Getting started

Table of Contents

This guide takes you through writing a simple application using GLFW 3. The application will create a window and OpenGL context, render a rotating triangle and exit when the user closes the window or presses *Escape*. This guide will introduce a few of the most commonly used functions, but there are many more.

This guide assumes no experience with earlier versions of GLFW. If you have used GLFW 2 in the past, read **Moving from GLFW 2 to 3**, as some functions behave differently in GLFW 3.

Step by step

Including the GLFW header

In the source files of your application where you use OpenGL or GLFW, you need to include the GLFW 3 header file.

```
#include <GLFW/glfw3.h>
```

This defines all the constants, types and function prototypes of the GLFW API. It also includes the OpenGL header from your development environment and defines all the constants and types necessary for it to work on your platform without including any platform-specific headers.

In other words:

- Do *not* include the OpenGL header yourself, as GLFW does this for you in a platform-independent way
- Do *not* include `windows.h` or other platform-specific headers unless you plan on using those APIs yourself

- If you *do* need to include such headers, include them *before* the GLFW header and it will detect this

On some platforms supported by GLFW the OpenGL header and link library only expose older versions of OpenGL. The most extreme case is Windows, which only exposes OpenGL 1.2. The easiest way to work around this is to use an **extension loader library**.

If you are using such a library then you should include its header *before* the GLFW header. This lets it replace the OpenGL header included by GLFW without conflicts. This example uses glad, but the same rule applies to all such libraries.

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
```

Initializing and terminating GLFW

Before you can use most GLFW functions, the library must be initialized. On successful initialization, `GLFW_TRUE` is returned. If an error occurred, `GLFW_FALSE` is returned.

```
if (!glfwInit())
{
    // Initialization failed
}
```

Note that `GLFW_TRUE` and `GLFW_FALSE` are and will always be just one and zero.

When you are done using GLFW, typically just before the application exits, you need to terminate GLFW.

```
glfwTerminate();
```

This destroys any remaining windows and releases any other resources allocated by GLFW. After this call, you must initialize GLFW again before using any GLFW functions that require it.

Setting an error callback

Most events are reported through callbacks, whether it's a key being pressed, a GLFW window being moved, or an error occurring. Callbacks are simply C functions (or C++ static methods) that are called by GLFW with arguments describing the event.

In case a GLFW function fails, an error is reported to the GLFW error callback. You can receive these reports with an error callback. This function must have the signature below. This simple error callback just prints the error description to `stderr`.

```
void error_callback(int error, const char* description)
{
```

```
    fprintf(stderr, "Error: %s\n", description);
}
```

Callback functions must be set, so GLFW knows to call them. The function to set the error callback is one of the few GLFW functions that may be called before initialization, which lets you be notified of errors both during and after initialization.

```
glfwSetErrorCallback(error_callback);
```

Creating a window and context

The window and its OpenGL context are created with a single call to **glfwCreateWindow**, which returns a handle to the created combined window and context object

```
GLFWwindow* window = glfwCreateWindow(640, 480, "My Title", NULL, NULL);
if (!window)
{
    // Window or OpenGL context creation failed
}
```

This creates a 640 by 480 windowed mode window with an OpenGL context. If window or OpenGL context creation fails, `NULL` will be returned. You should always check the return value. While window creation rarely fails, context creation depends on properly installed drivers and may fail even on machines with the necessary hardware.

By default, the OpenGL context GLFW creates may have any version. You can require a minimum OpenGL version by setting the `GLFW_CONTEXT_VERSION_MAJOR` and `GLFW_CONTEXT_VERSION_MINOR` hints *before* creation. If the required minimum version is not supported on the machine, context (and window) creation fails.

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
GLFWwindow* window = glfwCreateWindow(640, 480, "My Title", NULL, NULL);
if (!window)
{
    // Window or context creation failed
}
```

The window handle is passed to all window related functions and is provided to along to all window related callbacks, so they can tell which window received the event.

When a window and context is no longer needed, destroy it.

```
glfwDestroyWindow(window);
```

Once this function is called, no more events will be delivered for that window and its handle becomes invalid.

Making the OpenGL context current

Before you can use the OpenGL API, you must have a current OpenGL context.

```
glfwMakeContextCurrent(window);
```

The context will remain current until you make another context current or until the window owning the current context is destroyed.

If you are using an **extension loader library** to access modern OpenGL then this is when to initialize it, as the loader needs a current context to load from. This example uses glad, but the same rule applies to all such libraries.

```
gladLoadGLLoader((GLADloadproc) glfwGetProcAddress);
```

Checking the window close flag

Each window has a flag indicating whether the window should be closed.

When the user attempts to close the window, either by pressing the close widget in the title bar or using a key combination like Alt+F4, this flag is set to 1. Note that **the window isn't actually closed**, so you are expected to monitor this flag and either destroy the window or give some kind of feedback to the user.

```
while (!glfwWindowShouldClose(window))
{
    // Keep running
}
```

You can be notified when the user is attempting to close the window by setting a close callback with **glfwSetWindowCloseCallback**. The callback will be called immediately after the close flag has been set.

You can also set it yourself with **glfwSetWindowShouldClose**. This can be useful if you want to interpret other kinds of input as closing the window, like for example pressing the *Escape* key.

Receiving input events

Each window has a large number of callbacks that can be set to receive all the various kinds of events. To receive key press and release events, create a key callback function.

```
static void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
```

```
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GLFW_TRUE);
}
```

The key callback, like other window related callbacks, are set per-window.

```
glfwSetKeyCallback(window, key_callback);
```

In order for event callbacks to be called when events occur, you need to process events as described below.

Rendering with OpenGL

Once you have a current OpenGL context, you can use OpenGL normally. In this tutorial, a multi-colored rotating triangle will be rendered. The framebuffer size needs to be retrieved for `glViewport`.

```
int width, height;
glfwGetFramebufferSize(window, &width, &height);
glViewport(0, 0, width, height);
```

You can also set a framebuffer size callback using **glfwSetFramebufferSizeCallback** and be notified when the size changes.

Actual rendering with OpenGL is outside the scope of this tutorial, but there are many excellent tutorial sites that teach modern OpenGL. Some of them use GLFW to create the context and window while others use GLUT or SDL, but remember that OpenGL itself always works the same.

Reading the timer

To create smooth animation, a time source is needed. GLFW provides a timer that returns the number of seconds since initialization. The time source used is the most accurate on each platform and generally has micro- or nanosecond resolution.

```
double time = glfwGetTime();
```

Swapping buffers

GLFW windows by default use double buffering. That means that each window has two rendering buffers; a front buffer and a back buffer. The front buffer is the one being displayed and the back buffer the one you render to.

When the entire frame has been rendered, the buffers need to be swapped with one another, so the back buffer becomes the front buffer and vice versa.

```
glfwSwapBuffers(window);
```

The swap interval indicates how many frames to wait until swapping the buffers, commonly known as *vsync*. By default, the swap interval is zero, meaning buffer swapping will occur immediately. On fast machines, many of those frames will never be seen, as the screen is still only updated typically 60-75 times per second, so this wastes a lot of CPU and GPU cycles.

Also, because the buffers will be swapped in the middle the screen update, leading to screen tearing.

For these reasons, applications will typically want to set the swap interval to one. It can be set to higher values, but this is usually not recommended, because of the input latency it leads to.

```
glfwSwapInterval(1);
```

This function acts on the current context and will fail unless a context is current.

Processing events

GLFW needs to communicate regularly with the window system both in order to receive events and to show that the application hasn't locked up. Event processing must be done regularly while you have visible windows and is normally done each frame after buffer swapping.

There are two methods for processing pending events; polling and waiting. This example will use event polling, which processes only those events that have already been received and then returns immediately.

```
glfwPollEvents();
```

This is the best choice when rendering continually, like most games do. If instead you only need to update your rendering once you have received new input, **glfwWaitEvents** is a better choice. It waits until at least one event has been received, putting the thread to sleep in the meantime, and then processes all received events. This saves a great deal of CPU cycles and is useful for, for example, many kinds of editing tools.

Putting it together

Now that you know how to initialize GLFW, create a window and poll for keyboard input, it's possible to create a simple program.

This program creates a 640 by 480 windowed mode window and starts a loop that clears the screen, renders a triangle and processes events until the user either presses *Escape* or closes the window.

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>

#include "linmath.h"
```

```c
#include <stdlib.h>
#include <stdio.h>

static const struct
{
    float x, y;
    float r, g, b;
} vertices[3] =
{
    { -0.6f, -0.4f, 1.f, 0.f, 0.f },
    {  0.6f, -0.4f, 0.f, 1.f, 0.f },
    {   0.f,  0.6f, 0.f, 0.f, 1.f }
};

static const char* vertex_shader_text =
"uniform mat4 MVP;\n"
"attribute vec3 vCol;\n"
"attribute vec2 vPos;\n"
"varying vec3 color;\n"
"void main()\n"
"{\n"
"    gl_Position = MVP * vec4(vPos, 0.0, 1.0);\n"
"    color = vCol;\n"
"}\n";

static const char* fragment_shader_text =
"varying vec3 color;\n"
"void main()\n"
"{\n"
"    gl_FragColor = vec4(color, 1.0);\n"
"}\n";

static void error_callback(int error, const char* description)
{
    fprintf(stderr, "Error: %s\n", description);
}

static void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GLFW_TRUE);
}

int main(void)
{
    GLFWwindow* window;
    GLuint vertex_buffer, vertex_shader, fragment_shader, program;
```

```
    GLint mvp_location, vpos_location, vcol_location;

    glfwSetErrorCallback(error_callback);

    if (!glfwInit())
        exit(EXIT_FAILURE);

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);

    window = glfwCreateWindow(640, 480, "Simple example", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        exit(EXIT_FAILURE);
    }

    glfwSetKeyCallback(window, key_callback);

    glfwMakeContextCurrent(window);
    gladLoadGLLoader((GLADloadproc) glfwGetProcAddress);
    glfwSwapInterval(1);

    // NOTE: OpenGL error checks have been omitted for brevity

    glGenBuffers(1, &vertex_buffer);
    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    vertex_shader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex_shader, 1, &vertex_shader_text, NULL);
    glCompileShader(vertex_shader);

    fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment_shader, 1, &fragment_shader_text, NULL);
    glCompileShader(fragment_shader);

    program = glCreateProgram();
    glAttachShader(program, vertex_shader);
    glAttachShader(program, fragment_shader);
    glLinkProgram(program);

    mvp_location = glGetUniformLocation(program, "MVP");
    vpos_location = glGetAttribLocation(program, "vPos");
    vcol_location = glGetAttribLocation(program, "vCol");

    glEnableVertexAttribArray(vpos_location);
```

```
    glVertexAttribPointer(vpos_location, 2, GL_FLOAT, GL_FALSE,
                          sizeof(float) * 5, (void*) 0);
    glEnableVertexAttribArray(vcol_location);
    glVertexAttribPointer(vcol_location, 3, GL_FLOAT, GL_FALSE,
                          sizeof(float) * 5, (void*) (sizeof(float) * 2));

    while (!glfwWindowShouldClose(window))
    {
        float ratio;
        int width, height;
        mat4x4 m, p, mvp;

        glfwGetFramebufferSize(window, &width, &height);
        ratio = width / (float) height;

        glViewport(0, 0, width, height);
        glClear(GL_COLOR_BUFFER_BIT);

        mat4x4_identity(m);
        mat4x4_rotate_Z(m, m, (float) glfwGetTime());
        mat4x4_ortho(p, -ratio, ratio, -1.f, 1.f, 1.f, -1.f);
        mat4x4_mul(mvp, p, m);

        glUseProgram(program);
        glUniformMatrix4fv(mvp_location, 1, GL_FALSE, (const GLfloat*) mvp);
        glDrawArrays(GL_TRIANGLES, 0, 3);

        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwDestroyWindow(window);

    glfwTerminate();
    exit(EXIT_SUCCESS);
}
```

The program above can be found in the source package as `examples/simple.c` and is compiled along with all other examples when you build GLFW. If you built GLFW from the source package then already have this as `simple.exe` on Windows, `simple` on Linux or `simple.app` on OS X.

This tutorial used only a few of the many functions GLFW provides. There are guides for each of the areas covered by GLFW. Each guide will introduce all the functions for that category.

- **Introduction to the API**
- **Window guide**

- **Context guide**
- **Monitor guide**
- **Input guide**

You can access reference documentation for any GLFW function by clicking it and the reference for each function links to related functions and guide sections.

The tutorial ends here. Once you have written a program that uses GLFW, you will need to compile and link it. How to do that depends on the development environment you are using and is best explained by the documentation for that environment. To learn about the details that are specific to GLFW, see **Building applications**.

Last update on Thu Aug 18 2016 for GLFW 3.2.1