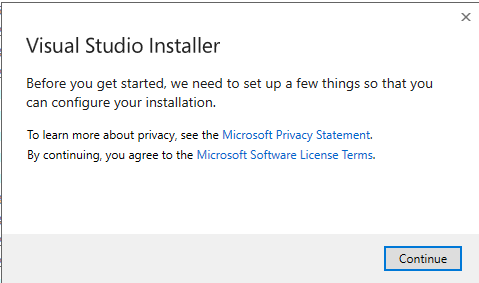
**OpenGL Environment Setup**

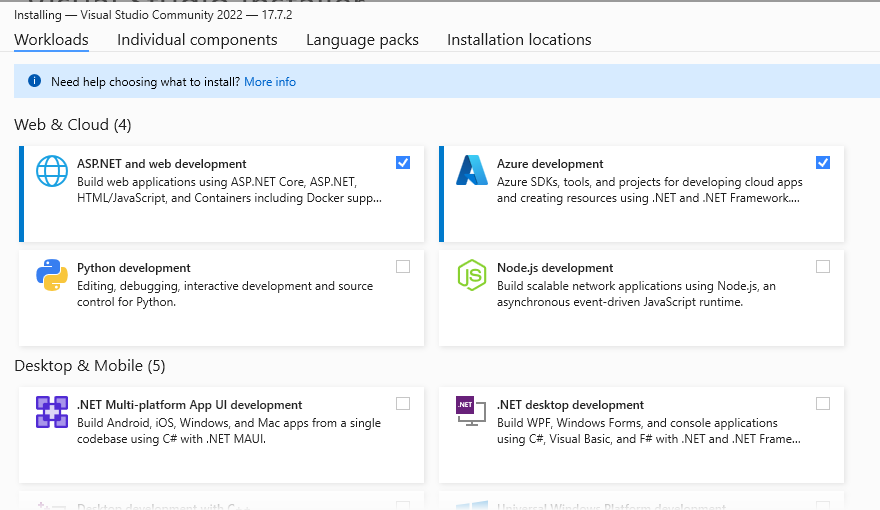
# Install Visual Studio Code

Step -1 : Double click on the VisualStudioSetup.exe file to install visual studio community 2022.

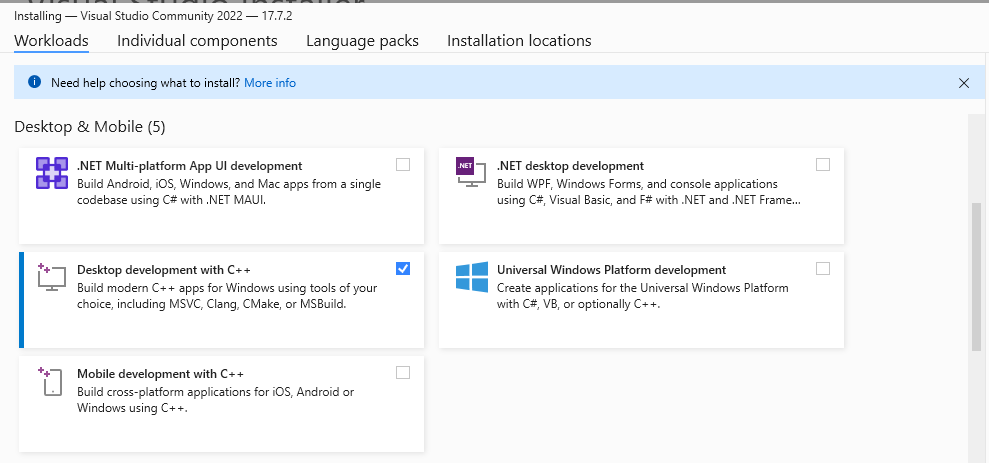
Step -2:



Click “Continue”.

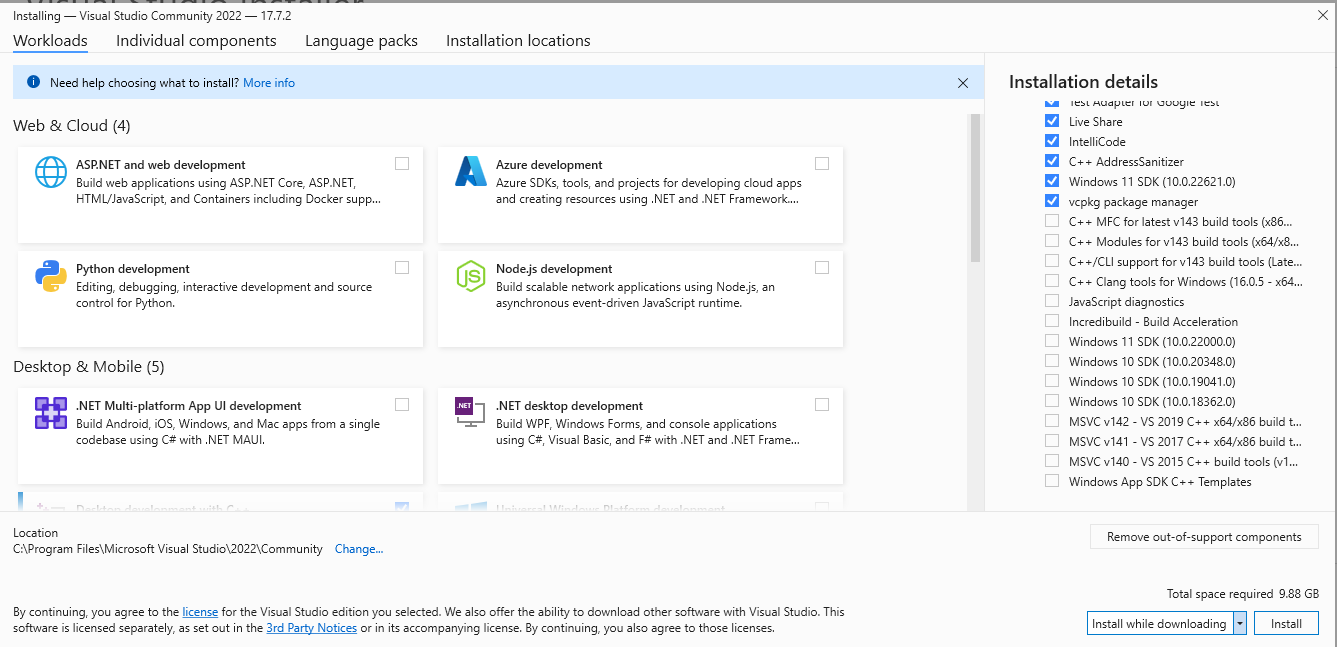
Step – 3: 

Uncheck ASP.NET and web development, Azure development.

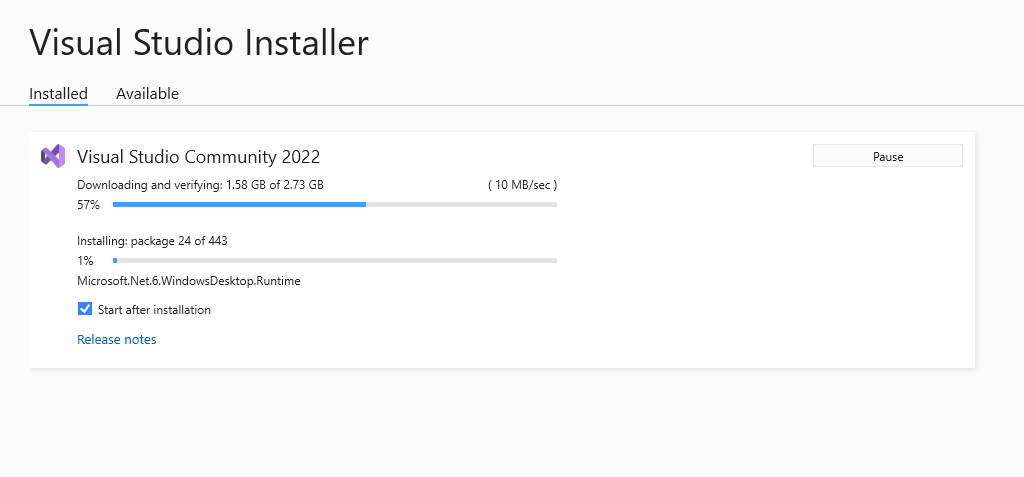


Check Desktop development with C++.

Step -4:

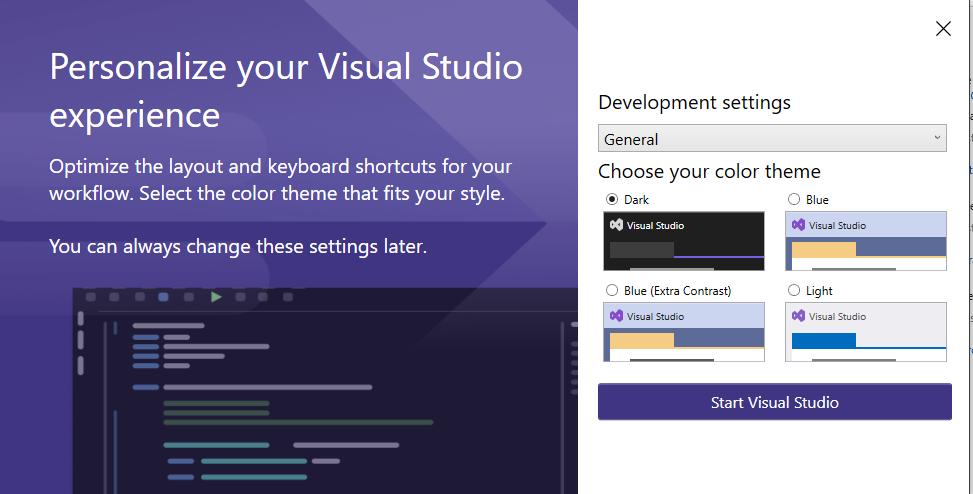


Click on the “Install” button.



Installation is going on………

Step -5:

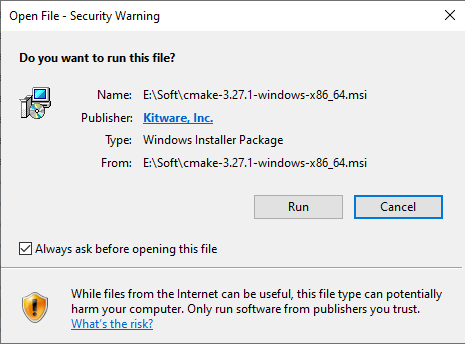


Choose your favorite color theme and click on “Start Visual Studio”.

# Install CMake

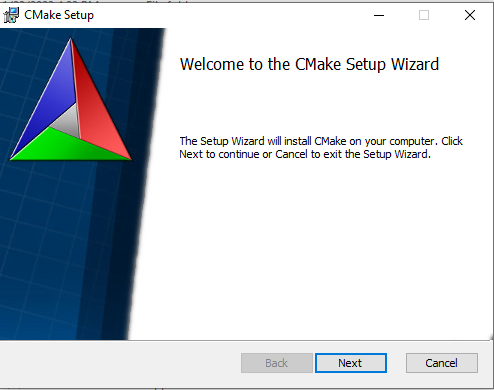
Step – 1: Double click on cmake-3.27.1-windows-x86\_64.msi.

Step -2:



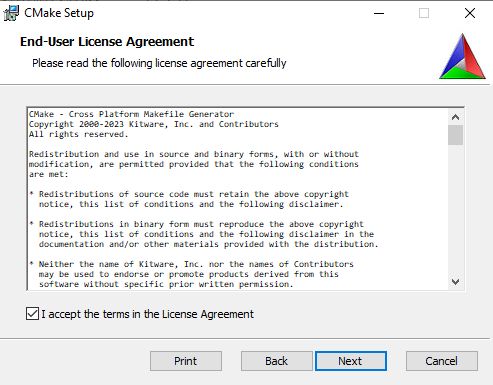
Click on “Run”.

Step -3:



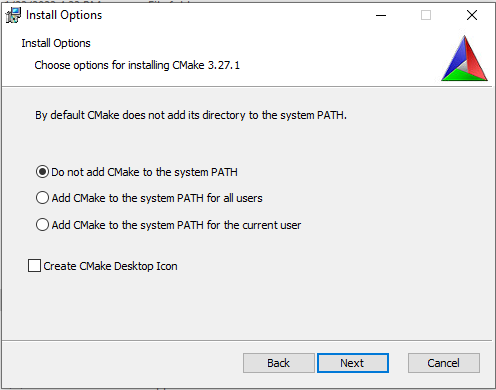
Click on “Next”.

Step – 4:



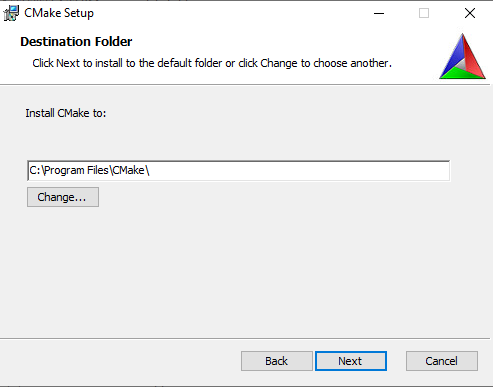
Accept the terms and license Agreement and click on “Next”.

Step – 5:



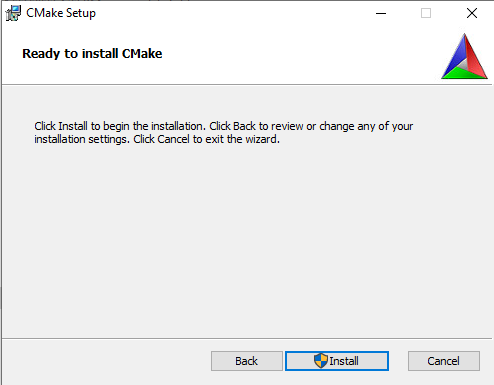
Leave everything as default and click on “Next”.

Step – 6:



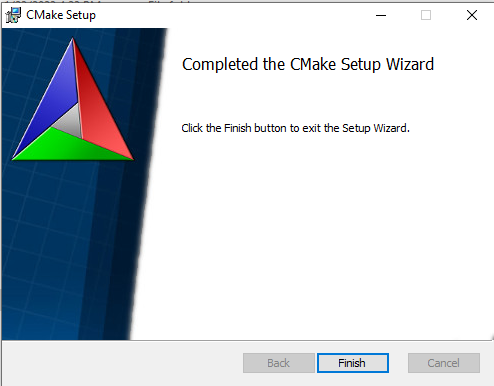
Click on “Next”.

Step – 7:



Click on “Install”.

Step -8:



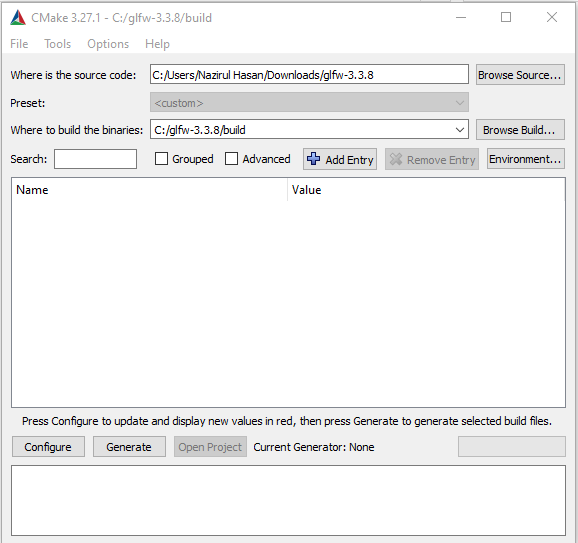
After the installation is over click on the “Finish” button.

# Install GLFW

Step – 1: Go to <https://www.glfw.org/download.html> and download the “source package”.

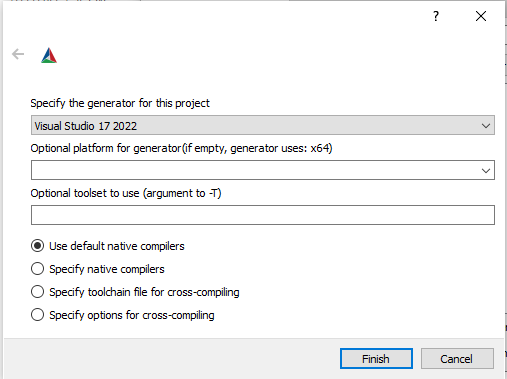
Step – 2: Unzip the downloaded file.

Step – 3:



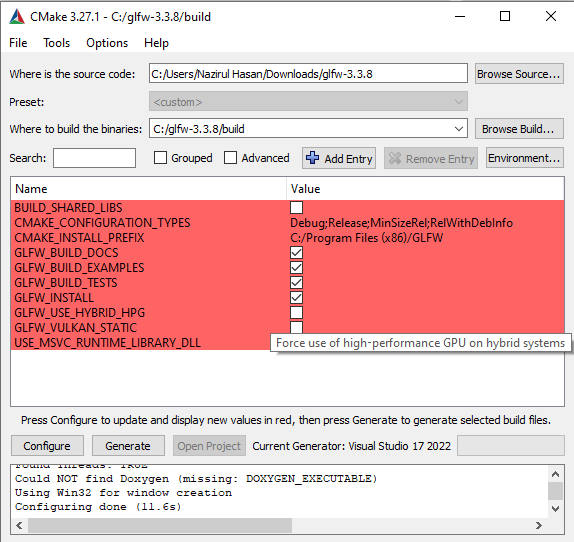
Open CMake. For the source code folder we’re going to choose the root folder of the downloaded GLFW source package and for the build folder we’re creating a new directory build and then select that directory. Once the source and destination folders have been set, click the Configure button.

Step – 4:



Choose visual Studio 17 2022. Leave everything as default and click “Finish”.

Step – 5:



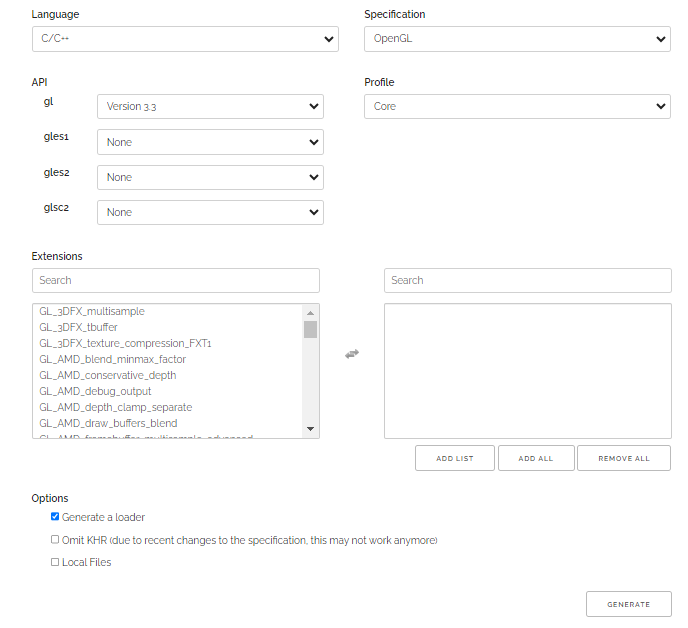
CMake will then display the possible build options to configure the resulting library. We can leave them to their default values and click Configure again to store the settings. Once the settings have been set, we click Generate and the resulting project files will be generated in your build folder.

Step -6: In the build folder a file named GLFW.sln can now be found and we open it with Visual Studio 2022. Now hit build solution. This will give us a compiled library file that can be found in build/src/Debug named glfw3.lib.

Step -7: Create a directory to store the header and library files. I named it “opengl”. Inside opengl, I created two more directories named “Include” and “Lib”. Copy the glfw3.lib file in this Lib folder. After that go to your downloaded glfw directory. Then go to the “include” folder and copy the “GLFW” folder. Now, go to “opengl/Include” directory and paste “GLFW” there.

# Install GLAD

Step – 1: Go to <https://glad.dav1d.de/> .

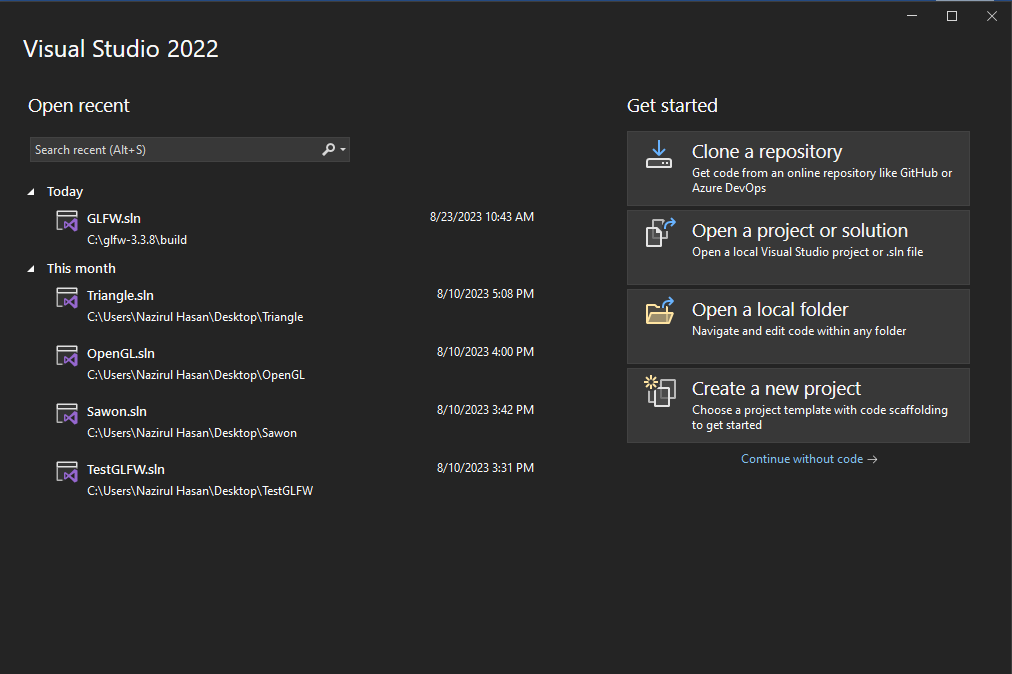


Select language as “c++”, specification “opengl”, opengl version “3.3” and profile “core”. Make sure that generate loader is checked. Now Click “GENERATE” button and download the glad.zip file.

Step – 2: Unzip the glad.zip file. Open it and copy the folders inside “include” file. Now, paste those folders inside “opengl/Include” directory. Again, open the “glad” folder. Inside “src” folder you will find glad.c file. Copy it and paste inside opengl folder.

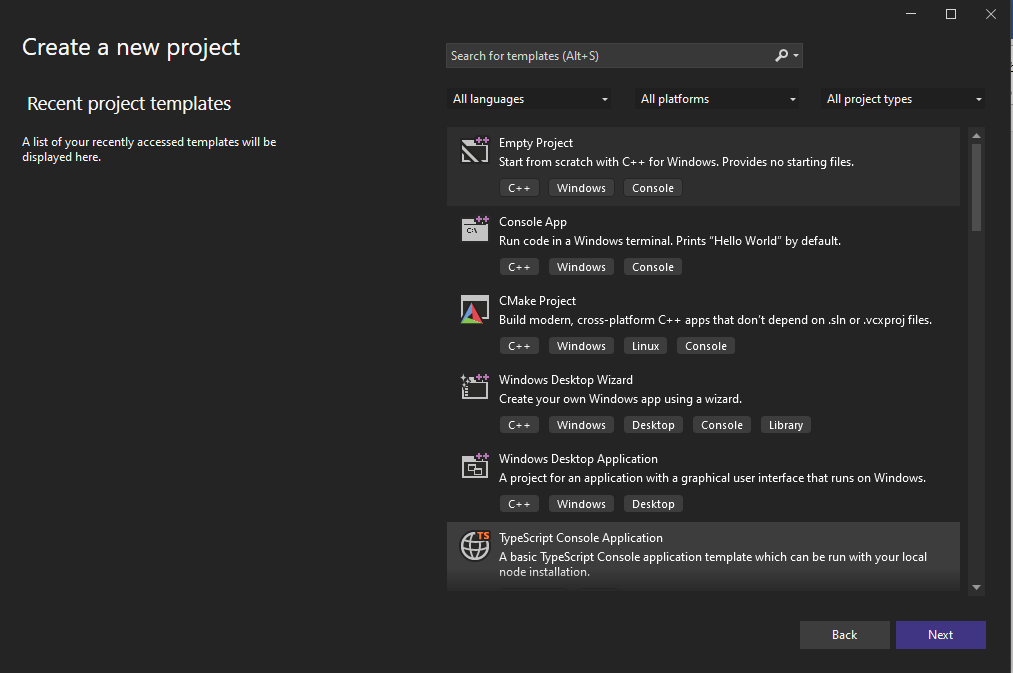
# Linking

Step – 1:



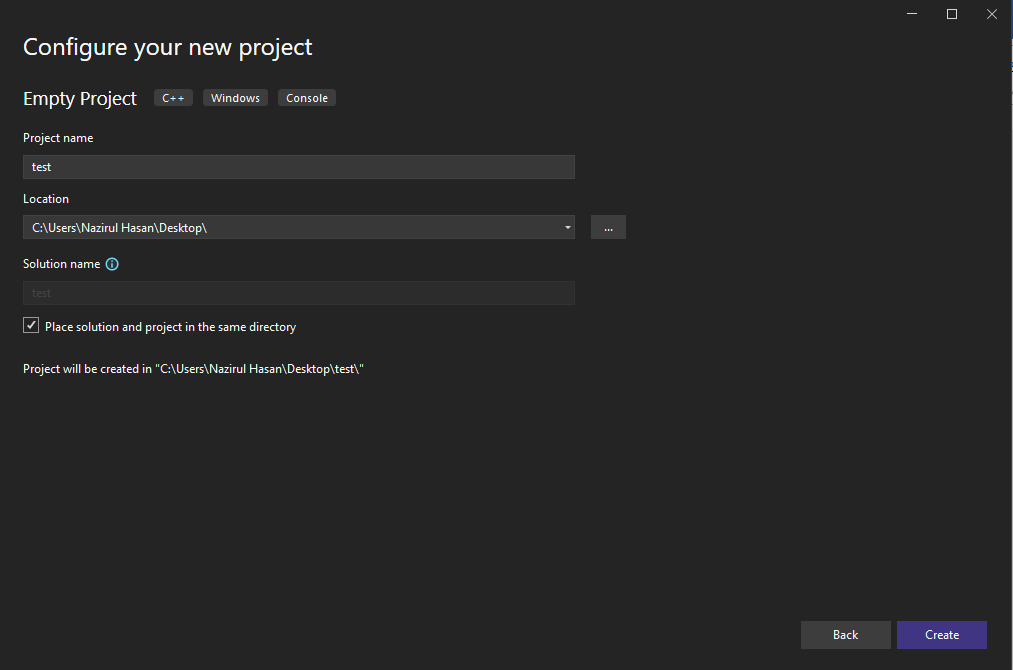
Open visual studio 2022 and click “Create a new project”.

Step -2:



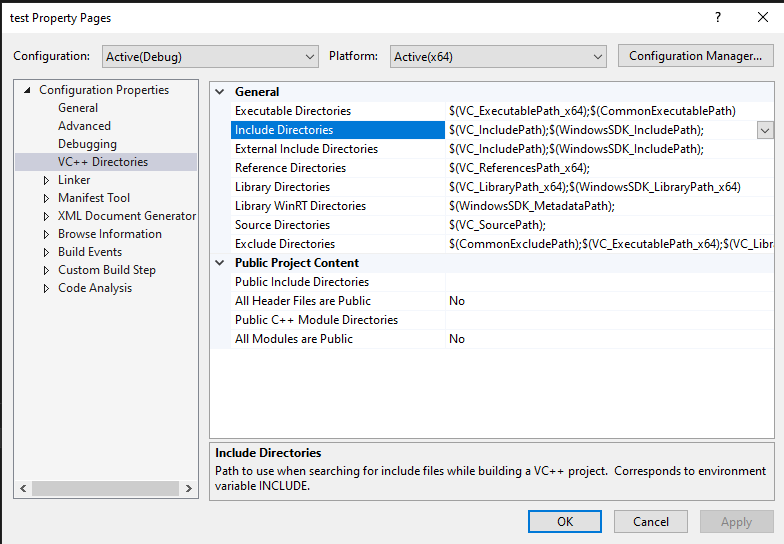
Select “Empty project” and then click “Next”.

Step – 3:

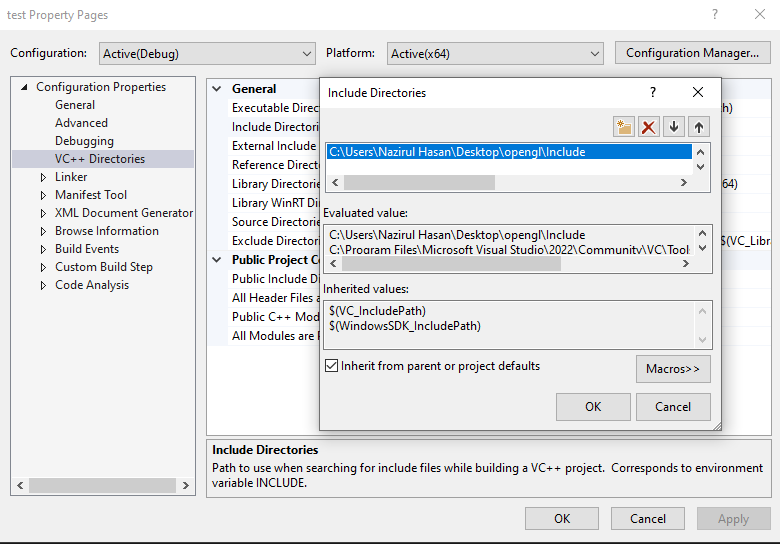


Give a name of your project. Select the location where you want to save it and then click “Create”.

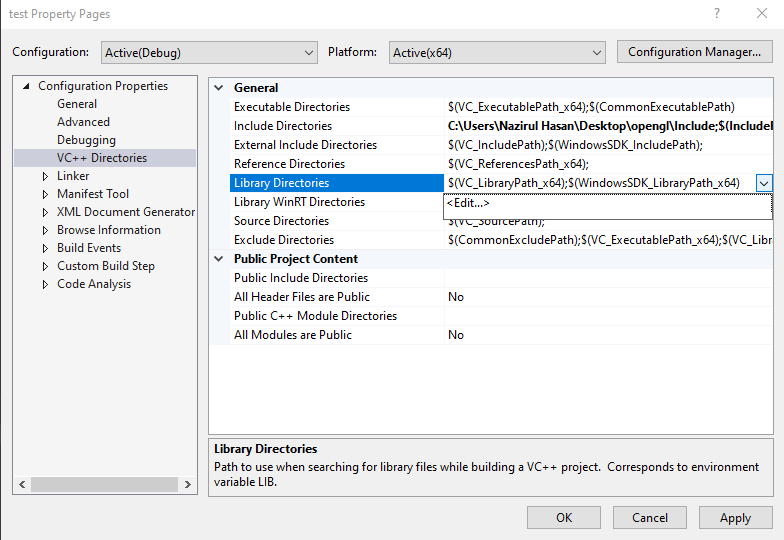
Step – 4: Right-click the project name in the solution explorer and then go to properties. Select VC++ Directories as seen in the image below:

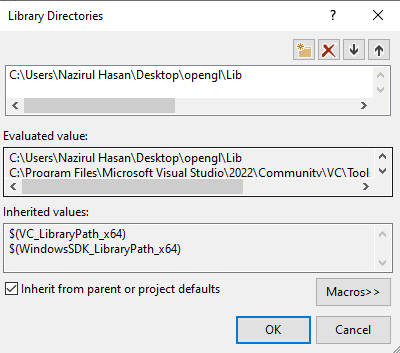


From there on out you can add your own directories to let the project know where to search.This can be done by manually inserting it into the text or clicking the appropriate location string and selecting the <Edit..> option. Do this for both the Library Directories and Include Directories:

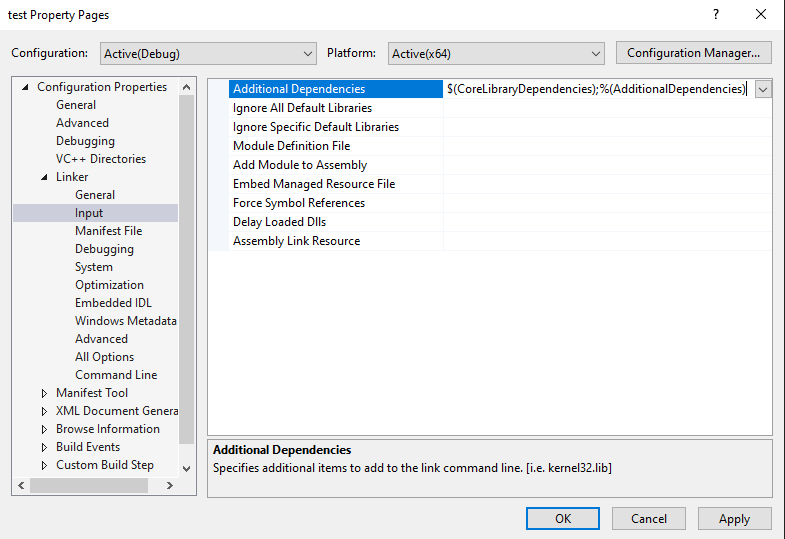


For Include Directories, paste the path of the “Include” folder which is inside your “opengl” folder. For Library Directories, paste the path of the “Lib” folder which is inside your “opengl” folder.

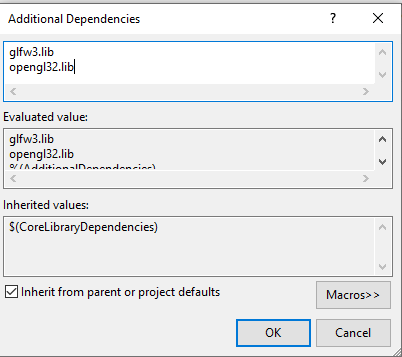




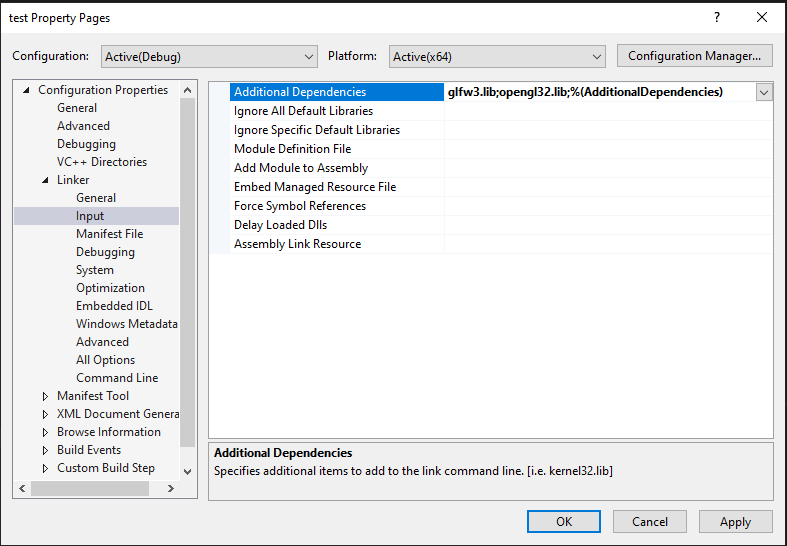
Since VS can now find all the required files we can finally link GLFW to the project by going to the Linker tab and Input:



To then link to a library you’d have to specify the name of the library to the linker. Since the library name is glfw3.lib, we add that to the Additional Dependencies field (either manually or using the <Edit..> option) and from that point on GLFW will be linked when we compile. In addition to GLFW we should also add a link entry to the OpenGL library, but this may differ per operating system.

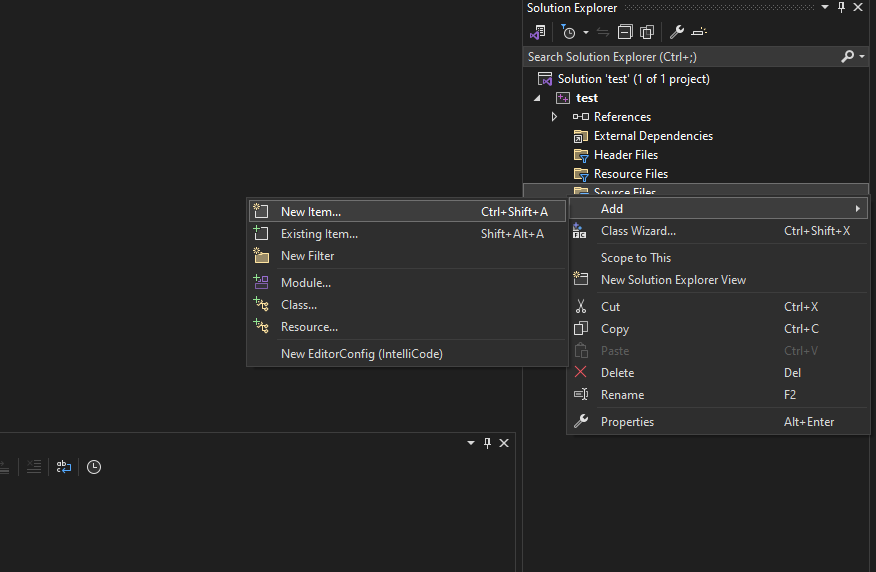


Click “OK”.



Now, Click “Apply”.

Step – 5 (Lab Task):



Now, In your project click on “Source Files”, the click on “Add” and click on “New item” to create a new cpp file. Paste the following code inside the file.

#include <glad/glad.h>

#include <GLFW/glfw3.h>

#include <iostream>

void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height);

void processInput(GLFWwindow\* window);

// settings

const unsigned int SCR\_WIDTH = 800;

const unsigned int SCR\_HEIGHT = 600;

const char\* vertexShaderSource = "#version 330 core\n"

"layout (location = 0) in vec3 aPos;\n"

"void main()\n"

"{\n"

" gl\_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"

"}\0";

const char\* fragmentShaderSource = "#version 330 core\n"

"out vec4 FragColor;\n"

"void main()\n"

"{\n"

" FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"

"}\n\0";

int main()

{

// glfw: initialize and configure

// ------------------------------

glfwInit();

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

#ifdef \_\_APPLE\_\_

glfwWindowHint(GLFW\_OPENGL\_FORWARD\_COMPAT, GL\_TRUE);

#endif

// glfw window creation

// --------------------

GLFWwindow\* window = glfwCreateWindow(SCR\_WIDTH, SCR\_HEIGHT, "LearnOpenGL", NULL, NULL);

if (window == NULL)

{

std::cout << "Failed to create GLFW window" << std::endl;

glfwTerminate();

return -1;

}

glfwMakeContextCurrent(window);

glfwSetFramebufferSizeCallback(window, framebuffer\_size\_callback);

// glad: load all OpenGL function pointers

// ---------------------------------------

if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))

{

std::cout << "Failed to initialize GLAD" << std::endl;

return -1;

}

// build and compile our shader program

// ------------------------------------

// vertex shader

unsigned int vertexShader = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);

glCompileShader(vertexShader);

// check for shader compile errors

int success;

char infoLog[512];

glGetShaderiv(vertexShader, GL\_COMPILE\_STATUS, &success);

if (!success)

{

glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);

std::cout << "ERROR::SHADER::VERTEX::COMPILATION\_FAILED\n" << infoLog << std::endl;

}

// fragment shader

unsigned int fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);

glCompileShader(fragmentShader);

// check for shader compile errors

glGetShaderiv(fragmentShader, GL\_COMPILE\_STATUS, &success);

if (!success)

{

glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);

std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION\_FAILED\n" << infoLog << std::endl;

}

// link shaders

unsigned int shaderProgram = glCreateProgram();

glAttachShader(shaderProgram, vertexShader);

glAttachShader(shaderProgram, fragmentShader);

glLinkProgram(shaderProgram);

// check for linking errors

glGetProgramiv(shaderProgram, GL\_LINK\_STATUS, &success);

if (!success) {

glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);

std::cout << "ERROR::SHADER::PROGRAM::LINKING\_FAILED\n" << infoLog << std::endl;

}

glDeleteShader(vertexShader);

glDeleteShader(fragmentShader);

// set up vertex data (and buffer(s)) and configure vertex attributes

// ------------------------------------------------------------------

float vertices[] = {

-0.5f, -0.5f, 0.0f, // left

0.5f, -0.5f, 0.0f, // right

0.0f, 0.5f, 0.0f // top

};

unsigned int VBO, VAO;

glGenVertexArrays(1, &VAO);

glGenBuffers(1, &VBO);

// bind the Vertex Array Object first, then bind and set vertex buffer(s), and then configure vertex attributes(s).

glBindVertexArray(VAO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);

glEnableVertexAttribArray(0);

// note that this is allowed, the call to glVertexAttribPointer registered VBO as the vertex attribute's bound vertex buffer object so afterwards we can safely unbind

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

// You can unbind the VAO afterwards so other VAO calls won't accidentally modify this VAO, but this rarely happens. Modifying other

// VAOs requires a call to glBindVertexArray anyways so we generally don't unbind VAOs (nor VBOs) when it's not directly necessary.

glBindVertexArray(0);

// uncomment this call to draw in wireframe polygons.

//glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

// render loop

// -----------

while (!glfwWindowShouldClose(window))

{

// input

// -----

processInput(window);

// render

// ------

glClearColor(0.2f, 0.3f, 0.3f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT);

// draw our first triangle

glUseProgram(shaderProgram);

glBindVertexArray(VAO); // seeing as we only have a single VAO there's no need to bind it every time, but we'll do so to keep things a bit more organized

glDrawArrays(GL\_TRIANGLES, 0, 3);

// glBindVertexArray(0); // no need to unbind it every time

// glfw: swap buffers and poll IO events (keys pressed/released, mouse moved etc.)

// -------------------------------------------------------------------------------

glfwSwapBuffers(window);

glfwPollEvents();

}

// optional: de-allocate all resources once they've outlived their purpose:

// ------------------------------------------------------------------------

glDeleteVertexArrays(1, &VAO);

glDeleteBuffers(1, &VBO);

glDeleteProgram(shaderProgram);

// glfw: terminate, clearing all previously allocated GLFW resources.

// ------------------------------------------------------------------

glfwTerminate();

return 0;

}

// process all input: query GLFW whether relevant keys are pressed/released this frame and react accordingly

// ---------------------------------------------------------------------------------------------------------

void processInput(GLFWwindow\* window)

{

if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)

glfwSetWindowShouldClose(window, true);

}

// glfw: whenever the window size changed (by OS or user resize) this callback function executes

// ---------------------------------------------------------------------------------------------

void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height)

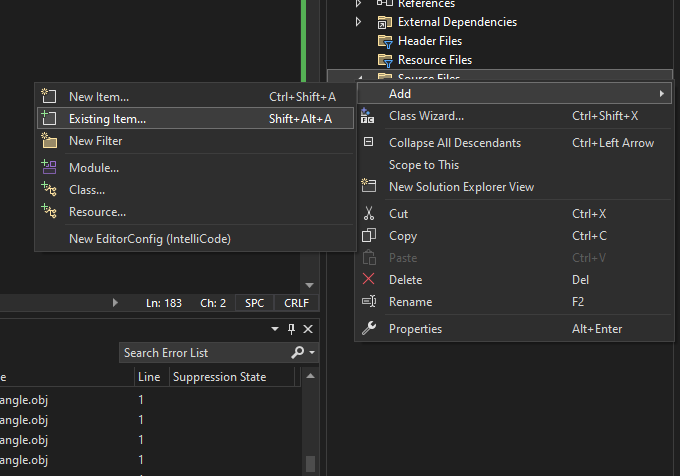
{

// make sure the viewport matches the new window dimensions; note that width and

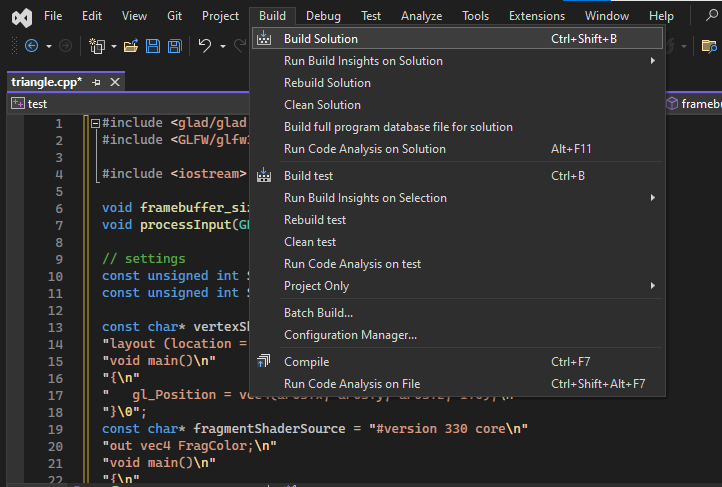
// height will be significantly larger than specified on retina displays.

glViewport(0, 0, width, height);

}



Again, In your project click on “Source Files”, the click on “Add” and click on “Existing item” to add the glad.c file.



Now, Build the solution and run it. If you see the below image as output you are good to go.

