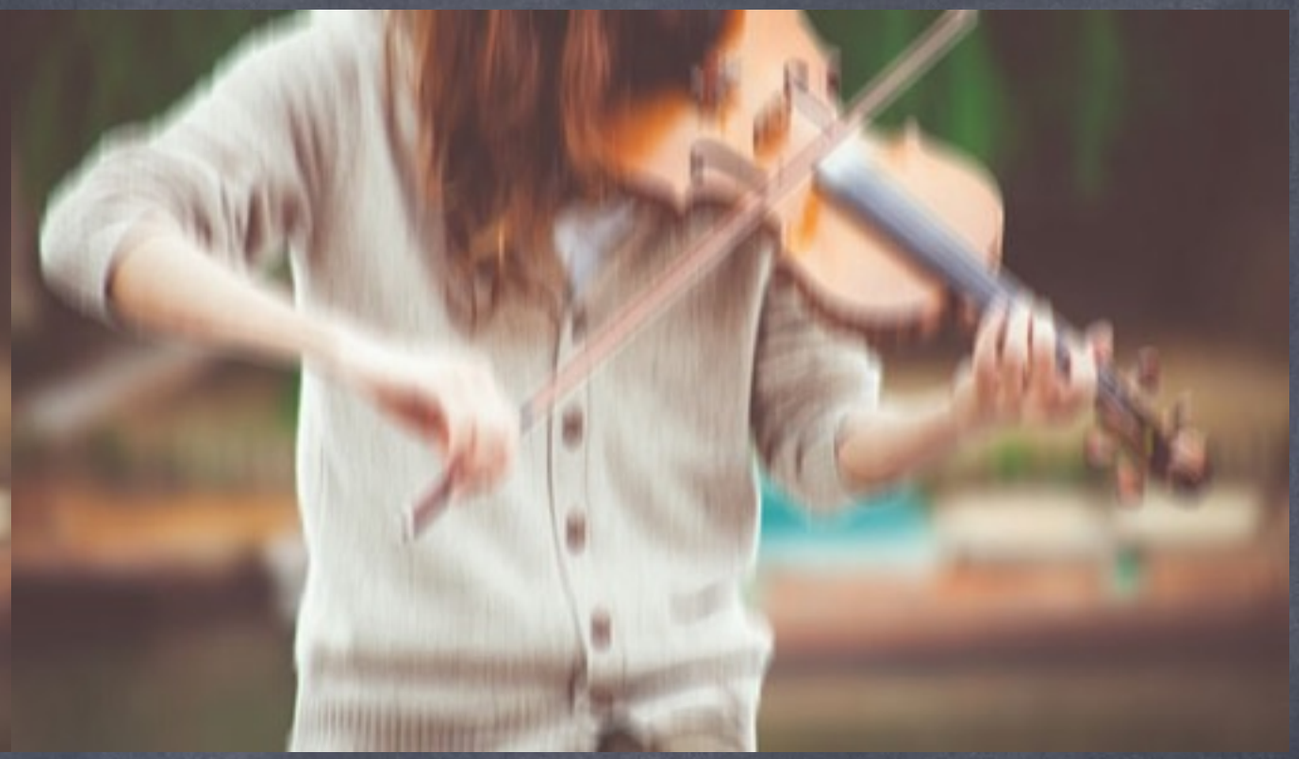
Java和Native C/C++方案

为了提高性能，做了以下优化。

- 分步模糊

模糊过程分为水平方向和垂直方向依次处理，因此每次模糊只需要一维模糊核即可。对于模糊核半径为r下，采用二维模糊核的复杂度为O(n * r * r)，而分步模糊的方式可以将复杂度减少至O(n * r)

横向模糊

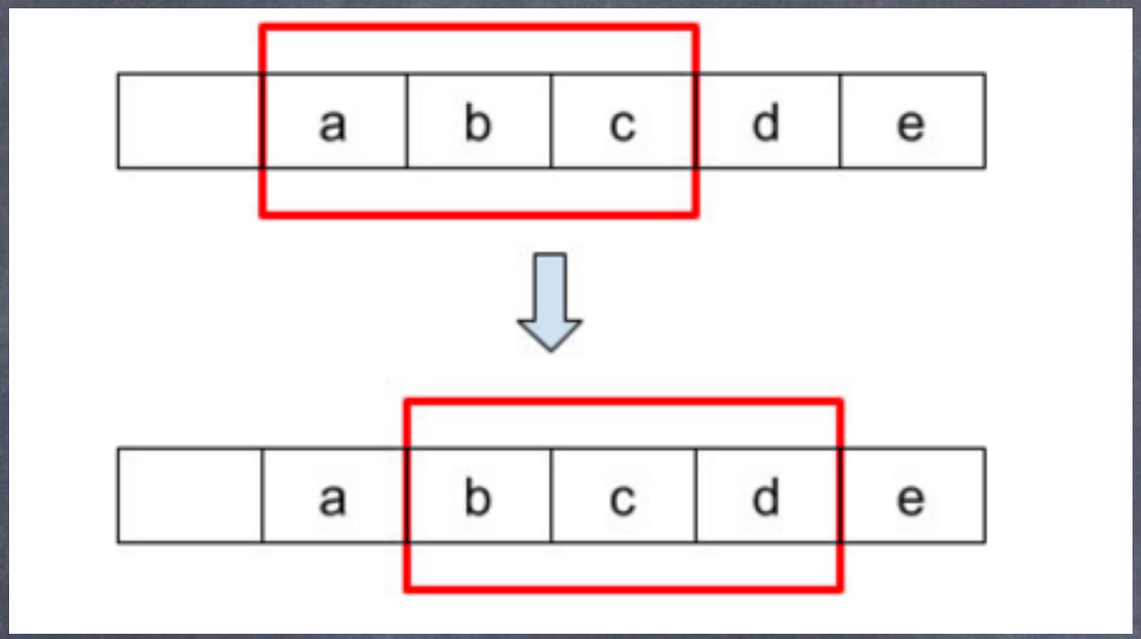


纵向模糊

双向模糊

Java和Native C/C++方案

- 滑动窗口算法



Box Blur实际并不需要重新计算窗口内所有元素的总和，只需要考虑移出窗口和进入窗口两个元素即可，也就是sum += d - a;

Stack Blur则需要考虑权重值，虽然每个元素权重值不同，但根据前面的分析，相邻元素权重值差1

```
outsum = a + b;
insum = c + d;
sum += insum - outsum;
```

Java和Native C/C++方案

建立查询表

```
int tableSize = 2 * radius + 1;
int divide[] = new int[256 * tableSize];

// 建立 0 到 255的查询表，0~256*tableSize 所有窗内元素权重和值的范围
for (int i = 0; i < 256 * tableSize; i++) {
    divide[i] = i / tableSize;
}
...

//不再需要除法，获得和值后，直接查表得到结果
out[outIndex] = (divide[suma] << 24) | (divide[sumr] << 16) | (divide[sumg] << 8) | divide[sumb];
```

提前计算好所有可能出现的取值。在每次加权平均的时候，不再需要做除法运算，只需要根据加权值进行查询即可获得最后的模糊结果。

RenderScript方案

Renderscript是Android系统上专门用于密集型计算的高性能框架。RenderScript可在设备上所有可用的处理器上并行执行，所以开发者可以专心写处理算法，而不需要关心调度和负载平衡的问题。

```
void __attribute__((kernel)) boxblur_h(uchar4 in, uint32_t x, uint32_t y) {

    float4 sum = 0;
    uchar4 result;
    int count = 0;
    int kernel = (2 * radius + 1);

    uchar4 center = rsGetElementAt_uchar4(input, x, y);

    for (int j = -radius; j <= radius; j++) {
        if (x >= 0 && x < width && y + j >= 0 && y + j < height) {
            uchar4 temp = rsGetElementAt_uchar4(input, x, y + j);
            sum += rsUnpackColor8888(temp);
            count++;
        }
    }


    sum = sum / count;
    result = rsPackColorTo8888(sum);
    result.a = center.a;
    rsSetElementAt_uchar4(output, result, x, y);

}
```

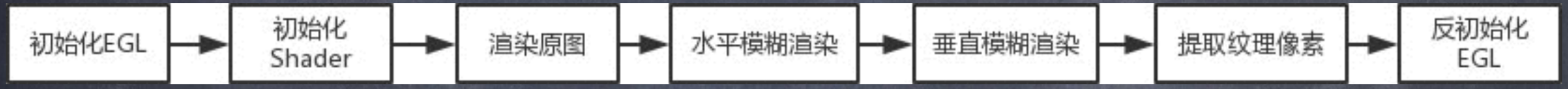
```
for (int i = -radius; i <= radius; i++) {
    if (x + i >= 0 && x + i < width) {
        uchar4 temp = rsGetElementAt_uchar4(input, x + i, y);
        int weight = radius + 1 - abs(i); //设置权重值
        sum += rsUnpackColor8888(temp) * weight;
        weightSum += weight;
    }

}
```

不再需要考虑滑动窗口之类的情况，只需要根据当前坐标（x,y)获取相邻像素值。

OpenGL方案

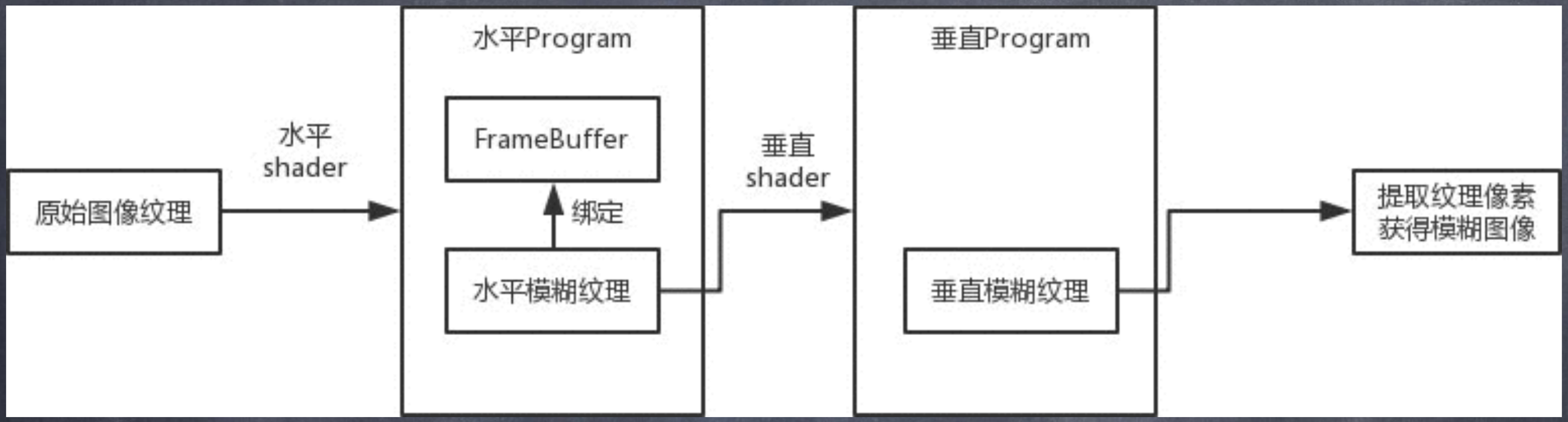
目标：直接获得模糊后的Bitmap。

这里选择离屏渲染。也可以采用屏上渲染，但需要创建GLSurfaceView或TextureView，GLSurfaceView和TextureView会消耗一定的系统资源



- 需要EGL的管理，但从EGLSurface，EGLDisplay和EGLContext的创建和设置，都需要程序自行实现
- shader实际控制了渲染方式，因此模糊是在Fragment shader当中处理的
- 分为水平和垂直方向的渲染

OpenGL方案



- 首先获得原始图像纹理；
- 将水平模糊纹理作为纹理附件与一个FrameBuffer绑定；
- 原始图像纹理(mInputTexture)经过shader的渲染时，模糊后的图像结果渲染到水平模糊纹理
- 然后利用该水平模糊纹理(mHorizontalTexture)作为输入纹理，再做一次渲染，得到最后的模糊图像

OpenGL ES Context在多线程环境下的管理

OpenGL在多线程环境下工作需要额外的处理，为了正常工作需要遵守两条规则：

- 一个线程只能有一个渲染上下文（Render context）
- 一个context只能绑定一个线程

因此在多线程环境下的实现context共享的处理方式如下：

1. 将context与线程1绑定:

```
eglMakeCurrent(display, surface, surface, context);
```

2. 在线程1执行OpenGL操作...
3. 将context与线程1解绑:

```
eglMakeCurrent(display, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT);
```

4. 将context与线程2绑定:

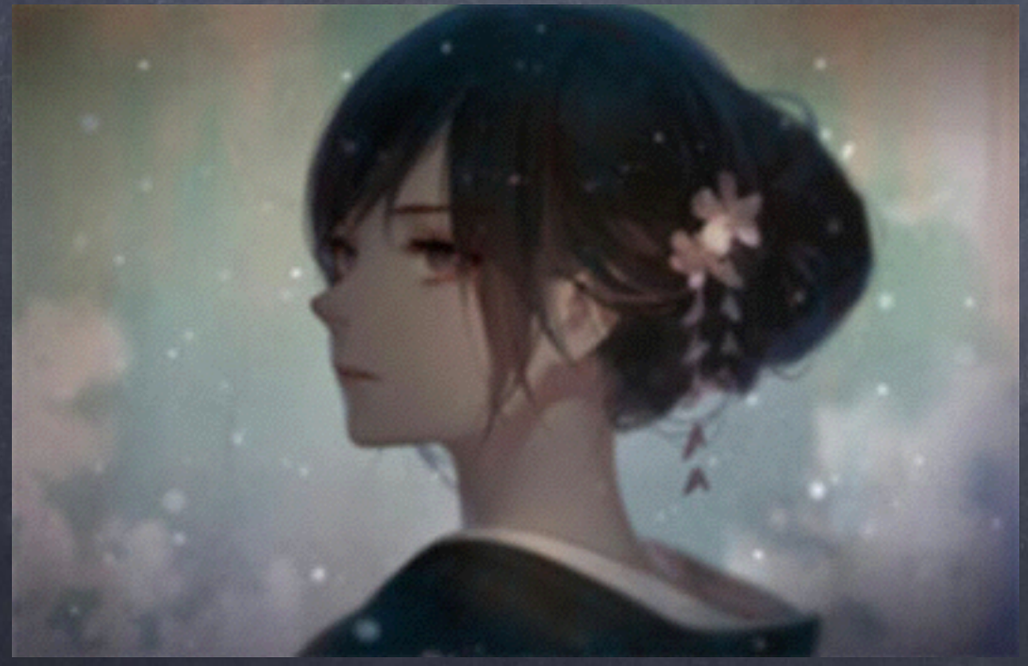
```
eglMakeCurrent(display, surface, surface, context);
```

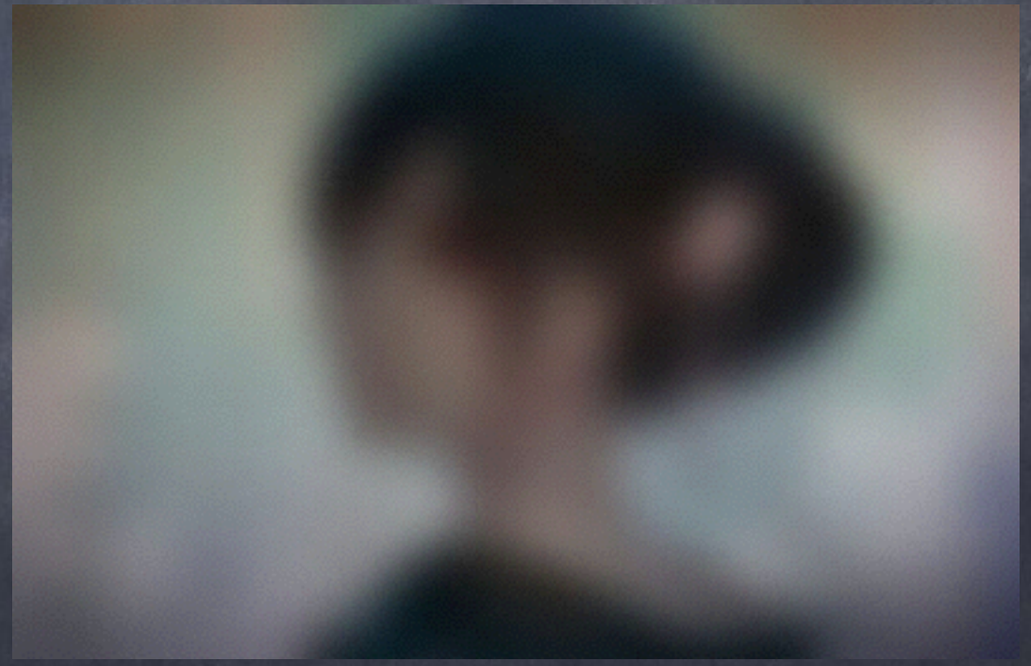
5. 在线程2执行OpenGL操作...

毛玻璃效果的实现

毛玻璃效果实际上是对原图片的严重劣化，要求具有朦胧感

原图 → 缩放尺寸 → 模糊处理 → 恢复尺寸





再优秀的模糊方案，当图片尺寸较大，且模糊核半径较大时，仍然会有性能上的问题。尺寸的缩小可大幅提高提高模糊的效率，实际缩放因子为5时，已能获得较好的毛玻璃效果和计算效率

性能对比

模糊1669×1080图片10次耗时对比（并未做resize）,模糊核半径10

- 缩放因子2，模糊核半径10

	Java	Native	RenderScript	OpenGL
Box Blur	1226ms	880ms	765ms	2730ms
Stack Blur	2690ms	1484ms	810ms	2780ms
Gaussian Blur	6747ms	3204ms	569ms	2686ms

- 缩放因子10，模糊核半径10

	Java	Native	RenderScript	OpenGL
Box Blur	325ms	298ms	277ms	545ms
Stack Blur	393ms	356ms	294ms	516ms
Gaussian Blur	762ms	428ms	272ms	568ms

性能对比

影响模糊耗时有几个因素

- 在获得模糊图像后，需要将图像尺寸恢复至原图尺寸，这一步实际较为耗时；
- 从纹理提取像素的操作，GlES20.glReadPixels()是一个相当耗时的操作；
- 原图尺寸本身较大，分辨率较高。

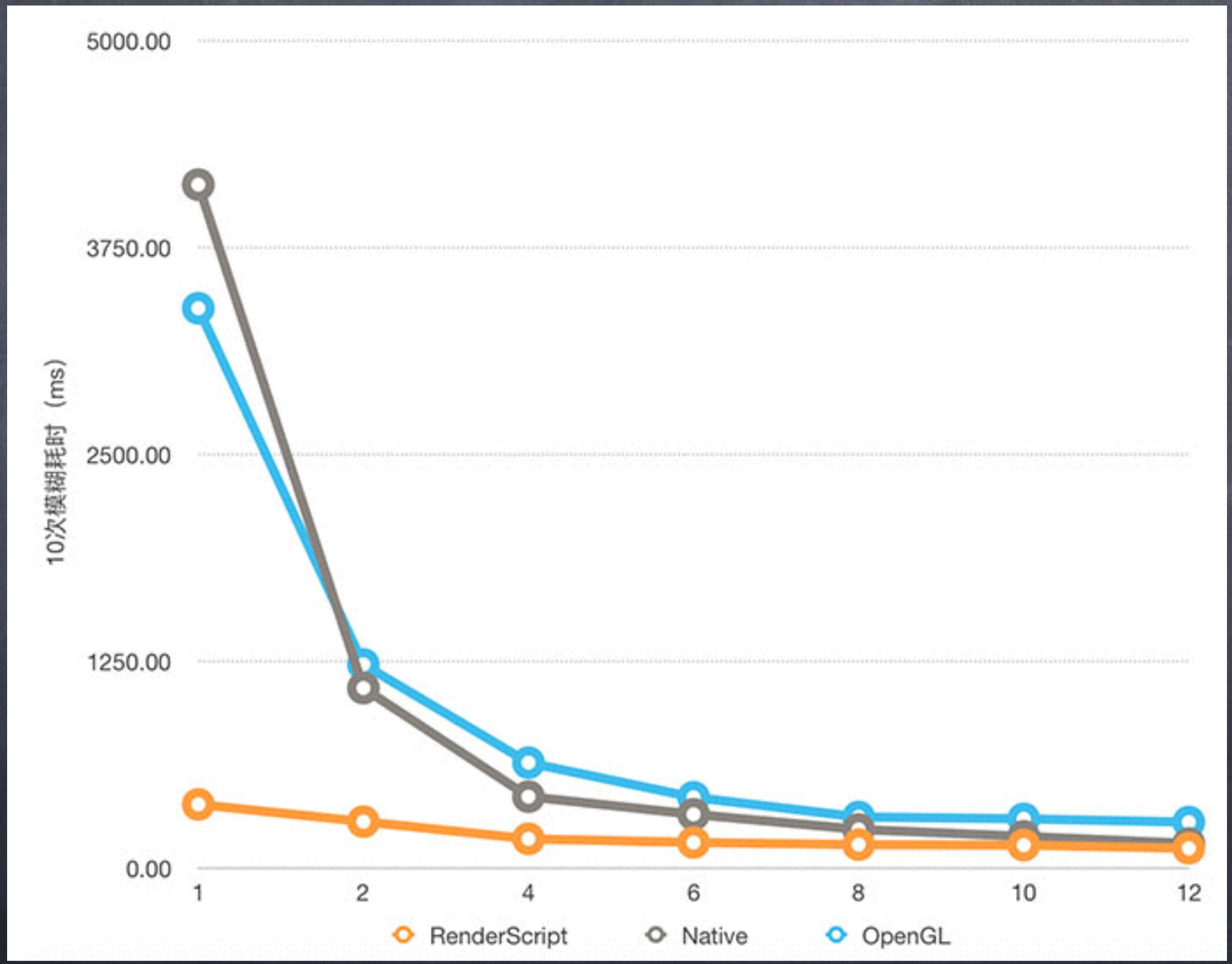
- **缩放因子10，模糊核半径10**

	Java	Native	RenderScript	OpenGL
Box Blur	183ms	172ms	178ms	278ms
Stack Blur	181ms	181ms	185ms	297ms
Gaussian Blur	213ms	195ms	142ms	301ms

对于1000×600的图像，则能达到60帧。实际开发应该避免直接使用大尺寸原图进行模糊。

缩放因子的影响

在模糊之前进行缩放能大幅提高模糊性能。



原图1000×600，模糊核半径10

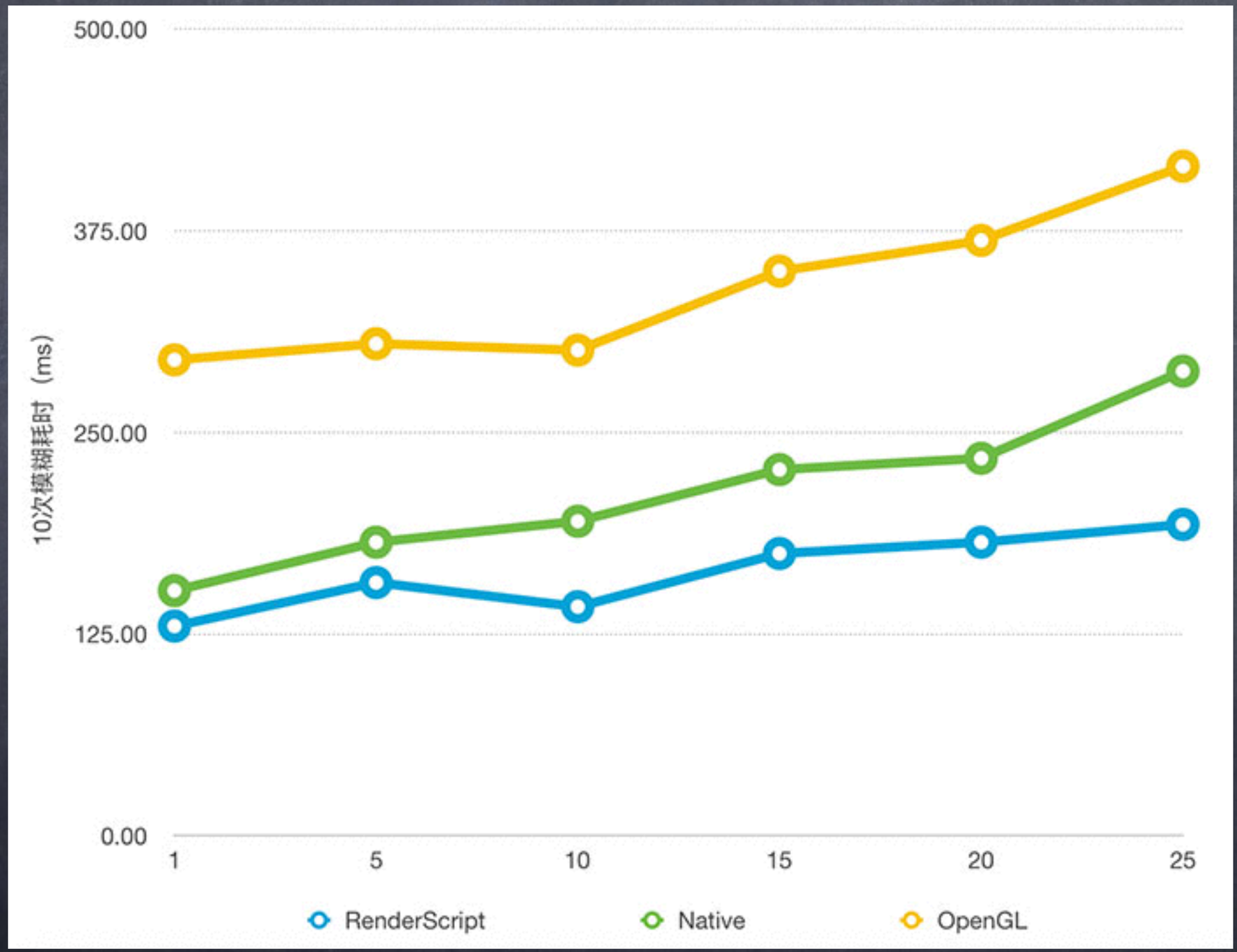
缩放因子的影响

在模糊之前进行所放能大幅提高模糊性能。

左图缩放因子5，右图缩放因子10

模糊半径的影响

实际模糊性能受模糊半径选择影响较小。

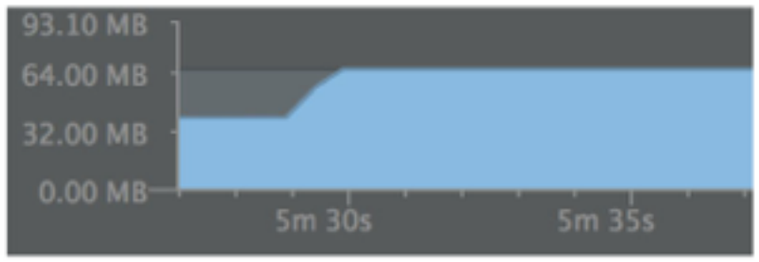


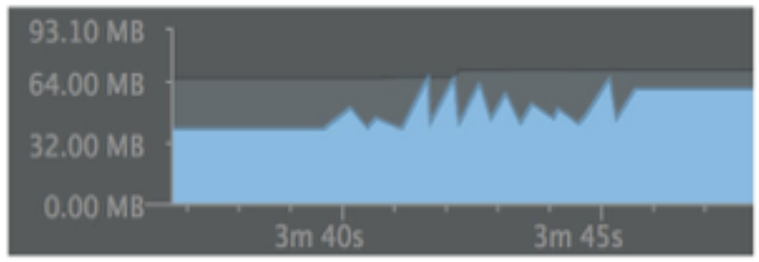
原图1000×600，缩放因子10

内存占用情况

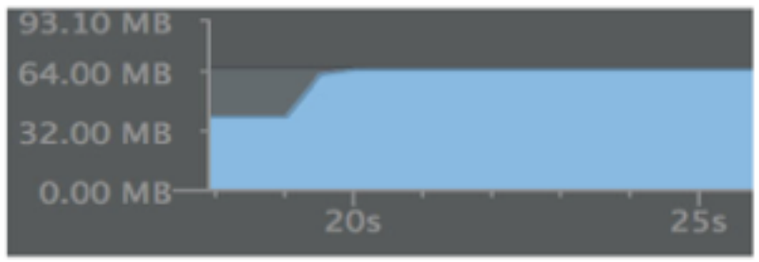
原图1000×600，缩放因子10，模糊核半径10。左为模糊10次，右为100次的内存占用情况。

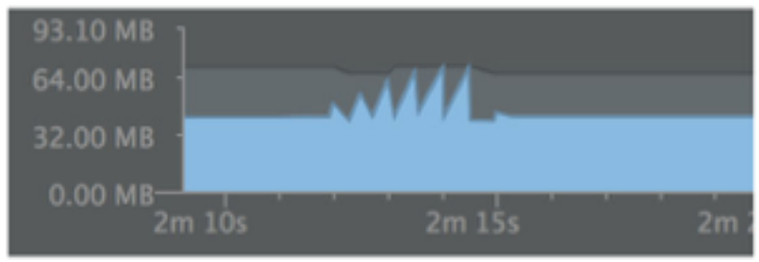
- Java



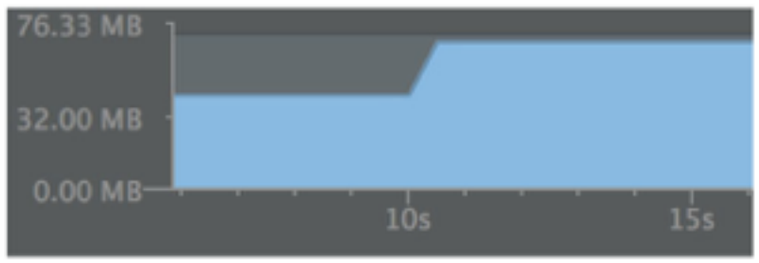


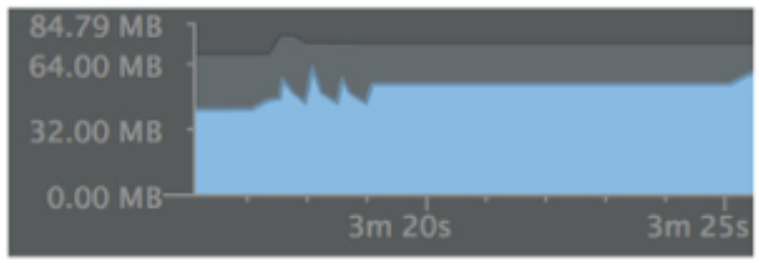
- Native



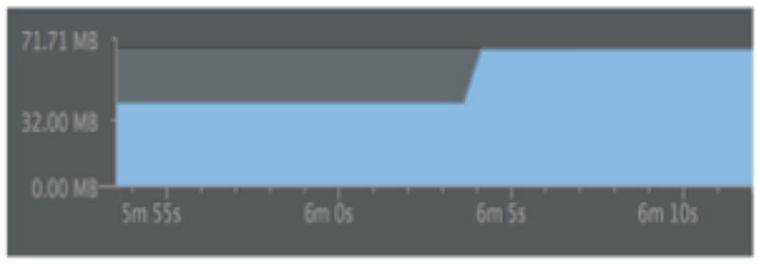


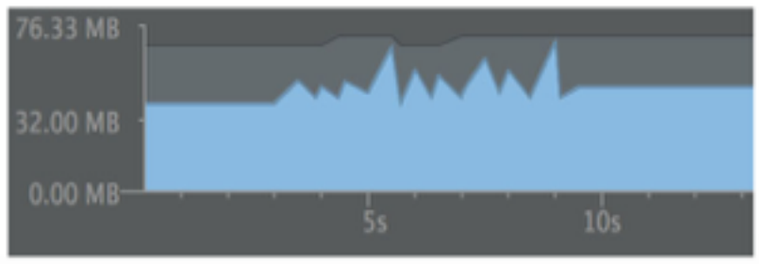
- RenderScript





- OpenGL

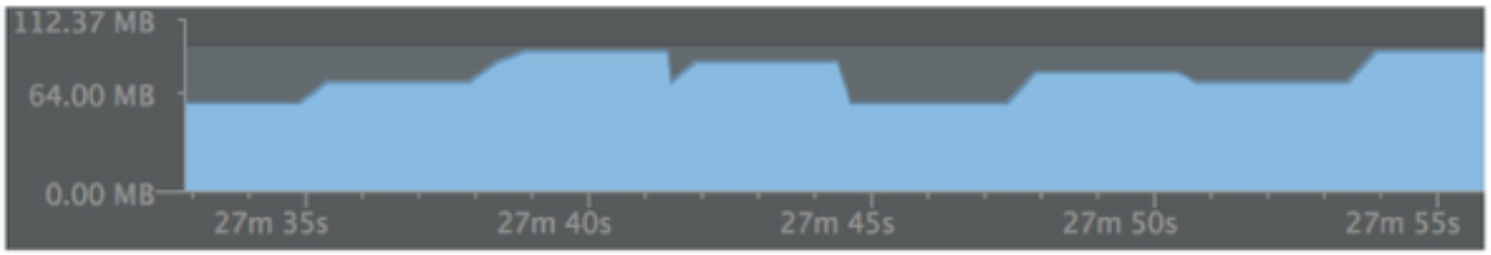




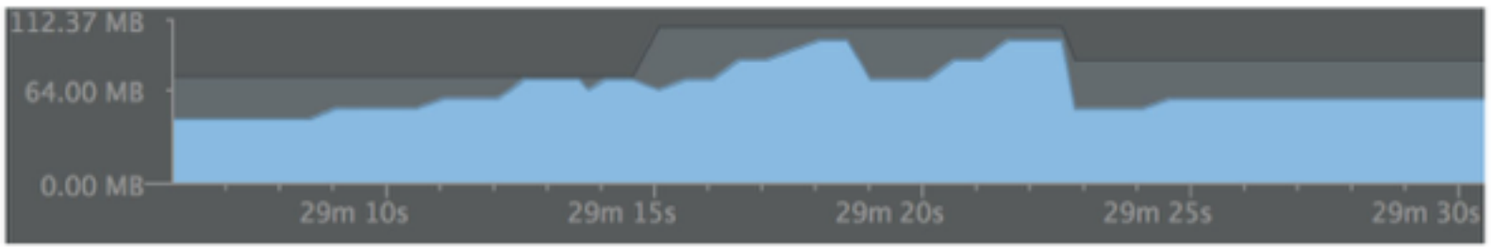
内存占用情况

原图1920×1080，缩放因子1，模糊核半径10。

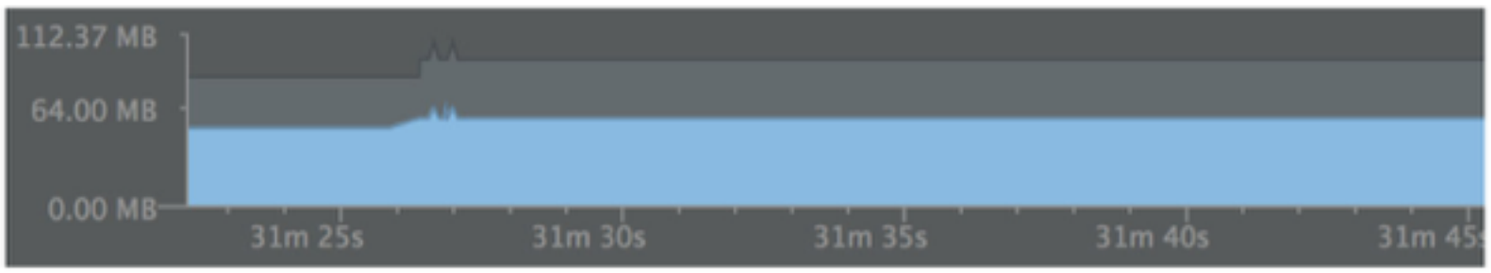
- Java



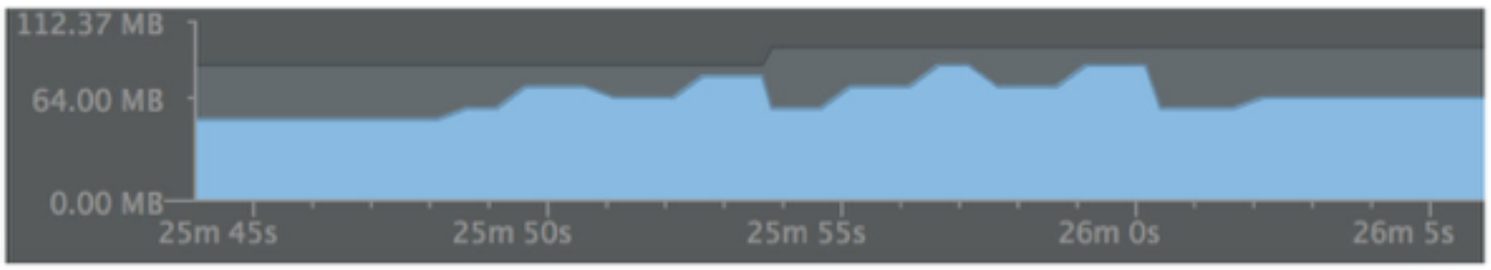
- Native



- RenderScript



- OpenGL



方案优缺点

- Java和Native方案，与其他方案相比具有更好的兼容性，其中Native方案比Java方案模糊效率稍高一些，运算耗时更久，且在低端机型环境下，更容易引起OOM；
- RenderScript方案的优点在于相当可观的模糊效率，但部分机型系统尤其是低版本系统会出现兼容性问题，常见为librsjni.so和libRSSupportIO.so缺失引起的错误（BuildToolsVersion 23.0.2）。另外，对于内置的高斯模糊，模糊半径大于25时会失效；
- 在使用高斯模糊处理大尺寸图像时，采用OpenGL方案会比Native高效一些。缺点是GPU的寄存器相对有限，部分低端机型在模糊半径较大时，由于寄存器不足而渲染失败。还有一个需要注意的问题，一个应用程序进程可以创建的Open GL纹理是有大小限制的。

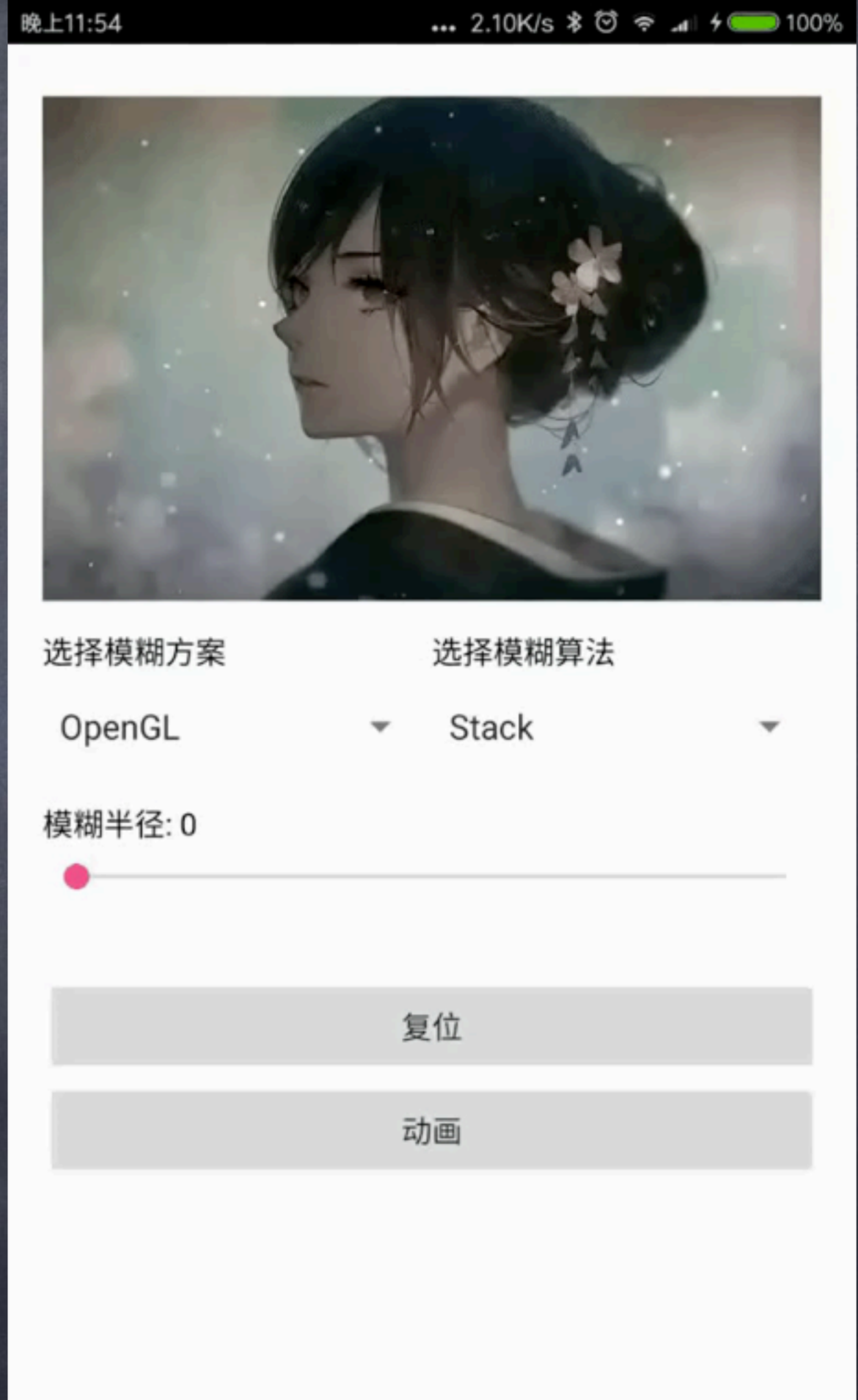
方案优先级

- 一般使用RenderScript方案实现高斯模糊即可解决问题，针对RenderScript有兼容性问题的机型，可以采用使用Native方案实现的Stack模糊；
- 在RenderScript方案失效时，还是需要评估Native的效果，低端机应该慎重考虑是否应用模糊，毕竟模糊是一个运算量大且相对耗时的工作；
- 对于模糊质量要求不高的场景，可以使用Box Blur替代Gaussian Blur，当然Stack Blur也是不错的选择。
- 并不能就此说明OpenGL是一个较差的实现方案，因为这里是通过获得模糊的Bitmap来实现模糊效果的展示，像素操作GLES20.glReadPixels()并不高效，利用OpenGL实现模糊可以使用其他方式，比如直接TextureView上的渲染，或者GLES20Canvas(HardwareCanvas)或者DisplayListCanvas的渲染嵌入实现
- 实际模糊性能还和机型有关，在小米Note上，OpenGL的方案相对效率低一些，但是在美图M4上OpenGL方案明显好于Native

动态模糊

- 每秒30~60帧的帧率能给人流畅的感觉，实现动态模糊应该尽量满足30~60帧的要求
- 原图1699*1080，并未做Resize处理。

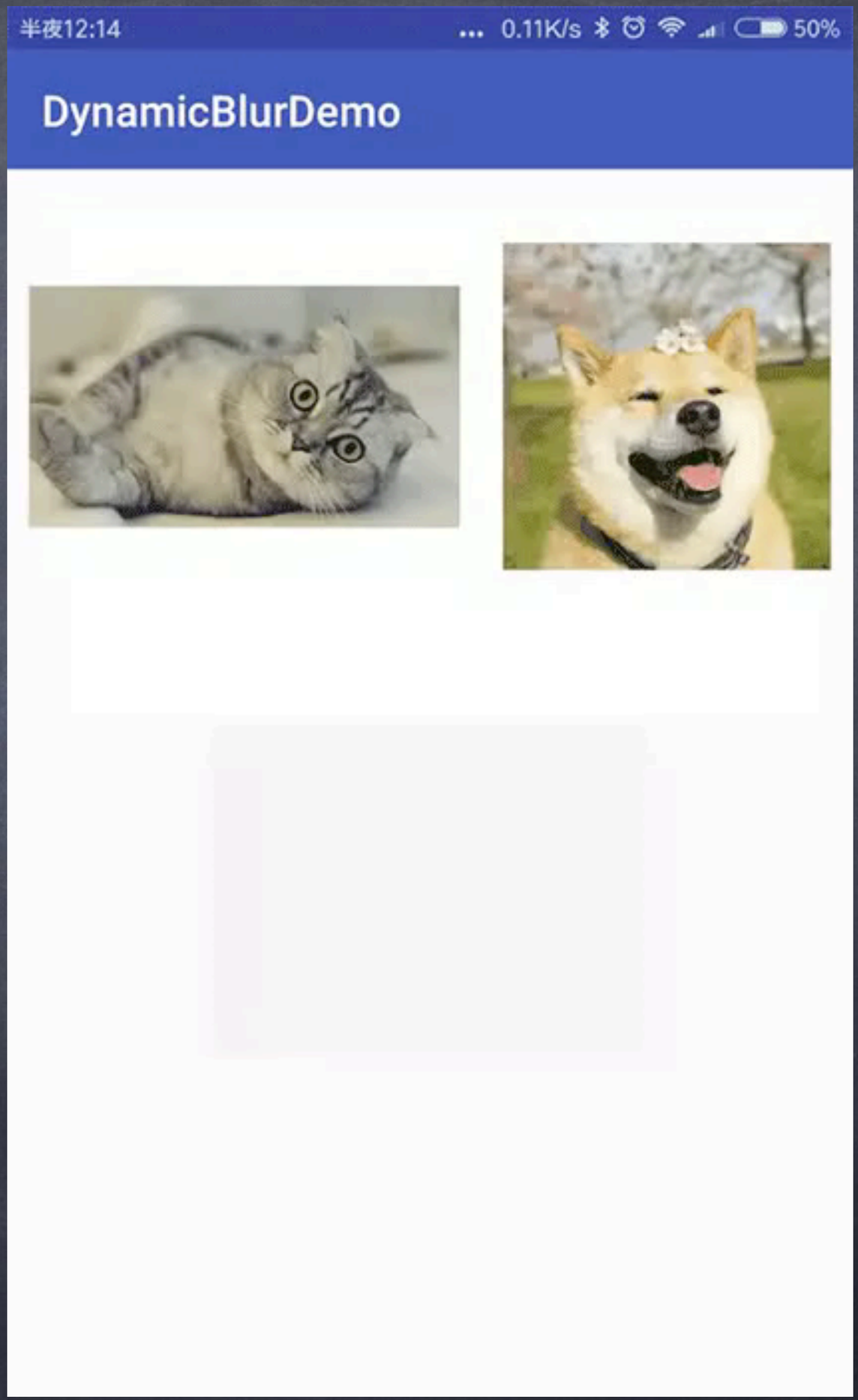
模糊动画



动态模糊

- 每秒30~60帧的帧率能给人流畅的感觉，实现动态模糊应该尽量满足30~60帧的要求
- 实际并不需要特别大尺寸的图只需要选取屏幕的一部分即可。

任意部位模糊



组件化

- 上述方案已整理成组件，使用姿势如下

```
Blur.with(context)
     .scheme(Blur.SCHEME_RENDER_SCRIPT) //设置模糊方案
     .mode(Blur.MODE_GAUSSIAN) //设置模糊算法类型
     .sampleFactor(10) //设置缩放因子
     .radius(10) //设置模糊半径
     .getBlurGenerator() //获得模糊实现类
     .doBlur(bitmap);    //模糊图片
```

TODO

- 实际测试可以发现，模糊图像尺寸upscale为原图尺寸，还是占用了一定的耗时，进一步优化效率可以从这方面入手；
- OpenGL方案中GLES20.glReadPixels()效率太低，可以考虑两种解决方案：
  - 对于支持OpenGL ES 3.0的Android 4.3以上系统，可以使用Pixel Buffer Object获取像素值；
  - Android 4.3以下系统，则采用EGL_KHRimage_base的解决方案。

Thanks for watching