Báo cáo thực tập

I) Tìm hiểu về thư viện GLFW

a) Giới thiệu

GLWF (Graphics Library Framework) là một thư viện mã nguồn mở được thiết kế để hỗ trợ việc tạo cửa sổ, quản lý ngữ cảnh OpenGL/Vulkan, xử lý đầu vào từ bàn phím, chuột và các thiết bị điều khiển(gamepad). Đây là một thư viện nhẹ, đa nền tảng được sử dụng trong phát triển đồ họa thời gian thực và trò chơi điện tử.

b) Đặt điểm nổi bật của GLFW

Đa nền tảng: hỗ trợ Windows, macOS và Linux

Hỗ **trợ OpenGL** và **Vulkan: GlFW** cung cấp API để tạo ngữ cảnh OpenGL và hỗ trợ khởi tạo Vulkan.

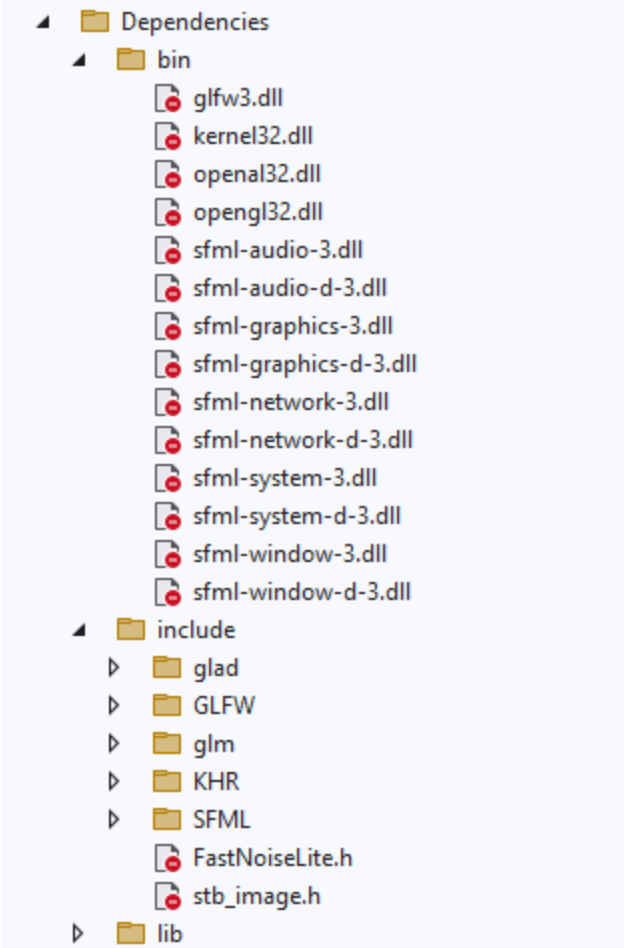
Quản lý cửa sổ: hỗ trợ bàn phím, chuột, con trỏ chuột, và tay cầm chơi game

Xử lý đầu vào: Cung cấp thông tin về các màn hình và cho phép thay đổi độ phân giải.

Cấu hình đơn giản: Không cần quá nhiều thiết lập để bắt đầu

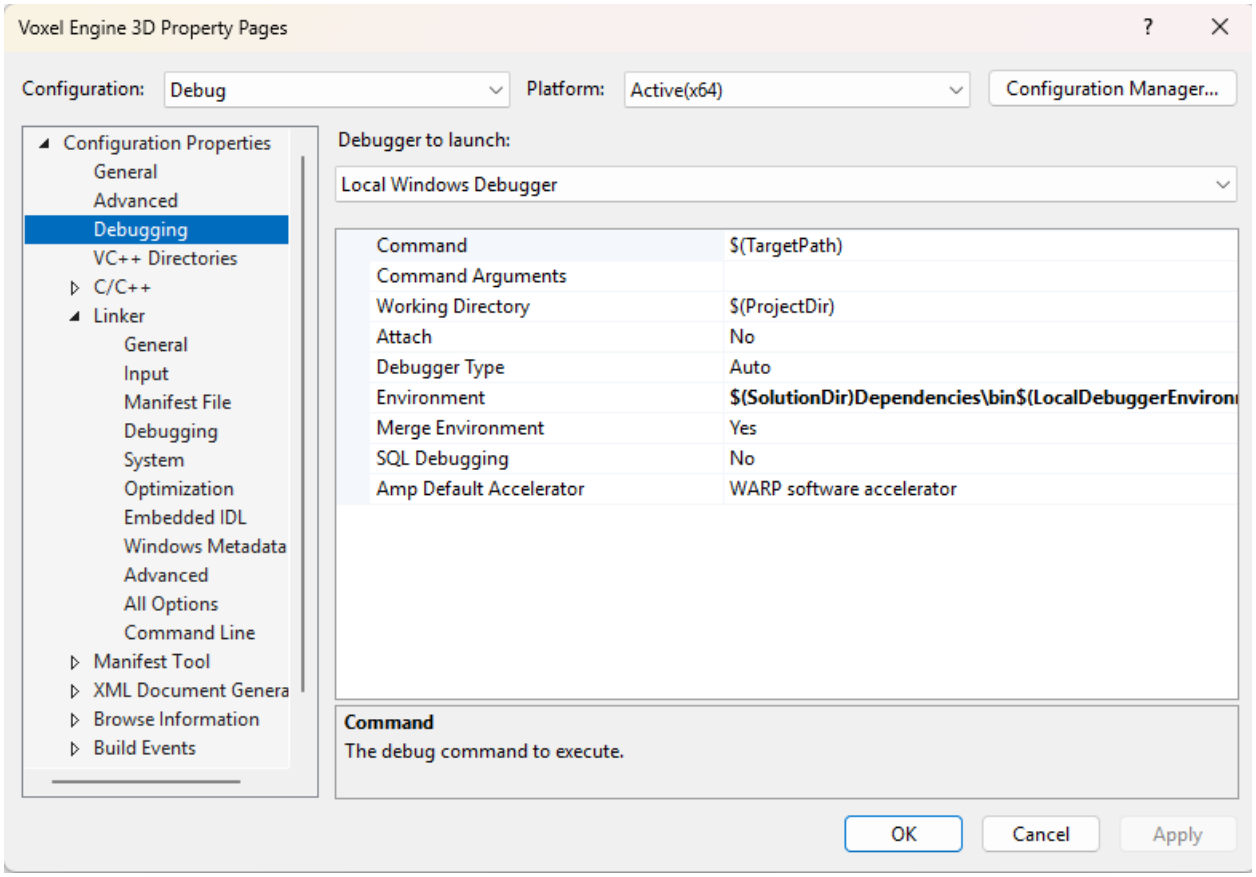
c) Cài đặt GLFW

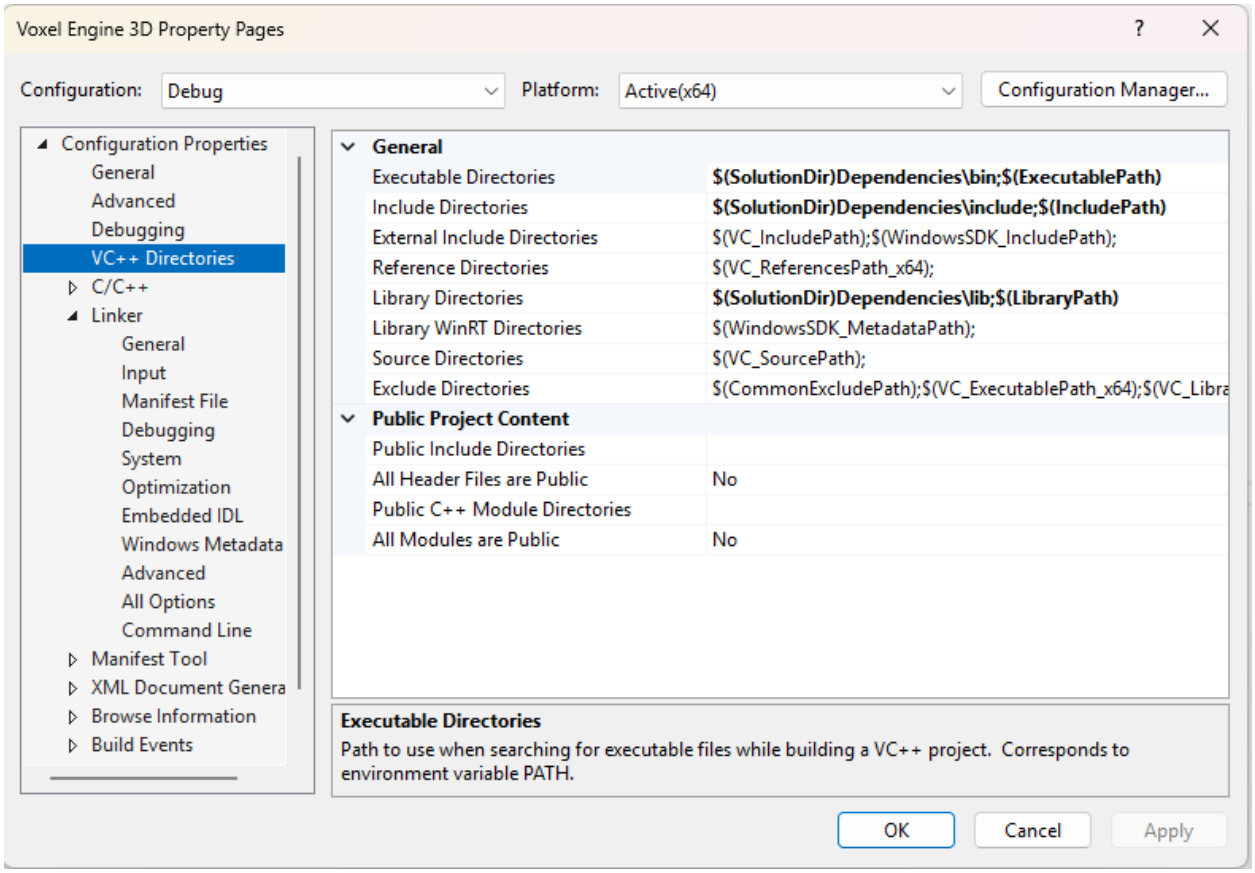
Vì dự án chọn tool hỗ trợ **là Visual Studio Community**, nên cài đặt **GLFW** sẽ được triển khai trên hệ điều hành **Windows.**

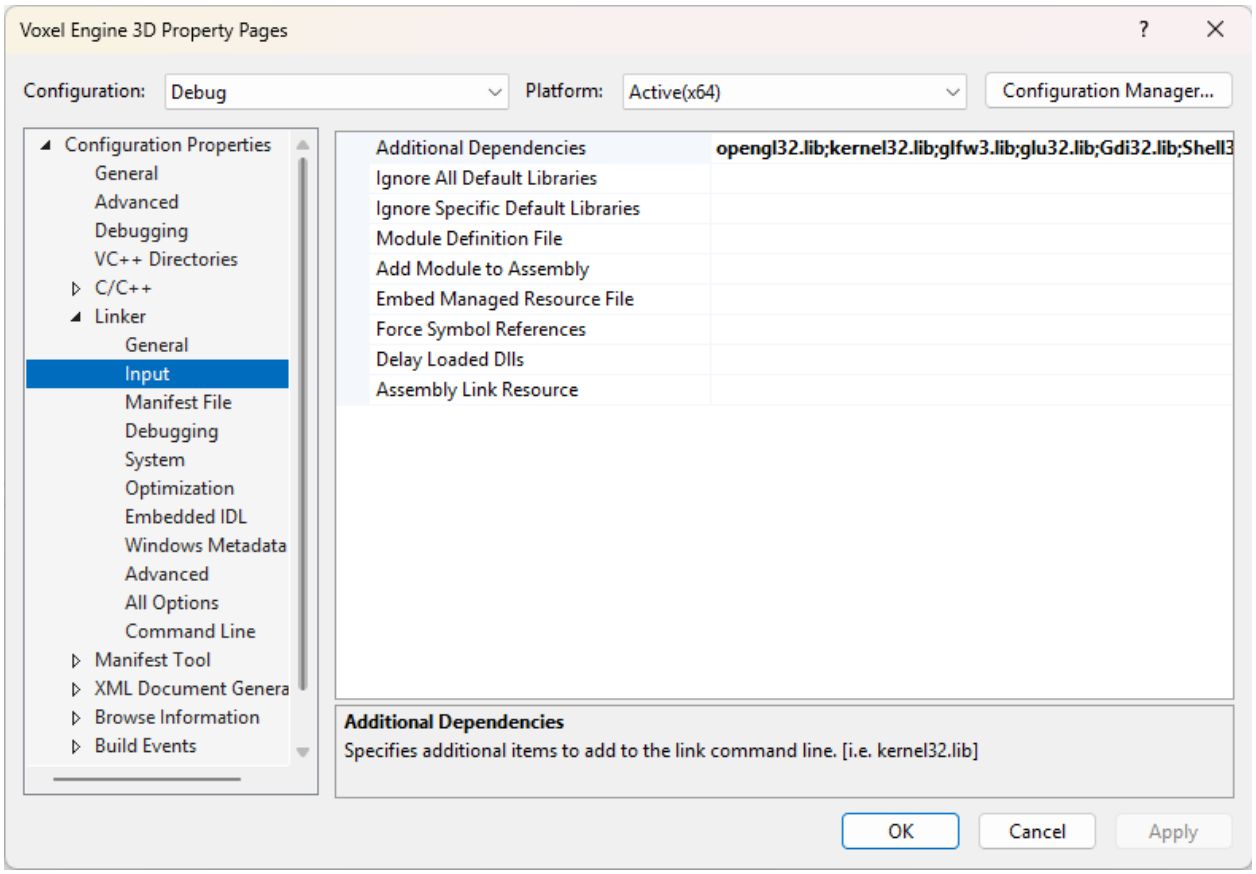


Cấu hình thư mục các thư viện hỗ trợ ở thư mục Dependencies. Bao gồm thư mục "include" chứa file header định nghĩa các hàm của thư viện. Thư mục "bin" chứa các file "**dynamic library**" sử dụng khi chương trình yêu cầu các file mã máy để chèn vào chương trình. Thư mục "lib" là nơi cài đặt các hàm và đối tượng đã được khai báo ở các file header.

Cấu hình **Visual Studio Community** cho chế độ DeBug:


Voxel Engine 3D Property Pages
Configuration: Debug
Platform: Active(x64)
Configuration Manager...
Configuration Properties
General
Advanced
Debugging
VC++ Directories
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Windows Metadata
Advanced
All Options
Command Line
Manifest Tool
XML Document Genera
Browse Information
Build Events
Debugger to launch:
Local Windows Debugger
Command $(TargetPath)
Command Arguments
Working Directory $(ProjectDir)
Attach No
Debugger Type Auto
Environment $(SolutionDir)Dependencies\bin$(LocalDebuggerEnviron
Merge Environment Yes
SQL Debugging No
Amp Default Accelerator WARP software accelerator
Command
The debug command to execute.
OK
Cancel
Apply


Voxel Engine 3D Property Pages
Configuration: Debug
Platform: Active(x64)
Configuration Manager...
Configuration Properties
General
Advanced
Debugging
VC++ Directories
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Windows Metadata
Advanced
All Options
Command Line
Manifest Tool
XML Document Genera
Browse Information
Build Events
General
Executable Directories $(SolutionDir)Dependencies\bin;$(ExecutablePath)
Include Directories $(SolutionDir)Dependencies\include;$(IncludePath)
External Include Directories $(VC_IncludePath);$(WindowsSDK_IncludePath);
Reference Directories $(VC_ReferencesPath_x64);
Library Directories $(SolutionDir)Dependencies\lib;$(LibraryPath)
Library WinRT Directories $(WindowsSDK_MetadataPath);
Source Directories $(VC_SourcePath);
Exclude Directories $(CommonExcludePath);$(VC_ExecutablePath_x64);$(VC_Libra
Public Project Content
Public Include Directories
All Header Files are Public No
Public C++ Module Directories
All Modules are Public No
Executable Directories
Path to use when searching for executable files while building a VC++ project. Corresponds to environment variable PATH.
OK
Cancel
Apply



Cấu hình **Visual Studio Community** cho chế độ Release tương tự chế độ DeBug.

d) Cấu hình API của GLFW

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

#include "WindowConfig.h"
#include "Singleton.h"

#define ENGINE Engine::getInstance()

//void framebuffer_size_callback(GLFWwindow* window, int width, int height);

class Engine : public Singleton<Engine>
{
friend class Singleton<Engine>;
public:
    Engine();
    ~Engine();
    void init();
    void run();
    void cleanup();

    static void framebuffer_size_callback(GLFWwindow* window, int width, int height) {
        glViewport(0, 0, width, height);
    }
private:
    GLFWwindow* m_window;
};
```

Engine được định nghĩa sử dụng Singleton design pattern.

```cpp
void Engine::init()
{
    // load glfw
    {
        if (!glfwInit()) {
            std::cerr << "Failed to initialize GLFW" << std::endl;
            return;
        }
        glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
        glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
        glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    }


    // init window
    {
        m_window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, WINDOW_TITLE, NULL, NULL);
        if (!m_window) {
            std::cerr << "Failed to create GLFW window" << std::endl;
            glfwTerminate();
            return;
        }
        WINDOW_MANAGER->setWindow(m_window);
    }

    // load glad
    {
        if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
            std::cerr << "Failed to initialize GLAD" << std::endl;
            return;
        }
    }

    // set viewport
    {
        glViewport(0, 0, 800, 600);
        glfwSetFramebufferSizeCallback(m_window, Engine::framebuffer_size_callback);
    }
}
```

Hàm khởi tạo của lớp Engine.

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

#include "../Singleton.h"

#define WINDOW_MANAGER WindowManager::getInstance()

class WindowManager : public Singleton<WindowManager>
{
    friend class Singleton<WindowManager>;
public:
    WindowManager();
    ~WindowManager();
    void render();

    void setWindow(GLFWwindow* window);
    GLFWwindow* getWindow();
private:
    GLFWwindow* m_window;
};
```

Lớp Window_Manager được định nghĩa theo Singleton Design pattern, dùng để quản lý của sổ GLFW.

- Cấu trúc API của GLFW bao gồm:
    - Khởi tạo:

```
if (!glfwInit()) {
    printf("Failed to initialize GLFW\n");
    return -1;
}
// Khi chương trình kết thúc
glfwTerminate();
```

    - Tạo cửa sổ và Ngữ cảnh OpenGL:

```
GLFWwindow* window = glfwCreateWindow(800, 600, "Hello GLFW", NULL, NULL);
if (!window) {
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
```

- Vòng lặp sự kiện:

```
while (!glfwWindowShouldClose(window)) {
    glClear(GL_COLOR_BUFFER_BIT);
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

- Xử lý đầu vào(ví dụ về xử lý sự kiện **resize** màn hình từ người dùng):

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

e) So sánh GLFW với các thư viện khác

Tính năng	GLFW	SDL2	SFML
Hỗ trợ OpenGL	Có	Có	Có
Hỗ trợ Vulkan	Có	Không	Không
Quản lý cửa sổ	Có	Có	Có
Xử lý đầu vào	Có	Có	Có
Hỗ trợ âm thanh	Không	Có	Có
Hỗ trợ mạng	Không	Có	Có

f) Ứng dụng thực tế của GLFW

- Phát triển game: GLFW thường được sử dụng trong các game engine trong OpenGL Engine
- Mô phỏng đồ họa: Được dùng để tạo các ứng dụng mô phỏng, visualizer
- Ứng dụng khoa học: Dùng trong nghiên cứu đồ họa máy tính

II) Tìm hiểu về thư viện glm

Giới thiệu về glm

Glm (**OpenGL Mathematics**) là một thư viện C++ để xử lý toán học cho đồ họa mô phỏng theo cách hoạt động của **OpenGL Shading language(GLSL)**. **GLM** là thư viện **header-only**, không cần build hay link, giúp người phát triển dễ dàng sử dụng trong project OpenGL, Vulkan, game engine hoặc các ứng dụng đồ họa khác.

Đặc điểm nổi bật của GLM

API tương đương GLSL: **Vector, maxtrix, quaternion**... giúp chuyển đổi qua lại giữa các shader và CPU dễ dàng.

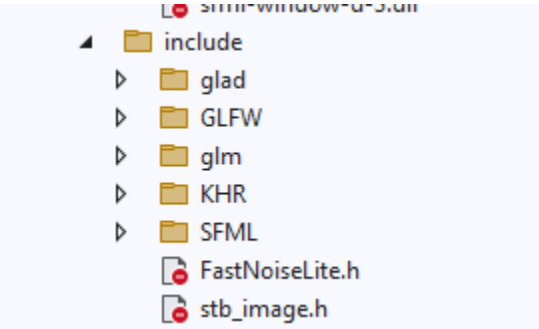
Header-only: Chỉ bao gồm các file ".hpp" không cần build thư viện.

Tích hợp C++ 11: Hỗ trợ template, constexpr, operator overloading...

Cung cấp nhiều module: Tranform, projection, trigonometry, matrix decomposition, random, noise...

Cài đặt thư viện

Thêm thư mục header của glm vào thư mục chứa các file header "include" trong "Dependencies".



Sử dụng GLM trong Project.

Vector:

```
#include <glm/glm.hpp>

glm::vec3 position(1.0f, 2.0f, 3.0f);
glm::vec3 direction(0.0f, 1.0f, 0.0f);
glm::vec3 result = position + direction;
```

Ma trận biến đổi

```
#include <glm/gtc/matrix_transform.hpp>

glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(0.0f, 5.0f, 0.0f));
model = glm::rotate(model, glm::radians(90.0f), glm::vec3(0.0f, 0.0f, 1.0f));
model = glm::scale(model, glm::vec3(2.0f));
```

Ma trận chiếu (Projection)

```
#include <glm/gtc/matrix_transform.hpp>

float aspect = 800.0f / 600.0f;
glm::mat4 projection = glm::perspective(glm::radians(45.0f), aspect, 0.1f, 100.0f);
```

Camera View

```
glm::vec3 cameraPos(0.0f, 0.0f, 3.0f);
glm::vec3 target(0.0f, 0.0f, 0.0f);
glm::vec3 up(0.0f, 1.0f, 0.0f);
glm::mat4 view = glm::lookAt(cameraPos, target, up);
```

III) Tọa độ và phép chiếu trong không gian 3D

Trong các khái niệm của OpenGL với **xử lý đồ họa 3D**, chúng ta đang áp dụng một hiện tương của mắt người đó chính là **khả năng phân biệt độ sâu** của **điểm ảnh** vào xử lý đồ họa máy tính.

Màn hình (monitor) là một mặt phẳng 2D, chỉ có chiều rộng và chiều cao. Tuy nhiên, khi nhìn vào một bức ảnh, phim, hay trò chơi 3D trên màn hình, chúng ta cảm nhận được độ sâu **(z-axis**). Đây là một ảo giác thị giác được tạo nên nhờ phép chiếu phối cảnh(**Perspective Projection**).

- **Vật thể ở gần mắt hơn sẽ trông nhỏ hơn**
- **Vật ở xa hơn sẽ trông nhỏ hơn**.
- **Đường thẳng song song trong thực tế(như đường tàu) sẽ hội tự về một điểm xa(gọi là điểm tụ - vanishing point).**

Công thức hình học:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \text{ProjectionMatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow (x', y') = \left(\frac{x'}{w}, \frac{y'}{w} \right)$$

Hệ thống tọa độ 3D (x, y, z) sẽ được chiếu lên mặt phẳng 2D(x', y') bằng **ma trận chiếu phối cảnh**. Đây chính là cách mà thư viện như **OpenGL**, DirectX hay **Vulkan** hiển thị hình ảnh 3D lên màn hình 2D.

Tìm độ sâu của mỗi **pixel** điểm ảnh trên màn hình(**monitor**)

Như chúng ta đã biết, một vật thể trong đời sống của thể được mô phỏng bằng một hệ tọa độ 3 chiều. Ta gán cho vật một gốc tọa độ tại một điểm bất kì trên vật thể. Sau đó chúng ta sẽ có **một số lượng tọa độ điểm minh hoạ cho những điểm nổi bật của vật thể**. Từ tập hợp các điểm đó, chúng ta sẽ xác định được **hình dạng của vật thể** như thế nào. Đối với những hình dạng đặc biệt, chúng ta thường sử dụng các hình tam giác(sinh ra bằng cách nối 3 điểm lại với nhau), và sử dụng **phép nội suy** để làm nhẵn mặt phẳng vật thể.

Vậy nên mỗi vật thể, đối tượng trong không gian 3D sẽ đều có **một hệ tọa độ riêng** dùng để miêu tả hình dạng, định nghĩa kích thước của vật thể.

Từ đó chúng ta cần một **gốc tọa độ chung** để biểu thị **vị trị đặt vật trong một thế giới chung**. Chúng ta gọi đó là **Ma trận thế giới – Model(World Matrix)** có tác dụng biến đổi từ local model -> world coordinates

$$M_{\text{model}} = T \cdot R \cdot S$$

Trong đó:

- T là ma trận tịnh tiến:

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- R là ma trận xoay (gồm xoay trục X, Y, Z – có thể nhân nhau)

  Ví dụ: xoay quanh trục Y:

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- S là ma trận co giãn (scale):

$$S = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Khi có tọa độ world, chúng ta cần xác định vị trí của các vật thể đến mắt người(hoặc **camera)**, từ đó chúng ta cần một ma trận biến đổi tọa độ khác gọi là **View Matrix** có tác

dụng xác định **vùng mà camera nhìn thấy**.

$$M_{\text{view}} = \text{LookAt}(eye, center, up)$$

Công thức chi tiết:

$$f = \text{normalize}(center - eye)$$
$$s = \text{normalize}(f \times up)$$
$$u = s \times f$$

$$M_{\text{view}} = \begin{bmatrix} s_x & s_y & s_z & -s \cdot eye \\ u_x & u_y & u_z & -u \cdot eye \\ -f_x & -f_y & -f_z & f \cdot eye \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Sau khi đã xác định được tọa độ các điểm vật thể so với camera. Chúng ta cần một ma trận để mô phỏng hiệu ứng thị giác như đã nói ở trên để mắt người có cảm giác như thật. **Vật thể càng xa thì kích thước càng bé**:

Phối cảnh (Perspective Projection):

$$M_{\text{perspective}} = \begin{bmatrix} \frac{1}{\tan(fov/2) \cdot aspect} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(fov/2)} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

`fov` : góc nhìn theo chiều dọc (radian)

`aspect` : tỉ lệ màn hình (width / height)

`n` : near plane

`f` : far plane

Tọa độ của các điểm của vật thể là **vector 3D**, tuy nhiên các ma trận trên là ma trận kích thước 4x4, vậy nên chúng ta cần xử lý một chút. Chúng ta có khái **niệm homogeneous coordinates**. Trong không gian **Euclidean 3D(x , y , z)** vector có 3 thành phần. Chỉ có **phép**

quay và **phép co giãn** có thể biểu diễn bằng ma trận 3 x 3. **Nhưng phép tịnh tiến (translation)** không thể biểu diễn bằng ma trận 3 x 3 thông thường, vậy nên không có cách nào để nhân ma trận với vector tọa độ của điểm. Vậy nên chúng ta cần thêm một chiều cho vector tọa độ:

$$\vec{v}_{\text{homogeneous}} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Giải thích **w = 1** là một điểm trong không gian (x , y , z), sau khi nhân ma trận chiếu phối cảnh, ta sẽ có vector dạng:

$$(x', y', z', w') \Rightarrow (x_{\text{screen}}, y_{\text{screen}}) = \left(\frac{x'}{w'}, \frac{y'}{w'} \right)$$

Từ đó chúng ta tìm được **(x', y')** là tọa độ tương ứng của tọa độ vật trên màn hình monitor.

OpenGL sau khi đã xác định được độ sâu của các tọa độ của vật thể sẽ tính toán điểm ảnh nào trong cùng tọa độ (x', y') trên màn hình có độ sâu bé hơn sẽ lấy màu của điểm ảnh đó hiển thị lên màn hình, các điểm ảnh có độ sâu lớn hơn (tức nằm phía sau sẽ không được hiển thị) từ đó tạo nên cảm giác chân thật cho người nhìn.