

Computer Graphics Through OpenGL: From Theory to Experiments

by Sumanta Guha

Chapman & Hall/CRC

CHAPTER 20

Programmable Pipelines

Click for [redSquare.cpp](#) [Program](#) [Windows Project](#)

Experiment 20.1. Fire up `redSquare.cpp` in the folder `Code/GLSL/Red-Square`.

Note: For how to set up the environment to run GLSL programs see Appendix B. Each of our GLSL programs is in a similarly named folder in the `GLSL` subdirectory of `Code`, with two accompanying shader files. Make sure, when running a GLSL program, to keep it in the same directory as its two shader files.

Now, `redSquare.cpp` is *exactly* `square.cpp` with the *barest* minimum amount of code added to be able to attach a vertex shader, called `passThrough.vs`, and a fragment shader, called `red.fs`. The output is a red square in the OpenGL window, as in Figure 20.2(a). **End**

Click for `redSquare.cpp` – vertex shader modified

[Program](#) [Windows Project](#)

Experiment 20.2. Replace the vertex shader code for `redSquare.cpp` with

```
void main()
{
    vec4 scaledPos = vec4(0.5 * gl_Vertex.xy, 0.0, 1.0);
    gl_Position = gl_ModelViewProjectionMatrix * scaledPos;
}
```

As expected, the *xy*-values of the square's vertices are both halved. See Figure 20.2(b) for a screenshot. **End**

Click for `multiColoredSquare1.cpp` Program Windows Project

Experiment 20.3. Run `multiColoredSquare1.cpp`. The program itself is a copy of `redSquare.cpp`, except for a different color at each square vertex *and* enabling of two-sided coloring with a call to `glEnable(GL_VERTEX_PROGRAM_TWO_SIDE)` in the setup routine. The output initially is a multi-colored square (Figure 20.3(a)).

The vertex shader `simpleColorizer.vs` writes out both a front and a back color to the built-in variables `gl_FrontColor` and `gl_BackColor`, respectively:

```
gl_FrontColor = gl_Color;
gl_BackColor = vec4(1.0, 0.0, 0.0, 1.0);
```

It reads the front color from the user-defined colors, which it accesses through the built-in state variable `gl_Color`, while the back color is a fixed red.

The fragment shader `passThrough.fs`, on the other hand, simply sets

```
gl_FragColor = gl_Color;
```

Now, the way the GLSL works, the fragment shader *does not* receive its `gl_Color` values from the program; rather they are computed by interpolation from either the `gl_FrontColor` or `gl_BackColor` values specified in the vertex shader, depending on the visible face. The use of the same name `gl_Color` to represent actually different variables in the two shaders – the vertex shader using `gl_Color` to access the program, while the fragment shader to access its sibling – can be a source of confusion. One needs to keep the context in mind when using this variable. As the current fragment shader does no more than assign colors interpolated from the vertex shader, it is called a *pass-through* fragment shader.

As the square itself is oriented counter-clockwise and, therefore, front-facing, the fragment shader computes its `gl_Color` values by interpolation from `gl_FrontColor`, which in turn tracks the vertex color values as specified in the program. Consequently, a multi-colored square is drawn.

A fun way to reverse the square’s orientation next is with a bit of swizzling. Accordingly, replace the vertex shader code with

```
void main()
{
    gl_FrontColor = gl_Color;
    gl_BackColor = vec4(1.0, 0.0, 0.0, 1.0);

    vec4 transposePos = gl_Vertex.yxzw; // Interchanges x and y
    // coordinate values, reversing the order of the vertices.

    gl_Position = gl_ModelViewProjectionMatrix * transposePos;
}
```

to see now a back-facing red square (Figure 20.3(b)).

End

Click for `multiColoredSquare2.cpp` Program Windows Project

Experiment 20.4. Run `multiColoredSquare2.cpp`. The code is exactly as `redSquare.cpp`, except this time the output is a multi-colored square because of the new shaders. Figure 20.5 is a screenshot. **End**

Click for `wavyCylinder1.cpp` Program Windows Project

Experiment 20.5. Run `wavyCylinder1.cpp`. This program, based on `cylinder.cpp`, draws a cylinder with a wavy surface, allowing the user to control the number of waves, as well as change its color from red to green. Press the up/down arrow keys to change the waviness, the left/right arrow keys to change the color and ‘x’-‘Z’ keys to turn the cylinder. Figure 20.6 shows the cylinder initially. **End**

Click for `wavyCylinder2.cpp` Program Windows Project

Experiment 20.6. Run `wavyCylinder2.cpp`. The output and controls are exactly as for `wavyCylinder1.cpp`. The difference between the two is in the mechanism by which the cross-section of the cylinder is scaled, which we discuss next. **End**

Click for `bumpMappingPerVertexLighting.cpp` Program Windows Project

Experiment 20.7. Run `bumpMappingPerVertexLighting.cpp`, which is code-wise almost exactly `bumpMapping.cpp`, but with a couple of shaders attached. Interaction is the same as well: press space to toggle between bump mapping on and off. Figures 20.7(a) and (b) are screenshots of `bumpMapping.cpp` and `bumpMappingPerVertexLighting.cpp`, respectively, doing bump mapping. Yes, they are exactly the same and we’ll see momentarily why! **End**

Click for `bumpMappingPerVertexLighting.cpp` – vertex shader modified
Program Windows Project

Experiment 20.8. If you are skeptical that we have actually replicated fixed-functionality lighting calculations in the vertex shader `perVertexLightingSimple.vs` and wondering if we are still somehow sneaking the output from fixed-functionality, then replace the `gl_FrontColor` specification in that shader with

```
gl_FrontColor = vec4(1.0, 0.0, 0.0, 1.0);
```

Figure 20.8 is a screenshot. There is no doubt, is there, that it's the vertex shader that's in charge of color calculation?! **End**

Click for `bumpMappingPerPixelLighting.cpp` **Program** **Windows**
Project

Experiment 20.9. Run `bumpMappingPerPixelLighting.cpp`. The program itself is identical to `bumpMappingPerVertexLighting.cpp` – the difference is in the shaders, which now implement Phong shading, or per-pixel lighting as it is called. Again, press space to toggle between bump mapping on and off. Figure 20.7(c) is a screenshot. **End**

Click for `interpolateTextures.cpp` **Program** **Windows** **Project**

Experiment 20.10. Run `interpolateTextures.cpp`, which allows the user to interpolate between (or, blend, if you like) two textures painted on a square. Figure 20.9 shows screenshots of the start, a part way and end configurations. **End**