

Lecture 17

Programmable Shader

- What's Programmable Shader?
- GLSL programming
- Pass-Through GLSL Program
- Screen coordinate as color
- Color as attribute

What's Programmable Shader?

- Opposite of fixed-function pipeline.
- In fixed-function pipeline, a vertex consists of a fixed set of attributes (position, color, normal, and texture coordinate), and goes through a fixed calculation.
- In programmable shader, you can define what attributes that a vertex consists of.

What's Programmable Shader?

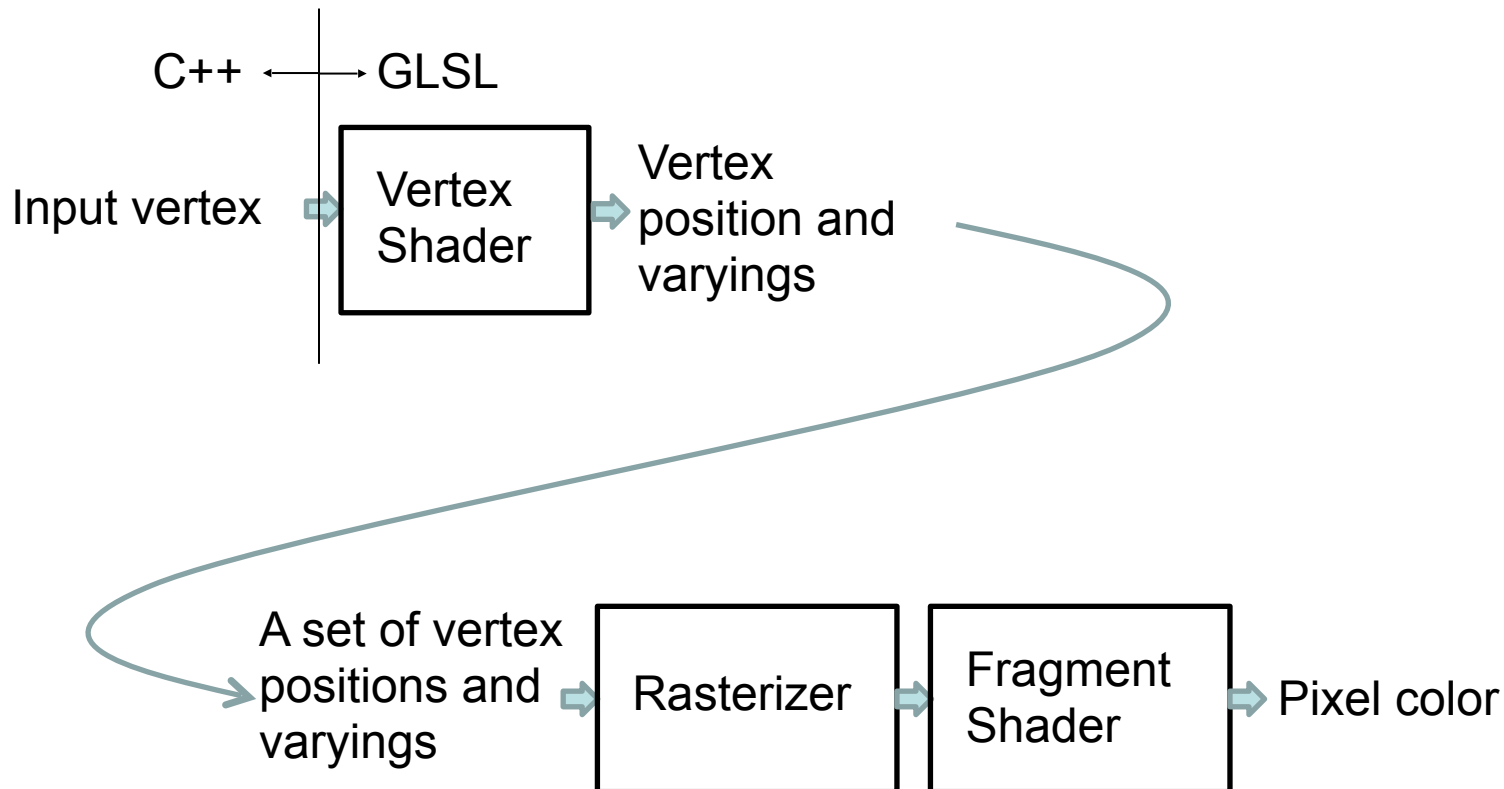
- A vertex can be as simple as just one XY coordinate if:
 - You are drawing 2D elements.
 - All elements need to be drawn in the same color.
- Such minimum vertex will substantially reduce CPU-GPU transaction.
- Or, a vertex can have additional information such as:
 - Multiple texture coordinates.
 - Offset in the camera coordinate (Billboard).
 - Different color for different lighting effect.
- Those information (what a vertex can carry) is called a ***vertex attribute***.

GLSL – GL Shader Language

- A programming language for programmable shader in OpenGL.
- Program-In-Program.
- Standard for OpenGL, OpenGL ES, and WebGL.
- You write separate programs in GLSL, and then compile in your program.
- Good news – GLSL is a C-like language.
- Bad news – Many C-language features are not available in GLSL.

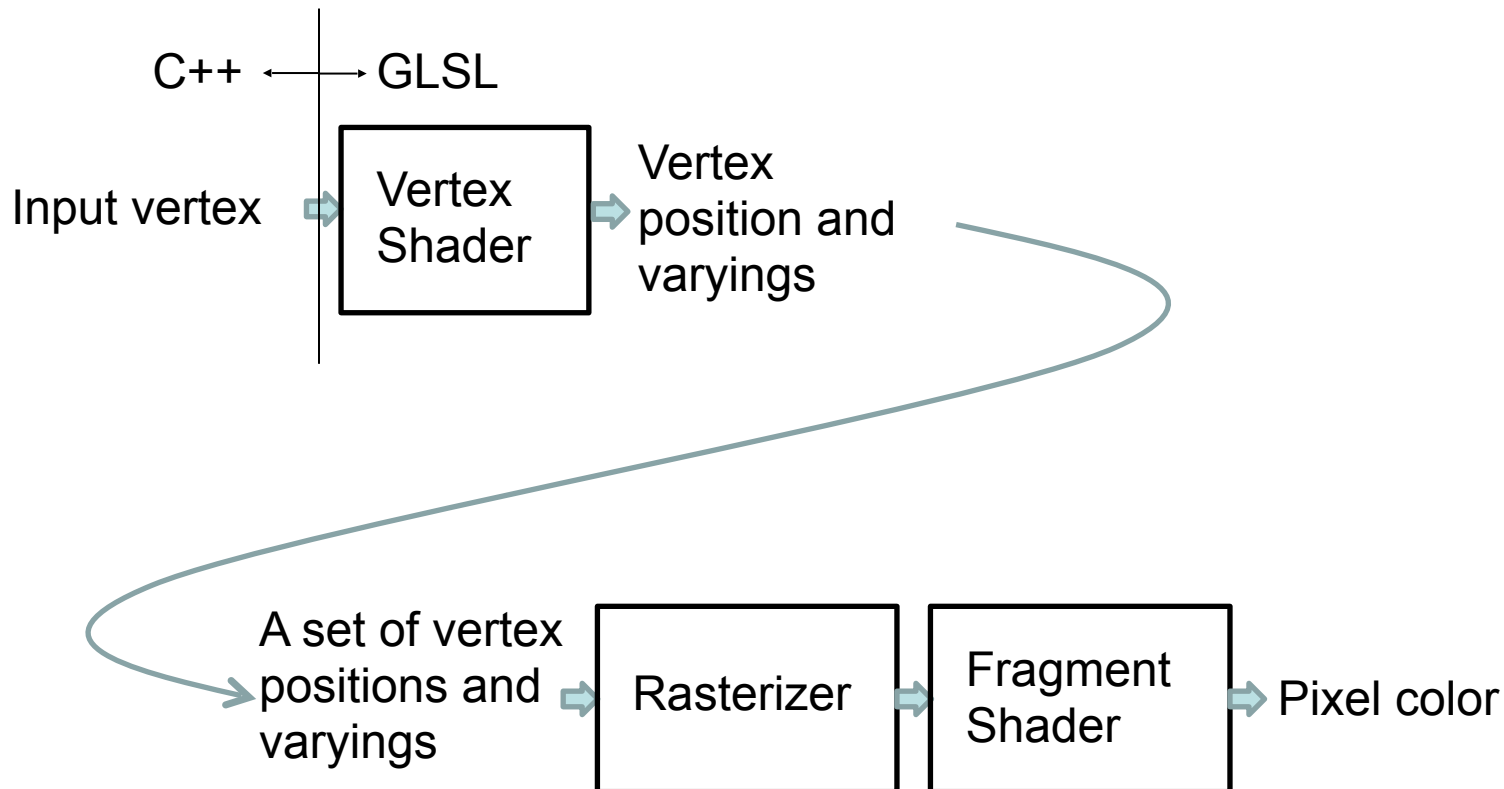
Vertex Shader and Fragment Shader

- You need to write two shaders for one program, Vertex Shader and Fragment Shader.



Vertex Shader and Fragment Shader

- The vertex attributes are passed from the C++ program to the vertex shader.
- Therefore, your vertex shader defines what a vertex looks like.

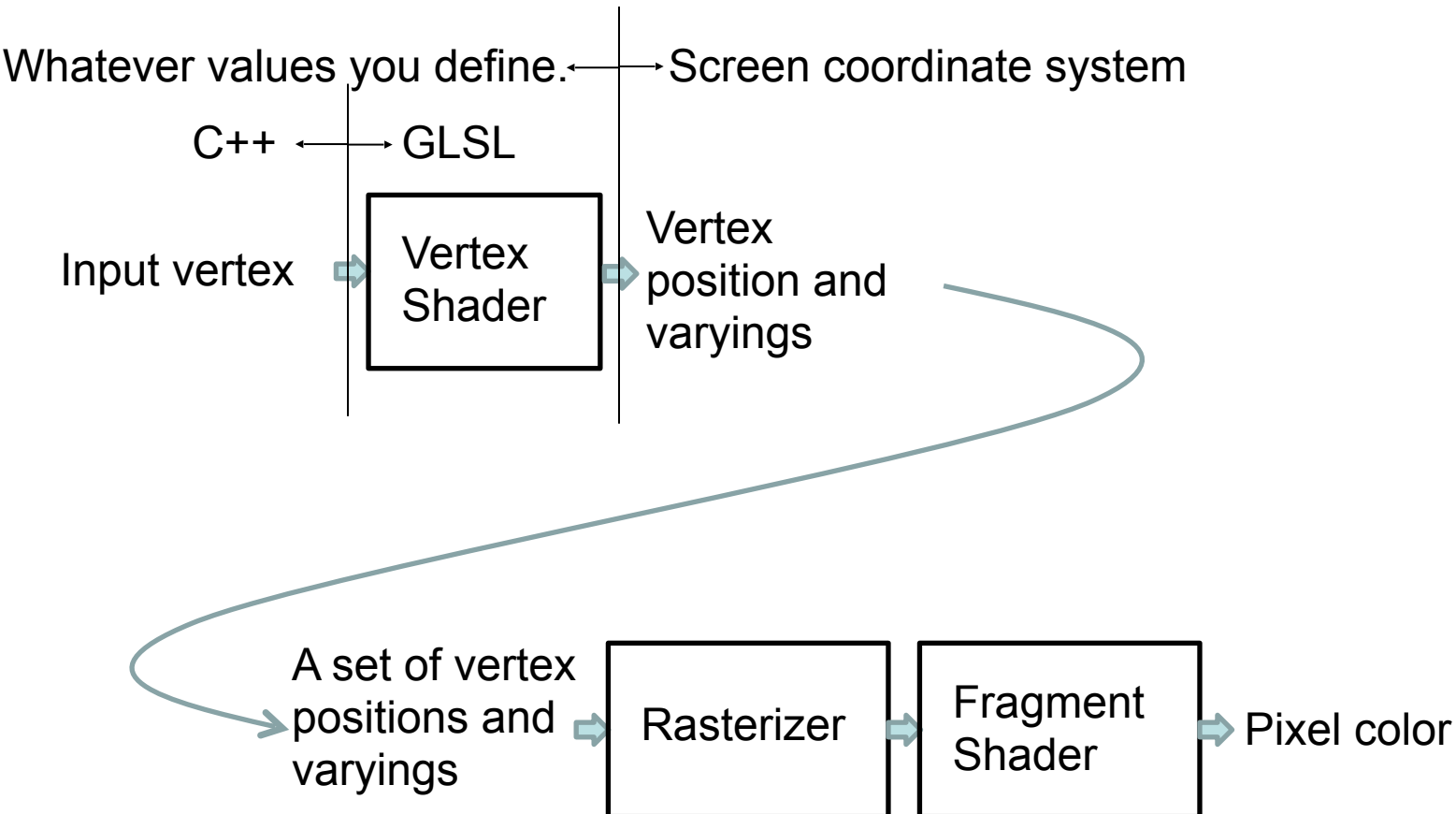


Vertex Shader

- Takes a set of vertex attributes as input.
- Outputs a vertex position (`gl_Position`) and varyings.
- Varying: A value that is assigned per vertex and interpolated within a primitive by the Rasterizer.
- Examples of a varying:
 - Color – When a color is assigned per vertex.
 - Normal – For Phong shading, normal must be interpolated within a primitive.
 - Texture Coordinate – Texture coordinate typically is independent of the vertex coordinate, and must be interpolated within a primitive.

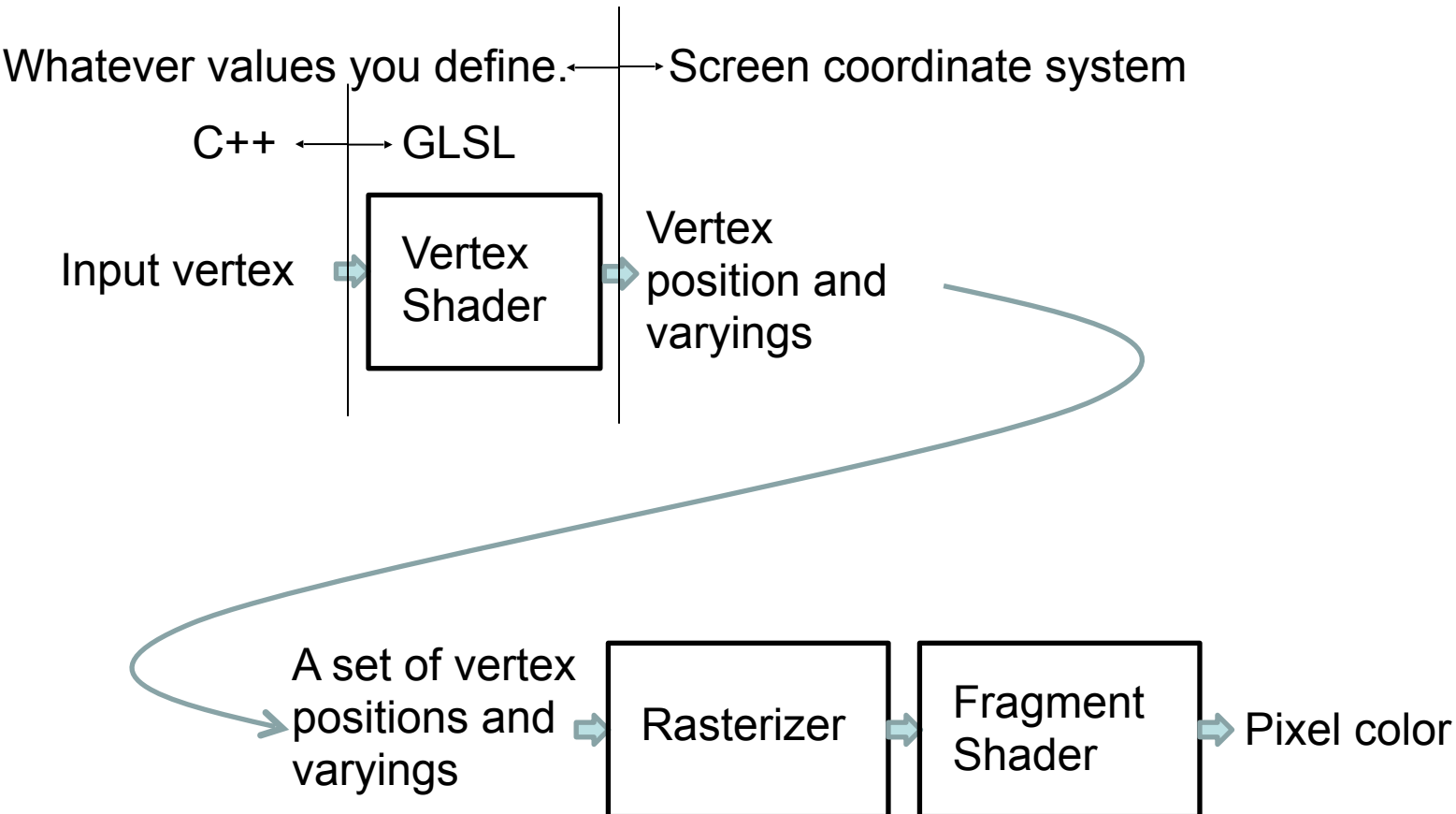
Fragment Shader

- Input are the varyings interpolated for the pixel location.
- Output is the pixel color (`gl_FragColor`)



Fragment Shader

- What happens between the vertex shader and a fragment shader? => Interpolation



Essentially, shaders are call-back functions.

- Vertex Shader: A function called per vertex.
- Fragment Shader: A function called per pixel.

- When the time comes, hopefully these can just be a C/C++ call-back function. But, there still is a wall between CPU and GPU.

What you need to prepare for GLSL program

- Two text files:
 - Vertex Shader
 - Fragment Shader
- In C++ program:
 - Reserve an identifier for each of:
 - A vertex shader
 - A fragment shader
 - A program (combination of vertex and fragment shaders)
 - Compile the shaders.
 - Link the shaders.

To use a GLSL program,

1. Enable the program (`glUseProgram`)
2. Enable vertex-attribute arrays.
3. Set vertex-attribute pointer.
4. `glDrawArrays`
5. Disable vertex-attribute arrays

Steps 3 and 4 can be repeated to save OpenGL function calls.

Pass-Through GLSL Program


- The minimum GLSL program.
 - Passes input vertex coordinate (x,y,z) to the rasterizer with no transformation.
 - The primitive color is constant (red).

Vertex Shader


- Written in vertexShader.glsl

```
attribute vec3 vertex;  
void main()  
{  
    gl_Position=vec4(vertex,1.0);  
}
```

This is what this vertex shader receives from your C++ program.
In this case, it means that one vertex consists of one 3D vector called vertex.



This is what the next stage receives from this vertex shader.
It needs to be a 4D vector (homogeneous coordinate). Therefore, 1.0 is added as the 4th dimension.



Vector calculation in GLSL


- GLSL is optimized for vector and matrix calculations.
- The following expression:
 `vec4(vertex,1.0)`
means that a 4D vector, first three dimensions taken from a 3D vector called `vertex`, and the last dimension is 1.0.
- The 3D vector of the first three component of a 4D vector can be extracted by writing (if `vertex` is a `vec4` variable):
 `vertex.xyz`
- Similarly, you can extract components of 3D vector to get a 2D vector by writing:
 `vertex.xy`
 `vertex.xz`
 `vertex.yz`

- Also a 3D or 4D vectors can be used as a color, (r,g,b) or (r,g,b,a).
- The first component vertex.x is same as vertex.r
- Also vertex.rgb is same as vertex.xyz

Fragment Shader

- Written in fragmentShader.glsl

```
void main()
{
    gl_FragColor=vec4(1,0,0,1);
}
```



This is the color of the pixel written to the frame buffer. In this case, always red.

One thing to remember when writing a shader program

- The compiler will not re-compile your program when you modify your shader source.
- When you modify your shader source, you also need to touch your source files so that the compiler builds your program, (and then copies the shader sources to the executable directory.)

How can the C++ program access these two files?

- These two files need to be copied to the location that the executable can access.
- Store these two files under data directory of the source directory, and
- Use:

```
set(DATA_FILE_LOCATION ${CMAKE_CURRENT_SOURCE_DIR}/data)
```

- By defining this variable, in the end of the CMakeLists.txt, it defines a Post-Build event that copies the files to the executable location (In MacOSX bundle, Contents/Resources sub-directory of the bundle.)

Going step by step

- First check if the two files are accessible.
- In Initialize function, just open and read the contents of the two shader source files.

Reading in the renderer to `std::vector <char>`

- Add `renderer.h` and `renderer.cpp`

opengl_header.h

- Problem (major frustration):
 - Apple put OpenGL headers in a non-standard location.
 - Windows needs windows.h be included before including OpenGL headers.
 - Visual C++ does not come with OpenGL 2.x headers. (For Visual C++, headers are in public/src/imported/include/GL. Redistribution of the headers are permitted.)
 - iOS uses OpenGL ES, whose headers are in a different location.
- You end up needing that many lines.
- In this class, if you are tired of cutting & pasting those lines, just include:
 `#include <yagl.h>`
and add yagl in LIB_DEPENDENCY.

```
#ifndef OPENGGL_HEADER_IS_INCLUDED
#define OPENGGL_HEADER_IS_INCLUDED
```

```
#ifdef _WIN32
```

```
    #ifndef WIN32_LEAN_AND_MEAN
        // Prevent inclusion of winsock.h
        #define WIN32_LEAN_AND_MEAN
        #include <windows.h>
        #undef WIN32_LEAN_AND_MEAN
    #else
        // Too late. Just include it.
        #include <windows.h>
    #endif
#endif
```

Visual C++ wants windows.h be included before OpenGL headers.

```
#ifndef GL_GLEXT_PROTOTYPES
#define GL_GLEXT_PROTOTYPES
#endif
```

This macro is needed for Visual C++, or OpenGL 2.x function prototypes are ignored.

```
#ifndef __APPLE__
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glext.h>
#else
```

This is the standard location. Everyone should put these three files in here!

```
    #include <TargetConditionals.h>
    #if TARGET_OS_EMBEDDED!=0 || TARGET_OS_IPHONE!=0 || TARGET_OS_IPHONE_SIMULATOR!=0
        #include <OpenGLES/ES2/gl.h>
        #include <OpenGLES/ES2/glext.h>
        typedef double GLdouble;
    #else
        #include <OpenGL/gl.h>
        #include <OpenGL/glu.h>
        #include <OpenGL/glext.h>
    #endif
#endif
#endif
```

OpenGL ES

Apple wanted to put these headers under OpenGL sub-directory, not GL.

Aaaaagh! Why everyone makes cross-platform development difficult!!!!

renderer.h

```
#ifndef RENDERER_IS_INCLUDED
#define RENDERER_IS_INCLUDED

#include "opengl_header.h"
#include <vector>
std::vector <char> ReadTextFile(const char fn[]);

#endif
```

renderer.cpp

```
#include "renderer.h"

std::vector <char> ReadTextFile(const char fn[])
{
    std::vector <char> fileContent;
    char str[256];

    FILE *fp=fopen(fn,"r");
    if(nullptr!=fp)
    {
        while(nullptr!=fgets(str,255,fp))
        {
            for(int i=0; str[i]!=0; ++i)
            {
                fileContent.push_back(str[i]);
            }
        }
        fclose(fp);
    }
    fileContent.push_back(0);
    return fileContent;
}
```

Compiling the shaders.

- You need three identifiers:
 - Program (Vertex shader + Fragment shader)
 - Vertex shader
 - Fragment shader
- Also you need to cache identifiers for:
 - Vertex Attributes (in this case, the attribute labeled as “vertex”)
 - Uniforms (in this example, not used yet.)

PassThroughRenderer class.

Add in renderer.h:

```
class RendererBase
{
public:
    GLuint programIdent;
    GLuint vertexShaderIdent,fragmentShaderIdent;

    bool Compile(const std::vector <char> &vtxShaderSource,const std::vector <char> &fragShaderSource);
protected:
    bool CompileShader(int shaderIdent);
    bool LinkShader(void);
    virtual void CacheAttributeAndUniformIdent(void)=0;
};

class PassThroughRenderer : public RendererBase
{
public:
    GLuint attribVertexPos;
    virtual void CacheAttributeAndUniformIdent(void);
};
```

Compiling and linking the shaders.

```
bool RendererBase::Compile(
    const std::vector<char> &vtxShaderSource,
    const std::vector<char> &fragShaderSource)
{
    bool res=true;

    vertexShaderId=glCreateShader(GL_VERTEX_SHADER);
    fragmentShaderId=glCreateShader(GL_FRAGMENT_SHADER);
    programId=glCreateProgram();

    const char *vtxShaderSourcePtr=vtxShaderSource.data();
    const char *fragShaderSourcePtr=fragShaderSource.data();
    glShaderSource(vertexShaderId,1,&vtxShaderSourcePtr,NULL);
    glShaderSource(fragmentShaderId,1,&fragShaderSourcePtr,NULL);

    if(true!=CompileShader(vertexShaderId))
    {
        res=false;
    }
    if(true!=CompileShader(fragmentShaderId))
    {
        res=false;
    }
    if(true!=LinkShader())
    {
        res=false;
    }

    CacheAttributeAndUniformId();

    return res;
}
```

} Reserve identifiers

} Connect shader identifier
and source.

```
bool RendererBase::CompileFile(const char vtxShaderFn[],const char fragShaderFn[])
{
    auto vtxShaderSource=ReadTextFile(vtxShaderFn);
    auto fragShaderSource=ReadTextFile(fragShaderFn);

    printf("Vertex Shader\n");
    printf("%s\n",vtxShaderSource.data());

    printf("Fragment Shader\n");
    printf("%s\n",fragShaderSource.data());

    return Compile(vtxShaderSource,fragShaderSource);
}
```

```
bool RendererBase::CompileShader(int shaderIdent)
{
    int compileSta=99999,infoLogLength=99999,acquiredErrMsgLen=99999;
    int linkSta=99999;
    const int errMsgLen=1024;
    char errMsg[1024];

    glCompileShader(shaderIdent);

    glGetShaderiv(shaderIdent,GL_COMPILE_STATUS,&compileSta);
    glGetShaderiv(shaderIdent,GL_INFO_LOG_LENGTH,&infoLogLength);
    glGetShaderInfoLog(shaderIdent,errMsgLen-1,&acquiredErrMsgLen,errMsg);
    printf("Compile Status %d Info Log Length %d Error Message Length %d\n",
        compileSta,infoLogLength,acquiredErrMsgLen);

    if(GL_TRUE!=compileSta)
    {
        int i;
        printf("Error Message: \n%s\n",errMsg);
        return false;
    }
    return true;
}
```

Compile



Were there
errors?



```
bool RendererBase::LinkShader(void)
```

```
{
```

```
    int compileSta=99999,infoLogLength=99999,acquiredErrMsgLen=99999;
```

```
    int linkSta=99999;
```

```
    const int errMsgLen=1024;
```

```
    char errMsg[1024];
```

```
    glAttachShader(programIdent,vertexShaderIdent);
```

```
    glAttachShader(programIdent,fragmentShaderIdent);
```

```
    glLinkProgram(programIdent);
```

```
    glGetProgramiv(programIdent,GL_LINK_STATUS,&linkSta);
```

```
    glGetProgramiv(programIdent,GL_INFO_LOG_LENGTH,&infoLogLength);
```

```
    glGetProgramInfoLog(programIdent,errMsgLen-1,&acquiredErrMsgLen,errMsg);
```

```
    printf("Link Status %d Info Log Length %d Error Message Length %d\n",
```

```
        linkSta,infoLogLength,acquiredErrMsgLen);
```

```
    if(GL_TRUE!=linkSta)
```

```
    {
```

```
        printf("Error Message: \n%s\n",errMsg);
```

```
        return false;
```

```
    }
```

```
    return true;
```

```
}
```

```
////////////////////////////////////
```

```
void PassThroughRenderer::CacheAttributeAndUniformIdent(void)
```

```
{
```

```
    attribVertexPos=glGetAttribLocation(programIdent,"vertex");
```

```
    printf("Attribute Vertex Position=%d\n",attribVertexPos);
```

```
}
```

} Link=Assemble shaders
into a program.

Using the shader program.

- Shader program must be compiled once.
- You can compile same program multiple times, but it just wastes GPU memory.
- Good timing is immediately after when the OpenGL context is ready.
- Add header inclusion:

```
#include "renderer.h"
```

- Add a member variable:

```
PassThroughRenderer passThrough;
```

- In Initialize function:

```
FsChangeToProgramDir();  
passThrough.CompileFile("vertex_shader.glsl", "fragment_shader.glsl");
```


- In Draw function:

```
/* virtual */ void FsLazyWindowApplication::Draw(void)
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

    glUseProgram(passThrough.programId);
    const GLfloat vtx[12]=
    {
        -1,-1,0,
        1,-1,0,
        1, 1,0,
        -1, 1,0
    };
    glEnableVertexAttribArray(passThrough.attribVertexPos);
    glVertexAttribPointer(passThrough.attribVertexPos,3,GLfloat,GL_FALSE,0,vtx);
    glDrawArrays(GL_TRIANGLE_FAN,0,4);
    glDisableVertexAttribArray(passThrough.attribVertexPos);

    FsSwapBuffers();

    needRedraw=false;
}
```

These values are passed to the attribute called vertex in the vertex shader.

Varying the color based on the screen coordinate

- In the previous example, no information is passed from the vertex shader to the fragment shader.
- Let's pass the projected vertex position to the fragment shader, and assign color based on the position.
- (Fragment, or pixel, coordinate is available in `gl_FragCoord` in the fragment shader in newer version of GLSL, but may not supported in some devices.)

- Copy vertex_shader.glsl and fragment_shader.glsl to:
 - color_by_coord_vertex_shader.glsl, and
 - color_by_coord_fragment_shader.glsl

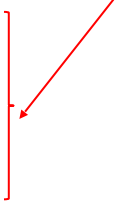
```
attribute vec3 vertex;  
varying vec4 projected_vertex;  
void main()  
{  
    projected_vertex=vec4(vertex,1.0);  
    gl_Position=vec4(vertex,1.0);  
}
```

Vertex Shader. Output gl_Position and projected_vertex.



```
varying vec4 projected_vertex;  
void main()  
{  
    gl_FragColor=projected_vertex;  
    gl_FragColor+=vec4(1.0,1.0,1.0,0.0);  
    gl_FragColor/=vec4(2.0,2.0,2.0,1.0);  
    gl_FragColor.w=1.0;  
}
```

Fragment Shader. Calculate pixel color based on the projected_vertex.



- In this case, since the input to the vertex shader is same as the pass-through renderer, only modification needed in the main.cpp is:

```
passThrough.CompileFile("vertex_shader.glsl","fragment_shader.glsl");
```



```
passThrough.CompileFile(  
    "color_by_coord_vertex_shader.glsl",  
    "color_by_coord_fragment_shader.glsl");
```

Color as Vertex Attribute

- In the previous example, a vertex had only one attribute, which is an xyz coordinate.
- In this example, a vertex has two attributes, an xyz coordinate and a rgba color.

Color as Vertex Attribute

- In the data directory, svn-copy fragment_shader.glsl and vertex_shader.glsl to:
 - color_as_attribute_vertex_shader.glsl, and
 - color_as_attribute_fragment_shader.glsl.

Vertex shader

```
attribute vec3 vertex;  
attribute vec4 color;  
  
varying vec4 colorOut;  
  
void main()  
{  
    colorOut=color;  
    gl_Position=vec4(vertex,1.0);  
}
```

Just passing attribute color to the colorOut.

Fragment shader

```
varying vec4 colorOut;  
  
void main()  
{  
    gl_FragColor=colorOut;  
}
```

Just passing colorOut from the vertex shader to the pixel color

- In renderer.h

```
class ColorAsAttributeRenderer : public RendererBase
{
public:
    GLuint attribVertexPos;
    GLuint attribColorPos;
    virtual void CacheAttributeAndUniformIdent(void);
};
```

- In renderer.cpp

```
void ColorAsAttributeRenderer::CacheAttributeAndUniformIdent(void)
{
    attribVertexPos=glGetAttribLocation(programIdent,"vertex");
    printf("Attribute Vertex Position=%d\n",attribVertexPos);
    attribColorPos=glGetAttribLocation(programIdent,"color");
    printf("Attribute Color Position=%d\n",attribColorPos);
}
```


- Add a member variable:

```
ColorAsAttributeRenderer colorAsAttrib;
```

- In Initialize

```
colorAsAttrib.CompileFile(  
    "color_as_attribute_vertex_shader.glsl",  
    "color_as_attribute_fragment_shader.glsl");
```

- In Draw

```
glUseProgram(colorAsAttrib.programIdent);  
const GLfloat vtx[12]=  
{  
    -1,-1,0,  
    1,-1,0,  
    1, 1,0,  
    -1, 1,0  
};  
const GLfloat col[16]=  
{  
    1,0,0,1,  
    0,1,0,1,  
    0,0,1,1,  
    1,0,1,1  
};  
glEnableVertexAttribArray(colorAsAttrib.attribVertexPos);  
glVertexAttribPointer(colorAsAttrib.attribVertexPos,3,GLfloat,GL_FALSE,0,vtx);  
glEnableVertexAttribArray(colorAsAttrib.attribColorPos);  
glVertexAttribPointer(colorAsAttrib.attribColorPos,4,GLfloat,GL_FALSE,0,col);  
glDrawArrays(GL_TRIANGLE_FAN,0,4);  
glDisableVertexAttribArray(colorAsAttrib.attribVertexPos);  
glDisableVertexAttribArray(colorAsAttrib.attribColorPos);
```