真香！阿里工程师的一段代码让我看饿了

原创 莱宁 阿里技术 2019-11-04

阿里妹导读：打开盒马app，相信你跟阿里妹一样，很难抵抗各种美味的诱惑。颜值即正义，盒马的图片视频技术逼真地还原了食物细节，并在短短数秒内呈现出食物的最佳效果。今天，我们请来阿里高级无线开发工程师莱宁，解密盒马app里那些“美味”视频是如何生产的。

一、前言

图片合成视频并产生类似PPT中每页过渡特效的能力是目前很多短视频软件带有的功能，比如抖音的影集。这个功能主要包括图片合成视频、转场时间线定义和OpenGL特效等三个部分。

其中图片转视频的流程直接决定了后面过渡特效的实现方案。这里主要有两种方案：

1. 图片预先合成视频，中间不做处理，记录每张图片展示的时间戳位置，然后在相邻图片切换的时间段用OpenGL做画面处理。
2. 图片合成视频的过程中，在画面帧写入时同时做特效处理。

方案1每个流程都比较独立，更方便实现，但是要重复处理两次数据，一次合并一次加特效，耗时更长。

方案2的流程是相互穿插的，只需要处理一次数据，所以我们采用这个方案。

下面主要介绍下几个重点流程，并以几个简单的转场特效作为例子，演示具体效果。

二、图片合成

1.方案

图片合成视频有多种手段可以实现。下面谈一下比较常见的几种技术实现。

I.FFMPEG

定义输出编码格式和帧率，然后指定需要处理的图片列表即可合成视频。

```
ffmpeg -r 1/5 -i img%03d.png -c:v libx264 -vf fps=25 -pix_fmt yuv420p out
```

II.MediaCodec

在使用Mediacodec进行视频转码时，需要解码和编码两个codec。解码视频后将原始帧数据按照时间戳顺序写入编码器生成视频。但是图片本身就已经是帧数据，如果将图片转换成YUV数据，然后配合一个自定义的时钟产生时间戳，不断将数据写入编码器即可达到图片转视频的效果。

III.MediaCodec&OpenGL

既然Mediacodec合成过程中已经有了处理图片数据的流程，可以把这个步骤和特效生成结合起来，把图片处理成特效序列帧后再按序写入编码器，就能一并生成转场效果。

2.技术实现

首先需要定义一个时钟，来控制图片帧写入的频率和编码器的时间戳，同时也决定了视频最终的帧率。

这里假设需要24fps的帧率，一秒就是1000ms，因此写入的时间间隔是1000/24=42ms。也就是每隔42ms主动生成一帧数据，然后写入编码器。

时间戳需要是递增的，从0开始，按照前面定义的间隔时间差deltaT，每写入一次数据后就要将这个时间戳加deltaT，用作下一次写入。

然后是设置一个EGL环境来调用OpenGL，在Android中一个OpenGl的执行环境是threadlocal的，所以在合成过程中需要一直保持在同一个线程中。Mediacodec的构造函数中有一个surface参数，在编码器中是用作数据来源。在这个surface中输入数据就能驱动编码器生产视频。通过这个surface用EGL获取一个EGLSurface，就达到了OpenGL环境和视频编码器数据绑定的效果。

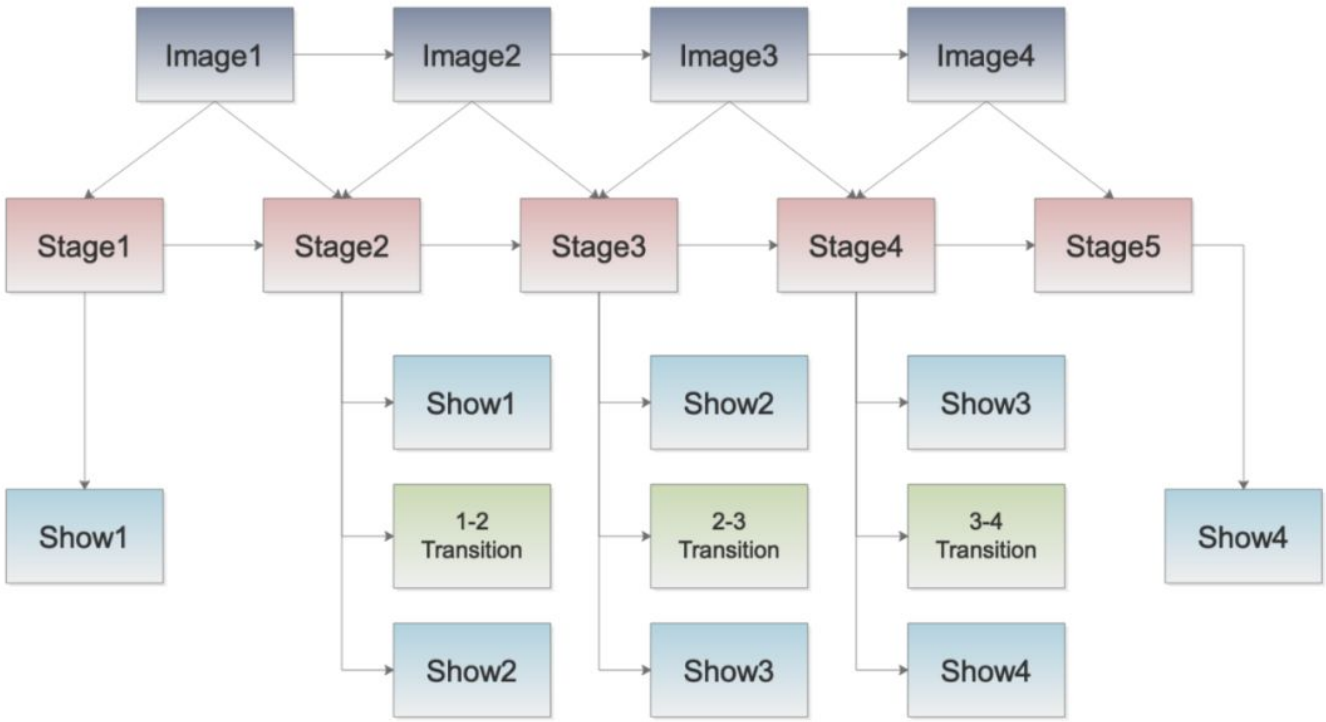
这里不需要手动将图片转换为YUV数据，先把图片解码为bitmap，然后通过texImage2D上传图片纹理到GPU中即可。

最后就是根据图片纹理的uv坐标，根据外部时间戳来驱动纹理变化，实现特效。

三、转场时间线

对于一个图片列表，在合成过程中如何衔接前后序列图片的展示和过渡时机，决定了最终的视频效果。

假设有图片合集{1,2,3,4}，按序合成，可以有如下的时间线：



每个Stage是合成过程中的一个最小单元，首尾的两个Stage最简单，只是单纯的显示图片。中间阶段的Stage，包括了过渡过程中前后两张图片的展示和过渡动画的时间戳定义。

假设每张图片的展示时间为showT(ms)，动画的时间为animT(ms)。

相邻Stage中同一张图的静态显示时间的总和为一张图的总显示时间，则首尾两个Stage的有效时长为showT/2，中间的过渡Stage有效时长为showT+animT。

其中过渡动画的时间段又需要分为：

- 前序退场起始点enterStartT，前序动画开始时间点。
- 前序退场结束点enterEndT，前序动画结束时间点。
- 后序入场起始点exitStartT，后序动画开始时间点。
- 后序入场结束点exitEndT，后序动画结束时间点。

动画时间线一般只定义为非淡入淡出外的其他特效使用。为了过渡的视觉连续性，前后序图片的淡入和淡出是贯穿整个动画时间的。考虑到序列的衔接性，退场完毕后会立刻入场，因此enterEndT=exitStartT。

四、OpenGL特效

1.基础架构

按照前面时间线定义回调接口，用于处理动画参数：

```
//参数初始化
protected abstract void onPhaseInit();
//前序动画,enterRatio(0-1)
protected abstract void onPhaseEnter(float enterRatio);
//后序动画,exitRatio(0-1)
protected abstract void onPhaseExit(float exitRatio);
//动画结束
protected abstract void onPhaseFinish();
//一帧动画执行完毕，步进
protected abstract void onPhaseStep();
```

定义几个通用的片段着色器变量，辅助过渡动画的处理：

```
//前序图片的纹理
uniform sampler2D preTexture
//后序图片的纹理
uniform sampler2D nextTexture;
//过渡动画总体进度，0到1
uniform float progress;
//窗口的长宽比例
uniform float canvasRatio;
//透明度变化
uniform float canvasAlpha;
```

前后序列的混合流程，根据动画流程计算出的两个纹理的UV坐标混合颜色值：

```
vec4 fromColor = texture2D(sTexture, fromUv);
vec4 nextColor = texture2D(nextTexture, nextUv);
vec4 mixColor = mix(fromColor, nextColor, mixIntensity);
gl_FragColor = vec4(mixColor.rgb, canvasAlpha);
```

解析图片，先读取Exif信息获取旋转值，再将旋转矩阵应用到bitmap上，保证上传的纹理图片与用户在相册中看到的旋转角度是一致的：

```
ExifInterface exif = new ExifInterface(imageFile);
orientation = exif.getAttributeInt(ExifInterface.TAG_ORIENTATION, Exif
int rotation = parseRotation(orientation);
Matrix matrix = new Matrix(rotation);
mImageBitmap = Bitmap.createBitmap(mOriginBitmap, 0, 0, mOriginBitmap.
```

在使用图片之前，还要根据最终的视频宽高调整OpenGL窗口尺寸。同时纹理的贴图坐标的起始(0,0)是在纹理坐标系的左下角，而Android系统上canvas坐标原点是在左上角，需要将图片做一次y轴的翻转，不然图片上传后是垂直镜像。

```
//根据窗口尺寸生成一个空的bitmap
mCanvasBitmap = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_
```

```
Canvas bitmapCanvas = new Canvas(mCanvasBitmap);
//翻转图片
bitmapCanvas.scale(1, -1, bitmapCanvas.getWidth() / 2f, bitmapCanvas.g
```

上传图片纹理，并记录纹理的handle：

```
int[] textures = new int[1];
GLES20.glGenTextures(1, textures, 0);
int textureId = textures[0];
GLES20.glBindTexture(textureType, textureId);
GLES20.glTexParameterf(textureType, GLES20.GL_TEXTURE_MIN_FILTER, GLES
GLES20.glTexParameterf(textureType, GLES20.GL_TEXTURE_MAG_FILTER, GLES
GLES20.glTexParameterf(textureType, GLES20.GL_TEXTURE_WRAP_S, GLES20.G
GLES20.glTexParameterf(textureType, GLES20.GL_TEXTURE_WRAP_T, GLES20.G
GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, bitmap, 0);
GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, 0);
```

加载第二张图片时要开启非0的其他纹理单元，过渡动画需要同时操作两个图片纹理：

```
GLES20.glActiveTexture(GLES20.GL_TEXTURE1);
```

最后是实际绘制的部分，因为用到了透明度渐变，要手动开启GL_BLEND功能，并注意切换正在操作的纹理：

```
//清除画布
GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT | GLES20.GL_DEPTH_BUFFER_BIT
GLES20.glUseProgram(mProgramHandle);

//绑定顶点坐标
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, mVertexBufferName);
GLES20.glVertexAttribPointer(getHandle(ATTRIBUTE_VEC4_POSITION), GLCon
        false, GLConstants.VERTICES_DATA_STRIDE_BYTES, GLConstants.VER
GLES20.glEnableVertexAttribArray(getHandle(ATTRIBUTE_VEC4_POSITION));
```

```
GLES20.glVertexAttribPointer(getHandle(ATTRIBUTE_VEC4_TEXTURE_COORD), (
        false, GLConstants.VERTICES_DATA_STRIDE_BYTES, GLConstants.VER
GLES20.glEnableVertexAttribArray(getHandle(ATTRIBUTE_VEC4_TEXTURE_COOR

//激活有效纹理
GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
//绑定图片纹理坐标
GLES20.glBindTexture(targetTexture, texName);
GLES20.glUniform1i(getHandle(UNIFORM_SAMPLER2D_TEXTURE), 0);

//开启透明度混合
GLES20.glEnable(GLES20.GL_BLEND);
GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA)

//绘制三角形条带
GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, 4);

//重置环境参数绑定
GLES20.glDisableVertexAttribArray(getHandle(ATTRIBUTE_VEC4_POSITION));
GLES20.glDisableVertexAttribArray(getHandle(ATTRIBUTE_VEC4_TEXTURE_COO
GLES20.glBindTexture(targetTexture, 0);
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, 0);
```

2.平移覆盖转场

I.着色器实现

```
uniform int direction;
void main(void) {
    float intensity;
    if (direction == 0) {
        intensity = step(0.0 + coord.x,progress);
    } else if (direction == 1) {
        intensity = step(1.0 - coord.x,progress);
    } else if (direction == 2) {
        intensity = step(1.0 - coord.y,progress);
    } else if (direction == 3) {
```

```
            intensity = step(0.0 + coord.y,progress);
        }
        vec4 mixColor = mix(fromColor, nextColor, intensity);
    }
```

GLSL中的step函数定义如下，当x<edge是返回0，反之则返回1：

```
Declaration:
genType step(genType edge, genType x);

Parameters:
edge Specifies the location of the edge of the step function.
x Specify the value to be used to generate the step function.
```

已知我们有前后两张图，将他们覆盖展示。然后从一个方向逐渐修改这一条轴上的所扫过的像素的intensity值，隐藏前图，展示后图。经过时钟动画驱动后就有了覆盖转场的效果。

再定义一个direction参数，控制扫描的方向，即可设置不同的转场方向，有PPT翻页的效果。

II.效果图

3.像素化转场

I.着色器实现

```
uniform float squareSizeFactor;
uniform float imageWidthFactor;
uniform float imageHeightFactor;
void main(void) {
    float revProgress = (1.0 - progress);
    float distFromEdges = min(progress, revProgress);
    float squareSize = (squareSizeFactor * distFromEdges) + 1.0;

    float dx = squareSize * imageWidthFactor;
```

```
    float dy = squareSize * imageHeightFactor;
    vec2 coord = vec2(dx * floor(uv.x / dx), dy * floor(uv.y / dy));
    vec4 fromColor = texture2D(preTexture, coord);
    vec4 nextColor = texture2D(nextTexture, coord);
    vec4 mixColor = mix(fromColor, nextColor, progress);
};
```

首先是定义像素块的效果，我们需要像素块逐渐变大，到动画中间值时再逐渐变小到消失。

通过对progress(0到1)取反向值1-progress，得到distFromEdges，可知这个值在progress从0到0.5时会从0到0.5，在0.5到1时会从0.5到0，即达到了我们需要的变大再变小的效果。

像素块就是一整个方格范围内的像素都是同一个颜色，视觉效果看起来就形成了明显的像素间隔。如果我们将一个方格范围内的纹理坐标都映射为同一个颜色，即实现了像素块的效果。

squareSizeFactor是影响像素块大小的一个参数值，设为50，即最大像素块为50像素。

imageWidthFactor和imageHeightFactor是窗口高宽取倒数，即1/width和1/height。

通过dx * floor(uv.x / dx)和dy * floor(uv.y / dy)的两次坐标转换，就把一个区间范围内的纹理都映射为了同一个颜色。

II.效果图

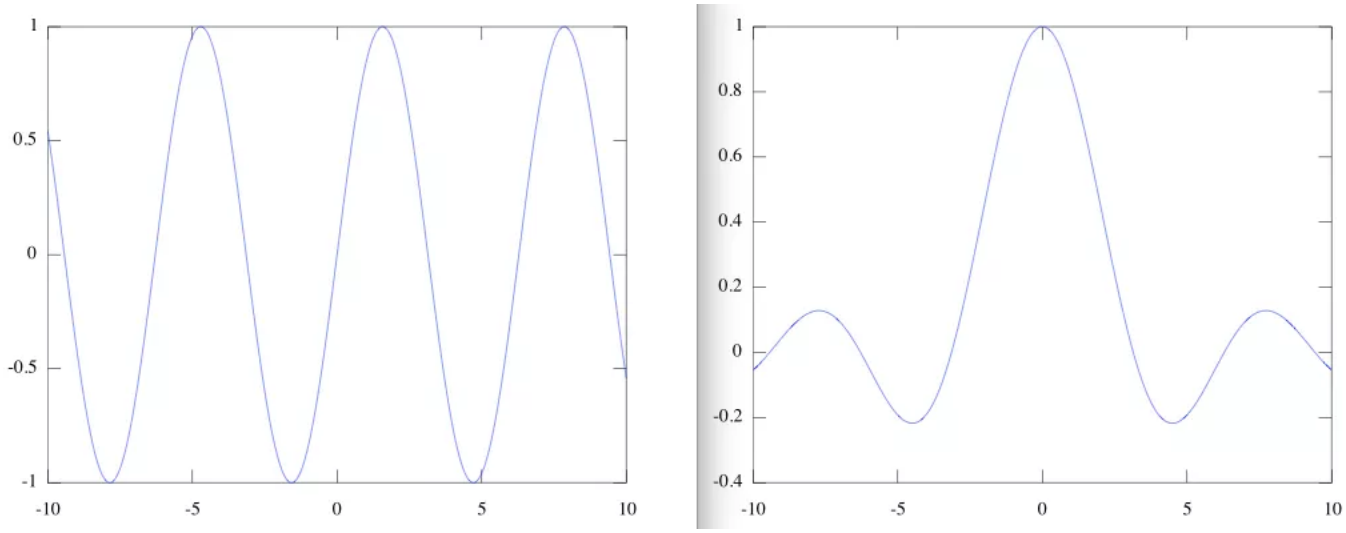
4.水波纹特效

I.数学原理

水波纹路的周期变化，实际就是三角函数的一个变种。目前业界最流行的简易水波纹实现，Adrian的博客中描述了基本的数学原理：

水波纹实际是Sombero函数的求值，也就是sinc函数的2D版本。

下图的左边是sin函数的图像，右边是sinc函数的图像，可以看到明显的水波纹特征。



博客中同时提供了一个WebGL版本的着色器实现，不过功能较简单，只是做了效果验证。

将其移植到OpenGLES中，并做参数调整，即可整合到图片转场特效中。

完整的水波纹片段着色器如下：

```
uniform float mixIntensity;
uniform float rippleTime;
uniform float rippleAmplitude;
uniform float rippleSpeed;
uniform float rippleOffset;
uniform vec2 rippleCenterShift;
void main(void) {
    //纹理位置坐标归一化
    vec2 curPosition = -1.0 + 2.0 * vTextureCoord;
    //修正相对波纹中心点的位置偏移
    curPosition -= rippleCenterShift;
    //修正画面比例
    curPosition.x *= canvasRatio;
    //计算波纹里中心点的长度
    float centerLength = length(curPosition);
    //计算波纹出现的纹理位置
    vec2 uv = vTextureCoord + (curPosition/centerLength)*cos(centerLen
    vec4 fromColor = texture2D(preTexture, uv);
    vec4 nextColor = texture2D(nextTexture, uv);
    vec4 mixColor = mix(fromColor, nextColor, mixIntensity);
    gl_FragColor = vec4(mixColor.rgb, canvasAlpha);
}
```

其中最关键的代码就是水波纹像素坐标的计算：

vTextureCoord + (curPosition/centerLength)*cos(centerLength*rippleAmplitude-rippleTime*rippleSpeed)*rippleOffset;

简化一下即：vTextureCoord + A*cos(L*x - T*y)*rippleOffset，一个标准的余弦函数。

vTextureCoord是当前纹理的归一化坐标(0,0)到(1,1)之间。

curPosition是(-1,-1)到(1,1)之间的当前像素坐标。

centerLength是当前点距离波纹中心的距离。

curPosition/centerLength即是线性代数中的单位矢量，这个参数用来决定波纹推动的方向。

cos(centerLength*rippleAmplitude-rippleTime*rippleSpeed)通过一个外部时钟rippleTime来驱动cos函数生成周期性的相位偏移。

rippleAmplitude是相位的扩大因子。

rippleSpeed调节函数的周期，即波纹传递速度。

最后将偏移值乘以一个最大偏移范围rippleOffset(一般为0.03)，限定单个像素的偏移范围，不然波纹会很不自然。

II.时间线动画

设定颜色混合，在整个动画过程中，图1逐渐消失(1到0)，图2逐渐展现(0到1)。

设定画布透明度，在起始时为1，逐渐变化到0.7，最后再逐渐回到1。

设定波纹的振幅，在起始时最大，过渡到动画中间点到最小，最后逐渐变大到动画结束。

设定波纹的速度，在起始时最大，过渡到动画中间点到最小，最后逐渐变大到动画结束。

设定波纹的像素最大偏移值，在起始时最大，过渡到动画中间点到最小，最后逐渐变大到动画结束。

```
protected void onPhaseInit() {
    mMixIntensity = MIX_INTENSITY_START;
    mCanvasAlpha = CANVAS_ALPHA_DEFAULT;
    mRippleAmplitude = 0;
    mRippleSpeed = 0;
    mRippleOffset = 0;
}
protected void onPhaseEnter(float enterRatio) {
    mMixIntensity = enterRatio * 0.5f;
    mCanvasAlpha = 1f - enterRatio;
    mRippleAmplitude = enterRatio * RIPPLE_AMPLITUDE_DEFAULT;
    mRippleSpeed = enterRatio * RIPPLE_SPEED_DEFAULT;
    mRippleOffset = enterRatio * RIPPLE_OFFSET_DEFAULT;
}
protected void onPhaseExit(float exitRatio) {
    mMixIntensity = exitRatio * 0.5f + 0.5f;
    mCanvasAlpha = exitRatio;
    mRippleAmplitude = (1f - exitRatio) * RIPPLE_AMPLITUDE_DEFAULT;
    mRippleSpeed = (1f - exitRatio) * RIPPLE_SPEED_DEFAULT;
    mRippleOffset = (1f - exitRatio) * RIPPLE_OFFSET_DEFAULT;
}
protected void onPhaseFinish() {
    mMixIntensity = MIX_INTENSITY_END;
    mCanvasAlpha = CANVAS_ALPHA_DEFAULT;
    mRippleAmplitude = 0;
    mRippleSpeed = 0;
    mRippleOffset = 0;
}
protected void onPhaseStep() {
    if (mCanvasAlpha < CANVAS_ALPHA_MINIMUN) {
        mCanvasAlpha = CANVAS_ALPHA_MINIMUN;
    }
}
```

将本次动画帧的参数更新到着色器：

```
long globalTimeMs = GLClock.get();
GLES20.glUniform1f(getHandle("rippleTime"), globalTimeMs / 1000f);
GLES20.glUniform1f(getHandle("rippleAmplitude"), mRippleAmplitude);
GLES20.glUniform1f(getHandle("rippleSpeed"), mRippleSpeed);
GLES20.glUniform1f(getHandle("rippleOffset"), mRippleOffset);
GLES20.glUniform2f(getHandle("rippleCenterShift"), mRippleCenterX, mRi
```

其中GLClock是一个与mediacodec编码时间戳绑定的外部时钟，用于同步合成时间和动画时间戳位置。

III.最终效果

图片展示时长：3s

过渡动画时长：1.5s

波纹中心为图片中心点

5.随机方格

I.噪声函数

我们想实现的效果是前一个画面上随机出现很多方块，每个方块中展示下一张图的画面，当图片上每一块位置都形成方块后就完成了画面的转换。

首先就需要解决随机函数的问题。虽然Java上有很多现成的随机函数，但是GLSL是个很底层的语言，基本上除了加减乘除其他的都需要自己想办法。这个着色器里用的rand函数是流传已久几乎找不到来源的一个实现，很有上古时期游戏编程代码的风格，有魔法数，代码只要一行，证明要写两页。

网上一个比较靠谱且简洁的说明是StackOverflow上的，这个随机函数实际是一个hash函数，对每一个相同的(x,y)输入都会有相同的输出。

II.着色器实现

```
uniform vec2 squares;
uniform float smoothness;
float rand(vec2 co) {
    return fract(sin(dot(co.xy ,vec2(12.9898,78.233))) * 43758.5453);
};
void main(void) {
    vec2 uv = vTextureCoord.xy;
    float randomSquare = rand(floor(squares * uv));
    float intensity = smoothstep(0.0, -smoothness, randomSquare - (pro
    vec4 fromColor = texture2D(preTexture, uv);
    vec4 nextColor = texture2D(nextTexture, uv);
    vec4 mixColor = mix(fromColor, nextColor, intensity);
    gl_FragColor = vec4(mixColor.rgb, canvasAlpha);
}
```

首先将当前纹理坐标乘以方格大小，用随机函数转换后获取这个方格区域的随机渐变值。

然后用smoothstep做一个厄米特插值，将渐变的intensity平滑化。

最后用这个intensity值mix前后图像序列。

III.效果图

想学区块链吗？

1000篇好文带你学懂区块链，更有认证问答官资格等你领！

识别下方二维码或点击“阅读原文”帮你破解区块链密码。