Hw1

作業環境:

Windows 10 / Visual Studio 2022

library	Version	Download link
glfw	3.3.6	https://github.com/JoeyDeVries/LearnOpenGL
glew	3.2.4	
KHR	32517	
GLM	0.9.9.3	
stb	v2.14	
glad	gl_verison_4_5	
assimp	3.3.0	
learnopengl	Aug 5, 2024	
mINI	V0.9.17	https://github.com/metayeti/mINI

Feature 說明：

1. 讀取 ini 檔：
2. 鍵盤控制鏡頭 (WASD)
3. 鍵盤關閉視窗(ESC)
4. 視窗改變自動更新長寬：
5. 滑鼠停用游標模式，可以自由的滑動不受視窗大小限制
6. 滑鼠滑動，改變視角
7. 滾輪滾動，縮放視野
8. 按 H，使用或取消線條模式。
9. 從 ini 檔，讀取 model path 並更換 model
10. 從 ini 檔，讀取 texture path 並綁定紋理到 model 上

文件說明：

base.h

鍵盤控制鏡頭：

1. ESC 鍵：關閉視窗
2. WSAD 鍵：控制相機位置
3. H 鍵：啟用線條模式、填充模式

```
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRE
        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.ProcessKeyboard(FORWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.ProcessKeyboard(BACKWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.ProcessKeyboard(LEFT, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.ProcessKeyboard(RIGHT, deltaTime);


    if (glfwGetKey(window, GLFW_KEY_H) == GLFW_PRESS){
        wireframeEnabled++;
        if ((wireframeEnabled) / 6 % 2){
            glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        }
        else{
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
        }
    }
}
```

4. 視窗改變自動更新長寬：

```
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);
}
```

5. 滑鼠滑動，改變視角

```cpp
// glfw: whenever the mouse moves, this callback is called
// -------------------------------------------------------
void mouse_callback(GLFWwindow* window, double xposIn, double yposIn)
{
    float xpos = static_cast<float>(xposIn);
    float ypos = static_cast<float>(yposIn);

    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-coordinates go from bottom to top

    lastX = xpos;
    lastY = ypos;

    camera.ProcessMouseMovement(xoffset, yoffset);
}
```

6. 滾輪滾動，縮放視野

```cpp
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    camera.ProcessMouseScroll(static_cast<float>(yoffset));
}
```

7. 讀取紋理

```cpp
// Load texture function
unsigned int loadTexture(const char* path) {
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);

    // Set texture wrapping parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    // Set texture filtering parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    // Load and generate the texture
    int width, height, nrChannels;
    unsigned char* data = stbi_load(path, &width, &height, &nrChannels, 0);
    if (data) {
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
    } else {
        std::cout << "Failed to load texture" << std::endl;
    }
    stbi_image_free(data);

    return textureID;
}
```

learnopengl/model.h

Model class 是來自於 <learnopengl/model.h> ，constructor 輸入 path string，加載 3D 模型，使用了 Assimp（Open Asset Import Library）庫來解析模型文件。

使用 Assimp 導入器：

Assimp::Importer importer;

宣告了一個 Assimp 的 Importer 導入器，用來處理和加載 3D 模型。

importer.ReadFile

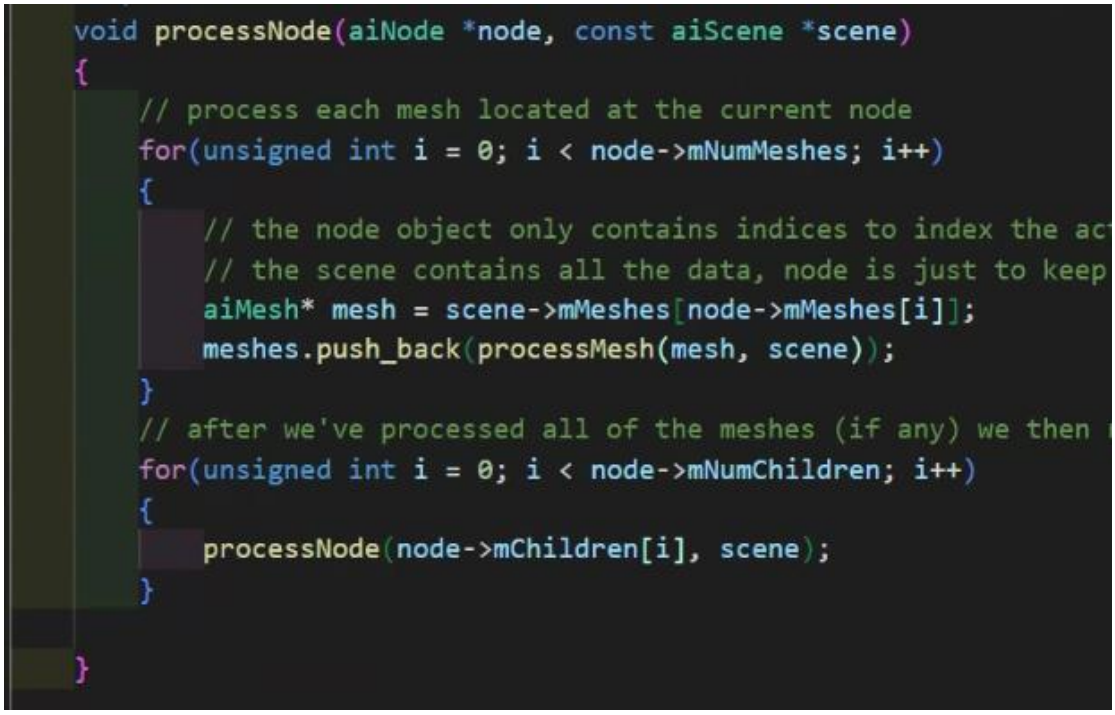
遞迴處理節點：

processNode(scene->mRootNode, scene);

遞迴處理模型的根節點及其子節點，這樣可以逐步解析並加載模型的所有部分

```
// loads a model with supported ASSIMP extensions from file and stores the result
void loadModel(string const &path)
{
    // read file via ASSIMP
    Assimp::Importer importer;
    const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProc
    // check for errors
    if(!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode)
    {
        cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << endl;
        return;
    }
    // retrieve the directory path of the filepath
    directory = path.substr(0, path.find_last_of('/'));

    // process ASSIMP's root node recursively
    processNode(scene->mRootNode, scene);
}
```



```
void processNode(aiNode *node, const aiScene *scene)
{
    // process each mesh located at the current node
    for(unsigned int i = 0; i < node->mNumMeshes; i++)
    {
        // the node object only contains indices to index the ac
        // the scene contains all the data, node is just to keep
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
        meshes.push_back(processMesh(mesh, scene));
    }
    // after we've processed all of the meshes (if any) we then
    for(unsigned int i = 0; i < node->mNumChildren; i++)
    {
        processNode(node->mChildren[i], scene);
    }

}
```

learnopengl/shader_m.h

```
Shader(const char* vertexPath, const char* fragmentPath)
{
    // 1. retrieve the vertex/fragment source code from filePath
    std::string vertexCode;
    std::string fragmentCode;
    std::ifstream vShaderFile;
    std::ifstream fShaderFile;
    // ensure ifstream objects can throw exceptions:
    vShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    fShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    try
    {
        // open files
        vShaderFile.open(vertexPath);
        fShaderFile.open(fragmentPath);
        std::stringstream vShaderStream, fShaderStream;
        // read file's buffer contents into streams
        vShaderStream << vShaderFile.rdbuf();
        fShaderStream << fShaderFile.rdbuf();
        // close file handlers
        vShaderFile.close();
        fShaderFile.close();
        // convert stream into string
        vertexCode = vShaderStream.str();
        fragmentCode = fShaderStream.str();
    }
    catch (std::ifstream::failure& e)
    {
        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESSFULLY_READ: " << e.what() << std::endl;
    }
    const char* vShaderCode = vertexCode.c_str();
    const char * fShaderCode = fragmentCode.c_str();
    // 2. compile shaders
    unsigned int vertex, fragment;
    // vertex shader
    vertex = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex, 1, &vShaderCode, NULL);
    glCompileShader(vertex);
    checkCompileErrors(vertex, "VERTEX");
    // fragment Shader
    fragment = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment, 1, &fShaderCode, NULL);
    glCompileShader(fragment);
    checkCompileErrors(fragment, "FRAGMENT");
    // shader Program
```

```
    // shader Program
    ID = glCreateProgram();
    glAttachShader(ID, vertex);
    glAttachShader(ID, fragment);
    glLinkProgram(ID);
    checkCompileErrors(ID, "PROGRAM");
    // delete the shaders as they're linked into our program now and no
    glDeleteShader(vertex);
    glDeleteShader(fragment);

}
```

前半處理讀取檔案，後半編譯著色器：

頂點著色器：

```
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vShaderCode, NULL);
glCompileShader(vertex);
checkCompileErrors(vertex, "VERTEX");
```

使用 glCreateShader 創建一個頂點著色器對象，然後用 glShaderSource 將頂點著色器源代碼傳遞給 OpenGL，最後使用 glCompileShader 進行編譯。編譯完成後，調用 checkCompileErrors 函數來檢查是否有編譯錯誤。

片段著色器：

```
fragment = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragment, 1, &fShaderCode, NULL);
glCompileShader(fragment);
checkCompileErrors(fragment, "FRAGMENT");
```

與頂點著色器類似，這段代碼創建並編譯片段著色器。

3. 創建和鏈接著色器程序：

著色器程序創建：

```
ID = glCreateProgram();
glAttachShader(ID, vertex);
glAttachShader(ID, fragment);
glLinkProgram(ID);
checkCompileErrors(ID, "PROGRAM");
```

創建一個著色器程序對象，將之前編譯的頂點和片段著色器附加到該程序，然後使用 glLinkProgram 將它們鏈接起來。鏈接完成後，調用 checkCompileErrors 來檢查鏈接是否成功。

4. 刪除著色器：

刪除不再需要的著色器對象：

```
glDeleteShader(vertex);
glDeleteShader(fragment);
```

learnopengl/mesh.h

```
// initializes all the buffer objects/arrays
void setupMesh()
{
    // create buffers/arrays
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);
    // load data into vertex buffers
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // A great thing about structs is that their memory layout is sequential for all its items.
    // The effect is that we can simply pass a pointer to the struct and it translates perfectly to a glm::vec3/
    // again translates to 3/2 floats which translates to a byte array.
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int), &indices[0], GL_STATIC_DRAW);

    // set the vertex attribute pointers
    // vertex Positions
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
    // vertex normals
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));
    // vertex texture coords
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, TexCoords));
    // vertex tangent
    glEnableVertexAttribArray(3);
    glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Tangent));
    // vertex bitangent
    glEnableVertexAttribArray(4);
    glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Bitangent));
    // ids
    glEnableVertexAttribArray(5);
    glVertexAttribIPointer(5, 4, GL_INT, sizeof(Vertex), (void*)offsetof(Vertex, m_BoneIDs));

    // weights
    glEnableVertexAttribArray(6);
    glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, m_Weights));
    glBindVertexArray(0);
}
```

這個 setupMesh 函數負責初始化並配置網格數據的緩衝區和頂點屬性。它使用 VAO 來儲存頂點數據的格式，VBO 來儲存頂點數據，EBO 來儲存索引數據。通過設置頂點屬性指針，OpenGL 知道如何解析這些數據並將它們應用到著色器程序中。

setupMesh 函數詳細解析：

1. 生成 VAO、VBO 和 EBO：

VAO 用於儲存頂點屬性配置。

VBO 儲存網格的頂點數據。

EBO 儲存繪製時使用的索引數據，優化網格繪製。

2. 綁定 VAO、VBO、EBO：

```
glBindVertexArray(VAO);
        // load data into vertex buffers
```

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex),
&vertices[0], GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned
int), &indices[0], GL_STATIC_DRAW);
```

3. 配置頂點屬性指針：

使用 glEnableVertexAttribArray 啟用頂點屬性，並使用 glVertexAttribPointer 設定每個頂點屬性。

```
// set the vertex attribute pointers
// vertex Positions
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
// vertex normals
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));
// vertex texture coords
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, TexCoords));
// vertex tangent
glEnableVertexAttribArray(3);
glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Tangent));
// vertex bitangent
glEnableVertexAttribArray(4);
glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Bitangent));
// ids
glEnableVertexAttribArray(5);
glVertexAttribIPointer(5, 4, GL_INT, sizeof(Vertex), (void*)offsetof(Vertex, m_BoneIDs));

// weights
glEnableVertexAttribArray(6);
glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, m_Weights));
```

頂點位置使用屬性索引 0。

頂點法線使用屬性索引 1。

貼圖坐標使用屬性索引 2。

頂點切線使用屬性索引 3。

頂點副切線使用屬性索引 4。

骨骼 ID 使用屬性索引 5。

骨骼權重使用屬性索引 6，

```cpp
// render the mesh
void Draw(Shader &shader)
{
    // bind appropriate textures
    unsigned int diffuseNr  = 1;
    unsigned int specularNr = 1;
    unsigned int normalNr   = 1;
    unsigned int heightNr   = 1;
    for(unsigned int i = 0; i < textures.size(); i++)
    {
        glActiveTexture(GL_TEXTURE0 + i); // active proper texture unit before binding
        // retrieve texture number (the N in diffuse_textureN)
        string number;
        string name = textures[i].type;
        if(name == "texture_diffuse")
            number = std::to_string(diffuseNr++);
        else if(name == "texture_specular")
            number = std::to_string(specularNr++); // transfer unsigned int to string
        else if(name == "texture_normal")
            number = std::to_string(normalNr++); // transfer unsigned int to string
         else if(name == "texture_height")
            number = std::to_string(heightNr++); // transfer unsigned int to string

        // now set the sampler to the correct texture unit
        glUniform1i(glGetUniformLocation(shader.ID, (name + number).c_str()), i);
        // and finally bind the texture
        glBindTexture(GL_TEXTURE_2D, textures[i].id);
    }

    // draw mesh
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, static_cast<unsigned int>(indices.size()), GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);

    // always good practice to set everything back to defaults once configured.
    glActiveTexture(GL_TEXTURE0);
}
```

綁定適當的貼圖：

首先，函數會遍歷每個材質貼圖並將它們綁定到相應的 OpenGL 貼圖單位 (texture unit) 上。

diffuseNr, specularNr, normalNr, heightNr

這些變數用來記錄已經綁定了多少張特定類型的貼圖（例如漫反射、鏡面反射、法線和高度貼圖），這是因為可能會有多個相同類型的貼圖（如 texture_diffuse1, texture_diffuse2 等）。

循環遍歷貼圖：

遍歷 textures 向量，為每個貼圖啟動一個 OpenGL 貼圖單位。

GL_TEXTURE0 + i 將不同的貼圖單位與貼圖索引 i 關聯。

取得貼圖類型和編號：

```cpp
if(name == "texture_diffuse")
    number = std::to_string(diffuseNr++);
```

```
else if(name == "texture_specular")
   number = std::to_string(specularNr++); // transfer unsigned int to
string
else if(name == "texture_normal")
   number = std::to_string(normalNr++); // transfer unsigned int to
string
else if(name == "texture_height")
   number = std::to_string(heightNr++); // transfer unsigned int to
string
```

依據貼圖的類型（如漫反射 texture_diffuse、鏡面反射 texture_specular、法線 texture_normal 或高度 texture_height），給每個類型分配一個編號。這樣可以生成類似 texture_diffuse1 的命名格式，並將計數器增加。

設置著色器取樣器：

```
glUniform1i(glGetUniformLocation(shader.ID, (name + number).c_str()),
i);
```

使用 glUniform1i 將每個貼圖的編號傳遞給著色器。glGetUniformLocation 用來獲取對應的 Uniform 變數的位置，將貼圖綁定到正確的貼圖單位。

綁定貼圖：

```
glBindTexture(GL_TEXTURE_2D, textures[i].id);
```

最後，通過 glBindTexture 將當前的貼圖綁定到對應的貼圖單位。

2. 繪製網格：

```
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, static_cast<unsigned int>(indices.size()),
GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
```

綁定儲存著網格數據的 VAO。這個 VAO 包含了頂點屬性配置和索引緩衝區的指針。

使用 glDrawElements 方法來繪製網格。這裡使用的是三角形作為基本繪製圖元（GL_TRIANGLES），索引數組的大小為 indices.size()，並且索引類型為無符號整數（GL_UNSIGNED_INT）。

繪製完畢後，解除 VAO 的綁定，以免意外修改。

3. 重置貼圖單位：

```
glActiveTexture(GL_TEXTURE0);
```

最後，將活動的貼圖單位設置回 GL_TEXTURE0。這是個良好的編程習慣，確保下次使用時貼圖單位處於預設狀態。

---

learnopengl/camera.h

```
// calculates the front vector from the Camera's (updated) Euler Angles
void updateCameraVectors()
{
    // calculate the new Front vector
    glm::vec3 front;
    front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
    front.y = sin(glm::radians(Pitch));
    front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
    Front = glm::normalize(front);
    // also re-calculate the Right and Up vector
    Right = glm::normalize(glm::cross(Front, WorldUp));  // normalize the vecto
    Up    = glm::normalize(glm::cross(Right, Front));
}
```

updateCameraVectors 函數根據攝影機的歐拉角來更新其方向向量，確保攝影機的視角能夠正確反映其旋轉。前向向量基於偏航角和俯仰角計算，而右向和上向向量則通過叉積計算來確保它們與前向垂直，形成正交坐標系。

```
    // returns the view matrix calculated using Euler Angles a
    glm::mat4 GetViewMatrix()
    {
        return glm::lookAt(Position, Position + Front, Up);
    }
```

GetViewMatrix 用於獲取攝影機的視圖矩陣。

```
// processes input received from any keyboard-like input system.
void ProcessKeyboard(Camera_Movement direction, float deltaTime)
{
    float velocity = MovementSpeed * deltaTime;
    if (direction == FORWARD)
        Position += Front * velocity;
    if (direction == BACKWARD)
        Position -= Front * velocity;
    if (direction == LEFT)
        Position -= Right * velocity;
    if (direction == RIGHT)
        Position += Right * velocity;
}
```

ProcessKeyboard 根據鍵盤輸入移動攝影機。

```
// processes input received from a mouse input system. Expects the offset value in both th
void ProcessMouseMovement(float xoffset, float yoffset, GLboolean constrainPitch = true)
{
    xoffset *= MouseSensitivity;
    yoffset *= MouseSensitivity;

    Yaw   += xoffset;
    Pitch += yoffset;

    // make sure that when pitch is out of bounds, screen doesn't get flipped
    if (constrainPitch)
    {
        if (Pitch > 89.0f)
            Pitch = 89.0f;
        if (Pitch < -89.0f)
            Pitch = -89.0f;
    }

    // update Front, Right and Up Vectors using the updated Euler angles
    updateCameraVectors();
}
```

ProcessMouseMovement 根據滑鼠移動更新攝影機的朝向。

```
// processes input received from a mouse scro
void ProcessMouseScroll(float yoffset)
{
    Zoom -= (float)yoffset;
    if (Zoom < 1.0f)
        Zoom = 1.0f;
    if (Zoom > 45.0f)
        Zoom = 45.0f;
}
```

ProcessMouseScroll 用於控制攝影機的視場縮放。

程式介紹：

1. 先初始化 glfw，建立視窗，初始化視窗，加入特性，包含：滑鼠控制、鍵盤控制、滾輪控制...
2. 讀取 config.ini 的參數，包含 modelPath、texturePath
3. 載入 Shader，包含 vertex shader、fragment shader
4. 從 modelPath 載入模型
5. 從 texturePath 載入紋理
6. 啟用並綁定紋理 model
7. 將帶有紋理的 model 輸出到視窗

程式如何執行：

1. 直接使用 cmake 產生 CG_Homework.sln
2. 在 Visual Studio 2022 開啟 CG_Homework.sln
3. 建置專案 “hw__hw1”
4. 執行檔在 ”bin/hw/Debug/hw__hw1.exe”
5. Config 在 “bin/hw/Debug/config.ini” 改變 model path、texture path
6. 直接執行 “hw__hw1.exe”