Underwater Volcano - Delivery 2 Report

Demonstration video: https://www.youtube.com/watch?v=53ZmWnFnnbc

`main.cpp`

Main implementation:

- The data structure of each mesh and its children meshes
- `load_mesh()` method would load up one mesh, and its own children meshes, as well as build hierarchical relationships. Because until now, the only hierarchical animation contained is the fish which is quite simple, it only has two layer: one for parent mesh (fish body), and one layer for all the other meshes(fin, head). That is why the code only map the hierarchy to two layer relationship.
- `generateObjectBufferMesh()` method would trigger the load process of all the models needed, and it would call a internal lambda function `SetUpModelBuffers()` to set up buffers for each mesh and recursively set up for its children meshes.
- `display()` function would pass the `view, model, proj` for each mesh to the shader. And if it's the smoke mesh, then it would pass more variables to shader such as smoke color, camera position to adjust the transparency of the smoke
- `updateScene()` would :
  - update the rotation values of different meshes along the hierarchy, so they could look more realistically
  - update each fish's translation with its own direction.
  - update camera position by checking if listened key being pressed, in this way, different key can be pressed together to produce a combined direction.

```
// Windows includes(For Time, IO, etc.)
#include <windows.h>
#include <mmsystem.h>
#include <iostream>
#include <string>
#include <stdio.h>
#include <math.h>
#include <vector> // STL dynamic memory.
#include <unordered_map>
#include <cstdlib>
#include <ctime>
```

```
// OpenGL includes
#include <GL/glew.h>
#include <GL/freeglut.h>

// Assimp includes
#include <assimp/cimport.h> // scene importer
#include <assimp/scene.h> // collects data
#include <assimp/postprocess.h> // various extra operations

// Project includes
#include "maths_funcs.h"
#include <filesystem>
namespace fs = std::filesystem;

#include <glm.hpp>
#define GLM_ENABLE_EXPERIMENTAL
#include <gtx/string_cast.hpp>
#include <gtc/type_ptr.hpp>
#include <gtc/matrix_transform.hpp>
#include <gtc/quaternion.hpp>

#include "CameraControl.hpp"
#include <functional>

/*----------------------------------------------------------------------------
MESH TO LOAD
----------------------------------------------------------------------------*/
// this mesh is a dae file format but you should be able to use any other format too
// put the mesh in your project directory, or provide a filepath for it here
#define TERRAIN_MESH "Assets/terrain.fbx"
#define SMOKE_MESH "Assets/Smoke.fbx"
#define ANIMATION_FOLDER "Assets/animationModels/"

#define TERRIAN 1
#define SKELETON 0
#define COLORANIMATION 0

/*----------------------------------------------------------------------------
----------------------------------------------------------------------------*/
using namespace std;

#pragma region SimpleTypes

typedef struct ModelData
{
```

```
    size_t mPointCount = 0;
    GLuint mVao = 0;
    vector<glm::vec3> mVertices;
    vector<glm::vec3> mNormals;
    vector<glm::vec4> mColors;
    vector<vec2> mTextureCoords;
    glm::mat4 mLocalTransform;
    vector<ModelData> mChildMeshes;

} ModelData;
#pragma endregion SimpleTypes



auto startTime = std::chrono::high_resolution_clock::now();

GLuint terrianShaderProgramID;

// all the meshes in the scene
ModelData volcano_terrian_mesh;
ModelData smoke_mesh;
vector<ModelData> animation_meshes;

int width = 1440;
int height = 720;

// fish hierarchical mesh rotation for each fish part
GLfloat rotate_head = 0.0f, rotate_fin = 0.0f, rotate_body = 0.f;

// whole translation for each fish model
vector<glm::vec3> fish_translations(9, glm::vec3(0.0f));

#pragma region MESH LOADING
/*----------------------------------------------------------------------------
MESH LOADING FUNCTION
----------------------------------------------------------------------------*/

glm::mat4 ConvertToGLMMat4(const aiMatrix4x4& aiMat) {
    return glm::mat4(
        aiMat.a1, aiMat.b1, aiMat.c1, aiMat.d1,
        aiMat.a2, aiMat.b2, aiMat.c2, aiMat.d2,
        aiMat.a3, aiMat.b3, aiMat.c3, aiMat.d3,
        aiMat.a4, aiMat.b4, aiMat.c4, aiMat.d4
    );
}
```

```
ModelData load_mesh(const char* file_name, bool b_hierarchical_mesh) {
    ModelData modelData;

    /* Use assimp to read the model file, forcing it to be read as    */
    /* triangles. The second flag (aiProcess_PreTransformVertices) is */
    /* relevant if there are multiple meshes in the model file that   */
    /* are offset from the origin. This is pre-transform them so      */
    /* they're in the right position.                                 */

    // if the model contains hierarchical meshes, then no aiProcess_PreTransformVert
    // because it would flatten the hierarchy
    unsigned int pFlags = b_hierarchical_mesh ? (aiProcess_FlipUVs | aiProcess_GenSm
        : (aiProcess_PreTransformVertices | aiProcess_GlobalScale);

    const aiScene* scene = aiImportFile(file_name, pFlags | aiProcess_Triangulate);

    if (!scene) {
        fprintf(stderr, "ERROR: reading mesh %s\n", filesystem::path(file_name).c_st
        return modelData;
    }

    printf("  %i materials\n", scene->mNumMaterials);
    printf("  %i meshes\n", scene->mNumMeshes);
    printf("  %i textures\n", scene->mNumTextures);
    printf("  %i animation\n", scene->mAnimations);

    for (unsigned int m_i = 0; m_i < scene->mNumMeshes; m_i++) {
        const aiMesh* mesh = scene->mMeshes[m_i];
        printf("    %i vertices in mesh\n", mesh->mNumVertices);
        printf("    %i bones in mesh\n", mesh->mNumBones);

        ModelData* modelPtr = nullptr;
        if (m_i == 0) // load root mesh
        {
            modelPtr = &modelData;
            modelPtr->mLocalTransform = glm::mat4(1.0f);
        }
        else if (m_i >= 1) // load child mesh
        {
            modelData.mChildMeshes.push_back(ModelData());
            modelPtr = &modelData.mChildMeshes.back();

            // each child mesh has a local transform from its parent
            if (b_hierarchical_mesh)
            {
                modelPtr->mLocalTransform = ConvertToGLMMat4(scene->mRootNode->mChil
```

```
            }
        }

        // Load vertices, normals, colors, and texture coordinates
        for (unsigned int v_i = 0; v_i < mesh->mNumVertices; v_i++) {
            if (mesh->HasPositions()) {
                const aiVector3D* vp = &(mesh->mVertices[v_i]);
                modelPtr->mVertices.push_back(glm::vec3(vp->x, vp->y, vp->z));
            }
            if (mesh->HasVertexColors(0)) { // Ensure vertex colors exist
                aiColor4D maskColor = mesh->mColors[0][v_i];
                modelPtr->mColors.push_back(glm::vec4(maskColor.r, maskColor.g, mask
            }
            else {
                modelPtr->mColors.push_back(glm::vec4(1.0, 1.0, 1.0, 0.0)); // Defau
            }
            if (mesh->HasNormals()) {
                const aiVector3D* vn = &(mesh->mNormals[v_i]);
                modelPtr->mNormals.push_back(glm::vec3(vn->x, vn->y, vn->z));
            }
            if (mesh->HasTextureCoords(0)) {
                const aiVector3D* vt = &(mesh->mTextureCoords[0][v_i]);
                modelPtr->mTextureCoords.push_back(vec2(vt->x, vt->y));
            }
            if (mesh->HasTangentsAndBitangents()) {
                /* You can extract tangents and bitangents here              */
                /* Note that you might need to make Assimp generate this     */
                /* data for you. Take a look at the flags that aiImportFile  */
                /* can take.                                                 */
            }
        }

        modelPtr->mPointCount += mesh->mNumVertices;
    }

    aiReleaseImport(scene);
    cout << "finish load mesh\n";
    return modelData;
}

#pragma endregion MESH LOADING

// Shader Functions- click on + to expand
#pragma region SHADER_FUNCTIONS
char* readShaderSource(const char* shaderFile) {
    FILE* fp;
```

```
    fopen_s(&fp, shaderFile, "rb");

    filesystem::path p(shaderFile);

    if (fp == NULL) {
        cout << "application current path " << filesystem::current_path() << endl;
        cout << "file path " << filesystem::absolute(p) << endl;
        return NULL;
    }

    fseek(fp, 0L, SEEK_END);
    long size = ftell(fp);

    fseek(fp, 0L, SEEK_SET);
    char* buf = new char[size + 1];
    fread(buf, 1, size, fp);
    buf[size] = '\0';

    fclose(fp);

    return buf;
}


static void AddShader(GLuint ShaderProgram, const char* pShaderText, GLenum ShaderTy
{
    // create a shader object
    GLuint ShaderObj = glCreateShader(ShaderType);

    if (ShaderObj == 0) {
        std::cerr << "Error creating shader..." << std::endl;
        std::cerr << "Press enter/return to exit..." << std::endl;
        std::cin.get();
        exit(1);
    }
    const char* pShaderSource = readShaderSource(pShaderText);

    // Bind the source code to the shader, this happens before compilation
    glShaderSource(ShaderObj, 1, (const GLchar**)&pShaderSource, NULL);
    // compile the shader and check for errors
    glCompileShader(ShaderObj);
    GLint success;
    // check for shader related errors using glGetShaderiv
    glGetShaderiv(ShaderObj, GL_COMPILE_STATUS, &success);
    if (!success) {
        GLchar InfoLog[1024] = { '\0' };
```

```
        glGetShaderInfoLog(ShaderObj, 1024, NULL, InfoLog);
        std::cerr << "Error compiling "
            << (ShaderType == GL_VERTEX_SHADER ? "vertex" : "fragment")
            << " shader program: " << InfoLog << std::endl;
        std::cerr << "Press enter/return to exit..." << std::endl;
        std::cin.get();
        exit(1);
    }
    // Attach the compiled shader object to the program object
    glAttachShader(ShaderProgram, ShaderObj);
}


GLuint CompileShaders()
{
    //Start the process of setting up our shaders by creating a program ID
    //Note: we will link all the shaders together into this ID

    auto linkShader = [](GLuint programID)
        {
            GLint Success = 0;
            GLchar ErrorLog[1024] = { '\0' };
            // After compiling all shader objects and attaching them to the program,
            glLinkProgram(programID);
            // check for program related errors using glGetProgramiv
            glGetProgramiv(programID, GL_LINK_STATUS, &Success);
            if (Success == 0) {
                glGetProgramInfoLog(programID, sizeof(ErrorLog), NULL, ErrorLog);
                std::cerr << "Error linking shader program: " << ErrorLog << std::en
                std::cerr << "Press enter/return to exit..." << std::endl;
                std::cin.get();
                exit(1);
            }
            // program has been successfully linked but needs to be validated to che
            glValidateProgram(programID);
            // check for program related errors using glGetProgramiv
            glGetProgramiv(programID, GL_VALIDATE_STATUS, &Success);
            if (!Success) {
                glGetProgramInfoLog(programID, sizeof(ErrorLog), NULL, ErrorLog);
                std::cerr << "Invalid shader program: " << ErrorLog << std::endl;
                std::cerr << "Press enter/return to exit..." << std::endl;
                std::cin.get();
                exit(1);
            }
            // Finally, use the linked shader program
            // Note: this program will stay in effect for all draw calls until you r
```

```
            glUseProgram(programID);
        };

    terrianShaderProgramID = glCreateProgram();
    if (terrianShaderProgramID == 0) {
        std::cerr << "Error creating shader program..." << std::endl;
        std::cerr << "Press enter/return to exit..." << std::endl;
        std::cin.get();
        exit(1);
    }
    AddShader(terrianShaderProgramID, "simpleVertexShader.txt", GL_VERTEX_SHADER);
    AddShader(terrianShaderProgramID, "simpleFragmentShader.txt", GL_FRAGMENT_SHADER
    linkShader(terrianShaderProgramID);
    return 1;
}
#pragma endregion SHADER_FUNCTIONS

// VBO Functions - click on + to expand



#pragma region VBO_FUNCTIONS

vector<string> GetAllAnimationModelPath()
{
    vector<string> animationModelPaths;
    // Iterate through the animation model folder and store the paths in a vector
    try {
        for (const auto& entry : fs::directory_iterator(ANIMATION_FOLDER)) {
            // Check if the entry is a file (not a directory)
            if (entry.is_regular_file()) {
                animationModelPaths.push_back(entry.path().string());
                std::cout << entry.path().filename() << std::endl;
            }
        }
    }
    catch (const fs::filesystem_error& e) {
        std::cerr << "Filesystem error: " << e.what() << std::endl;
    }
    catch (const std::exception& e) {
        std::cerr << "General exception: " << e.what() << std::endl;
    }
    return animationModelPaths;
}

void generateObjectBufferMesh() {
```

```
/*-----------------------------------------------------------------------
LOAD MESH HERE AND COPY INTO BUFFERS
-----------------------------------------------------------------------*/

//Note: you may get an error "vector subscript out of range" if you are using th
//Might be an idea to do a check for that before generating and binding the buff


// load terrain mesh
volcano_terrian_mesh = load_mesh(TERRAIN_MESH, false);

// load smoke mesh
smoke_mesh = load_mesh(SMOKE_MESH, false);

// get all the animation model paths
vector<string> animationModelPaths = GetAllAnimationModelPath();

// Load each animation model
for (size_t i = 0; i < animationModelPaths.size(); i++)
{
    animation_meshes.push_back(load_mesh(animationModelPaths[i].c_str(), true));
}

// Set up the VAO and VBOs for terrain and all animation models
GLuint loc1 = glGetAttribLocation(terrianShaderProgramID, "vertex_position");
GLuint loc2 = glGetAttribLocation(terrianShaderProgramID, "vertex_normal");
if (loc1 == -1 || loc2 == -1)
{
    std::cerr << "Error getting attribute location" << std::endl;
    exit(1);
}

function<void(ModelData&)> SetUpModelBuffers = [&](ModelData& model) {
    glGenVertexArrays(1, &model.mVao);
    glBindVertexArray(model.mVao);
    GLuint vbos[2] = { 0, 0 };
    glGenBuffers(2, vbos);

    // Vertices VBO
    glBindBuffer(GL_ARRAY_BUFFER, vbos[0]);
    glBufferData(GL_ARRAY_BUFFER, model.mPointCount * sizeof(glm::vec3), model.m
    glVertexAttribPointer(loc1, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), (void*
    glEnableVertexAttribArray(loc1);

    // Normals VBO
    glBindBuffer(GL_ARRAY_BUFFER, vbos[1]);
```

```
        glBufferData(GL_ARRAY_BUFFER, model.mPointCount * sizeof(glm::vec3), model.m
        glVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), (void*
        glEnableVertexAttribArray(loc2);

        glBindVertexArray(0); // unbind VAO
        glBindBuffer(GL_ARRAY_BUFFER, 0); // unbind VBO

        for (int j = 0; j < model.mChildMeshes.size(); ++j)
        {
            SetUpModelBuffers(model.mChildMeshes[j]);
        }
    };

    glUseProgram(terrianShaderProgramID);
    SetUpModelBuffers(volcano_terrian_mesh);
    SetUpModelBuffers(smoke_mesh);
    for (int i = 0; i < animationModelPaths.size(); ++i) {
        SetUpModelBuffers(animation_meshes[i]);
    }

    cout << "finish generate object buffer mesh\n";
}
#pragma endregion VBO_FUNCTIONS

void renderBitmapText(float x, float y, void* font, const char* text) {
    glRasterPos2f(x, y);
    while (*text) {
        glutBitmapCharacter(font, *text);
        text++;
    }
}


void display() {

    // tell GL to only draw onto a pixel if the shape is closer to the viewer
    glEnable(GL_DEPTH_TEST); // enable depth-testing
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glDepthFunc(GL_LESS); // depth-testing interprets a smaller value as "closer"
    glClearColor(0.004f, 0.361f, 0.588f, 0.8f); // background color to blue
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glUseProgram(terrianShaderProgramID);
    //Declare your uniform variables that will be used in your shader
    int matrix_location = glGetUniformLocation(terrianShaderProgramID, "model");
```

```
    int view_mat_location = glGetUniformLocation(terrianShaderProgramID, "view");
    int proj_mat_location = glGetUniformLocation(terrianShaderProgramID, "proj");
    int is_smoke_location = glGetUniformLocation(terrianShaderProgramID, "isSmoke");

    // projection matrix
    mat4 persp_proj = perspective(45.0f, (float)width / (float)height, 0.1f, 1000.0f

    // Update the view matrix by using the camera's position and orientation
    glm::vec3 forward(0.0);
    forward.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    forward.y = sin(glm::radians(pitch));
    forward.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
    forward = glm::normalize(forward);
    glm::vec3 cameraTarget = cameraPosition + forward;
    glm::mat4 view = glm::lookAt(cameraPosition, cameraTarget, glm::vec3(0.0f, 1.0f,



    // update terrain uniforms & draw
    function<void(const ModelData&, glm::mat4&, bool)> UpdateNormalMeshUniforms = [&
        glBindVertexArray(mesh.mVao);
        glUniformMatrix4fv(proj_mat_location, 1, GL_FALSE, persp_proj.m);
        glUniformMatrix4fv(view_mat_location, 1, GL_FALSE, glm::value_ptr(view));
        glUniformMatrix4fv(matrix_location, 1, GL_FALSE, glm::value_ptr(modelMatrix)
        glUniform1i(is_smoke_location, isSmoke);
        glDrawArrays(GL_TRIANGLES, 0, static_cast<GLsizei>(mesh.mPointCount));
    };

    glm::mat4 model(1.0f);
    // Draw terrain
    UpdateNormalMeshUniforms(volcano_terrian_mesh, model, false);

    // Draw smoke
    glUniform3f(glGetUniformLocation(terrianShaderProgramID, "smokeColor"), 0.5f, 0.
    glUniform1f(glGetUniformLocation(terrianShaderProgramID, "density"), 0.01f);
    glUniform3f(glGetUniformLocation(terrianShaderProgramID, "viewPos"), cameraPosit
    UpdateNormalMeshUniforms(smoke_mesh, model, true);

    // Draw animation models
    for (int i = 0; i < animation_meshes.size(); ++i)
    {
        // translate the root mesh
        glm::mat4 rootMesh = glm::mat4(1.f);
        rootMesh = glm::rotate(rootMesh, glm::radians(rotate_body), glm::vec3(0.0f,
        rootMesh = glm::translate(rootMesh, fish_translations[i]);
        UpdateNormalMeshUniforms(animation_meshes[i], rootMesh, false);
```

```
        // body rotation not effect the child meshes
        rootMesh = glm::mat4(1.f);
        rootMesh = glm::translate(rootMesh, fish_translations[i]);

        for(int j = 0; j < animation_meshes[i].mChildMeshes.size(); ++j)
        {
            // rotate the child mesh
            auto& childMesh = animation_meshes[i].mChildMeshes[j];
            glm::mat4 child(1.0);
            if (j == 0) // first child mesh is the head
            {
                child = glm::rotate(child, glm::radians(rotate_head), glm::vec3(0.0f
            }
            else // second child mesh is the fin
            {
                child = glm::rotate(child, glm::radians(rotate_fin), glm::vec3(0.0f,
            }

            // update the child transform matrix by multiplying the root mesh transf
            child = rootMesh * animation_meshes[i].mChildMeshes[j].mLocalTransform *

            UpdateNormalMeshUniforms(childMesh, child, false);
        }
    }


    renderBitmapText(-1.0, 0.9, GLUT_BITMAP_HELVETICA_18, "The scene consists of a s
    renderBitmapText(-1.0, 0.85, GLUT_BITMAP_HELVETICA_18, "All the fishes are movin
    renderBitmapText(-1.0, 0.8, GLUT_BITMAP_HELVETICA_18, "Each Fish contains hierar
    renderBitmapText(-1.0, 0.75, GLUT_BITMAP_HELVETICA_18, "Key Control : W(forward)
    renderBitmapText(-1.0, 0.7, GLUT_BITMAP_HELVETICA_18, "Mouse Control : Left clic

    glutSwapBuffers();
}


const float PI = 3.14159265358979323846f;
const vector<glm::vec3> fish_translate_directions = {
    glm::vec3(0.0f, 0.0f, -0.1f),
    glm::vec3(0.1f, 0.0f, -0.1f),
    glm::vec3(0.1f, 0.0f, -0.1f),
    glm::vec3(-0.1f, 0.0f, -0.1f),
    glm::vec3(-0.1f, 0.0f, 0.0f),
    glm::vec3(0.1f, 0.0f, -0.1f),
    glm::vec3(0.1f, 0.0f, -0.1f),
```

```
    glm::vec3(-0.1f, 0.0f, 0.1f),
    glm::vec3(-0.1f, 0.0f, -0.1f),
};

void updateScene() {
    auto currentTime = std::chrono::high_resolution_clock::now();
    std::chrono::duration<float> elapsedTime = currentTime - startTime;
    float timeInSeconds = elapsedTime.count(); // Get elapsed time in seconds

    // Fin and head of a fish rotate in a sinusoidal pattern
    rotate_fin = 0.4f * sin(timeInSeconds);
    rotate_body = 0.4f * sin(timeInSeconds + PI / 4.0f);
    rotate_head = 0.4f * sin(timeInSeconds + PI / 2.0f);

    for(int i = 0; i < fish_translations.size(); ++i)
    {
        fish_translations[i] += fish_translate_directions[i] * 0.01f;
    }

    keyControl::updateCameraPosition();

    // Draw the next frame
    glutPostRedisplay();
}


void init()
{
    // Set up the shaders
    GLuint shaderProgramID = CompileShaders();
    // load mesh into a vertex buffer array
    generateObjectBufferMesh();

}


int main(int argc, char** argv) {

    // Set up the window
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(width, height);
    glutCreateWindow("Underwater volcano");

    // Tell glut where the display function is
    glutDisplayFunc(display);
```

```
    glutIdleFunc(updateScene);

    glutKeyboardFunc(keyControl::keypress);
    glutKeyboardUpFunc(keyControl::keyRelease);
    glutSpecialFunc(keyControl::specialKeypress);
    glutSpecialUpFunc(keyControl::specialKeyRelease);
    glutMouseFunc(keyControl::mouseButton);
    glutMotionFunc(keyControl::mouseMotion);

    // A call to glewInit() must be done after glut is initialized!
    GLenum res = glewInit();
    // Check for any errors
    if (res != GLEW_OK) {
        fprintf(stderr, "Error: '%s'\n", glewGetErrorString(res));
        return 1;
    }
    // Set up your objects and shaders
    init();
    // Begin infinite event loop
    glutMainLoop();
    return 0;
}
```

CameraControl.hpp

Main implementation:

- Default values of camera position and rotations to initialize the camera and also support to reset the camera position(Key: R) and rotation(click Right mouse)
- A map that records the keys being pressed, so it can combine all the inputs when updating the camera position.
- When updating the camera position, it would retrieve the forward, up, right directions of current camera, so that the position can be updating accordingly.
- Listen to mouse press and move so that user can adjust the camera rotation by pressing and dragging the left mouse.

```
#include <unordered_map>
#include <iostream>
#include <GL/glut.h>
#include <GL/freeglut.h>

#include <glm.hpp>
```

```cpp
#include <gtc/type_ptr.hpp>
#include <gtc/matrix_transform.hpp>
#include <gtc/quaternion.hpp>

using namespace std;

const glm::vec3 cameraDefaultPosition(-4.0f, 8.0f, 30.0f);
glm::vec3 cameraPosition = cameraDefaultPosition; // Camera position in world space

const float defaultYaw = -80.f;
const float defaultPitch = -5.f;
float yaw = defaultYaw;  // Horizontal rotation (around Y-axis)
float pitch = defaultPitch; // Vertical rotation (around X-axis)

// camera and keyboard, mouse input
int lastMouseX = 0, lastMouseY = 0;
float angleX = 0.0f, angleY = 0.0f;

namespace keyControl
{
    // record the state of the keyboard
    std::unordered_map<unsigned char, bool> keyState;
    bool isShiftPressed = false; // record is shift key pressed
    bool isTranslationTriggered = false; // record is translation triggered

    // update kep pressed
    void keypress(unsigned char key, int x, int y) {
        key = std::tolower(key); // only record lower case
        keyState[key] = true;
        isTranslationTriggered = true;
    }

    // update kep released
    void keyRelease(unsigned char key, int x, int y) {
        key = std::tolower(key);
        keyState[key] = false;
        isTranslationTriggered = false;
    }

    // update shift statue
    void specialKeypress(int key, int x, int y) {
        if (key == GLUT_KEY_SHIFT_L || key == GLUT_KEY_SHIFT_R) {
            isShiftPressed = true;
        }
    }
```

```
// update shift statue
void specialKeyRelease(int key, int x, int y) {
    if (key == GLUT_KEY_SHIFT_L || key == GLUT_KEY_SHIFT_R) {
        isShiftPressed = false;
    }
}

const float normalSpeed = 0.05f;
// update camera position
void updateCameraPosition() {

    if (!isTranslationTriggered)
        return;
    // update speed based on shift key
    float currentSpeed = isShiftPressed ? (normalSpeed * 2) : normalSpeed;

    // calculate the forward, right, up vectors of camera
    glm::vec3 forward(0.0);
    forward.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    forward.y = sin(glm::radians(pitch));
    forward.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
    forward = glm::normalize(forward);
    glm::vec3 right = glm::normalize(glm::cross(forward, glm::vec3(0.0f, 1.0f, 0
    glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);

    // update camera position
    if (keyState['w']) {
        cameraPosition += forward * currentSpeed;
    }
    if (keyState['s']) {
        cameraPosition -= forward * currentSpeed;
    }
    if (keyState['a']) {
        cameraPosition -= right * currentSpeed;
    }
    if (keyState['d']) {
        cameraPosition += right * currentSpeed;
    }
    if (keyState['q']) {
        cameraPosition += up * currentSpeed;
    }
    if (keyState['e']) {
        cameraPosition -= up * currentSpeed;
    }
    if (keyState['r']) { // reset camera position
        cameraPosition = cameraDefaultPosition;
```

```
        }

        cout << "cameraPosition: " << cameraPosition.x << " " << cameraPosition.y <<
    }

    // record the location of just pressed mouse
    void mouseButton(int button, int state, int x, int y) {
        if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
            lastMouseX = x;
            lastMouseY = y;
        }
        else if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN) {
            yaw = defaultYaw;
            pitch = defaultPitch;
        }
    }

    // update camera rotation angle accroding to mouse movement compared to last tim
    const float sensitivity = 0.3f;
    void mouseMotion(int x, int y) {
        int deltaX = x - lastMouseX;
        int deltaY = y - lastMouseY;

        lastMouseX = x;
        lastMouseY = y;

        yaw += deltaX * sensitivity;
        pitch -= deltaY * sensitivity;

        cout << "deltaX: " << deltaX << " deltaY: " << deltaY << endl;
        cout << "yaw: " << yaw << " pitch: " << pitch << endl;

        // Constrain the pitch to avoid gimbal lock
        if (pitch > 87.0f) pitch = 87.0f;
        if (pitch < -87.0f) pitch = -87.0f;

        glutPostRedisplay();
    }

};
```

Vertex shader

```
#version 440

in vec3 vertex_position;
in vec3 vertex_normal;

out vec4 EyeCoords;
out vec3 Normal;

uniform mat4 view;
uniform mat4 proj;
uniform mat4 model;

void main() {

    // Model-view matrix and normal transformation
    mat4 ModelViewMatrix = view * model;
    mat3 NormalMatrix = mat3(ModelViewMatrix);

    // Transform normal to view space and normalize it
    Normal = normalize(NormalMatrix * vertex_normal);

    // Transform vertex position to view space
    EyeCoords = ModelViewMatrix * vec4(vertex_position, 1.0);

    // Convert position to clip coordinates and pass along
    gl_Position = proj * view * model * vec4(vertex_position, 1.0);
}
```

Fragment Shader

There are 5 lights in total, one from the center of the volcano which is red, the other four are blue lights at the corners of the terrain. For most of the models, they will be lighted with the mentioned lightings to calculate the color, but as for the smoke, all the lightings are replaced with the color of smoke.

```
#version 330

in vec4 EyeCoords;
in vec3 Normal;

uniform vec3 smokeColor;          // Base color for fog
uniform float density;           // Density for fog effect
```

```
uniform vec3 viewPos;            // Camera position
uniform bool isSmoke;              // Flag to indicate fog vs. regular object

// Volcano and Sea Lights
vec4 VolcanoLightPosition = vec4(0.0, 2.0, 0.0, 1.0);
vec3 VolcanoLd = vec3(1.0, 1.0, 1.0);

vec4 SeaLightPosition1 = vec4(-25.0, -10.0, -25.0, 1.0);
vec4 SeaLightPosition2 = vec4(25.0, -10.0, 25.0, 1.0);
vec4 SeaLightPosition3 = vec4(-25.0, -10.0, 25.0, 1.0);
vec4 SeaLightPosition4 = vec4(25.0, -10.0, -25.0, 1.0);
vec3 SeaLd = vec3(1.0, 1.0, 1.0);

// Diffuse colors for volcano and sea lights
vec3 VolcanoKd = vec3(1.0, 0.0, 0.0);
vec3 SeaKd = vec3(0.004f, 0.361f, 0.588f);

vec3 CalcLightIntensity(vec4 LightPosition, vec3 normal, vec3 Kd, vec3 Ld, vec4 eyeC
    float distance = length(LightPosition.xyz - eyeCoords.xyz);
    float attenuation = 1.0 / (constant + linear * distance + quadratic * (distance
    vec3 s = normalize(vec3(LightPosition - eyeCoords));
    return Ld * Kd * max(dot(s, normal), 0.0) * attenuation;
}

void main() {
    vec3 LightIntensity;

    if (isSmoke) {
        // Smoke effect calculations
        float distance = length(viewPos - EyeCoords.xyz);
        float smokeFactor = exp(-density * distance);  // Exponential falloff

        // Calculate smoke lighting using the same light sources
        LightIntensity = CalcLightIntensity(VolcanoLightPosition, Normal, smokeColor
        LightIntensity += CalcLightIntensity(SeaLightPosition1, Normal, smokeColor,
        LightIntensity += CalcLightIntensity(SeaLightPosition2, Normal, smokeColor,
        LightIntensity += CalcLightIntensity(SeaLightPosition3, Normal, smokeColor,
        LightIntensity += CalcLightIntensity(SeaLightPosition4, Normal, smokeColor,

        // Apply smoke color blending based on smokeFactor
        vec3 finalSmokeColor = mix(smokeColor, LightIntensity, smokeFactor);
        gl_FragColor = vec4(finalSmokeColor, smokeFactor);
    } else {
        // Standard lighting for regular objects
        LightIntensity = CalcLightIntensity(VolcanoLightPosition, Normal, VolcanoKd,
        LightIntensity += CalcLightIntensity(SeaLightPosition1, Normal, SeaKd, SeaLd
```

```
        LightIntensity += CalcLightIntensity(SeaLightPosition2, Normal, SeaKd, SeaLd
        LightIntensity += CalcLightIntensity(SeaLightPosition3, Normal, SeaKd, SeaLd
        LightIntensity += CalcLightIntensity(SeaLightPosition4, Normal, SeaKd, SeaLd

        gl_FragColor = vec4(LightIntensity, 1.0);  // Solid alpha for regular object
    }
}
```