# COSC 4370 - HW 4

Omar Watad

## Objective:

The assignment requires the implementation of texture mapping in OpenGL. The uv data is provided in the main function and needs to be transferred to the OpenGL buffer. The goal is to bind a texture and draw it using shader code to produce a rotating textured cube.

## Camera.h:

This header file defines an abstract Camera class that processes input and calculates the corresponding Euler angles, vectors, and matrices for use in OpenGL. It provides methods for processing keyboard and mouse input for camera movement, as well as methods for calculating and updating the camera's view matrix. This header file can be used in OpenGL projects that require a camera to navigate and view the 3D environment.

We define the Camera class with its constructor and member functions for processing input and updating the camera attributes. It includes standard headers like vector and GL headers like glew.h and glm.hpp.

The Camera class has attributes like Position, Front, Up, Right, WorldUp, Yaw, Pitch, MovementSpeed, MouseSensitivity, and Zoom. The camera class has two constructors: one that accepts vector values and another that accepts scalar values.

The header file also defines an enum named Camera_Movement, which contains possible options for camera movement, including FORWARD, BACKWARD, LEFT, and RIGHT.

The Camera class has several member functions to process inputs received from different input systems. The ProcessKeyboard() function processes input from any keyboard-like input system, and the ProcessMouseMovement() function processes input received from a mouse input system. The ProcessMouseScroll() function processes input received from a mouse scroll-wheel event.

The GetViewMatrix() function returns the view matrix calculated using Eular Angles and the LookAt Matrix. The updateCameraVectors() function calculates the front vector from the Camera's updated Eular Angles.

Overall, this header file defines a camera class that can process input from different input systems and update the camera attributes accordingly. It provides a way to handle camera movement and view matrix calculation for use in OpenGL applications.

Int this file, we only had to create the GetViewMatrix(). As mentioned above, this function returns the 4X4 view matrix with the help of the glm library, This

portion of code defines a member function named "GetViewMatrix()" that returns a 4x4 view matrix using the "glm" library.The view matrix is used in computer graphics to transform a 3D scene into 2D so that it can be displayed on a screen. It defines the position and orientation of the camera in the 3D space, as well as the direction the camera is looking at.

The function uses the "lookAt" function from the "glm" library to create the view matrix. It takes three parameters:
"this->Position" is the position of the camera in the 3D space.
"this->Position + this->Front" is the point the camera is looking at, which is calculated by adding the camera's front vector to its position vector.
"this->Up" is the camera's up vector, which defines the orientation of the camera.

The "lookAt" function creates a view matrix based on these parameters, which can be used to transform the scene into a 2D image from the camera's perspective. The view matrix is then returned by the "GetViewMatrix()" function.

## texture.frag:

The shader It takes in a 2D texture sample and applies a vertical tiling effect by repeating the texture along the y-axis. The resulting output color is stored in the "color" output variable and passed to the next stage of the rendering pipeline.

"color" is an output variable of type vec4, which represents the color of the current fragment being processed.
"UV" is an input variable of type vec2, which represents the texture coordinates of the current fragment being processed.
"myTextureSampler" is a uniform variable of type sampler2D, which represents the texture sampler used to access the texture data.

Implementation:
We calculate the new y-coordinate of the texture coordinate using the "mod" function to repeat the texture along the y-axis. The second argument of "mod" is 0.66666777, which means the texture will repeat every 2/3 of the texture height.
Use the "texture2D" function to sample the texture data using the modified texture coordinate. The result is a vec4 containing the color data of the sampled texel. Then store the resulting vec4 color data in the "color" output variable.

## texture.vs:

The purpose of this vertex shader is to define how the vertices of a 3D object should be transformed, based on the model, view, and projection matrices provided as input uniforms.
We declare input variables for the shader. position is a 3-component vector representing the position of the vertex in 3D space, and vertexUV is a 2-component vector representing the texture coordinates of the vertex.
The out keyword declares a variable UV as an output of the shader, which will be interpolated for each fragment. This variable will be used to pass the texture coordinates to the fragment shader.
The three uniform variables model, view, and projection are matrices that will be used to transform the vertices. model represents the model matrix, which defines the position, rotation, and scale of the object. view represents the view matrix, which defines the position and orientation of the camera. projection represents the projection matrix, which defines how the 3D scene should be projected onto a 2D surface.
In the main() function of the shader, the gl_Position variable is set to the final position of the vertex, after being transformed by the model, view, and projection matrices. This is done by multiplying the position vector by the matrices, and then adding a 1.0 as the fourth component of the resulting vector.
The UV output variable is set to the vertexUV input variable, which will be interpolated and passed to the fragment shader.
We compute the final position of the vertex in clip space with the help of gl_Position = projection * view * model * vec4(position, 1.0). This is achieved by multiplying the vertex's position by a series of transformation matrices: model, view, and projection.
By multiplying the vertex's position by these matrices in the specified order, we get its final position in clip space, which is a 4D homogeneous coordinate.
The line UV = vertexUV; is used to pass the texture coordinates of the vertex to the fragment shader. These texture coordinates are interpolated for each fragment, allowing us to sample the texture at the correct location for each pixel.
The UV variable is an out variable, meaning it will be passed as an input to the fragment shader.

## Main.cpp:

The program initializes GLFW and creates a window using GLFW with a specific size and title. It also initializes GLEW to retrieve OpenGL function pointers and extensions. After that, the program defines the viewport and enables depth testing.

Next, we load the shader files "texture.vs" and "texture.frag" using the "Shader" class. These shaders are used to render the cube with the texture.
Then, the program loads the texture file "texture.dds" using the "loadDDS" function, The texture is then assigned a texture ID using the "glGetUniformLocation" function and the "textureShader.Program" program.
The vertex data for the cube is defined in the "vertices" array. The array defines the positions of each vertex for the cube, which is comprised of 12 triangles.
The next step is to create a vertex buffer object (VBO) to store the vertex data and upload it to the GPU. Additionally, a texture coordinate buffer object is created to store the texture coordinates for each vertex.
In the rendering loop, the program binds the VBO, TCBO and texture, sets the uniforms in the shader, and draws the cube with the texture.
The program ends by cleaning up GLFW and exiting the application**.**

The glGenBuffers function generates a buffer object name for a new VBO, which is passed as the first parameter, and stores it in the UVBO variable, which is a reference to the VBO. The second line binds the newly created buffer object to the current context's GL_ARRAY_BUFFER target, so that subsequent OpenGL calls operate on this buffer. The glBufferData function then allocates and initializes a buffer's data store with the size, data, and usage flags specified in the parameters. In this case, the data being loaded is a set of UV coordinates, which are 2D texture coordinates used to map an image to a 3D model.
The glEnableVertexAttribArray call enables the vertex attribute array with index 1, which is the UV attribute in this case. The glVertexAttribPointer call then sets up the data format for the UV attribute, specifying that it has two components (u and v), is of type GL_FLOAT, has no normalization, has a stride of 0 (implying tightly packed data), and has an offset of 0 bytes.
The glBindTexture function binds a named texture object (passed as the second parameter) to a texture target (in this case, GL_TEXTURE_2D). This call sets up the texture object to be used in subsequent OpenGL calls.