1. **Include một số thư viện cần thiết**

- ```
  // Include standard headers
  #include <stdio.h>
  #include <stdlib.h>
  ```
- ```
  // Include GLEW
  #include <GL/glew.h>
  ```
- GLEW: cross-platform open-source dùng để load - link C/C++ lib (opengl extension nào phù hợp với platform nào)
- ```
  // Include GLFW
  #include <GLFW/glfw3.h>
  GLFWwindow* window;
  ```
- GLFW: Open Source, multi-platform library cung cấp các high-level API của openGL (openGL ES) -> chủ yếu là dùng mở window, contexts, events, …
- ```
  // Include GLM
  #include <glm/glm.hpp>
  using namespace glm;
  ```
- OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications -> Những cái mat4, vec, ...

2. **Initialize**

- Initialise GLFW
- ```
  // Initialise GLFW
  if( !glfwInit() )
  {
      fprintf( stderr, "Failed to initialize GLFW\n" );
      getchar();
      return -1;
  }

  glfwWindowHint(GLFW_SAMPLES, 4);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // To make MacOS happy; should not be needed
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

  // Open a window and create its OpenGL context
  window = glfwCreateWindow( 1024, 768, "Tutorial 02 - Red triangle", NULL, NULL);
  if( window == NULL ){
      fprintf( stderr, "Failed to open GLFW window. If you have an Intel GPU, they are not 3.3 compatible. Try
      getchar();
      glfwTerminate();
      return -1;
  }
  glfwMakeContextCurrent(window);
  ```
  ```

- Initialize GLEW

```
// Initialize GLEW
glewExperimental = true; // Needed for core profile
if (glewInit() != GLEW_OK) {
    fprintf(stderr, "Failed to initialize GLEW\n");
    getchar();
    glfwTerminate();
    return -1;
}
```

- Một số cái khác:

-
```
// Ensure we can capture the escape key being pressed below
glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);

// Dark blue background
glClearColor(0.0f, 0.0f, 0.4f, 0.0f);
```

-
```
GLuint VertexArrayID;
glGenVertexArrays(1, &VertexArrayID);
glBindVertexArray(VertexArrayID);
```

- VAO (Vertex Array Objects): là opengl object chỉ được tạo 1 lần trong toàn ctr và đóng vai trò là container cho VBOs

-
```
// Cleanup VBO
glDeleteBuffers(1, &vertexbuffer);
glDeleteVertexArrays(1, &VertexArrayID);
glDeleteProgram(programID);
```

3. Nạp dữ liệu

- Khai báo:

-
```
static const GLfloat g_vertex_buffer_data[] = {
    -1.0f, -1.0f, 0.0f,
     1.0f, -1.0f, 0.0f,
     0.0f,  1.0f, 0.0f,
};
```

- 3 số liên tiếp ứng với tọa độ 1 đỉnh trong coordinate system
- Nạp vào GPU

```
GLuint vertexbuffer;
glGenBuffers(1, &vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data), g_vertex_buffer_data, GL_STATIC_DRAW);
```

- Gluint: là 1 số uint nhưng kích thước cố định trên các platform khác nhau
- glGenbuffers: Tạo 1 vùng buffer và gán định danh (resulting identifier) vào vertexbuffer
- glBindBuffer: gán GL_ARRAY_BUFFER với vertexbuffer => để thay đổi vertexbuffer sẽ thao tác trên GL_ARRAY_BUFFER
- Cái nhìn tổng quan hơn: glGenbuffers tạo 1 buffer object trong GPU chứa unformatted data => The behavior of a buffer object depends on its binding points (For example, when a buffer object's binding point is GL_ARRAY_BUFFER, it behaves as a Vertex Buffer Object - VBO)

- VBO là 1 loại object trong openGL. openGL object thường gắn với các thao tác:
  - glGen
  - glDelete
  - glBind
- VBO: Thông thường data được lưu trong VBO dưới dạng vector. VD, một đa giác sẽ gồm các loại data sau:
  - Position: represented as (x, y, z)
  - Normal: represented as (n1, n2 ,n3)
  - Texture: represented as (u, v)

4. Shader

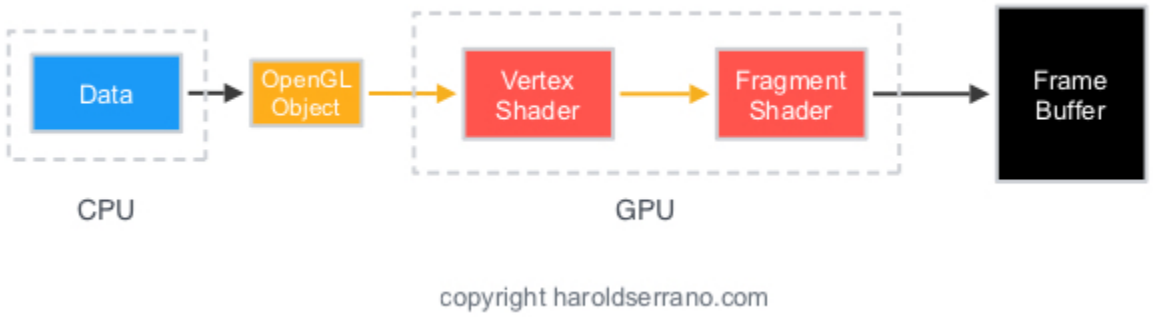
- Load shader:

```
// Create and compile our GLSL program from the shaders
GLuint programID = LoadShaders( "TransformVertexShader.vertexshader", "ColorFragmentShader.fragmentshader" );
```

```
// Use our shader
glUseProgram(programID);
```

```
// Cleanup VBO and shader
glDeleteBuffers(1, &vertexbuffer);
glDeleteBuffers(1, &colorbuffer);
glDeleteProgram(programID);
glDeleteVertexArrays(1, &VertexArrayID);
```

- Connect VBO với shader:



-
- Là cặp chương trình hoạt động trong GPU, có nhiệm vụ rendering the polygon mesh lên màn hình.
- Mặc dù VBO ở trong GPU nhưng chưa có cách nào để connect VBO với shader program => tạo connect bằng function **glVertexAttribPointer()**
-

```
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glVertexAttribPointer(
    0,                  // attribute 0. No particular reason for 0, but must match the layout in the shader.
    3,                  // size
    GL_FLOAT,           // type
    GL_FALSE,           // normalized?
    0,                  // stride
    (void*)0            // array buffer offset
);
```

-

```
//Data representing Position of a polygon

glVertexAttribPointer(location_of_position_attribute,3,GL_FLOAT,GL_FALSE,0,0);
```

-
- glEnableVertexAttribArray: enables the generic vertex attribute array specified by *index*. By default, all client-side capabilities are disabled, including all generic vertex attribute arrays. If enabled, the values in the generic vertex attribute array will be accessed and used for rendering when calls are made to vertex array commands such as glDrawArrays or glDrawElements
- Size: Vì ví dụ trên cần dữ liệu tọa độ => second param là 3 ~ biểu thị 3 trục tọa độ x, y, z
- Type: vertex data type

- Vertex Shader

```
#version 330 core

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec3 vertexColor;

// Output data ; will be interpolated for each fragment.
out vec3 fragmentColor;
// Values that stay constant for the whole mesh.
uniform mat4 MVP;
```

- #version => khai báo sử dụng opengl 3 syntax
- Vec3: là vector 3 components trong glsl
- layout(location = 0): chỉ cái vùng buffer khi **glVertexAttribPointer của** vertexPosition_modelspace. Mỗi vertex có thể có nhiều attribute: color, coordinate, ... => Specify vùng nhớ nào tương ứng với attribute nào (layout(location = 0): refers to the buffer we use to feed the vertexPosition_modelspace attribute. Each vertex can have numerous attributes : A position, one or several colours, one or several texture coordinates, lots of other things. OpenGL doesn't know what a colour is : it just sees a vec3. So we have to tell him which buffer corresponds to which input. We do that by setting the layout to the same value as the first parameter to glVertexAttribPointer. The value "0" is not important, it could be 12 (but no more than glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &v)), the important thing is that it's the same number on both sides.)
- "vertexPosition_modelspace" could have any other name. It will contain the position of the vertex for each run of the vertex shader.
- "in" means that this is some input data

```
void main(){

    // Output position of the vertex, in clip space : MVP * position
    gl_Position =  MVP * vec4(vertexPosition_modelspace,1);

    // The color of each vertex will be interpolated
    // to produce the color of each fragment
    fragmentColor = vertexColor;
}
```

- gl_Position là một trong những built-in variables => xác định vị trí
- fragmentColor là output cho fragment shader nhận vào.

- Fragment shader

```
#version 330 core

// Interpolated values from the vertex shaders
in vec3 fragmentColor;

// Ouput data
out vec3 color;

void main(){

    // Output color = color specified in the vertex shader,
    // interpolated between all 3 surrounding vertices
    color = fragmentColor;

}
```

- Texture

- Trong fragment shader(uniform giống kiểu biến global trong glsl):

```
// Values that stay constant for the whole mesh.
uniform sampler2D myTextureSampler;

void main(){

    // Output color = color of the texture at the specified UV
    color = texture( myTextureSampler, UV ).rgb;
}
```

- Load texture vào

```
// Load the texture using any two methods
//GLuint Texture = loadBMP_custom("uvtemplate.bmp");
GLuint Texture = loadDDS("uvtemplate.DDS");
```

- Lấy textureID ứng với biến myTextureSampler (type: uniform) trong FragmentShader

```
// Get a handle for our "myTextureSampler" uniform
GLuint TextureID  = glGetUniformLocation(programID, "myTextureSampler");
```

- Bind texture với texture unit 0 (có nhiều unit bởi vì shader có thể có nhiều texture). glUniform1i gắn cái TextureID (ứng với myTextureSampler) vào texture unit0

```
// Bind our texture in Texture Unit 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, Texture);
// Set our "myTextureSampler" sampler to use Texture Unit 0
glUniform1i(TextureID, 0);
```

- Khi có nhiều texture:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture0);
glUniform1i(_textureUniform, 0);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture1);
glUniform1i(_textureUniform, 1);
```

```
// Cleanup VBO and shader
glDeleteBuffers(1, &vertexbuffer);
glDeleteBuffers(1, &uvbuffer);
glDeleteProgram(programID);
glDeleteTextures(1, &Texture);
glDeleteVertexArrays(1, &VertexArrayID);
```

-

5. Tạo Model, View và Projection matrices

- Tạo model, view và Projection matrices

```
// Get a handle for our "MVP" uniform
GLuint MatrixID = glGetUniformLocation(programID, "MVP");

// Projection matrix : 45° Field of View, 4:3 ratio, display range : 0.1 unit <-> 100 units
glm::mat4 Projection = glm::perspective(glm::radians(45.0f), 4.0f / 3.0f, 0.1f, 100.0f);
// Camera matrix
glm::mat4 View       = glm::lookAt(
                               glm::vec3(4,3,3), // Camera is at (4,3,3), in World Space
                               glm::vec3(0,0,0), // and looks at the origin
                               glm::vec3(0,1,0)  // Head is up (set to 0,-1,0 to look upside-down)
                          );
// Model matrix : an identity matrix (model will be at the origin)
glm::mat4 Model      = glm::mat4(1.0f);
// Our ModelViewProjection : multiplication of our 3 matrices
glm::mat4 MVP        = Projection * View * Model; // Remember, matrix multiplication is the other way around
```

-

- Handle thay đổi model, view, projection matrices
- Quy trình tính toán:

```
do{

    // ...

    // Compute the MVP matrix from keyboard and mouse input
    computeMatricesFromInputs();
    glm::mat4 ProjectionMatrix = getProjectionMatrix();
    glm::mat4 ViewMatrix = getViewMatrix();
    glm::mat4 ModelMatrix = glm::mat4(1.0);
    glm::mat4 MVP = ProjectionMatrix * ViewMatrix * ModelMatrix;

    // ...
}
```

- Có 3 function mới là:
    + computeMatricesFromInputs(): đọc vị trí bàn phím và chuột để tính lại Projection và View matrices
    + getProjectionMatrix(): returns the computed Projection matrix.
    + getViewMatrix() just returns the computed View matrix.
- Vì phần này có thể thường được tái sử dụng nên hay được phân thành folder common/controls.cpp

```
glm::mat4 ViewMatrix;
glm::mat4 ProjectionMatrix;

glm::mat4 getViewMatrix(){
    return ViewMatrix;
}
glm::mat4 getProjectionMatrix(){
    return ProjectionMatrix;
}


// Initial position : on +Z
glm::vec3 position = glm::vec3( 0, 0, 5 );
// Initial horizontal angle : toward -Z
float horizontalAngle = 3.14f;
// Initial vertical angle : none
float verticalAngle = 0.0f;
// Initial Field of View
float initialFoV = 45.0f;

float speed = 3.0f; // 3 units / second
float mouseSpeed = 0.005f;



void computeMatricesFromInputs(){ ···
}
```