Getting Started with LLGL

Lukas Hermanns

September 16, 2016

Introduction

LLGL (Low Level Graphics Library) is a thin abstraction layer for graphics APIs such as OpenGL, Direct3D, and Vulkan. It is meant to abstract these rendering technologies to one uniform interface. The library is written entirely in C++11, so you'll need a modern C++ compiler, i.e. at least **VisualC++ 2013** for Windows, **g++ 4.8** for Linux, or **Clang 3.1** for MacOS.

Prerequisites

To use this library you should be familiar with these subjects:

- **Basic C++11 Programming**
  Since the library is written in C++11, you should know something about *smart pointers*, *raw pointers*, and basic *object-oriented programming* (OOP) in C++.
- **Basic Linear Algebra**
  You should be familiar with at least *Vectors* and *Matrices*.
- **Fundamentals in Graphics Programming**
  You should be familiar with the fundamentals of graphics programming, since this is only a low-level graphics library. You should also be familiar with at least one of the major graphics APIs, i.e. *OpenGL*, *Direct3D*, *Vulkan*, *Mantle*, or *Metal*, because LLGL does only little to no higher abstractions.
- **Shading Languages**
  LLGL forces you to always write your own shaders, so you should be familiar with GLSL, HLSL, or Cg.

Progress

This project is still in its early steps. Here is a short overview of its progress:

- **Windows support**
  Windows 10 is the main development environment of the author, so this platform has the best support.
- **Linux support**
  Kubuntu Linux 16 is used inside a virtual machine by the author to develop the linux port. This platform is partially supported (i.e. anti-aliasing or other features are not complete on this platform).
- **MacOS support**
  MacOS is currently not supported, but it's part of the plan ;-)
- **OpenGL renderer**
  OpenGL renderer is ~90% complete
- **Direct3D12 renderer**
  Direct3D 12 renderer is ~5% complete
- **Direct3D11 renderer**
  Direct3D 11 renderer is not supported yet.
- **Vulkan renderer**
  Vulkan renderer is not supported yet.

Build Process

Dependencies

GaussianLib

The only required dependency is the header-only library GaussianLib, which is used for basic linear algebra computations with vectors and matrices.

OpenGL

To build the OpenGL render system you need the OpenGL extension header files and an up-to-date graphics driver. For Windows the header files `glext.h` and `wglext.h` are required. For Linux the header files `glext.h` and `glxext.h` are required. For MacOS no header files need to be downloaded, since the OpenGL version depends on the OS version. You can find the header files at the OpenGL registry page. Place the header files in the `include/GL/` folder of your compiler environment or add the include path later in your build settings.

Direct3D

Since VisualStudio 2013, the DirectX framework (of which Direct3D is a part of) is included within the VisualStudio setup, so no further SDK needs to be installed.

Vulkan

To build the Vulkan render system you need the Vulkan SDK, and of course a graphics driver which supports at least Vulkan 1.0.

Build Tool

To build the LLGL project files you need the build tool CMake 2.8 or later. The build process is now demonstrated with the CMake GUI on Windows, but it can also be configured on a command line (more about this see cmake.org/runningcmake).

Set the source directory ("Where is the source code:") to the LLGL repository and set the build directory ("Where to build the binaries") where you want your project files. In this example (see 1) the source directory is `<...>/LLGL/repository` and the build directory is `<...>/LLGL/build_msvc14` because the project files are build for MSVC14 (VisualStudio 2015).

Now set the GaussianLib include directory (in this example `<...>/GaussianLib/repository/include`) and click on "Configure". If everything worked quite well, you should see the message "Configuring done" in the lower box. To finally create the project files, clock on "Generate". Then your project files should be located in the build directory you just set up previously.

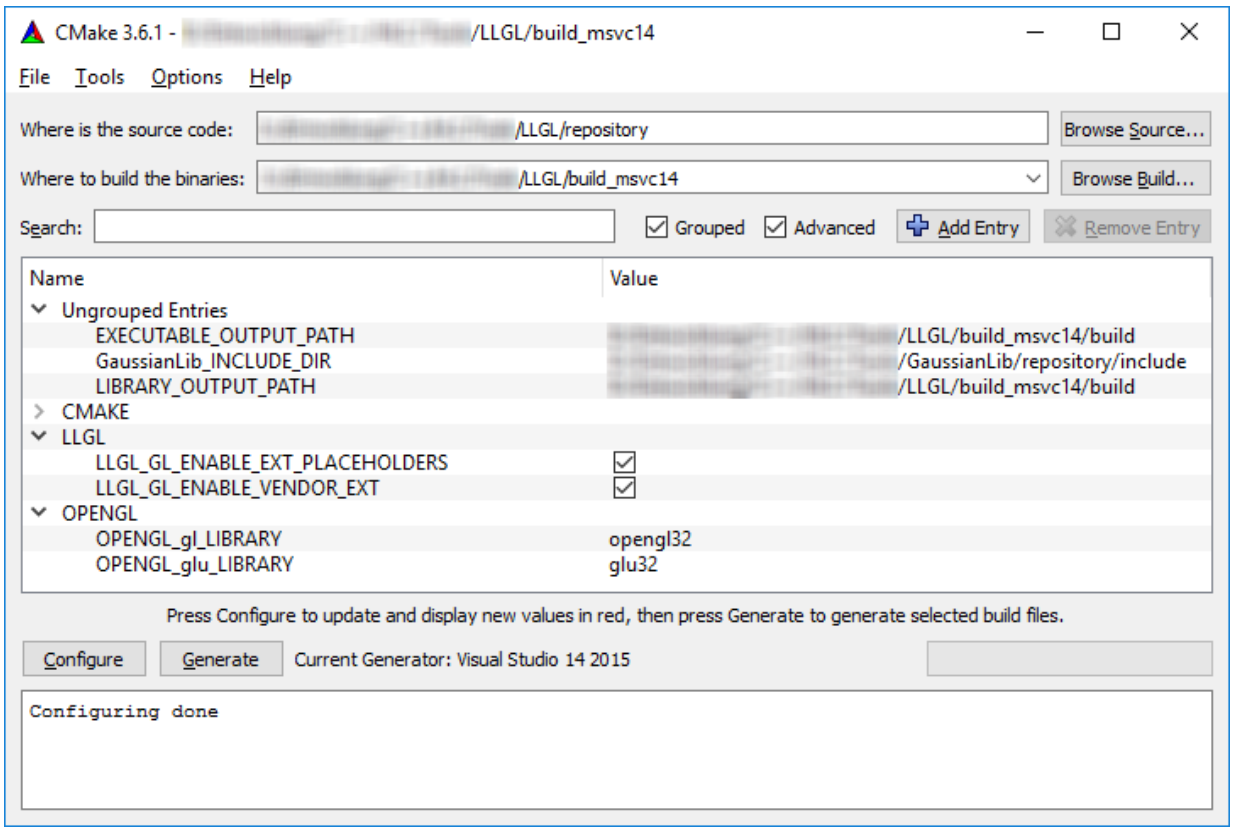

Figure 1: CMake GUI mask to set up the project files for VisualStudio 2015 (MSVC14).

There are a few options you can switch on and off, which will enable or disable the respective macro when you compile the project:

- `LLGL_GL_ENABLE_EXT_PLACEHOLDERS`
  Specifies whether OpenGL extensions should be replaced by placeholder procedures when they are not available. This may help debugging and should not influence the runtime performance.
- `LLGL_GL_ENABLE_VENDOR_EXT`
  Specifies whether vendor specific OpenGL extensions should be enabled or disabled. One of these extensions is for conservative rasterization (`GL_NV_conservative_raster` and `GL_INTEL_conservative_rasterization`) for instance. These extensions will only be loaded and used by the runtime if they are available on the host platform.

Hello Triangle

After we have set up and build the library, we can start rendering some geometry. Our example will consist of a single C++ source file (e.g. "`main.cpp`") and two shader files (e.g. "`vertex.glsl`" and "`fragment.glsl`"). The include path for a project, that uses LLGL, must be set to *<your-LLGL-repository>*`/include/`, and the LLGL library file must be added to the linker (for Windows this is "`LLGL.lib`" when compiling in Release mode and "`LLGLD.lib`" when compiling in Debug mode). All the other library files don't need to be added to the linker (e.g. "`LLGL_OpenGL.lib`"), since the respective renderer module is loaded at runtime.

Now let's start with a small example. At first we need to include the header files. The main header we need is `LLGL/LLGL.h` where the entire LLGL interface is declared. For our example we also need basic I/O classes from `iostream` for standard output, `fstream` to read the shader files, and `Gauss/Gauss.h` for some matrix classes:

```
#include <LLGL/LLGL.h>
#include <Gauss/Gauss.h>
#include <iostream>
#include <fstream>
```

Next we define the C++ main function and wrap the entire example code into a large `try-catch` block, to quit the application with a meaningful error message if any failure happens:

```
int main() {
    try {
        /* example code here ... */
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
    }
    return 0;
}
```

To create an LLGL renderer instance in our main function, we load a renderer module (a *module* here denotes a dynamic shared library) from the static `Load` function of the `RenderSystem` interface:

```
std::shared_ptr<LLGL::RenderSystem> renderer = LLGL::RenderSystem::Load("OpenGL");
```

Here we could actually use the C++11 keyword `auto` to simplify the code, but for explanation purposes we keep the types explicit. Most creation or load functions return a new instance wrapped in a `std::unique_ptr`, but in this case LLGL needs to keep track of the instance to check if it has already been expired, which is only feasible with an `std::shared_ptr`.

Moreover, most functions use enumerations instead of strings to specify some type, but in this case a module can be loaded dynamically at runtime and further modules can be added or removed independently. Therefore the renderer module is specified by a string (here "OpenGL"). Other modules are named "Direct3D11", "Direct3D12", and "Vulkan". Whenver you load a new renderer, there must not remain any references to this shared object, because only a single renderer can be loaded at a time. When this shared object expires, all objects allocated by this renderer will be deleted automatically.

If loading the renderer failed, an `std::runtime_error` exception will be thrown, which can be cached to show an erorr message and/or load another renderer module instead. This can be very handy if a specific Direct3D version is not installed on the host Windows platform, so another Direct3D version or OpenGL renderer can be loaded as fallback without disturbing the user with akward error messages and program crashes.

After we created the renderer we need a graphics context to draw into. This is done by the `CreateRenderContext` function which takes a descriptor structure:

```
LLGL::RenderContextDescriptor contextDesc;
{
    contextDesc.videoMode.resolution = { 800, 600 };
}
LLGL::RenderContext* context = renderer->CreateRenderContext(contextDesc);
```

This is a minimal example for the render context descriptor. There are much more attributes to specify anti-aliasing, vertical-synchronisation, etc. See the API documentation for more information about these attributes.

This render context will create its own window, but we could also specifiy our own one. However, in this example we keep it simple. To access this window and change the title, we use the `GetWindow` function of the `RenderContext` interface which returns a constant reference to its window:

```
context->GetWindow().SetTitle(L"LLGL Tutorial 01: Hello Triangle");
```

Since some platforms (such as Win32) support Unicode window titles, our string literal starts with the 'L' token.

Next we create a vertex buffer to store our geometry. For this example we give up an index buffer, since we only draw a single triangle and no complex models:

```
LLGL::VertexBuffer* vertexBuffer = renderer->CreateVertexBuffer();
```

Now we have an empty and unspecified vertex buffer. Next we define our vertex data and the vertex format, which is required to tell the rendering API how the vertex attributes are located within the vertex buffer:

```
// Vertex data structure
struct Vertex {
    Gs::Vector2f    position;
    LLGL::ColorRGBf color;
};

// Vertex data (3 vertices for our triangle)
Vertex vertices[] = {
    { {  0,  1 }, { 1, 0, 0 } }, // 1st vertex: center-top, red
    { {  1, -1 }, { 0, 1, 0 } }, // 2nd vertex: right-bottom, green
    { { -1, -1 }, { 0, 0, 1 } }, // 3rd vertex: left-bottom, blue
};

// Vertex format
LLGL::VertexFormat vertexFormat;
vertexFormat.AddAttribute("position", LLGL::DataType::Float, 2); // position has 2 float components
vertexFormat.AddAttribute("color",    LLGL::DataType::Float, 3); // color has 3 float components
```

The `AddAttribute` function adds the attributes to the vertex format. The order of these function calls determines the location in the vertex data, so they must match the order of the member fields in the vertex data structure (here "`struct Vertex`").

The final step for our vertex buffer generation is to upload the data to the GPU. For all hardware buffers and textures, this is done with the respective "`Setup...`" function of the `RenderSystem` interface:

```
renderer->SetupVertexBuffer(
    *vertexBuffer,              // Pass the vertex buffer as reference
    vertices,                   // Pointer to the vertex data
    sizeof(vertices),           // Size (in bytes) of the vertex buffer
    LLGL::BufferUsage::Static, // Buffer usage is static since we won't change it frequently
    vertexFormat                // Vertex format
);
```

To update an entire hardware buffer or texture (or only a portion of it) there is a respective "`Write...`" function.

Now we have the vertex buffer complete and we can cross over to shader creation. In LLGL the shaders (Vertex, Tessellation-Control, Tessellation-Evaluation, Geometry, Fragment, and Compute shaders) are created independently, and then attached and linked together with a shader program. For our example we only need a Vertex- and Fragment (also called "Pixel") shader:

```
LLGL::Shader* vertexShader = renderer->CreateShader(LLGL::ShaderType::Vertex);
LLGL::Shader* fragmentShader = renderer->CreateShader(LLGL::ShaderType::Fragment);
```

Now we have two empty shaders. Next we load the shader code from file with our custom "`ReadFileContent`" lambda function:

```
// Define the lambda function to read an entire text file
auto ReadFileContent = [](const std::string& filename) {
    std::ifstream file(filename);
    return std::string( ( std::istreambuf_iterator<char>(file) ),
                        ( std::istreambuf_iterator<char>(    ) ) );
};

// Load vertex- and fragment shader code from file
std::string vertexShaderCode = ReadFileContent("vertex.glsl");
std::string fragmentShaderCode = ReadFileContent("fragment.glsl");
```

After reading the shader code into strings we can compile the shaders:

```
auto CompileShader = [](LLGL::Shader* shader, const std::string& code) {
    // Compile shader
    shader->Compile(code);

    // Print info log (warnings and errors)
    std::string log = shader->QueryInfoLog();
    if (!log.empty()) {
        std::cerr << log << std::endl;
    }
};

CompileShader(vertexShader, vertexShaderCode);
CompileShader(fragmentShader, fragmentShaderCode);
```

Shader compilation works a little different between GLSL (for OpenGL) and HLSL (for Direct3D). For HLSL shader compilation there is a second overloaded "`Compile`" function with more parameters, to specifiy the entry point and shader version target.

Having the shaders compiled, we can now create a shader program, attach all shaders to it, bind the vertex attribute layout, and finally link the shader program:

```
// Create shader program which is used as composite
LLGL::ShaderProgram* shaderProgram = renderer->CreateShaderProgram();

// Attach vertex- and fragment shader to the shader program
shaderProgram->AttachShader(*vertexShader);
shaderProgram->AttachShader(*fragmentShader);

// From now on we only use the shader program, so the shaders can be released
renderer->Release(*vertexShader);
renderer->Release(*fragmentShader);

// Bind vertex attribute layout (this is not required for a compute shader program)
shaderProgram->BindVertexAttributes(vertexFormat.GetAttributes());

// Link shader program and check for errors
if (!shaderProgram->LinkShaders()) {
    throw std::runtime_error(shaderProgram->QueryInfoLog());
}
```

The "`BindVertexAttributes`" function binds the vertex layout to the shader program. This must be called after shader attachment but before shader linking (except a compute shader program is used).

Before we continue with the C++ code, we first take a look at the shader code:

`vertex.glsl`

```
// GLSL shader version 1.30 (for OpenGL 3.1)
#version 130

// Vertex attributes (these names must match our vertex format attributes)
in vec2 position;
in vec3 color;

// Vertex output to the fragment shader
out vec3 vertexColor;

// Vertex shader main function
void main() {
    gl_Position = vec4(position, 0, 1);
    vertexColor = color;
}
```

This simple vertex shader only passes the vertex position and color (with default interpolation) to the fragment shader.

And here is the fragment shader:

fragment.glsl

```
// GLSL shader version 1.30 (for OpenGL 3.1)
#version 130

// Fragment input from the vertex shader
in vec3 vertexColor;

// Fragment output color
out vec4 fragColor;

// Fragment shader main function
void main() {
    fragColor = vec4(vertexColor, 1);
}
```

Now we are finally done with setting up the shader. Next we need to create a graphics pipeline. This concept is derived from modern graphics APIs such as Direct3D 12 and Vulkan. The major pipeline state is stored inside this pipeline state object. For older graphics APIs (such as OpenGL) LLGL will set the respective render states by its internal state manager, to reduce state changes.

The graphics pipeline specifies the depth-, stencil-, rasterizer-, blending-, and shader states and is created as follows:

```
LLGL::GraphicsPipelineDescriptor pipelineDesc;
{
    pipelineDesc.shaderProgram = shaderProgram;
}
LLGL::GraphicsPipeline* pipeline = renderer->CreateGraphicsPipeline(pipelineDesc);
```

In our example we can use all default settings of the graphics pipeline descriptor except the shader program, which must always be set by the client programmer.

We now have all render objects, so we can start with the main loop:

```
// Run main loop until the main window is closed
while (context->GetWindow().ProcessEvents()) {
    /* render code here ... */
}
```

This main loop will run until the user clicks the close button on the window of the render context. The rest of the code is pretty simple, since the most work is done during initialization. What we need to do is to clear the color buffer of the previous frame, set the graphics pipeline, set the vertex buffer, set the primitive topology, draw the primitives, and present the result on the frame buffer:

```
// Clear color buffer
context->ClearBuffers(LLGL::ClearBuffersFlags::Color);

// Bind graphics pipeline
context->SetGraphicsPipeline(*pipeline);

// Bind vertex buffer
context->SetVertexBuffer(*vertexBuffer);

// Set primitive topology (tell the renderer what primitives to draw)
context->SetPrimitiveTopology(LLGL::PrimitiveTopology::TriangleList);

// Draw triangle with 3 vertices
context->Draw(3, 0);

// Present the result on the frame buffer (or rather on the screen)
context->Present();
```

When you have done everything right, you should see something like shown in figure 2 after compilation and program start.

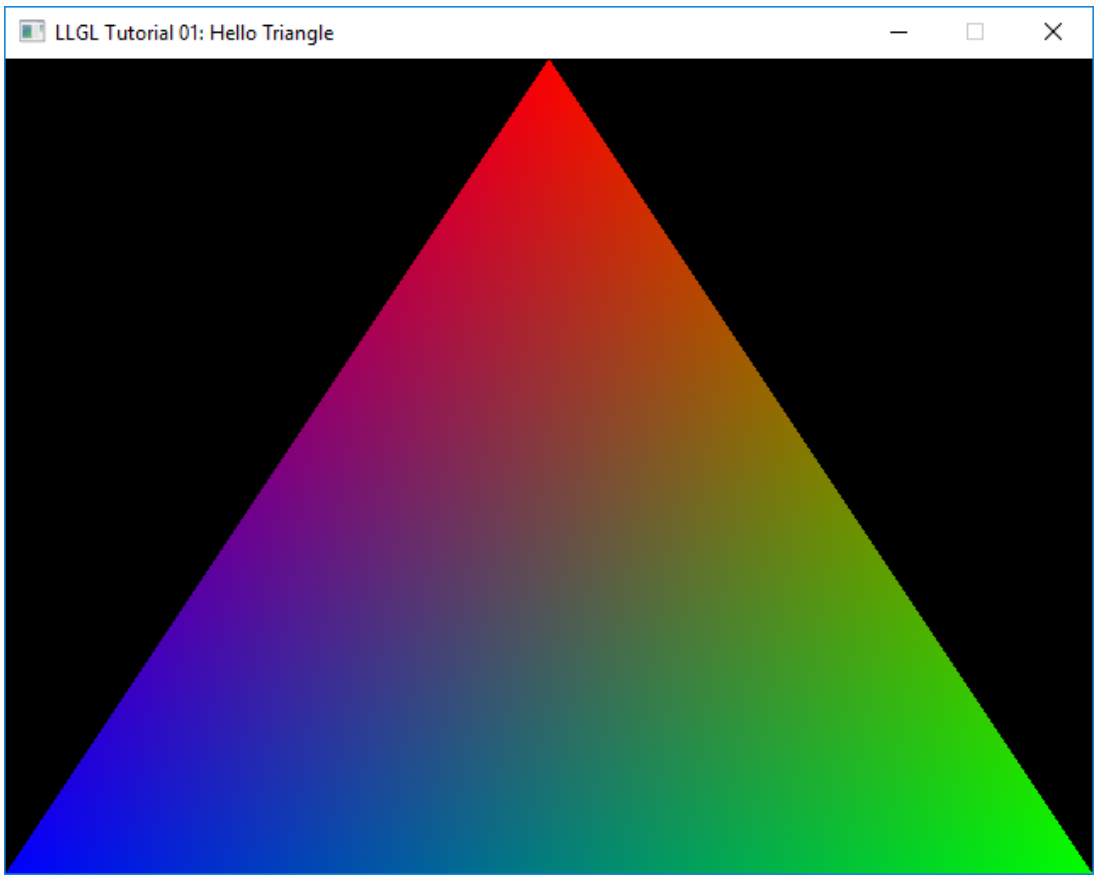

Figure 2: Output of the “Tutorial01: HelloTriangle” running on Windows 10.