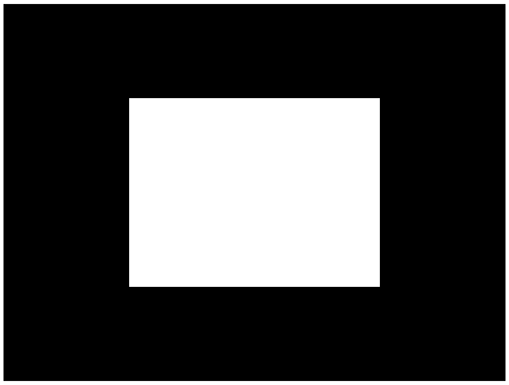
Lazy Foo' Productions

SDL Forums SDL Tutorials Articles OpenGL Tutorials OpenGL Forums
News FAQs Contact Donations

SDL and Modern OpenGL



Last Updated 6/19/19

With OpenGL 3 there was a massive overhaul that made everything shader based. In this tutorial we'll be rendering a quad using core modern OpenGL.

```
//Using SDL, SDL OpenGL, GLEW, standard IO, and strings
#include <SDL.h>
#include <gl\glew.h>
#include <SDL_opengl.h>
#include <gl\glu.h>
#include <stdio.h>
#include <string>
```

For this tutorial we'll be using the OpenGL Extension Wrangler. Certain operating systems like windows only support a limited amount of OpenGL by default. Using GLEW you can get the latest functionality. If you use GLEW, make sure to include the GLEW header before any OpenGL headers.

GLEW is an extension library and if you can set up any of the SDL extension libraries you can set up GLEW.

```
//Shader loading utility programs
void printProgramLog( GLuint program );
void printShaderLog( GLuint shader );
```

Here are some custom functions we're making to report any errors when making our shader programs.

```
//Graphics program
GLuint gProgramID = 0;
GLint gVertexPos2DLocation = -1;
GLuint gVBO = 0;
GLuint gIBO = 0;
```

The way modern OpenGL works is that we create shader programs (gProgramID) that process vertex attributes like positions (gVertexPos2DLocation). We put vertices in Vertex Buffer Objects (gVBO) and specify the order in which to draw them using Index Buffer Objects.

```
        //Use OpenGL 3.1 core
        SDL_GL_SetAttribute( SDL_GL_CONTEXT_MAJOR_VERSION, 3 );
        SDL_GL_SetAttribute( SDL_GL_CONTEXT_MINOR_VERSION, 1 );
        SDL_GL_SetAttribute( SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_PROFILE_CORE );
```

Here we're initializing for a version 3.1 core context. 3.1 core gets rid off al the old functionality. We specify the major and minor version like before and make it a core context by setting the profile mask to core.

```
            //Create context
            gContext = SDL_GL_CreateContext( gWindow );
            if( gContext == NULL )
            {
                printf( "OpenGL context could not be created! SDL Error: %s\n", SDL_GetError() );
                success = false;
            }
            else
            {
                //Initialize GLEW
                glewExperimental = GL_TRUE;
                GLenum glewError = glewInit();
                if( glewError != GLEW_OK )
                {
                    printf( "Error initializing GLEW! %s\n", glewGetErrorString( glewError ) );
                }

                //Use Vsync
                if( SDL_GL_SetSwapInterval( 1 ) < 0 )
                {
                    printf( "Warning: Unable to set VSync! SDL Error: %s\n", SDL_GetError() );
                }

                //Initialize OpenGL
                if( !initGL() )
                {
                    printf( "Unable to initialize OpenGL!\n" );
                    success = false;
                }
            }
```

After we create our context we initialize GLEW. Since we want the latest features, we have to set glewExperimental to true. After that we call glewInit() to initialize GLEW.

```
bool initGL()
{
    //Success flag
    bool success = true;

    //Generate program
    gProgramID = glCreateProgram();
```

In our initialization function we're going to create our shader program to render with along with the VBO and IBO data.

If you've never worked with OpenGL shaders, this function is probably going to go over your head. It's OK because this tutorial is about how to use SDL's 3.0+ context controls, not so much the detail about how OpenGL 3.0+ works. Just try to get a general idea on how a shader works.

```
    //Create vertex shader
    GLuint vertexShader = glCreateShader( GL_VERTEX_SHADER );

    //Get vertex source
    const GLchar* vertexShaderSource[] =
    {
        "#version 140\nin vec2 LVertexPos2D; void main() { gl_Position = vec4( LVertexPos2D.x, LVertexPos2D.y, 0, 1 );
    };

    //Set vertex source
    glShaderSource( vertexShader, 1, vertexShaderSource, NULL );

    //Compile vertex source
    glCompileShader( vertexShader );

    //Check vertex shader for errors
    GLint vShaderCompiled = GL_FALSE;
    glGetShaderiv( vertexShader, GL_COMPILE_STATUS, &vShaderCompiled );
    if( vShaderCompiled != GL_TRUE )
    {
        printf( "Unable to compile vertex shader %d!\n", vertexShader );
        printShaderLog( vertexShader );
        success = false;
    }
```

Here we are loading a vertex shader from an in code source. If the vertex shader failed to load and compile we use our log printing function to spit out the error.

```
    else
    {
        //Attach vertex shader to program
        glAttachShader( gProgramID, vertexShader );


        //Create fragment shader
        GLuint fragmentShader = glCreateShader( GL_FRAGMENT_SHADER );

        //Get fragment source
        const GLchar* fragmentShaderSource[] =
        {
            "#version 140\nout vec4 LFragment; void main() { LFragment = vec4( 1.0, 1.0, 1.0, 1.0 ); }"
        };

        //Set fragment source
        glShaderSource( fragmentShader, 1, fragmentShaderSource, NULL );

        //Compile fragment source
        glCompileShader( fragmentShader );

        //Check fragment shader for errors
        GLint fShaderCompiled = GL_FALSE;
        glGetShaderiv( fragmentShader, GL_COMPILE_STATUS, &fShaderCompiled );
        if( fShaderCompiled != GL_TRUE )
        {
            printf( "Unable to compile fragment shader %d!\n", fragmentShader );
            printShaderLog( fragmentShader );
            success = false;
        }
```

If the vertex shader loaded successfully we attach it to the program and then compile the fragment shader.

```
        else
        {
            //Attach fragment shader to program
            glAttachShader( gProgramID, fragmentShader );


            //Link program
            glLinkProgram( gProgramID );

            //Check for errors
            GLint programSuccess = GL_TRUE;
            glGetProgramiv( gProgramID, GL_LINK_STATUS, &programSuccess );
            if( programSuccess != GL_TRUE )
            {
                printf( "Error linking program %d!\n", gProgramID );
                printProgramLog( gProgramID );
                success = false;
            }
```

If the fragment shader compiled, we attach it to the shader program and link it.

```
            else
            {
                //Get vertex attribute location
                gVertexPos2DLocation = glGetAttribLocation( gProgramID, "LVertexPos2D" );
                if( gVertexPos2DLocation == -1 )
                {
                    printf( "LVertexPos2D is not a valid glsl program variable!\n" );
                    success = false;
                }
```

If the program linked successfully we then get the attribute from the shader program so we can send it vertex data.

```
                else
                {
                    //Initialize clear color
                    glClearColor( 0.f, 0.f, 0.f, 1.f );

                    //VBO data
                    GLfloat vertexData[] =
                    {
                        -0.5f, -0.5f,
                         0.5f, -0.5f,
                         0.5f,  0.5f,
                        -0.5f,  0.5f
                    };

                    //IBO data
                    GLuint indexData[] = { 0, 1, 2, 3 };

                    //Create VBO
                    glGenBuffers( 1, &gVBO );
                    glBindBuffer( GL_ARRAY_BUFFER, gVBO );
                    glBufferData( GL_ARRAY_BUFFER, 2 * 4 * sizeof(GLfloat), vertexData, GL_STATIC_DRAW );

                    //Create IBO
                    glGenBuffers( 1, &gIBO );
                    glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, gIBO );
                    glBufferData( GL_ELEMENT_ARRAY_BUFFER, 4 * sizeof(GLuint), indexData, GL_STATIC_DRAW );
                }
            }
        }
    }

    return success;
}
```

After we get the shader program working, we create the VBO and IBO. As you can see, the VBO has the same positions as the quad from the last tutorial.

```
void printProgramLog( GLuint program )
{
    //Make sure name is shader
    if( glIsProgram( program ) )
    {
        //Program log length
        int infoLogLength = 0;
        int maxLength = infoLogLength;

        //Get info string length
        glGetProgramiv( program, GL_INFO_LOG_LENGTH, &maxLength );

        //Allocate string
        char* infoLog = new char[ maxLength ];

        //Get info log
        glGetProgramInfoLog( program, maxLength, &infoLogLength, infoLog );
        if( infoLogLength > 0 )
        {
            //Print Log
            printf( "%s\n", infoLog );
        }

        //Deallocate string
        delete[] infoLog;
    }
    else
    {
        printf( "Name %d is not a program\n", program );
    }
}

void printShaderLog( GLuint shader )
{
    //Make sure name is shader
    if( glIsShader( shader ) )
    {
        //Shader log length
        int infoLogLength = 0;
        int maxLength = infoLogLength;

        //Get info string length
        glGetShaderiv( shader, GL_INFO_LOG_LENGTH, &maxLength );

        //Allocate string
        char* infoLog = new char[ maxLength ];

        //Get info log
        glGetShaderInfoLog( shader, maxLength, &infoLogLength, infoLog );
        if( infoLogLength > 0 )
        {
            //Print Log
            printf( "%s\n", infoLog );
        }
        //Deallocate string
        delete[] infoLog;
    }
    else
    {
        printf( "Name %d is not a shader\n", shader );
    }
}
```

Here are our log printing functions. These grab the shader log from the given shader or program and spit it out to the console.

```
void render()
{
    //Clear color buffer
    glClear( GL_COLOR_BUFFER_BIT );

    //Render quad
    if( gRenderQuad )
    {
        //Bind program
        glUseProgram( gProgramID );

        //Enable vertex position
        glEnableVertexAttribArray( gVertexPos2DLocation );

        //Set vertex data
        glBindBuffer( GL_ARRAY_BUFFER, gVBO );
        glVertexAttribPointer( gVertexPos2DLocation, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GLfloat), NULL );

        //Set index data and render
        glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, gIBO );
        glDrawElements( GL_TRIANGLE_FAN, 4, GL_UNSIGNED_INT, NULL );

        //Disable vertex position
        glDisableVertexAttribArray( gVertexPos2DLocation );

        //Unbind program
        glUseProgram( NULL );
    }
}
```

In our rendering function, we bind our shader program, enable vertex positions, bind the VBO, set the data offset, bind the IBO, and draw the quad as a triangle fan. Once we're done we disable the vertex attribute and unbind the program.

Again this tutorial is more for people with some OpenGL experience that want to know how to switch over to core functionality. The fact is that this code will work with an OpenGL 2.1 context as well as a 3.0 context (Well, except for the shader code because OpenGL 2.1 only supports up to #version 120). Core OpenGL just removes OpenGL calls that don't reflect modern hardware.

If you want to learn more about modern opengl, I have OpenGL shader tutorials too.

Also, I get e-mails of how this code is broken because if you set the version to 3.2+ it won't work because it doesn't use vertex array objects (or VAOs). The thing is this code works fine for version 3.1 core, which it is designed to be. However, OpenGL 3.2+ requires you create a VAO. Fortunately I cover VAOs in the OpenGL tutorial.

Download the media and source code for this tutorial here.

Back to SDL Tutorials

Share 237 Tweet

SDL Forums SDL Tutorials Articles OpenGL Tutorials OpenGL Forums

News FAQs Contact Donations

Copyright Lazy Foo' Productions 2004-2021