111550139

郭芷安

ICG HW2 report

Create shader

```
unsigned int createShader(const string &filename, const string &type) {
    GLuint shader;
    if (type == "frag")
        shader = glCreateShader(GL_FRAGMENT_SHADER);
    else if (type == "vert")
        shader = glCreateShader(GL_VERTEX_SHADER);

    std::ifstream ShaderFile;
    std::stringstream ShaderStream;
    std::string code;
    try {
        ShaderFile.open(filename);
        ShaderStream << ShaderFile.rdbuf();
        ShaderFile.close();
        code = ShaderStream.str();
    }
    catch (std::ifstream::failure& e) {
        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESSFULLY_READ: " << e.what() << "\n";
    }
    const char* shaderCode = code.c_str();
    glShaderSource(shader, 1, &shaderCode, NULL);
    glCompileShader(shader);
    // debug
    int success;
    char infoLog[512];
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if (!success) {
        glGetShaderInfoLog(shader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::" << type << "::COMPILATION_FAILED\n" << infoLog << std::endl;
        std::cout << "Shader code:\n" << code << std::endl;
    } else {
        std::cout << "Shader " << type << " compiled successfully" << std::endl;
    }

    return shader;
}
```

The task in this part to create fragment shader and vertex shader. If type is vert, then use glCreateShader to create vertex shader, and if type is frag, then use glCreateShader to create fragment shader. After this, load the shader source from vertexShader.vert or fragmentShader.frag, bind the source with the created shader, and compile it. In the end, return the compiled shader.

Create program

```
unsigned int createProgram(unsigned int vertexShader, unsigned int fragmentShader) {
    GLuint ID = glCreateProgram();
    glAttachShader(ID, vertexShader);
    glAttachShader(ID, fragmentShader);
    glLinkProgram(ID);

    // debug
    int success;
    char infoLog[512];
    glGetProgramiv(ID, GL_LINK_STATUS, &success);
    if (!success) {
        glGetProgramInfoLog(ID, 512, NULL, infoLog);
        std::cout << "ERROR::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
    } else {
        std::cout << "Shader program linked successfully" << std::endl;
    }

    // check valid
    glValidateProgram(ID);
    glGetProgramiv(ID, GL_VALIDATE_STATUS, &success);
    if (!success) {
        glGetProgramInfoLog(ID, 512, NULL, infoLog);
        std::cout << "ERROR::PROGRAM::VALIDATION_FAILED\n" << infoLog << std::endl;
    }

    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);

    return ID;
}
```

The Shader program is to link shader objects to a executable GPU processing unit. To create a shader program, we use glCreateProgram and get the returned program id. Use the program id to attach the program with vertex shader and fragment shader. And link the program. After the above actions are done, we can delete vertexShader and fragmentShader using glDeleteShader, because the shaders are no longer needed.

Create VAO and VBO

```
unsigned int modelVAO(Object &model) {
    vector<float> positions = model.positions;
    vector<float> normals = model.normals;
    vector<float> texcoords = model.texcoords;

    unsigned int VBO[3];
    unsigned int VAO;
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);
    glGenBuffers(3, VBO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT)  * (positions.size()), &(positions[0]), GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 3, 0);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT)  * (normals.size()), &(normals[0]), GL_STATIC_DRAW);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 3, 0);
    glEnableVertexAttribArray(1);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    glBindBuffer(GL_ARRAY_BUFFER, VBO[2]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT)  * (texcoords.size()), &(texcoords[0]), GL_STATIC_DRAW);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 2, 0);
    glEnableVertexAttribArray(2);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    glBindVertexArray(0);
    // glDrawArrays();
    return VAO;
}
```

Observe Object.h in directory headers, we can know that an object contains three parts of data, position, normal and textcoords. So, use glGenVertexArrays to generate VAO and bind it. use glGenBuffers to generate VBO and bind it first, then fill the buffers with data using glBufferData. Use glVertexAttribPointer to link the buffer with the vertex shader input, specify the index of generic vertex attribute, number of components per generic vertex attribute, data type of each component in the array, set false to fixed-point data normalization, byte offset between consecutive generic vertex attributes, and the offset of the first component of the first generic vertex attribute. After that, use glEnableVertexAttribArray to enable the vertex attribute, told the VAO to read the data from the specified VBO when rendering.

Load texture

```cpp
unsigned int loadTexture(const string &filename) {
    GLuint texture;
    glEnable(GL_TEXTURE_2D);
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    int width, height, nrChannels;
    if (filename == "..\\..\\src\\asset\\texture\\earth.jpg") stbi_set_flip_vertically_on_load(1);
    unsigned char* data = stbi_load(filename.c_str(), &width, &height, &nrChannels, 0);
    if (data) {
        std::cout << "texture loaded successfully: " << filename << std::endl;
        std::cout << "width: " << width << ", height: " << height << ", channels: " << nrChannels << std::endl;
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
        stbi_image_free(data);
    }
    else {
        std::cout << "ERROR::TEXTURE::FAILED_TO_LOAD_TEXTURE: " << filename << std::endl;
        stbi_image_free(data);
    }
    return texture;
}
```

The goal of this part is to load textures of the object. Use glEnable to enable the texture, and use glGenTextures to store the textures generated in an unsigned int array, and use glBindTexture to bind the texture. Let a unsigned char pointer data store the texture data loaded using stbi_load, and use glTexImage2D to generate two-dimensional texture image.

Draw model

```
void drawModel(Render renPara) {
    glUseProgram(shaderProgram);
    glActiveTexture(GL_TEXTURE0);
    if (renPara.modelName == "plane") glBindTexture(GL_TEXTURE_2D, airplaneTexture);
    else if (renPara.modelName == "earth") glBindTexture(GL_TEXTURE_2D, earthTexture);
    glUniformMatrix4fv(renPara.modelLoc, 1, GL_FALSE, glm::value_ptr(renPara.model));
    glUniformMatrix4fv(renPara.viewLoc, 1, GL_FALSE, glm::value_ptr(renPara.view));
    glUniformMatrix4fv(renPara.projLoc, 1, GL_FALSE, glm::value_ptr(renPara.projection));
    glUniform1f(renPara.squeezLoc, glm::radians(squeezeFactor));
    glUniform3fv(renPara.rainbowLoc, 1, glm::value_ptr(renPara.color));
    glUniform1i(renPara.useRainbowColorLoc, renPara.useRainbow);
    glUniform1i(renPara.useSqueezeLoc, renPara.useSqueeze);
    glUniform1i(renPara.useHeliLoc, renPara.useHeli);
    glUniform1f(renPara.textureLoc, 0);
    if (renPara.modelName == "plane") {
        glBindVertexArray(airplaneVAO);
        glDrawArrays(GL_TRIANGLES, 0, airplaneObject->positions.size());
    }
    else if (renPara.modelName == "earth") {
        glBindVertexArray(earthVAO);
        glDrawArrays(GL_TRIANGLES, 0, earthObject->positions.size());
    }
    else {  // - draw cube
        glBindVertexArray(cubeVAO);
        glDrawArrays(GL_TRIANGLES, 0, cubeObject->positions.size());
    }
    glBindVertexArray(0);
}
```

```
typedef struct Render{
    string modelName;
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 projection;
    bool useRainbow;
    bool useSqueeze;
    bool useHeli;
    glm::vec3 color;
    GLint modelLoc, viewLoc, projLoc, squeezLoc, rainbowLoc, textureLoc, useRainbowColorLoc, useSqueezeLoc, useHeliLoc;
}Render;
```

To reduce the redundancy, I define a function drawModel to handle the drawing action. And the parameter Render renPara is defined in the following picture. Every time we want to draw model, we only need to change the modelName, model, useRainbow, useSqueeze, useHeli, and color of renPara, and let drawModel to handle the remaining processes.

In drawModel function, we use glUseProgram to specify the shaderprogram we want to use, anduse glActiveTexture to active the texture, and bind texture based on the modelName. After that, use glUniform to set uniform parameters to let main process communicate with the shader program. Finally, use glBindVertexArray to bind the vertexArray of different models, and use glDrawArrays to draw the model.

Problems I met

1. At first, I failed to render earth and plane, the only thing on my screen is black and white blocks, so I add debug sections after every action, to see if the build is success. After that, I found that the problem occurred due to that when using glGetUniformLocation to load the uniform location, nothing is loaded, and this is because I failed to compile the shader program. So I read the code in shader code carefully, and found that I misspelled the argument. After that, everything worked.
2. When writing bonus, I want to reduce the redundancy of drawing models, because the helicopter contains too many cubes. I tried to pass the parameters separately to drawModel function, but when I start to build the project, there always an error said that the reference to the function is not defined. To solve this problem, I wrap all the arguments in a struct, and pass only the struct to the drawModel function. This solve the error, and also made the function easier to use.