

Getting Started with LLGL

L. Hermanns

February 19, 2023



Contents

I	Introduction	3
1	In a nutshell	3
1.1	What can LLGL do for me?	3
1.2	What can LLGL <i>not</i> do for me?	3
2	Prerequisites	4
3	Progress	5
3.1	Platforms	5
3.2	Renderers	5
3.3	Known issues	5
4	Build Process	6
4.1	Cloning Repository	6
4.2	Dependencies	6
4.2.1	OpenGL	6
4.2.2	Direct3D	6
4.2.3	Vulkan	6
4.3	Build Tool	7
5	API Overview	9
5.1	Rendering Interfaces	9
5.1.1	RenderSystem	9
5.1.2	SwapChain	9
5.1.3	CommandBuffer	9
5.2	Windowing System	9
II	Tutorials	10
6	Hello Triangle	10
III	Extensibility	15
7	Custom Surface Class	15
8	Custom Render System	17

Part I

Introduction

1 In a nutshell

1.1 What can **LLGL** do for me?

- **Unification**

LLGL provides a unified interface across all supported renderers. Write your graphics render passes once and use them across multiple rendering APIs and platforms.

- **Low Overhead**

LLGL is meant to be a thin abstraction layer which only adds as less overhead between your application and the underlying rendering API as possible. At best, a function is just a wrapper that passes the parameters on to the native rendering API. For example, the implementation of the `CommandBuffer::DrawIndexed` interface for Direct3D 11 looks something like this:

```
void D3D11CommandBuffer::DrawIndexed(std::uint32_t numVertices, std::uint32_t firstIndex) {
    context_>DrawIndexed(numVertices, firstIndex, 0);
}
```

- **Compatibility**

LLGL provides various compatibility functionalities between the renderers. For example, some image formats that are supported by OpenGL but not by Direct3D are converted ‘on the fly’ by **LLGL**. However, some compromises are inevitable, due to different hardware restrictions. Having said that, all incompatibilities are well documented.

- **Simplification**

Where close access to the hardware is not necessary, **LLGL** provides useful simplifications. For example, creating a swap-chain as well as a device context across multiple platforms can be a high effort when done manually. With **LLGL**, this can be done with a few lines of code while maintaining a high degree of control thanks to the rich descriptor structures.

- **Debug Layer**

In debug mode, **LLGL** makes use of the native debug layers for the respective renderer. These debug layers, however, vary in quality depending on the API. **LLGL** offers another debug layer on top of it that can be easily enabled or disabled. This layer performs extensive state-, parameter-, and descriptor validation before each draw call. This can be a powerful tool to find invalid render states or erroneous descriptors.

- **Extension Support**

Especially OpenGL is a complex system due to its many extensions. For texturing alone, **LLGL** manages eight different extensions. Depending on which extension is available on the host platform, **LLGL** uses the most efficient or otherwise emulates the functionality. Even a few vendor specific extensions are supported. Extensions in form of API revisions in Direct3D are also supported such as Direct3D 11.3.

1.2 What can **LLGL** *not* do for me?

- **Shader Cross Compilation**

LLGL unifies the underlying rendering APIs as far as possible, but since each rendering API has its own shading language, all shaders need to be provided for each selected renderer explicitly. Having said that, there are several shader cross compilers available:

- XShaderCompiler (github.com/LukasBanana/XShaderCompiler),
- SPIRV-Cross (github.com/KhronosGroup/SPIRV-Cross),
- hlsl2glsl (github.com/aras-p/hlsl2glslfork).
- HLSLCrossCompiler (github.com/James-Jones/HLSLCrossCompiler)

- **Scene Management**

LLGL is a low-level render system which does not provide any high-level scene management or animation system. It provides you with a set of functions to submit draw commands and render states to the graphics hardware as well as managing raw hardware buffers. ‘Game engine’-like functionality or anything beyond graphics is not part of **LLGL**.

2 Prerequisites

[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++ 2015** for Windows, **g++ 4.8** for Linux, or **Clang 3.1** for MacOS. To take advantage of 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*, or *Metal*, because [LLGL](#) does only little to no higher abstractions.

- **Shading Languages**

[LLGL](#) requires you to always write your own shaders, so you should be familiar with GLSL, HLSL, or Metal.

3 Progress

As with many open-source projects that people work on in their spare time, this project does *not* claim to be complete nor that it ever will. The best source to convince yourself of its capabilities is to take a look at the examples that are included in the [LLGL](#) repository.

3.1 Platforms

- **Windows**
Windows 10 is the main development environment of the author, so this platform has the best support.
- **GNU/Linux**
Ubuntu 17 (GNU/Linux) is used by the author to develop the linux port. GNU/Linux is supported for the most part. Both OpenGL and the Vulkan back-end are available.
- **macOS**
The MacOS port is in its early steps, but a few tutorials are already running. The development environment for the MacOS port is *macOS Sierra*.
- **iOS**
Experimental state for the iOS mobile platform.

3.2 Renderers

- **Direct3D 12** on Windows only
- **Direct3D 11** on Windows only
- **Vulkan** on Windows and GNU/Linux
- **Metal** on macOS and iOS
- **OpenGL** on Windows, GNU/Linux, and macOS

3.3 Known issues

A major feature with quite a lot of known issues is multi-threaded command recording. One of the big selling points of Vulkan and Direct3D 12 for better multi-threaded rendering support does not work correctly in [LLGL](#) as of 2023. Ironically, this is currently only fully support in the OpenGL backend as it servers as the reference implementation of [LLGL](#) since it's supported on most platforms.

4 Build Process

4.1 Cloning Repository

There are some submodules in the [LLGL](#) repository, all of which are optional. These submodules are located in the `external/` directory. To clone the repository with git, enter the following line in a command prompt:

```
git clone https://github.com/LukasBanana/LLGL.git
```

To clone the repository with all its submodules, enter the following:

```
git clone --recursive https://github.com/LukasBanana/LLGL.git
```

4.2 Dependencies

Since version 0.02, LLGL has no longer any required base dependencies except the one for the respective renderers. Previously, the `GAUSSIANLIB` was required but has been replaced by a few plain-old-data structures. The `GaussianLib` is now only required for the Examples and Test projects.

4.2.1 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 `glxext.h` and `wglxext.h` are required. For Linux the header files `glxext.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.

4.2.2 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.

4.2.3 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.

4.3 Build Tool

To build the [LLGL](#) project files you need the build tool CMake 3.7 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 Figure 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, click on “Generate”. Then your project files should be located in the build directory you just set up previously.

There are several options you can enable or disable to build the project:

- `LLGL_BUILD_(TESTS/EXAMPLES/RENDERER_...)`
Specifies whether to include all test, all examples, or the respective renderer projects.
- `LLGL_D3D11_ENABLE_FEATURELEVEL`
Specifies which feature level is enabled in the Direct3D 11 renderer. For example, feature level 11.1 requires that your development environment can find the `<d3d11_1.h>` header file. Feature level 11.1 enables logic fragment operations, and feature level 11.3 enables conservative rasterization for the D3D11 renderer.
- `LLGL_ENABLE_CHECKED_CAST`
Specifies whether to enable or disable dynamic checked casts. This is only available in debug mode.
- `LLGL_ENABLE_DEBUG_LAYER`
Specifies whether to enable or disable the debug layer. This is a wrapper around the render system and all other render objects for effective debugging.
- `LLGL_ENABLE_SPIRV_REFLECT`
Specifies whether to include the submodule `external/SPIRV` to enable code reflection of SPIR-V shader modules for the Vulkan renderer.
- `LLGL_ENABLE_UTILITY`
Specifies whether to enable or disable utility functions, which can be used to easily initialize descriptor structures. They must be included separately with the `LLGL/Utility.h` header file.
- `LLGL_GL_ENABLE_DSA_EXT`
Specifies whether to enable or disable the `GL_ARB_direct_state_access` extension that spans the entire render system.
- `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.
- `LLGL_GL_INCLUDE_EXTERNAL`
Specifies whether to include the OpenGL header files from the `external/GL/` directory of the LLGL repository. Otherwise the `GL/` directory must be elsewhere specified in your include search paths.

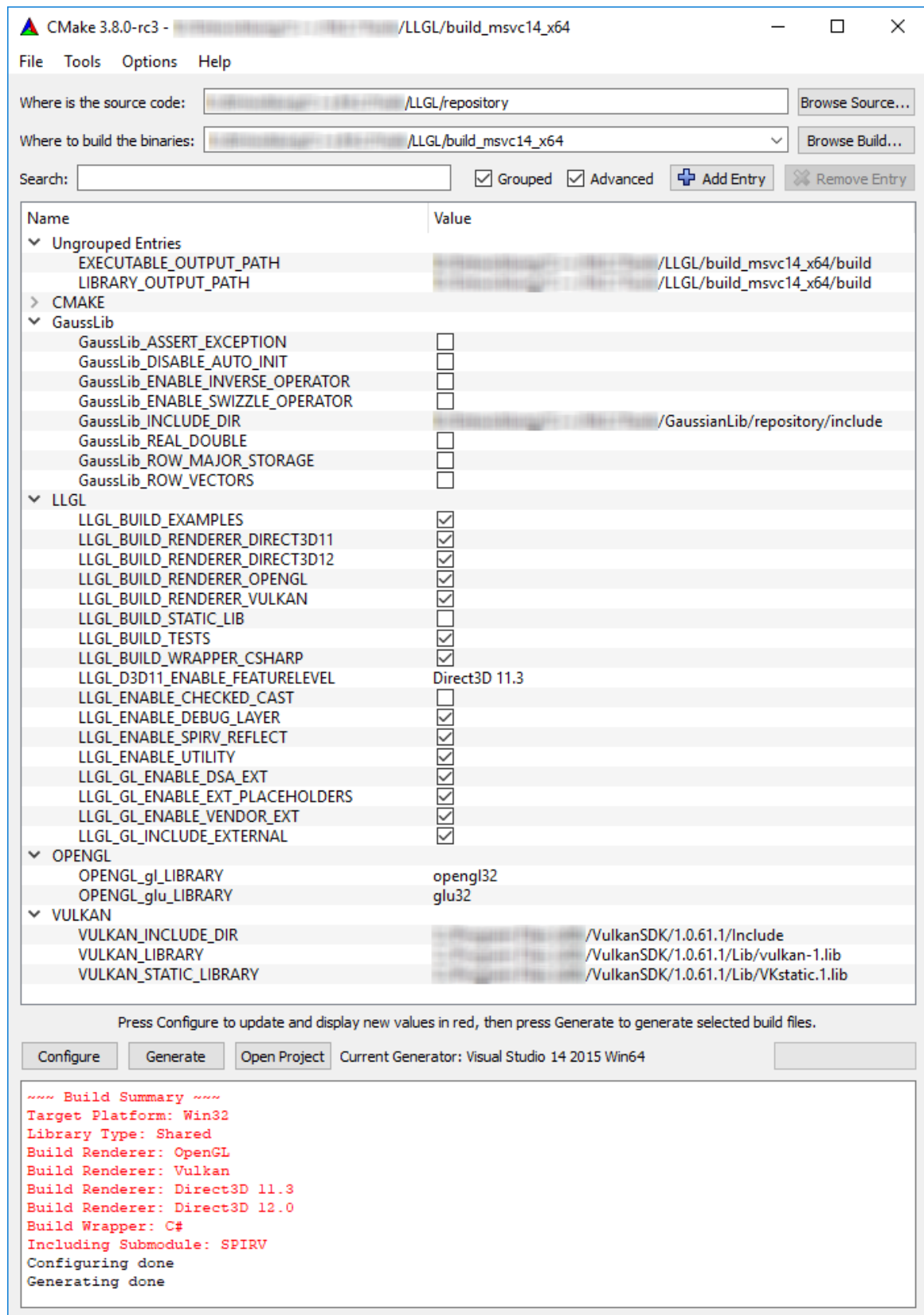


Figure 1: CMake GUI mask to set up the project files for VisualStudio 2015 (MSVC14).

5 API Overview

LLGL has a very simple and unified API design. For object creation, there is commonly a “. . .Descriptor” structure (e.g. `LLGL:BufferDescriptor`), which contains all necessary information to describe the object which is to be created.

5.1 Rendering Interfaces

There are three major interfaces for rendering: *RenderSystem* is mainly used for object creation and memory read/write operations, *SwapChain* is used to configure each framebuffer and its back buffers, and *CommandBuffer* which is used to set render states, draw primitives, and dispatch compute commands.

5.1.1 RenderSystem

In the *RenderSystem* interface there are several functions of the following form:

```
// Create a new object
Create...

// Update the data of an object
Write...

// Read the data from an object
Read...

// Map the memory of an object from GPU to CPU memory space
Map...

// Unmap the memory from an object
Unmap...

// Release an object
Release
```

5.1.2 SwapChain

The most important function in the *SwapChain* interface is *Present*, to show the content of the back buffer on the screen. There are a few other functions to access the context window and change the video mode:

```
// Access the surface to render into ('Window' on desktop/ 'Canvas' on mobile)
GetSurface

// Resizes all front- and back buffers (i.e. resolution, fullscreen/windowed mode etc.)
ResizeBuffers

// Set vertical synchronization (Vsync) interval
SetVsyncInterval
```

5.1.3 CommandBuffer

There are several overloaded functions for drawing operations with the naming convention “Draw”, “DrawIndexed”, “DrawInstanced”, or “DrawIndexedInstanced”. The most other functions are used to configure the command buffer of the graphics API, which have the following form:

```
// Set a hardware buffer/ texture/ sampler etc.
Set...

// Begin and always end a state (e.g. BeginQuery/ EndQuery)
Begin...
End...

// Draw primitives
Draw...
```

5.2 Windowing System

LLGL has the *Window* interface for a very basic but platform independent windowing system. Use its static function *Create* to create an instance of this interface for the host platform. A custom implementation of this interface can also be written and used for any renderer. In section 7 an example of a custom implementation with GLFW is shown.

Part II

Tutorials

6 Hello Triangle

NOTE: this section is not up-to-date!

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:

```
LLGL::RenderSystemPtr myRenderer = 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”. Whenever 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 caught to show an error 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 awkward error messages and program crashes.

After we created the renderer we need a framebuffer to draw into and present the result on the screen. This is done by the `CreateSwapChain` function which takes the following descriptor structure:

```
LLGL::SwapChainDescriptor mySwapChainDesc;
{
    mySwapChainDesc.resolution = { 640, 480 };
}
LLGL::SwapChain* mySwapChain = myRenderer->CreateSwapChain(mySwapChainDesc);
```

This is a minimal example for the swap-chain descriptor. There are several more attributes to specify multi-sampled anti-aliasing, bit depth, etc. See the API documentation for more information about these attributes.

This swap-chain will create its own surface, but we could also specify our own one. However, in this example we keep it simple. To access this surface and change the title, we use the `GetSurface` function of the `SwapChain` interface and interpret it as a `Window` on desktop platforms:

```

LLGL::Window& myWindow = LLGL::Cast<LLGL::Window>(mySwapChain->GetSurface());
myWindow.SetTitle("LLGL Tutorial 01: Hello Triangle");

```

The window that was created automatically is hidden by default and we have to show it explicitly.

```

myWindow.Show();

```

This way, modifications like changing the window title only become visible once the window is explicitly shown.

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. At first 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 {
    float   position[2];
    uint8_t color[4];
};

// Vertex data (3 vertices for our triangle)
Vertex myVertices[] = {
    { { +0.0f, +0.5f }, { 255, 0, 0, 255 } }, // 1st vertex: center-top, red
    { { +0.5f, -1.0f }, { 0, 255, 0, 255 } }, // 2nd vertex: right-bottom, green
    { { -0.5f, -1.0f }, { 0, 0, 255, 255 } }, // 3rd vertex: left-bottom, blue
};

// Vertex format
LLGL::VertexFormat vertexFormat;
vertexFormat.AppendAttribute({ "position", LLGL::Format::RG32Float }); // 2D float vector
vertexFormat.AppendAttribute({ "color", LLGL::Format::RGBA8UNorm }); // 4D unsigned normalized vector

```

The `AppendAttribute` 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”).

Now we can create our vertex buffer and upload the vertex data to the GPU by passing a pointer to the `initialData` parameter:

```

LLGL::BufferDescriptor myVertexBufferDesc;

// Define size (in bytes) of the buffer
myVertexBufferDesc.size = sizeof(vertices);

// We will bind it as a vertex buffer (can be a bitwise OR combination)
myVertexBufferDesc.bindFlags = LLGL::BindFlags::VertexBuffer;

// Set vertex buffer specific attributes
myVertexBufferDesc.vertexAttribs = vertexFormat.attributes;

LLGL::Buffer* myVertexBuffer = myRenderer->CreateBuffer(myVertexBufferDesc, vertices);

```

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 completed and we can move 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 pipeline state object (PSO). For our example we only need a Vertex- and Fragment (also called “Pixel”) shader:

```

LLGL::ShaderDescriptor vertexShaderDesc;

// Specifies the shader type. Here, we create a vertex shader
vertexShaderDesc.type = LLGL::ShaderType::Vertex;

// Pointer to the shader source. This can point to a null terminated string,
// or a buffer with shader byte code depending on the input language.
// Whether or not this must be a null terminated string is specified by the next field.
vertexShaderDesc.source = "vertex.glsl";

// We just load our shader from a file, so we specify 'CodeFile' as source type.
// CodeFile and BinaryFile denotes that our source must be a null terminated string.
vertexShaderDesc.sourceType = LLGL::ShaderSourceType::CodeFile;

// Specifiy the vertex input attributes. This is obviously only necessary for the vertex shader.
vertexShaderDesc.vertex.inputAttribs = vertexFormat.attributes;

```

Now can create our vertex shader. For simplicity, we just create our fragment shader // with one of the `ShaderDescriptor` constructors and use their default arguments:

```
LLGL::Shader* vertexShader = myRenderer->CreateShader(vertexShaderDesc);
LLGL::Shader* fragmentShader = myRenderer->CreateShader({ LLGL::ShaderType::Fragment, "fragment.glsl" });
```

This will load and compile the shaders. Once they are created, we can query whether the compilation was successful and what error or warning reports have been produced during compilation:

```
for (LLGL::Shader* shader : { vertexShader, fragmentShader }) {
    if (const LLGL::Report* report = shader->GetReport()) {
        if (report->HasErrors() {
            std::cerr << report->GetText() << std::endl; // Print errors
        } else {
            std::cout << report->GetText() << std::endl; // Print warnings
        }
    }
};
```

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 vec4 color;

// Vertex output to the fragment shader
out vec4 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 vec4 vertexColor;

// Fragment output color
out vec4 fragColor;

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

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 myPSODesc;

myPSODesc.vertexShader = myVertexShader;
myPSODesc.fragmentShader = myFragmentShader;
myPSODesc.renderPass = mySwapChain->GetRenderPass();

LLGL::PipelineState* myPSO = myRenderer->CreatePipelineState(myPSODesc);
```

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.

Before we start with the render main loop, we need a command buffer which can send render state and draw commands to the GPU:

```
// Create command buffer to submit subsequent graphics commands to the GPU
LLGL::CommandBuffer* myCmdBuffer = myRenderer->CreateCommandBuffer();
```

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

```
// Run main loop until the main window is closed
while (myWindow.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, draw the primitives, and present the result on the frame buffer:

```
// Begin recording GPU commands
myCmdBuffer->Begin();

// Begin the render pass with our swap-chain to render directly into the back buffer.
myCmdBuffer->BeginRenderPass(*mySwapChain);

// Set viewport (left: 0, top: 0, width: 640, height: 480)
myCmdBuffer->SetViewport(mySwapChain->GetResolution());

// Clear color buffer
myCmdBuffer->Clear(LLGL::ClearFlags::Color);

// Bind graphics PSO
myCmdBuffer->SetPipelineState(*myPSO);

// Bind vertex buffer
myCmdBuffer->SetVertexBuffer(*myVertexBuffer);

// Generate 3 vertices to draw a triangle
myCmdBuffer->Draw(3, 0);

// Finish render pass and command recording
myCmdBuffer->EndRenderPass();
myCmdBuffer->End();

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

When you have done everything right, you should see something like shown in figure 2 after compilation and program start (set working directory to tutorial/Tutorial01_HelloTriangle).

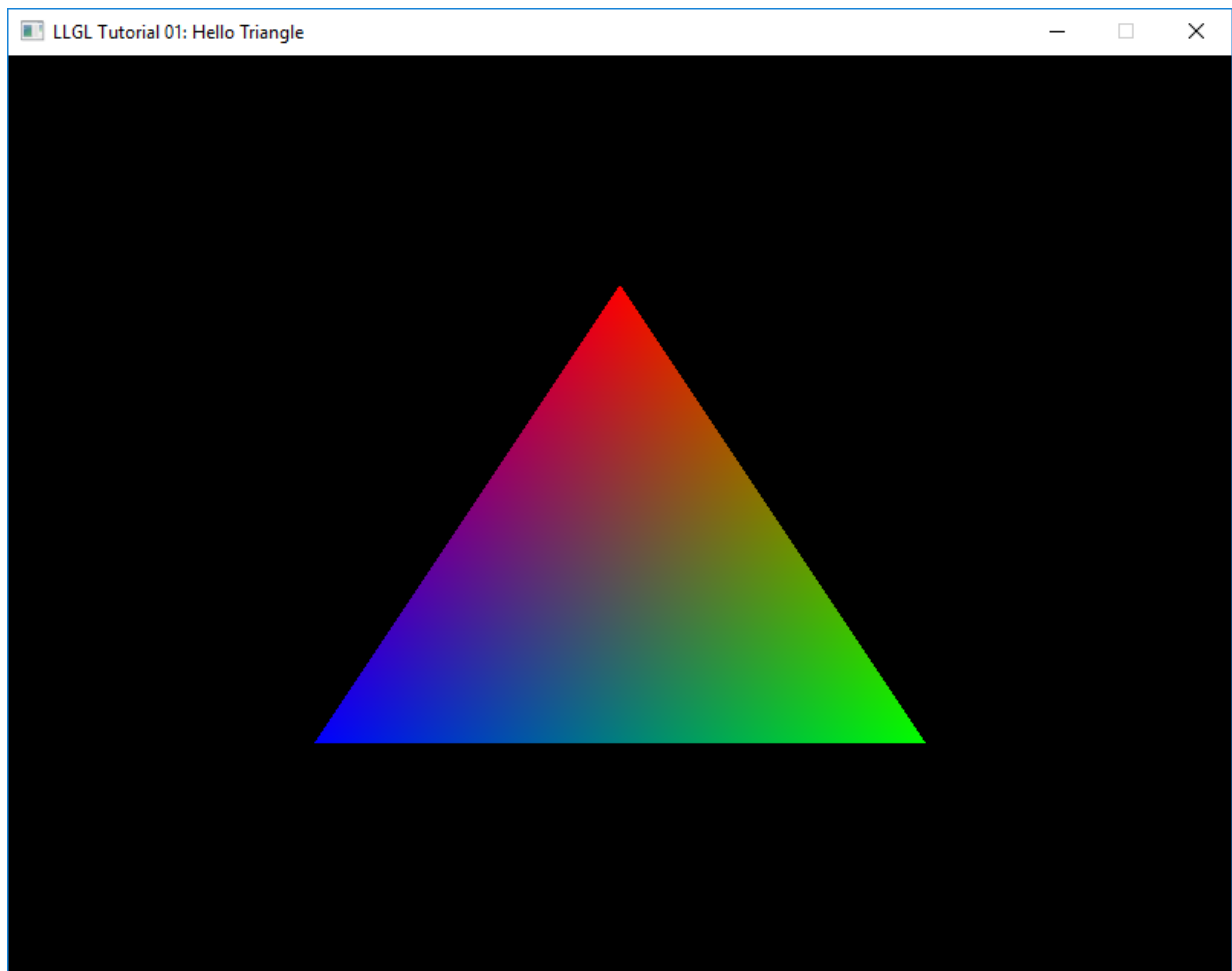


Figure 2: Output of the “Tutorial01: HelloTriangle” running on Windows 10.

Part III

Extensibility

7 Custom Surface Class

In this example a simple custom implementation of the Surface interface is demonstrated, to show how LLGL can be used with other windowing system libraries. Here we will use the popular cross-platform library GLFW.

We start with the include files of GLFW and LLGL:

```
// Include GLFW library (in this example we use the Win32 platform)
#define GLFW_EXPOSE_NATIVE_WIN32
#include <GLFW/glfw3.h>
#include <GLFW/glfw3native.h>

// Include LLGL and also the native handle structures
#include <LLGL/LLGL.h>
#include <LLGL/Platform/NativeHandle.h>
```

Now we declare our custom surface class and override all necessary interface functions. We could also inherit from the Window interface but this is not really meaningful here:

```
class CustomSurface : public LLGL::Surface {
public:
    // Constructor and destructor
    CustomSurface(const LLGL::Extent2D& size, const char* title);
    ~CustomSurface();

    // Interface implementation
    void GetNativeHandle(void* nativeHandle) const override;
    LLGL::Extent2D GetContentSize() const override;
    bool AdaptForVideoMode(LLGL::Extent2D* resolution, bool* fullscreen) override;
    void ResetPixelFormat() override;
    LLGL::Display* FindResidentDisplay() const override;

    // Additional class functions
    void PollEvents();

private:
    GLFWwindow* CreateGLFWWindow();

    std::string title_;
    LLGL::Extent2D size_;
    GLFWwindow* wnd_ = nullptr; // GLFW window pointer
};
```

This is an example of a minimal interface implementation. We start implementing the constructor, destructor, and the CreateGLFWWindow function:

```
CustomSurface::CustomSurface(const LLGL::Extent2D& size, const char* title) :
    title_ { title },
    size_ { size },
    wnd_ { CreateGLFWWindow() }
{
}

CustomSurface::~CustomSurface() {
    // Destroy GLFW window
    glfwDestroyWindow(wnd_);
}

GLFWwindow* CustomSurface::CreateGLFWWindow() {
    // Create GLFW window with class members
    auto wnd = glfwCreateWindow(size_.width, size_.height, title_.c_str(), nullptr, nullptr);
    if (!wnd) {
        throw std::runtime_error("failed to create GLFW window");
    }
    return wnd;
}
```

Now we cross over to the major functions in the Surface interface. The first one is ResetPixelFormat. It is mainly used by the OpenGL renderer when a multi-sampled render context is created on Win32. In our custom class we

merely destroy the GLFW window and create a new one. It's the only way on Win32 to reset the internal pixel format of the native window object:

```
void CustomSurface::ResetPixelFormat() {  
    // Destroy and recreate GLFW window  
    glfwDestroyWindow(wnd_);  
    wnd_ = CreateGLFWWindow();  
}
```

Next we implement the `GetNativeHandle` function. This is also very important, because any renderer needs access to the native window handle. On MS/Windows this is from the type `HWND` from the Win32 API. On GNU/Linux we need to pass three parameters: `Display*` from the X11 lib, `::Window` from the X11 lib (unfortunately the same name but in the global scope), and `XVisualInfo*` from GLX. On MacOS it is from the type `NSWindow*` from the Cocoa API. In this example we only implement it for Win32:

```
void CustomSurface::GetNativeHandle(void* nativeHandle) const {  
    // This function must always return a valid native handle!  
    auto handle = reinterpret_cast<LLGL::NativeHandle*>(nativeHandle);  
    handle->window = glfwGetWin32Window(wnd_);  
}
```

There are two (rather secondary) interface functions left. They are used inside the `RenderContext` class, to update or query the video mode suitable for the surface:

```
LLGL::Extent2D CustomSurface::GetContentSize() const {  
    // Actually the client-area size of the window must be returned,  
    // but for this example the entire window size is sufficient.  
    return size_;  
}  
  
bool CustomSurface::AdaptForVideoMode(LLGL::Extent2D* resolution, bool* fullscreen) {  
    // Resize GLFW window for the new video mode resolution.  
    size_ = resolution;  
    glfwSetWindowSize(wnd_, size_.width, size_.height);  
    return true;  
}  
  
LLGL::Display* CustomSurface::FindResidentDisplay() const {  
    // Just return primary display  
    return Display::GetPrimary();  
}
```

The last function we have to implement is the additional class function `PollEvents`. Here we poll all window events from GLFW and return false when the user clicked the close button to terminate the application:

```
bool CustomSurface::PollEvents() {  
    // Poll events from GLFW windowing system  
    glfwPollEvents();  
  
    // Return true until the user pressed the close button  
    return !glfwWindowShouldClose(wnd_);  
}
```

Finally we are done with our custom window class :-). Now we can start using it with a render context in our main function:

```
// Initialize GLFW  
if (!glfwInit()) {  
    return -1;  
}  
  
// Create an instance of our custom window class  
auto mySurface = std::make_shared<CustomSurface>({ 640, 480 }, L"LLGL test with GLFW");  
  
// Create render context and pass the custom window  
LLGL::SwapChainDescriptor mySwapChainDesc;  
mySwapChainDesc.resolution = { 640, 480 };  
LLGL::SwapChain* mySwapChain = myRenderer->CreateSwapChain(contextDesc, surface);  
  
// Scene construction ...  
  
while (mySurface->PollEvents()) { /* Rendering ... */ }
```

That's all folks. The rest can be seen in the tutorials.

8 Custom Render System

This is only a very short explanation on how to implement your own render system with LLGL, since it will take a lot of familiarization time anyways.

First of all you have to extend the `CMakeLists.txt` file in your local LLGL repository for a new library. Here is an example on how it could look for a custom OpenGL renderer:

```
# Find OpenGL headers and libraries
FIND_PACKAGE(OpenGL)

IF(OpenGL_FOUND)
    # Add OpenGL include path
    INCLUDE_DIRECTORIES(${OPENGL_INCLUDE_DIR})

    # Add "LLGL_CustomOpenGL" library project to the solution,
    # where "FilesGL" denotes the variable containing all filenames for this render system.
    # The render system can later be loaded with the name "CustomOpenGL".
    ADD_LIBRARY(LLGL_CustomOpenGL SHARED "${FilesGL}")

    # Add "D" postfix to debug mode (this is required so that LLGL loads the correct module)
    SET_TARGET_PROPERTIES(
        LLGL_CustomOpenGL PROPERTIES
        LINKER_LANGUAGE CXX # CXX is for C++
        DEBUG_POSTFIX "D" # Postfix "D" for debug mode
    )

    # Add library dependencies (On Win32 this is LLGL.lib and opengl32.lib)
    TARGET_LINK_LIBRARIES(LLGL_CustomOpenGL LLGL "${OPENGL_LIBRARIES}")

    # Tell CMake to build a project with C++11 support
    TARGET_COMPILE_FEATURES(LLGL_CustomOpenGL PRIVATE cxx_range_for)
ELSE()
    MESSAGE("Missing OpenGL -> LLGL_CustomOpenGL renderer will be excluded from project")
ENDIF()
```

After you integrated your source files into the LLGL project solution, you have to implement a few module interface functions. They are the only functions that need to be exported from the shared library, and they need to be exported as extern "C" functions:

```
// Include the custom render system
#include "CustomOpenGLRenderSystem.h"

// Declare the following functions to be exported as "C" functions
extern "C" {

    // This function must always return 'LLGL_BUILD_ID' to ensure the render system,
    // when loaded from a module, has been compiled with the same compiler toolchain.
    LLGL_EXPORT int LLGL_RenderSystem_BuildID() {
        return LLGL_BUILD_ID;
    }

    // Here we can return our own renderer ID or one of the pre-defined values.
    LLGL_EXPORT int LLGL_RenderSystem_RendererID() {
        return LLGL::RendererID::OpenGL;
    }

    // Here we can return the name of our renderer.
    // This does not have to match the name of the module.
    LLGL_EXPORT const char* LLGL_RenderSystem_Name() {
        return "Custom OpenGL";
    }

    // Here we have to return a pointer to our custom implementation of the RenderSystem interface.
    // If we don't allocate it via 'new' operator, LLGL_RenderSystem_Free must also be provided (see next).
    // 'renderSystemDesc' can be casted to 'LLGL::RenderSystemDescriptor'.
    LLGL_EXPORT void* LLGL_RenderSystem_Alloc(const void* renderSystemDesc, int renderSystemDescSize) {
        return new CustomOpenGLRenderSystem();
    }

    // The deleter function is only required if our render system was *not* allocated via 'new' operator.
    LLGL_EXPORT void LLGL_RenderSystem_Free(void* renderSystem) {
        delete reinterpret_cast<LLGL::RenderSystem*>(renderSystem);
    }

} // end of extern "C"
```

That's all you have to do, except of implementing the entire `RenderSystem` interface in your `CustomOpenGLRenderSystem` class ;-). After that, you can load your render system like this:

```
// Load custom render system (library name is "LLGL_CustomOpenGL", so the passed name is "CustomOpenGL")
LLGL::RenderSystemPtr renderer = LLGL::RenderSystem::Load("CustomOpenGL");

// Start using the custom renderer ...
```