FBO

Frame Buffer object

为什么要用FBO

> 我们需要对纹理进行多次渲染采样时，而这些渲染采样是不需要展示给用户看的，所以我们就可以用一个单独的缓冲对象（离屏渲染）来存储我们的这几次渲染采样的结果，等处理完后才显示到窗口上

优势

> 提高渲染效率，避免闪屏，可以很方便的实现纹理共享等。

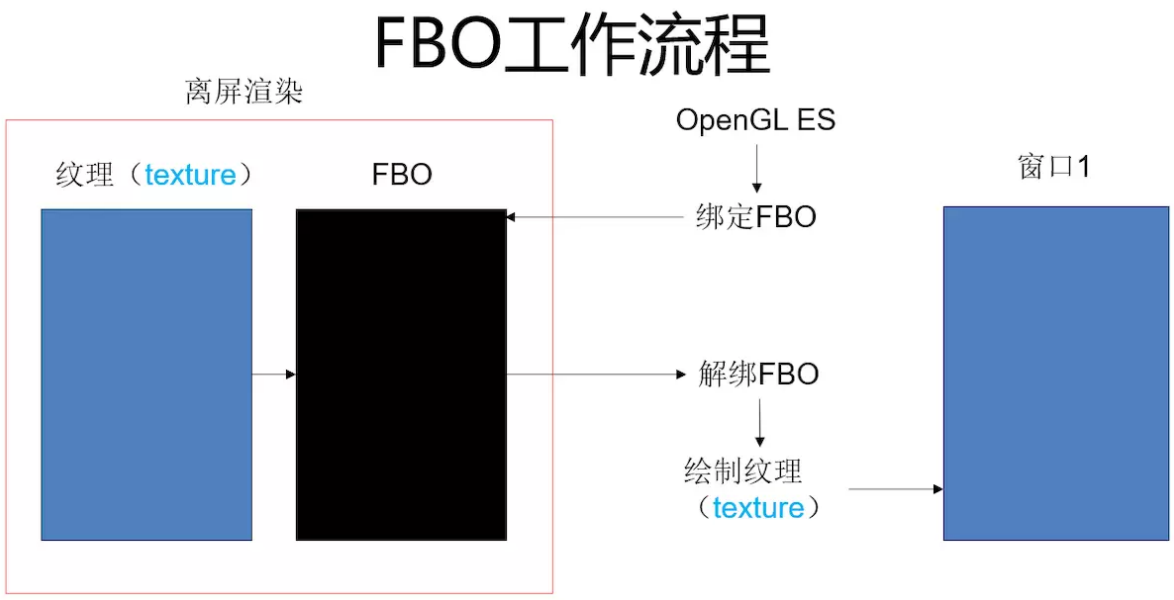
渲染方式

> 1. 渲染到纹理（Texture）- 图像渲染
> 2. 渲染到缓冲区（Render）- 深度测试和模板测试

FBO纹理的坐标系

FBO坐标系

渲染到纹理



工作流程

创建FBO的步骤:

```
//1. 创建FBO
int[] frameBuffer = new int[1];
GLES20.glGenFramebuffers(1, frameBuffer, 0);
fboId = frameBuffer[0];

//2. 绑定FBO
GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, fboId);

//3. 创建FBO纹理
fboTextureId = createTexture();

 //4. 把纹理绑定到FBO
GLES20.glFramebufferTexture2D(GLES20.GL_FRAMEBUFFER,
GLES20.GL_COLOR_ATTACHMENT0,GLES20.GL_TEXTURE_2D, fboTextureId, 0);

//5. 设置FBO分配内存大小
GLES20.glTexImage2D(GLES20.GL_TEXTURE_2D, 0, GLES20.GL_RGBA,bitmap.getWidth(),
bitmap.getHeight(),0, GLES20.GL_RGBA, GLES20.GL_UNSIGNED_BYTE, null);

//6. 检测是否绑定从成功
if (GLES20.glCheckFramebufferStatus(GLES20.GL_FRAMEBUFFER)!=
GLES20.GL_FRAMEBUFFER_COMPLETE) {
    Log.e("zzz", "glFramebufferTexture2D error");
}

//7. 解绑纹理和FBO
GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, 0);
GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, 0);
```

使用FBO的步骤:

```
//1. 绑定fbo
```

```
GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, fboId);

//2. FBO绘制
GLES20.glUseProgram(program);
//绑定渲染纹理
GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, imageTextureId);
//...
//解绑纹理
GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, 0);
//解绑fbo
GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, 0);

//3. 根据绑定到fbo上的纹理id，渲染
 GLES20.glUseProgram(program);
//绑定渲染纹理
GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureId);
```

示例代码如下:

TexureRender.java

```
import android.content.Context;
import android.graphics.BitmapFactory;
import android.opengl.GLES20;

public class TexureRender implements EglSurfaceView.Renderer {
    private BitmapFboTexture bitmapFboTexture;
    private BitmapRenderTexture bitmapRenderTexture;

    public TexureRender(Context context) {
        bitmapFboTexture = new BitmapFboTexture(context);

bitmapFboTexture.setBitmap(BitmapFactory.decodeResource(context.getResources(),R
.mipmap.bg));

        bitmapRenderTexture = new BitmapRenderTexture(context);
    }

    @Override
    public void onSurfaceCreated() {
        bitmapFboTexture.onSurfaceCreated();
        bitmapRenderTexture.onSurfaceCreated();
    }

    @Override
    public void onSurfaceChanged(int width, int height) {
        //宽高
        GLES20.glViewport(0, 0, width, height);

        bitmapFboTexture.onSurfaceChanged(width, height);
        bitmapRenderTexture.onSurfaceChanged(width, height);
    }

    @Override
    public void onDrawFrame() {
```

```
        //清空颜色
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
        //设置背景颜色
        GLES20.glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
        //FBO处理
        bitmapFboTexture.draw();
        //通过FBO处理之后，拿到纹理id，然后渲染
        bitmapRenderTexture.draw(bitmapFboTexture.getFboTextureId());
    }
}
```

GL_TEXTURE_MIN_FILTER和GL_TEXTURE_MAG_FILTER

当纹理的大小和渲染屏幕的大小不一致时会出现两种情况：

第一种情况：纹理大于渲染屏幕，将会有一部分像素无法映射到屏幕上，对应于GL_TEXTURE_MIN_FILTER。

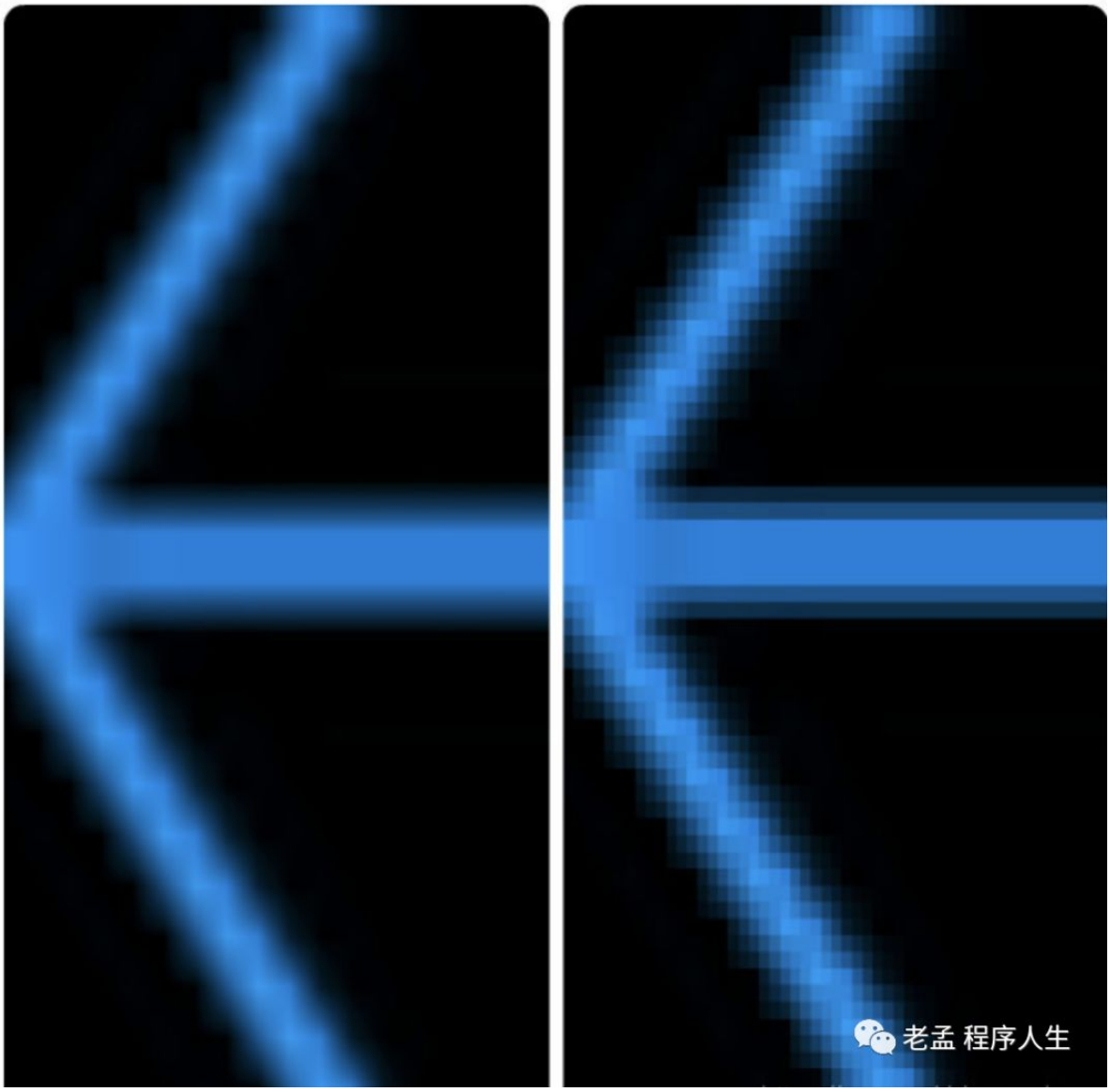
第二种情况：纹理小于渲染屏幕，没有足够的像素映射到屏幕上，GL_TEXTURE_MAG_FILTER。

可设置的值为GL_NEAREST 、GL_LINEAR。

GL_NEAREST：使用纹理中坐标最接近的一个像素的颜色作为需要绘制的像素颜色。

GL_LINEAR：使用纹理中坐标最接近的若干个颜色，通过加权平均算法得到需要绘制的像素颜色，与GL_NEAREST比较速度较慢。

视觉效果如下图：



左面的图为GL_LINEAR的效果，是放大模糊的效果。右面的图是GL_NEAREST的效果，锯齿比较明显，但没有模糊。

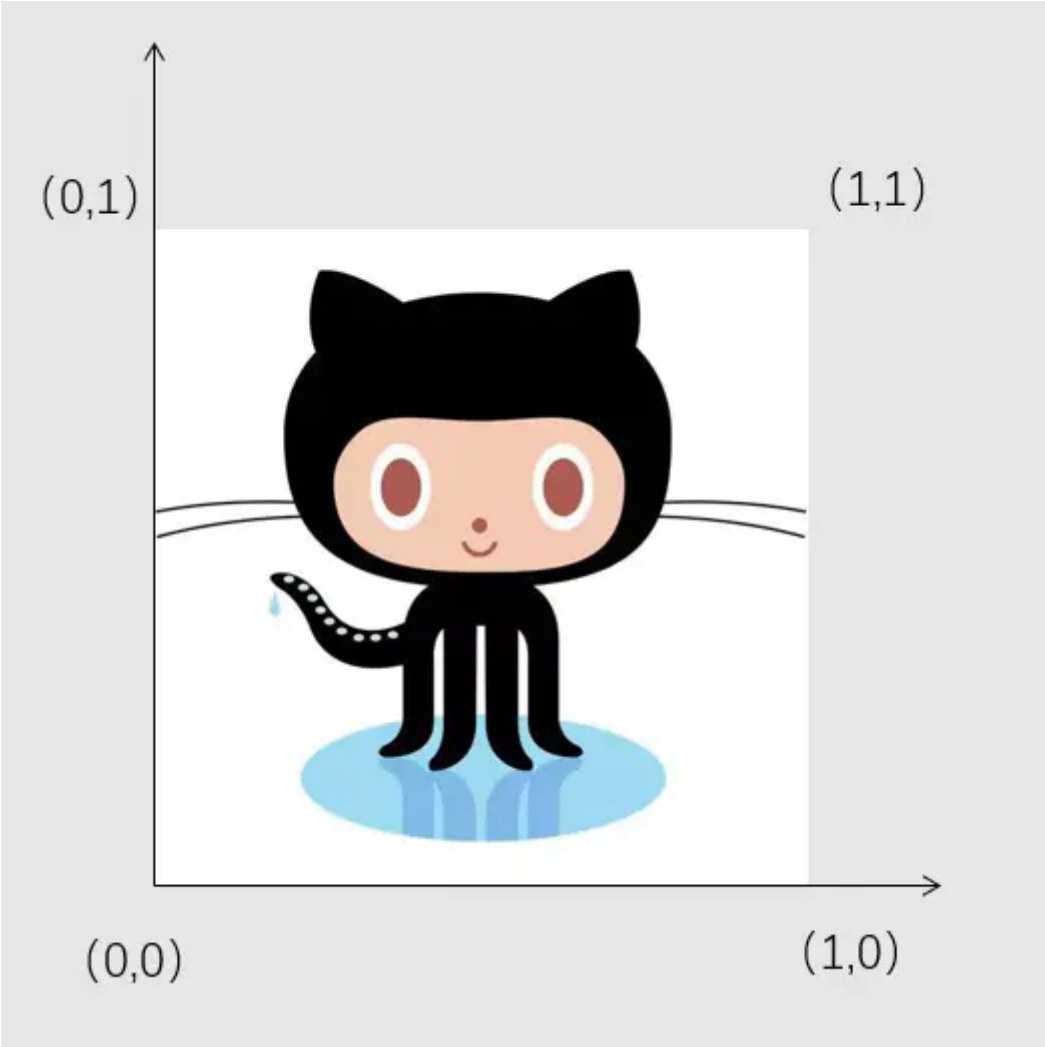
纹理映射到屏幕

纹理是什么

纹理将指定纹理图像的一部分映射到纹理化为活动的每个图形基元上。当前片段着色器或顶点着色器使用内置纹理查找函数时，纹理处于活动状态。

纹理（TEXTURE），即物体表面的样子。在计算机的世界中，我们能够绘制的仅仅是一些非常基础的形状，比如点、线、三角形，这些基础显然是无法将一个现实世界中的物体很好的描述在屏幕上的。通常我们通过纹理映射将物体表面图片贴到物体的几何图形上面，完成贴图的过程，将物体从现实世界中模拟到虚拟世界中。

纹理的基础单元是纹素（Texel，即texture element或texture pixel的合成字），亦如屏幕的基础单元是像素。屏幕上有自己的坐标系，纹理也有，即纹理坐标，一个维度称为 s，另一个维度称为 t，其范围都在 [0,1]之间。纹理坐标上，是纹素。



纹理坐标

如上图所示，在 OpenGL的二维世界中，本质上，纹理就是一个二维数组（图像数据），而纹素就是这个二维数组中的值。

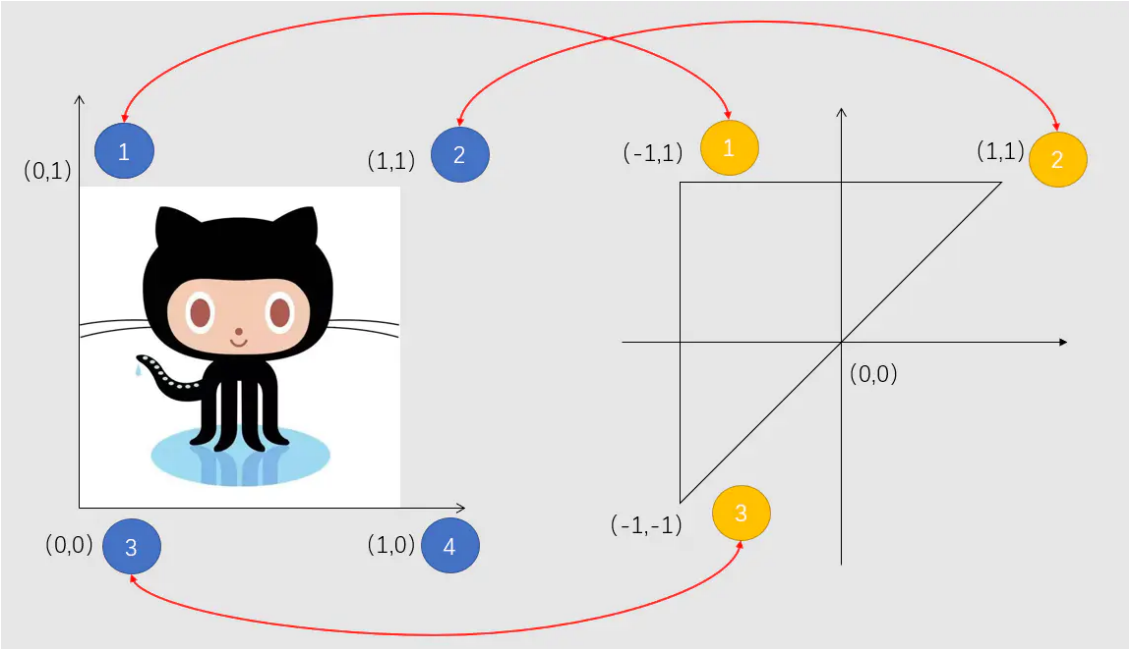
纹理映射到屏幕

接下来，首先我们讨论下图像数据是如何从纹理坐标下被投射到屏幕上的。

坐标映射

上面我们就讲过纹理坐标系。两个维度，范围都在[0,1]之间。无论是使用纯色渲染，还是图片渲染，实际上都是在进行颜色渲染，最终，都需要将那块颜色映射到归一化坐标系中。而归一化坐标系中两个维度都是在[-1,1]之间。
由于这种坐标系的不一致，就需要将纹理坐标系中的图片映射到归一化坐标系中。这里，首先我们会遇到两个问题：

- 将纹理坐标系中的哪个区域块取出来进行映射。
- 取出来的区域块又该如何如何对应到归一化坐标系中通过顶点定义的外形中。

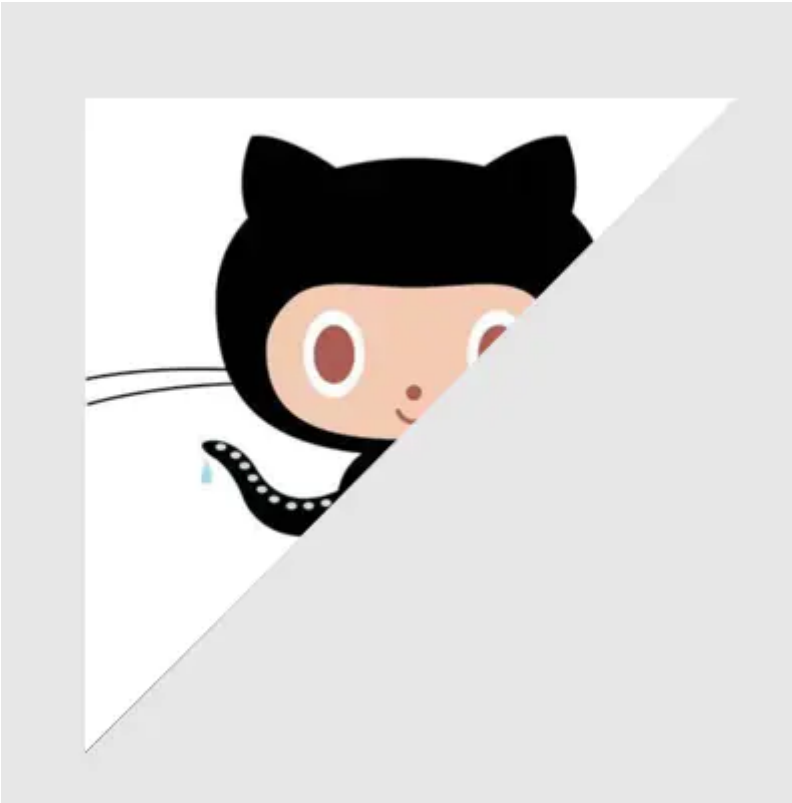


纹理坐标和归一化坐标的对应关系

如上图所示，我们可以得到以下几点：

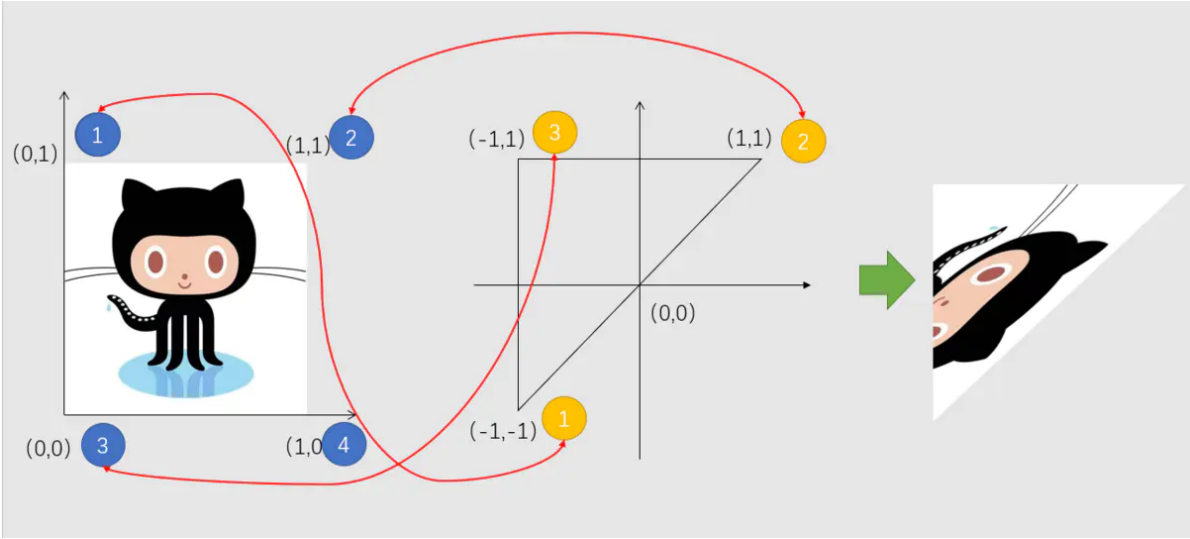
- 要选取图片中的哪个区域是由我们通过**纹理顶点坐标**定义的。上图中，我们通过定义(0,0),(0,1),(1,1),(1,0)，实际上选取了整个图片部分作为贴片。
- 要进行贴图的部分定义好了，它所要贴到的地方，实际上和我们定义的顶点坐标按照**定义顺序**一一对应（映射）。即，我们定义的第一个纹理顶点坐标，对应我们定义的第一个顶点坐标。

最终，我们可以得到：



屏幕上显示的图形

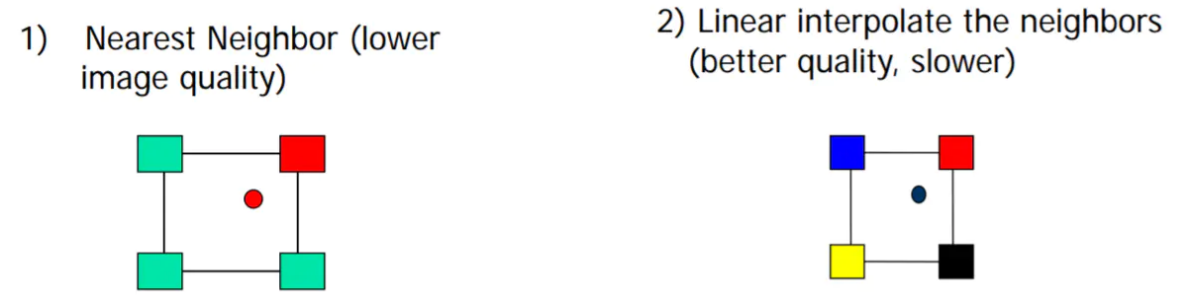
为了明显起见，这里再给出一张图（改变顶点定义的顺序）：



改变映射顺序后的结果

纹理过滤(Texture Filtering)

纹理坐标和归一化坐标通常并不能实现一比一的映射关系，**纹理可能会比归一化坐标系中的几何图形大，也可能更小。**这样就带来了这样的问题：在不能实现一比一映射的时候，该如何在纹理坐标中**采样**显示到归一化坐标中。这就是纹理过滤，实际上是在处理放大和缩小纹理的操作。
常见的过滤方式有两种：最邻近过滤和双线性过滤。
对于最邻近过滤，顾名思义，是为每个片段选择最近的纹素；而双线性过滤，使用双线性插值平滑像素之间的过滤。如下图所示，显然线性过滤会有更好的效果，但是也会消耗更多的计算能力：



最邻近过滤和双线性过滤

我们可以看到，对于双线性过滤这种方式，特别是在进行缩小采样的时候，当缩小比例过大，会丢掉很多的细节，因为不管这个比例如何，它只会选取周围的四个点（即四个纹理元素）。为了解决这样的问题，可以使用 MIP 贴图的方式解决。关于 MIP 更多信息可以参考这里

纹理环绕(Texture Wrapping)

上面坐标中，我们指定的纹理坐标范围均是在 [0,1]之间，但是如果超出了这个范围， 此时刻，我们选择的纹理区域大于了实际图片大小，纹理区域空白部分该如何处理，就是纹理环绕问题。下图参考

纹理环绕模式

综上，我们了解了纹理坐标及其映射。下面，我们可以进行实际地编码工作了。

在程序中使用纹理

通常，我们在程序中使用纹理的步骤是这样的：