

## 原 最简单的视音频播放示例6：OpenGL播放YUV420P（通过Texture，使用Shader）

2014年10月28日 00:35:40 阅读数：35170

=====

最简单的视音频播放示例系列文章列表：

[最简单的视音频播放示例1：总述](#)

[最简单的视音频播放示例2：GDI播放YUV, RGB](#)

[最简单的视音频播放示例3：Direct3D播放YUV，RGB（通过Surface）](#)

[最简单的视音频播放示例4：Direct3D播放RGB（通过Texture）](#)

[最简单的视音频播放示例5：OpenGL播放RGB/YUV](#)

[最简单的视音频播放示例6：OpenGL播放YUV420P（通过Texture，使用Shader）](#)

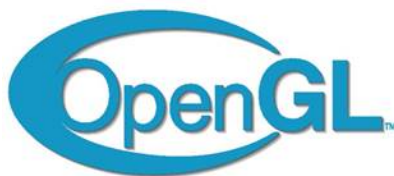
[最简单的视音频播放示例7：SDL2播放RGB/YUV](#)

[最简单的视音频播放示例8：DirectSound播放PCM](#)

[最简单的视音频播放示例9：SDL2播放PCM](#)

=====

本文记录OpenGL播放视频的技术。上一篇文章中，介绍了一种简单的使用OpenGL显示视频的方式。但是那还不是OpenGL显示视频技术的精髓。和Direct3D一样，OpenGL更好的显示视频的方式也是通过纹理（Texture）。本文介绍OpenGL通过纹理的方式显示视频的技术。



### OpenGL中坐标和Direct3D坐标的不同

OpenGL中的纹理的坐标和Direct3D中的坐标是不一样的。

在Direct3D中。纹理坐标如下图所示。取值是0到1。坐标系原点在左上角。



物体表面坐标如下图所示。取值是实际的像素值。坐标系原点在左上角。



OpenGL纹理坐标取值范围是0-1，坐标原点位于左下角。这一点和Direct3D是不同的，Direct3D纹理坐标的取值虽然也是0-1，但是他的坐标原点位于左上角。



在OpenGL中，物体表面坐标取值范围是-1到1。坐标系原点在中心位置。



## OpenGL视频显示的流程

有关纹理方面的知识已经在文章《[最简单的视音频播放示例4：Direct3D播放RGB（通过Texture）](#)》中有详细的记录。OpenGL中纹理的概念和Direct3D中纹理的概念基本上是等同的，因此不再重复记录了。

本文记录的程序，播放的是YUV420P格式的像素数据。上一篇文章中的程序也可以播放YUV420P格式的像素数据。但是它们的原理是不一样的。上一篇文章中，输入的YUV420P像素数据通过一个普通的函数转换为RGB数据后，发送给OpenGL播放。也就是像素的转换是通过CPU完成的。本文的程序，输入的YUV420P像素数据通过Shader转换为YUV数据，发送给OpenGL播放。像素的转换是通过显卡上的GPU完成的。通过本程序，可以了解使用OpenGL进行GPU编程的基础知识。

使用Shader通过OpenGL的纹理（Texture）播放视频一般情况下需要如下步骤：

1.
  - 初始化
    - 1) 初始化
    - 2) 创建窗口
    - 3)

设置绘图函数

4)

设置定时器

5)

初始化Shader

初始化Shader的步骤比较多，主要可以分为3步：创建Shader，创建Program，初始化Texture。

(1)

创建一个Shader对象

- 1) 编写Vertex Shader和Fragment Shader源码。
- 2) 创建两个shader 实例。
- 3) 给Shader实例指定源码。
- 4) 在线编译shaer源码。

(2)

创建一个Program对象

- 1) 创建program。
- 2) 绑定shader到program。
- 3) 链接program。
- 4) 使用porgram。

(3)

初始化Texture。可以分为以下步骤。

- 1) 定义定点数组
- 2) 设置顶点数组
- 3) 初始化纹理

6)

进入消息循环

2.

**循环显示画面**

1)

设置纹理

2)

绘制

3)

显示

下面详述一下使用Shader通过OpenGL的纹理的播放YUV的步骤。有些地方和上一篇文章是重复的，会比较简单的提一下。

1.

**初始化**

1)

**初始化**

glutInit()用于初始化glut库。它原型如下：

```
1. void glutInit(int *argcp, char **argv);
```

它包含两个参数：argcp和argv。一般情况下，直接把main()函数中的argc, argv传递给它即可。

glutInitDisplayMode()用于设置初始显示模式。它的原型如下。

```
1. void glutInitDisplayMode(unsigned int mode);
```

需要注意的是，如果使用双缓冲（GLUT\_DOUBLE），则需要用glutSwapBuffers ()绘图。如果使用单缓冲（GLUT\_SINGLE），则需要用glFlush()绘图。

在使用OpenGL播放视频的时候，我们可以使用下述代码：

```
1. glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB );
```

2)

**创建窗口**

glutInitWindowPosition()用于设置窗口的位置。可以指定x, y坐标。

glutInitWindowSize()用于设置窗口的大小。可以设置窗口的宽, 高。

glutCreateWindow()创建一个窗口。可以指定窗口的标题。

上述几个函数十分基础，不再详细叙述。直接贴出一段示例代码：

```
[cpp]
1.  glutInitWindowPosition(100, 100);
2.  glutInitWindowSize(500, 500);
3.  glutCreateWindow("Simplest Video Play OpenGL");
```

3)

### 设置绘图函数

glutDisplayFunc()用于设置绘图函数。操作系统在必要时刻就会调用该函数对窗体进行重新绘制操作。类似于windows程序设计中处理WM\_PAINT消息。例如，当把窗口移动到屏幕边上，然后又移动回来的时候，就会调用该函数对窗口进行重绘。它的原型如下。

```
[cpp]
1.  void glutDisplayFunc(void (*func)(void));
```

其中(\*func)用于指定重绘函数。

例如在视频播放的时候，指定display()函数用于重绘：

```
[cpp]
1.  glutDisplayFunc(&display);
```

4)

### 设置定时器

播放视频的时候，每秒需要播放一定的画面（一般是25帧），因此使用定时器每间隔一段时间调用一下绘图函数绘制图形。定时器函数glutTimerFunc()的原型如下。

```
[cpp]
1.  void glutTimerFunc(unsigned int millis, void (*func)(int value), int value);
```

它的参数含义如下：

millis：定时的时间，单位是毫秒。1秒=1000毫秒。

(\*func)(int value)：用于指定定时器调用的函数。

value：给回调函数传参。比较高端，没有接触过。

如果只在主函数中写一个glutTimerFunc()函数的话，会发现只会调用该函数一次。因此需要在回调函数中再写一个glutTimerFunc()函数，并调用回调函数自己。只有这样才能实现反反复复循环调用回调函数。

例如在视频播放的时候，指定每40毫秒调用一次timeFunc ()函数：

主函数中：

```
[cpp]
1.  glutTimerFunc(40, timeFunc, 0);
```

而后在timeFunc()函数中如下设置。

```
[cpp]
1.  void timeFunc(int value){
2.      display();
3.      // Present frame every 40 ms
4.      glutTimerFunc(40, timeFunc, 0);
5.  }
```

这样就实现了每40ms调用一次display()。

5)

### 初始化Shader

初始化Shader的步骤比较多，主要可以分为3步：创建Shader，创建Program，初始化Texture。它们的步骤如下所示。

(1)

#### 创建一个Shader对象

Shader有点类似于一个程序的编译器。创建一个Shader可以分成以下4步：

- 1) 编写Vertex Shader和Fragment Shader源码。
- 2) 创建两个shader 实例:glCreateShader()。
- 3) 给Shader实例指定源码:glShaderSource()。
- 4) 在线编译shader源码 glCompileShader()。

下面详细分析这4步。

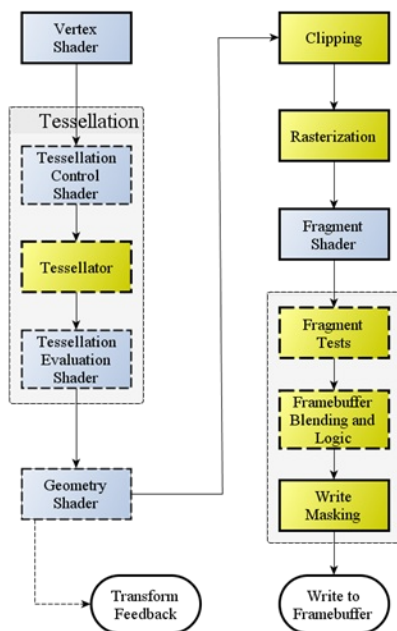
1)

#### 编写Vertex Shader和Fragment Shader源码。

在这里用到了一种新的语言：OpenGL Shader Language,简称GLSL。它是一种类似于C语言的专门为GPU设计的语言，它可以放在GPU里面被并行运行。

OpenGL的着色器有.fsh和.vsh两个文件。这两个文件在被编译和链接后就可以产生可执行程序与GPU交互。.vsh 是Vertex Shader（顶点着色器），用于顶点计算，可以理解控制顶点的位置，在这个文件中我们通常会传入当前顶点的位置，和纹理的坐标。.fsh 是Fragment Shader（片元着色器），在这里面我可以对于每一个像素点进行重新计算。

下面这张图可以更好的解释Vertex Shader和Fragment Shader的作用。这张图是OpenGL的渲染管线。其中的信息太多先不一一记录了。从图中可以看出，Vertex Shader在前，Fragment Shader在后。



在这里贴出本文的示例程序的fsh和vsh的代码。

Shader.vsh

```
[plain]
1. attribute vec4 vertexIn;
2. attribute vec2 textureIn;
3. varying vec2 textureOut;
4. void main(void)
5. {
6.     gl_Position = vertexIn;
7.     textureOut = textureIn;
8. }
```

Shader.fsh

```
[plain]
1. varying vec2 textureOut;
2. uniform sampler2D tex_y;
3. uniform sampler2D tex_u;
4. uniform sampler2D tex_v;
5. void main(void)
6. {
7.     vec3 yuv;
8.     vec3 rgb;
9.     yuv.x = texture2D(tex_y, textureOut).r;
10.    yuv.y = texture2D(tex_u, textureOut).r - 0.5;
11.    yuv.z = texture2D(tex_v, textureOut).r - 0.5;
12.    rgb = mat3( 1, 1, 1,
13.               0, -0.39465, 2.03211,
14.               1.13983, -0.58060, 0) * yuv;
15.    gl_FragColor = vec4(rgb, 1);
16. }
```

从上述代码中可以看出GLSL的语法和C语言很类似。每一个Shader程序都有一个main函数，这一点和c语言是一样的。这里的变量命名规则保持跟c一样就行了，注意gl开头的变量名是系统内置的变量。有以下几种变量：

attribute：外部传入vsh文件的变量，每一个顶点都会有这两个属性。变化率高，用于定义每个点。

varying：用于 vsh和fsh之间相互传递的参数。

uniform：外部传入vsh文件的变量。变化率较低，对于可能在整个渲染过程没有改变，只是个常量。

上文代码中使用了以下数据类型：

vec2：包含了2个浮点数的向量

vec3：包含了3个浮点数的向量

vec4：包含了4个浮点数的向量

sampler1D：1D纹理着色器

sampler2D：2D纹理着色器

sampler3D：3D纹理着色器

mat2：2\*2维矩阵

mat3：3\*3维矩阵

mat4：4\*4维矩阵

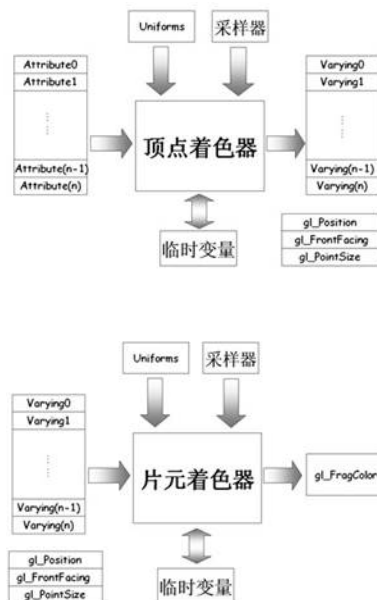
上文代码中还使用到了OpenGL的几个全局变量：

gl\_Position：原始的顶点数据在Vertex Shader中经过平移、旋转、缩放等数学变换后，生成新的顶点位置（一个四维（vec4）变量，包含顶点的 x、y、z 和 w 值）。新的顶点位置通过在Vertex Shader中写入gl\_Position传递到渲染管线的后继阶段继续处理。

gl\_FragColor：Fragment Shader的输出，它是一个四维变量（或称为 vec4）。gl\_FragColor 表示在经过着色器代码处理后，正在呈现的像素的 R、G、B、A 值。

Vertex Shader是作用于每一个顶点的，如果Vertex有三个点，那么Vertex Shader会被执行三次。Fragment Shader是作用于每个像素的，一个像素运行一次。从源代码中可以看出，像素的转换在Fragment Shader中完成。

在网上看到两张图可以很好地说明Vertex Shader和Fragment Shader的作用：



Vertex Shader（顶点着色器）主要是传入相应的Attribute变量、Uniforms变量、采样器以及临时变量，最后生成Varying变量，以及gl\_Posizion等变量。Fragment Shader（片元着色器）可以执行纹理的访问、颜色的汇总、雾化等操作，最后生成gl\_FragColor变量。有高手总结如下：“vsh负责搞定像素位置，填写gl\_Posizion；fsh负责搞定像素外观，填写 gl\_FragColor。”

2)

#### 创建两个shader 实例。

创建一个容纳shader的容器。用glCreateShader ()创建一个容纳shader的容器，它的原型如下：

```
1. int glCreateShader (int type)
```

其中type包含2种：

GLES20.GL\_VERTEX\_SHADER：Vertex Shader.

GLES20.GL\_FRAGMENT\_SHADER：Fragment Shader.

如果调用成功的话，函数将返回一个整形的正整数作为Shader容器的id。

3)

#### 给Shader实例指定源码。

Shader容器中添加shader的源代码。源代码应该以字符串数组的形式表示。glShaderSource函数的原型如下：

```
1. void glShaderSource (int shader, String string)
```

参数含义如下：

shader：是代表shader容器的id（由glCreateShader()返回的整形数）。



strings：是包含源程序的字符串数组。

如果感觉通过“字符串数组”的方式写源代码不太习惯的话，可以把源代码写到单独的一个文本文件里。然后在需要源代码的时候，读取该文本文件中的所有内容。

4)



#### 在线编译Shader源码。

使用glCompileShader()对shader容器中的源代码进行编译。函数的原型如下：

```
[cpp]    
1. void glCompileShader (int shader)
```

其中shader是代表Shader容器的id。

在编译完成后，可能需要调试。调试一个Shader是非常困难的。Shader的世界里没有printf，无法在控制台中打印调试信息。但是可以通过一些OpenGL提供的函数来获取编译和连接过程中的信息。在编译阶段使用glGetShaderiv获取编译情况。glGetShaderiv()函数原型如下：

```
[cpp]    
1. void glGetShaderiv (int shader, int pname, int[] params, int offset)
```

参数含义：

- shader：一个shader的id；
- pname：使用GL\_COMPILE\_STATUS；
- params：返回值，如果一切正常返回GL\_TRUE代，否则返回GL\_FALSE。

(2)

#### 创建一个Program对象

Program有点类似于一个程序的链接器。program对象提供了把需要做的事连接在一起的机制。在一个program中，shader对象可以连接在一起。

创建一个Program可以分成以下4步：

- 1) 创建program：glCreateProgram()
- 2) 绑定shader到program：glAttachShader()。

\*每个program必须绑定一个Vertex Shader 和一个Fragment Shader。

- 3) 链接program：glLinkProgram()。
- 4) 使用program：glUseProgram()。



下面详细分析这4步。

1)

#### 创建program。

首先使用glCreateProgram ()创建一个容纳程序（Program）的容器，我们称之为程序容器。

函数的原型如下：

```
[cpp]    
1. int glCreateProgram ()
```



如果函数调用成功将返回一个整形正整数作为该着色器程序的id。

2)

#### 绑定shader到program。

使用glAttachShader()将shader容器添加到程序中。这时的shader容器不一定需要被编译，他们甚至不需要包含任何的代码。

函数的原型如下：

```
[cpp]    
1. void glAttachShader (int program, int shader)
```

参数含义：

- program：着色器程序容器的id。
- shader：要添加的顶点或者片元shader容器的id。



Vertex Shader和Fragment Shader需要分别将他们各自的两个shader容器添加到程序容器中。

3)

#### 链接program。

使用glLinkProgram()链接程序对象。



函数的原型如下：

```
[cpp]    
1. void glLinkProgram (int program)
```

program是着色器程序容器的id。

如果任何类型为GL\_VERTEX\_SHADER的shader对象连接到program，它将产生在“顶点着色器”（Vertex Shader）上可执行的程序；如果任何类型为GL\_FRAGMENT\_SHADER的shader对象连接到program，它将产生在“像素着色器”（Pixel Shader）上可执行的程序。

在链接阶段使用glGetProgramiv()获取编译情况。glGetProgramiv ()函数原型如下：



```
[cpp]    
1. void glGetProgramiv (int program, int pname, int[] params, int offset)
```

参数含义：

- program：一个着色器程序的id；

pname : GL\_LINK\_STATUS ;  
param : 返回值, 如果一切正常返回GL\_TRUE代, 否则返回GL\_FALSE。

通过glBindAttribLocation()把“顶点属性索引”绑定到“顶点属性名”。

```
[cpp]    
1. void glBindAttribLocation(GLuint program,GLuint index,const GLchar* name);
```



参数含义：

program : 着色器程序容器的id。  
index : 顶点属性索引。  
name : 顶点属性名。

4)

**使用program。**

在链接了程序以后, 我们可以使用glUseProgram()函数来加载并使用链接好的程序。glUseProgram函数原型如下：

```
[cpp]    
1. void glUseProgram (int program)
```

其中program是要使用的着色器程序的id。

(3)

**初始化Texture**

初始化Texture可以分为以下步骤。

1)

**定义顶点数组**



这一步需要初始化两个数组,

2)

**设置顶点数组**

这一步通过glVertexAttribPointer()完成。glVertexAttribPointer()定义一个通用顶点属性数组。当渲染时, 它指定了通用顶点属性数组从索引index处开始的位置和数据格式。



glVertexAttribPointer()原型如下。

```
[cpp]    
1. void glVertexAttribPointer(  
2.     GLuint index,  
3.     GLint size,  
4.     GLenum type,  
5.     GLboolean normalized,  
6.     GLsizei stride,  
7.     const GLvoid * pointer);
```

每个参数的含义：

index : 指示将被修改的通用顶点属性的索引  
size : 指点每个顶点元素个数(1~4)  
type : 数组中每个元素的数据类型  
normalized : 指示定点数据值是否被归一化(归一化<[-1,1]或[0,1]> : GL\_TRUE,直接使用:GL\_FALSE)  
stride : 连续顶点属性间的偏移量, 如果为0, 相邻顶点属性间紧紧相邻  
pointer:顶点数组

使用函数glEnableVertexAttribArray()启用属性数组。默认状态下, 所有客户端的能力被Disabled, 包括所有通用顶点属性数组。如果被Enable, 通用顶点属性数组中的值将被访问并被用于Rendering。函数的原型如下：

```
[cpp]    
1. void glEnableVertexAttribArray( GLuint index);
```

其中index用于指定通用顶点属性的索引。

3)

**初始化纹理**

使用glGenTextures()初始化纹理, 其原型如下。



```
[cpp]
1. glGenTextures(GLsizei n, GLuint *textures)
```

参数含义：

n：用来生成纹理的数量  
textures：存储纹理索引的数组

glGenTextures()就是用来产生你要操作的纹理对象的索引的，比如你告诉OpenGL，我需要5个纹理对象，它会从没有用到的整数里返回5个给你。  
产生纹理索引之后，需要使用glBindTexture()绑定纹理，才能对该纹理进行操作。glBindTexture()告诉OpenGL下面对纹理的任何操作都是针对它所绑定的纹理对象的，比如glBindTexture(GL\_TEXTURE\_2D,1)即告诉OpenGL下面代码中对2D纹理的任何设置都是针对索引为1的纹理的。  
glBindTexture()函数的声明如下所示：

```
[cpp]
1. void glBindTexture(GLenum target, GLuint texture );
```

函数参数的含义：

target：纹理被绑定的目标，它只能取值GL\_TEXTURE\_1D、GL\_TEXTURE\_2D、GL\_TEXTURE\_3D或者GL\_TEXTURE\_CUBE\_MAP。  
texture：纹理的名称，并且，该纹理的名称在当前的应用中不能被再次使用。

绑定纹理之后，就可以设置该纹理的一些属性了。

纹理过滤函数glTexParameteri()可以用来确定如何把图像从纹理图像空间映射到帧缓冲图像空间。即把纹理像素映射成像素。glTexParameteri()的原型如下。

```
[cpp]
1. void glTexParameteri(GLenum target, GLenum pname, GLint param);
```

部分参数功能说明如下：

pname：参数。可以指定为GL\_TEXTURE\_MAG\_FILTER（放大过滤），GL\_TEXTURE\_MIN\_FILTER（缩小过滤）等。  
param：参数的值。例如GL\_LINEAR（线性插值。使用距离当前渲染像素中心最近的4个纹素加权平均值），GL\_NEAREST（临近像素插值。该方法质量较差）

6)

## 进入消息循环

glutMainLoop()将会进入GLUT事件处理循环。一旦被调用，这个程序将永远不会返回。视频播放的时候，调用该函数之后即开始播放视频。

2.

## 循环显示画面

1)

## 设置纹理

使用glActiveTexture()选择可以由纹理函数进行修改的当前纹理单位。后续的操作都是对选择的纹理进行的。glActiveTexture()的原型如下。

```
[cpp]
1. void glActiveTexture(GLenum texUnit);
```

接着使用glBindTexture()告诉OpenGL下面对纹理的任何操作都是针对它所绑定的纹理对象的，这一点前文已经记录，不再重复。  
然后使用glTexImage2D()根据指定的参数，生成一个2D纹理（Texture）。相似的函数还有glTexImage1D、glTexImage3D。glTexImage2D()原型如下。

```
[cpp]
1. void glTexImage2D(  GLenum target,
2.     GLint level,
3.     GLint internalformat,
4.     GLsizei width,
5.     GLsizei height,
6.     GLint border,
7.     GLenum format,
8.     GLenum type,
9.     const GLvoid * data);
```

参数说明如下：

target：指定目标纹理，这个值必须是GL\_TEXTURE\_2D。  
level：执行细节级别。0是最基本的图像级别，n表示第N级贴图细化级别。  
internalformat：指定纹理中的颜色格式。可选的值有GL\_ALPHA, GL\_RGB, GL\_RGBA, GL\_LUMINANCE, GL\_LUMINANCE\_ALPHA 等几种。

width：纹理图像的宽度。

height：纹理图像的高度。

border：边框的宽度。必须为0。

format：像素数据的颜色格式, 不需要和internalformat取值必须相同。可选的值参考internalformat。

type：指定像素数据的数据类型。可以使用的值有GL\_UNSIGNED\_BYTE, GL\_UNSIGNED\_SHORT\_5\_6\_5, GL\_UNSIGNED\_SHORT\_4\_4\_4\_4, GL\_UNSIGNED\_SHORT\_5\_5\_5\_1等。


pixels：指定内存中指向图像数据的指针

glUniform()为当前程序对象指定Uniform变量的值。（注意，由于OpenGL由C语言编写，但是C语言不支持函数的重载，所以会有很多名字相同后缀不同的函数版本存在。其中函数名中包含数字（1、2、3、4）表示接受该数字个用于更改uniform变量的值，i表示32位整形，f表示32位浮点型，ub表示8位无符号byte，ui表示32位无符号整形，v表示接受相应的指针类型。）

2)

### 绘制

使用glDrawArrays()进行绘制。glDrawArrays()原型如下。

```
[cpp]    
1. void glDrawArrays (GLenum mode, GLint first, GLsizei count);
```

参数说明：

mode：绘制方式，提供以下参数：GL\_POINTS、GL\_LINES、GL\_LINE\_LOOP、GL\_LINE\_STRIP、GL\_TRIANGLES、GL\_TRIANGLE\_STRIP、GL\_TRIANGLE\_FAN。

first：从数组缓存中的哪一位开始绘制，一般为0。

count：数组中顶点的数量。

3)

### 显示

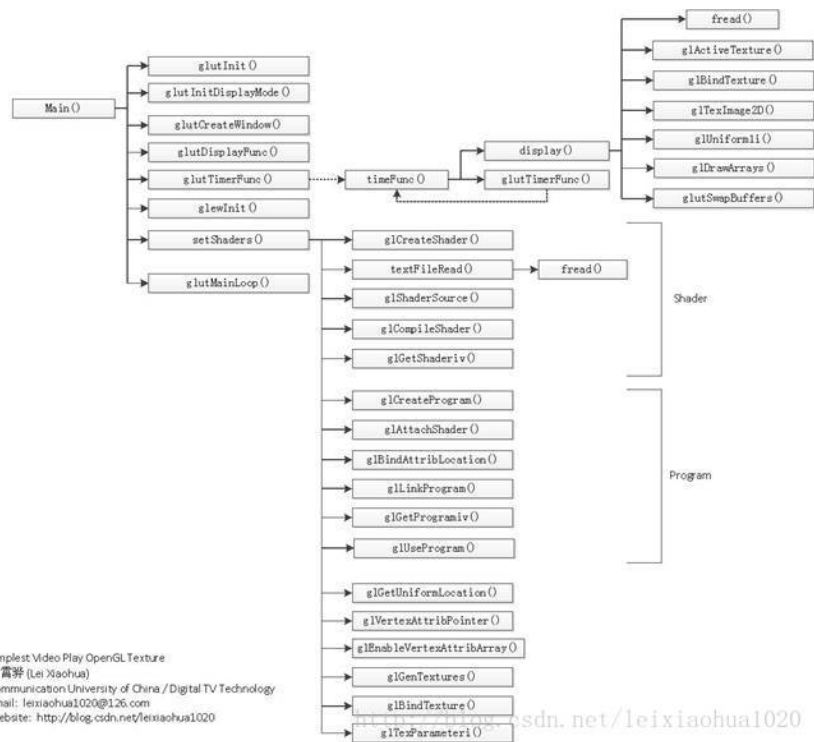
如果使用“双缓冲”方式的话，使用glutSwapBuffers()绘制。如果使用“单缓冲”方式的话，使用glFlush()绘制。glutSwapBuffers()的功能是交换两个缓冲区指针，表现的形式即是把画面呈现到屏幕上。

简单解释一下双缓冲技术。当我们进行复杂的绘图操作时，画面便可能有明显的闪烁。这是由于绘制的东西没有同时出现在屏幕上而导致的。使用双缓冲可以解决这个问题。所谓双缓冲技术，是指使用两个缓冲区：前台缓冲和后台缓冲。前台缓冲即我们看到的屏幕，后台缓冲则在内存当中，对我们来说是不可见的。每次的所有绘图操作不是在屏幕上直接绘制，而是在后台缓冲中进行，当绘制完成时，再把绘制的最终结果显示到屏幕上。

glutSwapBuffers()函数执行之后，缓冲区指针交换，两个缓冲的“角色”也发生了对调。原先的前台缓冲变成了后台缓冲，等待进行下一次绘制。而原先的后台缓冲变成了前台缓冲，展现出绘制的结果。

## 视频显示（使用Texture）流程总结

上文流程的函数流程可以用下图表示。



## 代码

源代码如下所示。

```

1.  /**
2.   * 最简单的OpenGL播放视频的例子 (OpenGL播放YUV) [Texture]
3.   * Simplest Video Play OpenGL (OpenGL play YUV) [Texture]
4.   *
5.   * 雷霄骅 Lei Xiaohua
6.   * leixiaohua1020@126.com
7.   * 中国传媒大学/数字电视技术
8.   * Communication University of China / Digital TV Technology
9.   * http://blog.csdn.net/leixiaohua1020
10.  *
11.  * 本程序使用OpenGL播放YUV视频像素数据。本程序支持YUV420P的
12.  * 像素数据作为输入，经过转换后输出到屏幕上。其中用到了多种
13.  * 技术，例如Texture, Shader等，是一个相对比较复杂的例子。
14.  * 适合有一定OpenGL基础的初学者学习。
15.  *
16.  * 函数调用步骤如下：
17.  *
18.  * [初始化]
19.  * glutInit(): 初始化glut库。
20.  * glutInitDisplayMode(): 设置显示模式。
21.  * glutCreateWindow(): 创建一个窗口。
22.  * glwInit(): 初始化glew库。
23.  * glutDisplayFunc(): 设置绘图函数（重绘的时候调用）。
24.  * glutTimerFunc(): 设置定时器。
25.  * InitShaders(): 设置Shader。包含了一系列函数，暂不列出。
26.  * glutMainLoop(): 进入消息循环。
27.  *
28.  * [循环渲染数据]
29.  * glActiveTexture(): 激活纹理单位。
30.  * glBindTexture(): 绑定纹理
31.  * glTexImage2D(): 根据像素数据，生成一个2D纹理。
32.  * glUniform1i():
33.  * glDrawArrays(): 绘制。
34.  * glutSwapBuffers(): 显示。
35.  *
36.  * This software plays YUV raw video data using OpenGL.
37.  * It support read YUV420P raw file and show it on the screen.
38.  * It's use a slightly more complex technologies such as Texture,
39.  * Shaders etc. Suitable for beginner who already has some
40.  * knowledge about OpenGL.
41.  *
42.  * The process is shown as follows:
43.  *
44.  * [Init]
45.  * glutInit(): Init glut library.
46.  * glutInitDisplayMode(): Set display mode.
47.  * glutCreateWindow(): Create a window.
48.  * glwInit(): Init glew library.
49.  * glutDisplayFunc(): Set the display callback.

```

```

50.  * glutTimerFunc(): Set timer.
51.  * InitShaders(): Set Shader, Init Texture. It contains some functions about Shader.
52.  * glutMainLoop(): Start message loop.
53.  *
54.  * [Loop to Render data]
55.  * glActiveTexture(): Active a Texture unit
56.  * glBindTexture(): Bind Texture
57.  * glTexImage2D(): Specify pixel data to generate 2D Texture
58.  * glUniform1i():
59.  * glDrawArrays(): draw.
60.  * glutSwapBuffers(): show.
61.  */
62.
63. #include <stdio.h>
64.
65. #include "glew.h"
66. #include "glut.h"
67.
68. #include <stdio.h>
69. #include <stdlib.h>
70. #include <malloc.h>
71. #include <string.h>
72.
73. //Select one of the Texture mode (Set '1'):
74. #define TEXTURE_DEFAULT 0
75. //Rotate the texture
76. #define TEXTURE_ROTATE 0
77. //Show half of the Texture
78. #define TEXTURE_HALF 1
79.
80. const int screen_w=500,screen_h=500;
81. const int pixel_w = 320, pixel_h = 180;
82. //YUV file
83. FILE *infile = NULL;
84. unsigned char buf[pixel_w*pixel_h*3/2];
85. unsigned char *plane[3];
86.
87.
88. GLuint p;
89. GLuint id_y, id_u, id_v; // Texture id
90. GLuint textureUniformY, textureUniformU,textureUniformV;
91.
92.
93. #define ATTRIB_VERTEX 3
94. #define ATTRIB_TEXTURE 4
95.
96. void display(void){
97.     if (fread(buf, 1, pixel_w*pixel_h*3/2, infile) != pixel_w*pixel_h*3/2){
98.         // Loop
99.         fseek(infile, 0, SEEK_SET);
100.         fread(buf, 1, pixel_w*pixel_h*3/2, infile);
101.     }
102.     //Clear
103.     glClearColor(0.0,255,0.0,0.0);
104.     glClear(GL_COLOR_BUFFER_BIT);
105.     //Y
106.     //
107.     glActiveTexture(GL_TEXTURE0);
108.
109.     glBindTexture(GL_TEXTURE_2D, id_y);
110.
111.     glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, pixel_w, pixel_h, 0, GL_RED, GL_UNSIGNED_BYTE, plane[0]);
112.
113.     glUniform1i(textureUniformY, 0);
114.     //U
115.     glActiveTexture(GL_TEXTURE1);
116.     glBindTexture(GL_TEXTURE_2D, id_u);
117.     glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, pixel_w/2, pixel_h/2, 0, GL_RED, GL_UNSIGNED_BYTE, plane[1]);
118.     glUniform1i(textureUniformU, 1);
119.     //V
120.     glActiveTexture(GL_TEXTURE2);
121.     glBindTexture(GL_TEXTURE_2D, id_v);
122.     glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, pixel_w/2, pixel_h/2, 0, GL_RED, GL_UNSIGNED_BYTE, plane[2]);
123.     glUniform1i(textureUniformV, 2);
124.
125.     // Draw
126.     glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
127.     // Show
128.     //Double
129.     glutSwapBuffers();
130.     //Single
131.     //glFlush();
132. }
133.
134. void timeFunc(int value){
135.     display();
136.     // Timer: 40ms
137.     glutTimerFunc(40, timeFunc, 0);
138. }
139.
140. char *textFileRead(char * filename)

```

```

141. {
142.     char *s = (char *)malloc(8000);
143.     memset(s, 0, 8000);
144.     FILE *infile = fopen(filename, "rb");
145.     int len = fread(s, 1, 8000, infile);
146.     fclose(infile);
147.     s[len] = 0;
148.     return s;
149. }
150.
151. //Init Shader
152. void InitShaders()
153. {
154.     GLint vertCompiled, fragCompiled, linked;
155.
156.     GLint v, f;
157.     const char *vs,*fs;
158.     //Shader: step1
159.     v = glCreateShader(GL_VERTEX_SHADER);
160.     f = glCreateShader(GL_FRAGMENT_SHADER);
161.     //Get source code
162.     vs = textFileRead("Shader.vsh");
163.     fs = textFileRead("Shader.fsh");
164.     //Shader: step2
165.     glShaderSource(v, 1, &vs,NULL);
166.     glShaderSource(f, 1, &fs,NULL);
167.     //Shader: step3
168.     glCompileShader(v);
169.     //Debug
170.     glGetShaderiv(v, GL_COMPILE_STATUS, &vertCompiled);
171.     glCompileShader(f);
172.     glGetShaderiv(f, GL_COMPILE_STATUS, &fragCompiled);
173.
174.     //Program: Step1
175.     p = glCreateProgram();
176.     //Program: Step2
177.     glAttachShader(p,v);
178.     glAttachShader(p,f);
179.
180.     glBindAttribLocation(p, ATTRIB_VERTEX, "vertexIn");
181.     glBindAttribLocation(p, ATTRIB_TEXTURE, "textureIn");
182.     //Program: Step3
183.     glLinkProgram(p);
184.     //Debug
185.     glGetProgramiv(p, GL_LINK_STATUS, &linked);
186.     //Program: Step4
187.     glUseProgram(p);
188.
189.
190.     //Get Uniform Variables Location
191.     textureUniformY = glGetUniformLocation(p, "tex_y");
192.     textureUniformU = glGetUniformLocation(p, "tex_u");
193.     textureUniformV = glGetUniformLocation(p, "tex_v");
194.
195.     #if TEXTURE_ROTATE
196.     static const GLfloat vertexVertices[] = {
197.         -1.0f, -0.5f,
198.         0.5f, -1.0f,
199.         -0.5f, 1.0f,
200.         1.0f, 0.5f,
201.     };
202.     #else
203.     static const GLfloat vertexVertices[] = {
204.         -1.0f, -1.0f,
205.         1.0f, -1.0f,
206.         -1.0f, 1.0f,
207.         1.0f, 1.0f,
208.     };
209.     #endif
210.
211.     #if TEXTURE_HALF
212.     static const GLfloat textureVertices[] = {
213.         0.0f, 1.0f,
214.         0.5f, 1.0f,
215.         0.0f, 0.0f,
216.         0.5f, 0.0f,
217.     };
218.     #else
219.     static const GLfloat textureVertices[] = {
220.         0.0f, 1.0f,
221.         1.0f, 1.0f,
222.         0.0f, 0.0f,
223.         1.0f, 0.0f,
224.     };
225.     #endif
226.     //Set Arrays
227.     glVertexAttribPointer(ATTRIB_VERTEX, 2, GL_FLOAT, 0, 0, vertexVertices);
228.     //Enable it
229.     glEnableVertexAttribArray(ATTRIB_VERTEX);
230.     glVertexAttribPointer(ATTRIB_TEXTURE, 2, GL_FLOAT, 0, 0, textureVertices);
231.     glEnableVertexAttribArray(ATTRIB_TEXTURE);
232. }

```

```

232.
233.
234.     //Init Texture
235.     glGenTextures(1, &id_y);
236.     glBindTexture(GL_TEXTURE_2D, id_y);
237.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
238.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
239.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
240.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
241.
242.     glGenTextures(1, &id_u);
243.     glBindTexture(GL_TEXTURE_2D, id_u);
244.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
245.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
246.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
247.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
248.
249.     glGenTextures(1, &id_v);
250.     glBindTexture(GL_TEXTURE_2D, id_v);
251.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
252.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
253.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
254.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
255.
256. }
257.
258.
259.
260. int main(int argc, char* argv[])
261. {
262.     //Open YUV420P file
263.     if((infile=fopen("../test_yuv420p_320x180.yuv", "rb"))==NULL){
264.         printf("cannot open this file\n");
265.         return -1;
266.     }
267.
268.     //YUV Data
269.     plane[0] = buf;
270.     plane[1] = plane[0] + pixel_w*pixel_h;
271.     plane[2] = plane[1] + pixel_w*pixel_h/4;
272.
273.     //Init GLUT
274.     glutInit(&argc, argv);
275.     //GLUT_DOUBLE
276.     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA /*| GLUT_STENCIL | GLUT_DEPTH*/);
277.     glutInitWindowPosition(100, 100);
278.     glutInitWindowSize(screen_w, screen_h);
279.     glutCreateWindow("Simplest Video Play OpenGL (Texture)");
280.     printf("Lei Xiaohua\n");
281.     printf("http://blog.csdn.net/leixiaohua1020\n");
282.     printf("Version: %s\n", glGetString(GL_VERSION));
283.     GLenum l = glewInit();
284.
285.     glutDisplayFunc(&display);
286.     glutTimerFunc(40, timeFunc, 0);
287.
288.     InitShaders();
289.
290.     // Begin!
291.     glutMainLoop();
292.
293.     return 0;
294. }

```

#### Shader.vsh

```

[plain]
1. attribute vec4 vertexIn;
2. attribute vec2 textureIn;
3. varying vec2 textureOut;
4. void main(void)
5. {
6.     gl_Position = vertexIn;
7.     textureOut = textureIn;
8. }

```

#### Shader.fsh

```
[plain]
1.  varying vec2 textureOut;
2.  uniform sampler2D tex_y;
3.  uniform sampler2D tex_u;
4.  uniform sampler2D tex_v;
5.  void main(void)
6.  {
7.      vec3 yuv;
8.      vec3 rgb;
9.      yuv.x = texture2D(tex_y, textureOut).r;
10.     yuv.y = texture2D(tex_u, textureOut).r - 0.5;
11.     yuv.z = texture2D(tex_v, textureOut).r - 0.5;
12.     rgb = mat3( 1,      1,      1,
13.               0,      -0.39465, 2.03211,
14.               1.13983, -0.58060, 0) * yuv;
15.     gl_FragColor = vec4(rgb, 1);
16. }
```

## 代码注意事项

1. 目前支持读取YUV420P格式的像素数据。
2. 窗口的宽高为screen\_w, screen\_h。像素数据的宽高为pixel\_w,pixel\_h。它们的定义如下。

```
[cpp]
1.  //Width, Height
2.  const int screen_w=500,screen_h=500;
3.  const int pixel_w=320,pixel_h=180;
```

3. 通过代码前面的宏，可以选择几种不同的纹理映射方式

```
[cpp]
1.  //Select one of the Texture mode (Set '1'):
2.  #define TEXTURE_DEFAULT 1
3.  //Rotate the texture
4.  #define TEXTURE_ROTATE 0
5.  //Show half of the Texture
6.  #define TEXTURE_HALF 0
```

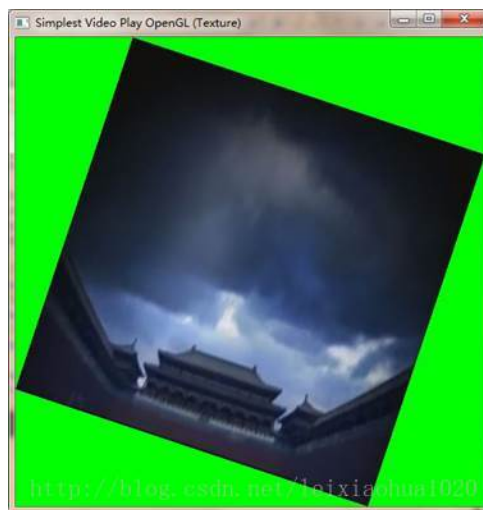
第一种是正常的映射方式，第二种是“旋转”的方式，第三种是只映射一半的方式。

## 结果

程序运行结果如下。默认的纹理映射：



“旋转”：



一半纹理：



## 下载

代码位于“Simplest Media Play”中

SourceForge项目地址：<https://sourceforge.net/projects/simplestmediaplay/>

CSDN下载地址：<http://download.csdn.net/detail/leixiaohua1020/8054395>

注：

该项目会不定时的更新并修复一些小问题，最新的版本请参考该系列文章的总述页面：

《最简单的视音频播放示例1：总述》

上述工程包含了使用各种API（Direct3D，OpenGL，GDI，DirectSound，SDL2）播放多媒体例子。其中音频输入为PCM采样数据。输出至系统的声卡播放出来。视频输入为YUV/RGB像素数据。输出至显示器上的一个窗口播放出来。

通过本工程的代码初学者可以快速学习使用这几个API播放视频和音频的技术。

一共包括了如下几个子工程：

simplest\_audio\_play\_directsound:

使用DirectSound播放PCM音频采样数据。

simplest\_audio\_play\_sdl2:

使用SDL2播放PCM音频采样数据。

simplest\_video\_play\_direct3d:

使用Direct3D的Surface播放RGB/YUV视频像素数据。

simplest\_video\_play\_direct3d\_texture:

使用Direct3D的Texture播放RGB视频像素数据。

simplest\_video\_play\_gdi:



使用GDI播放RGB/YUV视频像素数据。  
simplest\_video\_play\_opengl:

使用OpenGL播放RGB/YUV视频像素数据。  
simplest\_video\_play\_opengl\_texture:  
使用OpenGL的Texture播放YUV视频像素数据。  
simplest\_video\_play\_sdl2:  
使用SDL2播放RGB/YUV视频像素数据。

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/leixiaohua1020/article/details/40379845>

文章标签：[OpenGL](#) [Texture](#) [YUV](#) [渲染](#) [Shader](#)

个人分类：[OpenGL](#) [我的开源项目](#)

此PDF由spygg生成,请尊重原作者版权!!!  
我的邮箱:liushidc@163.com