Game Project Code Demo

This document contains the mainn source code for three of my game projects. The overall sourscode and project structure can be found on my github repositry.

https://github.com/ynCAOr06/USC_Appication_Game_demo

Dropnumber_2048

Overview

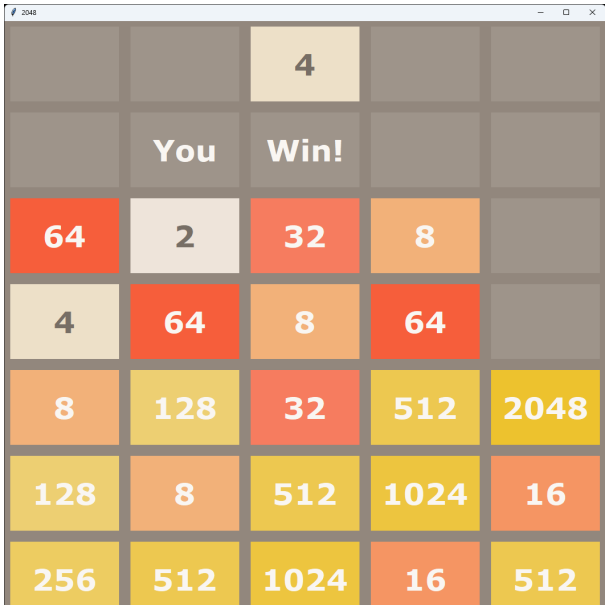
This is a python game project.

Run game.py to play the game.

```
Use "←" and "→" to decide the location of the current dropping number.
Use  "↓"  to drop the current number.
```

If a number block is dropped, the two adjacent identical number blocks will be synthesized into one block.

If the height of the number blocks exceed 5 blocks, then the game is over. If a number block 2048 is synthesized, then you win the game.



Code Display

game logic

```
# logic.py
import random
import constants as c

current_con = 0

def new_game():
    matrix = []
    for i in range(c.GRID_HEIGHT):
        matrix.append([0] * c.GRID_LEN)
    matrix = ran_init(matrix)
    return matrix

def ran_init(mat):
    num_list = [2, 4, 8, 16]
    val = random.choice(num_list)
    mat[0][2] = val
    return mat

def eli_check(mat):
    for i in range(len(mat)-1):
        for j in range(len(mat[0])-1):
            if (mat[i][j] == mat[i+1][j] or mat[i][j+1] == mat[i][j]) and mat[i]
[j] != 0:
                return True
    j = len(mat[0])-1
    for i in range(len(mat)-1):
        if mat[i][j] == mat[i+1][j] and mat[i][j] != 0:
            return True
    i = len(mat)-1
    for j in range(len(mat[0])-1):
        if mat[i][j+1] == mat[i][j] and mat[i][j] != 0:
            return True
    return False

def game_state(mat):
    # check for win cell
    for i in range(len(mat)):
        for j in range(len(mat[0])):
            if mat[i][j] == 2048:
                return 'win'
    # check whether the play area exceeds, if yes, consider grace conditions
    flag = False
    for j in range(c.GRID_LEN):
        if mat[1][j] != 0:
            flag = True
    if not flag:
        return 'not over'
```

```
        # if area exceeds, check whether if it is still playable on some point
        flag = False
        for j in range(c.GRID_LEN):
            if mat[1][j] != 0:
                if mat[2][j] != mat[1][j]:
                    flag = True
        if not flag:
            return 'not over'
        # or, very trickily, you can generate the above case:
        flag = False
        for j in range(c.GRID_LEN):
            if mat[1][j] != 0:
                if mat[0][j] != mat[1][j]:
                    flag = True
        if not flag:
            return 'not over'

        return 'lose'

    def move_down(mat, con = None):
        if con is None:
            for j in range(len(mat[0])):
                for i in range(len(mat)-1):
                    if mat[i][j] != 0 and mat[i+1][j] == 0:
                        for k in range(i+1):
                            if mat[i-k][j] != 0:
                                mat[i-k][j], mat[i-k+1][j] = mat[i-k+1][j], mat[i-k]
    [j]
        else:
            for i in range(len(mat)-1):
                if mat[i][con] != 0 and mat[i+1][con] == 0:
                    for k in range(i+1):
                        if mat[i-k][con] != 0:
                            mat[i-k][con], mat[i-k+1][con] = mat[i-k+1][con], mat[i-k]
    [con]
        return mat

    def merge(mat):
        while eli_check(mat):
            # 1st, merge column to reduce the height
            for i in range(len(mat)-1):
                for j in range(len(mat[0])-1):
                    if mat[i][j] == mat[i+1][j] and mat[i][j] != 0:
                        mat[i+1][j] *= 2
                        mat[i][j] = 0
                    if mat[i][j] == mat[i][j+1] and mat[i][j] != 0:
                        (con_1, con_2) = (j, j+1) if current_con == j+1 else (j+1, j)
                        mat[i][con_1] *= 2
                        mat[i][con_2] = 0
            j = len(mat[0])-1
            for i in range(len(mat) - 1):
                if mat[i][j] == mat[i+1][j] and mat[i][j] != 0:
                    mat[i + 1][j] *= 2
                    mat[i][j] = 0
```

```
        i = len(mat)-1
        for j in range(len(mat[0])-1):
            if mat[i][j] == mat[i][j+1] and mat[i][j] != 0:
                (con_1, con_2) = (j, j + 1) if current_con == j + 1 else (j + 1,
j)
                mat[i][con_1] *= 2
                mat[i][con_2] = 0

        mat = move_down(mat)
    return mat

def down(game):
    """ Release the generated block """
    is_down = True
    for i in range(c.GRID_LEN):
        if game[0][i] != 0:
            current_con = i
            game = move_down(game, con = i)

    game = merge(game)
    return game, is_down

def left(game):
    is_down = False
    for i in range(c.GRID_LEN):
        if game[0][i] != 0 and i > 0:
            game[0][i], game[0][i-1] = game[0][i-1], game[0][i]
    return game, is_down

def right(game):
    is_down = False
    for i in range(c.GRID_LEN):
        if game[0][i] != 0 and i < c.GRID_LEN-1:
            game[0][i], game[0][i+1] = game[0][i+1], game[0][i]
            # to avoid loop reassignments
            break
    return game, is_down
```

game initialization

```
#puzzle.py
from tkinter import Frame, Label, CENTER
import random
import logic
import constants as c



def gen():
    return random.randint(0, c.GRID_LEN - 1)
```

```
class GameGrid(Frame):
    def __init__(self):
        Frame.__init__(self)

        self.grid()
        self.master.title('2048')
        self.master.bind("<Key>", self.key_down)

        self.commands = {
            c.KEY_DOWN: logic.down,
            c.KEY_LEFT: logic.left,
            c.KEY_RIGHT: logic.right,
            c.KEY_DOWN_ALT1: logic.down,
            c.KEY_LEFT_ALT1: logic.left,
            c.KEY_RIGHT_ALT1: logic.right,
        }

        self.grid_cells = []
        self.init_grid()
        self.matrix = logic.new_game()
        self.history_matrixs = [self.matrix]
        self.update_grid_cells()

        self.mainloop()

    def init_grid(self):
        background = Frame(self, bg=c.BACKGROUND_COLOR_GAME, width=c.SIZE,
height=c.SIZE)
        background.grid()

        for i in range(c.GRID_HEIGHT):
            grid_row = []
            for j in range(c.GRID_LEN):
                cell = Frame(
                    background,
                    bg=c.BACKGROUND_COLOR_CELL_EMPTY,
                    width=c.SIZE / c.GRID_LEN,
                    height=c.SIZE / c.GRID_HEIGHT
                )
                cell.grid(
                    row=i,
                    column=j,
                    padx=c.GRID_PADDING,
                    pady=c.GRID_PADDING
                )
                t = Label(
                    master=cell,
                    text="",
                    bg=c.BACKGROUND_COLOR_CELL_EMPTY,
                    justify=CENTER,
                    font=c.FONT,
                    width=5,
                    height=2)
                t.grid()
```

```
                grid_row.append(t)
            self.grid_cells.append(grid_row)

    def update_grid_cells(self):
        for i in range(c.GRID_HEIGHT):
            for j in range(c.GRID_LEN):
                new_number = self.matrix[i][j]
                if new_number == 0:
                    self.grid_cells[i][j].configure(text="",
bg=c.BACKGROUND_COLOR_CELL_EMPTY)
                else:
                    self.grid_cells[i][j].configure(
                        text=str(new_number),
                        bg=c.BACKGROUND_COLOR_DICT[new_number],
                        fg=c.CELL_COLOR_DICT[new_number]
                    )
        self.update_idletasks()

    def key_down(self, event):
        key = event.keysym
        print(event)
        if key == c.KEY_QUIT:
            exit()
        if key == c.KEY_BACK and len(self.history_matrixs) > 0:
            self.matrix = self.history_matrixs.pop()
            self.update_grid_cells()
            print('back on step total step:', len(self.history_matrixs))
        elif key in self.commands:
            self.matrix, is_down = self.commands[key](self.matrix)
            self.update_grid_cells()
            if is_down:
                self.matrix = logic.ran_init(self.matrix)
                # record last move
                self.history_matrixs.append(self.matrix)
                self.update_grid_cells()
                if logic.game_state(self.matrix) == 'win':
                    self.grid_cells[1][1].configure(text="You",
bg=c.BACKGROUND_COLOR_CELL_EMPTY)
                    self.grid_cells[1][2].configure(text="Win!",
bg=c.BACKGROUND_COLOR_CELL_EMPTY)
                elif logic.game_state(self.matrix) == 'lose':
                    self.grid_cells[1][1].configure(text="You",
bg=c.BACKGROUND_COLOR_CELL_EMPTY)
                    self.grid_cells[1][2].configure(text="Lose!",
bg=c.BACKGROUND_COLOR_CELL_EMPTY)


    def generate_next(self):
        index = (gen(), gen())
        while self.matrix[index[0]][index[1]] != 0:
            index = (gen(), gen())
        self.matrix[index[0]][index[1]] = 2

game_grid = GameGrid()
```

game grid

```
#game.py
import numpy as np
from tkinter import Frame, Label, CENTER
import random
import logic
import constants as c


def gen():
    return random.randint(0, c.GRID_LEN - 1)


class GameGrid(Frame):
    def __init__(self):
        Frame.__init__(self)

        self.grid()
        self.master.title('2048')
        self.master.bind("<Key>", self.key_down)

        self.commands = {
            c.KEY_DOWN: logic.down,
            c.KEY_LEFT: logic.left,
            c.KEY_RIGHT: logic.right,
            c.KEY_DOWN_ALT1: logic.down,
            c.KEY_LEFT_ALT1: logic.left,
            c.KEY_RIGHT_ALT1: logic.right,
        }

        self.grid_cells = []
        self.init_grid()
        self.matrix = logic.new_game()
        self.history_matrixs = [self.matrix]
        self.update_grid_cells()

        self.mainloop()

    def init_grid(self):
        background = Frame(self, bg=c.BACKGROUND_COLOR_GAME, width=c.SIZE,
height=c.SIZE)
        background.grid()

        for i in range(c.GRID_HEIGHT):
            grid_row = []
            for j in range(c.GRID_LEN):
                cell = Frame(
                    background,
                    bg=c.BACKGROUND_COLOR_CELL_EMPTY,
```

```
                    width=c.SIZE / c.GRID_LEN,
                    height=c.SIZE / c.GRID_HEIGHT
                )
                cell.grid(
                    row=i,
                    column=j,
                    padx=c.GRID_PADDING,
                    pady=c.GRID_PADDING
                )
                t = Label(
                    master=cell,
                    text="",
                    bg=c.BACKGROUND_COLOR_CELL_EMPTY,
                    justify=CENTER,
                    font=c.FONT,
                    width=5,
                    height=2)
                t.grid()
                grid_row.append(t)
            self.grid_cells.append(grid_row)

    def update_grid_cells(self):
        for i in range(c.GRID_HEIGHT):
            for j in range(c.GRID_LEN):
                new_number = self.matrix[i][j]
                if new_number == 0:
                    self.grid_cells[i][j].configure(text="",
bg=c.BACKGROUND_COLOR_CELL_EMPTY)
                else:
                    self.grid_cells[i][j].configure(
                        text=str(new_number),
                        bg=c.BACKGROUND_COLOR_DICT[new_number],
                        fg=c.CELL_COLOR_DICT[new_number]
                    )
        self.update_idletasks()

    def key_down(self, event):
        key = event.keysym
        print(event)
        if key == c.KEY_QUIT:
            exit()
        if key == c.KEY_BACK and len(self.history_matrixs) > 0:
            self.matrix = self.history_matrixs.pop()
            self.update_grid_cells()
            print('back on step total step:', len(self.history_matrixs))
        elif key in self.commands:
            self.matrix, is_down = self.commands[key](self.matrix)
            self.update_grid_cells()
            if is_down:
                self.matrix = logic.ran_init(self.matrix)
                # record last move
                self.history_matrixs.append(self.matrix)
                self.update_grid_cells()
                if logic.game_state(self.matrix) == 'win':
```

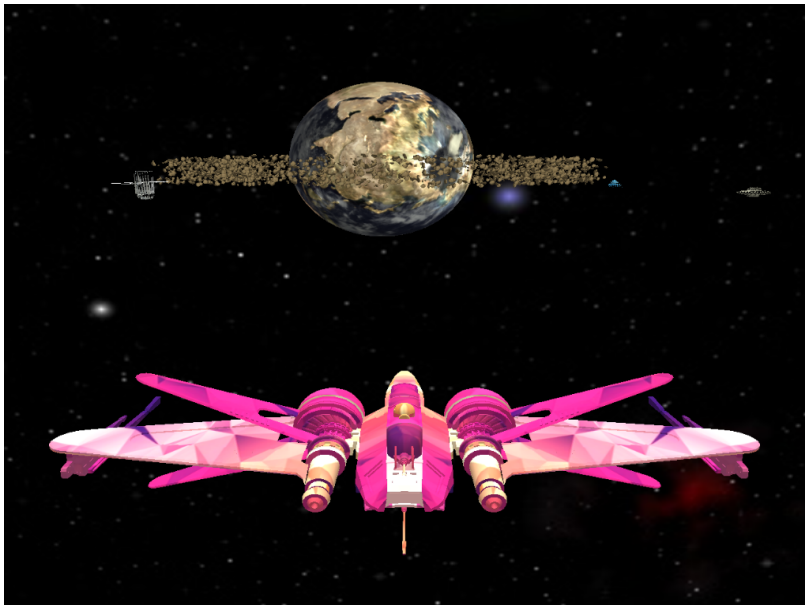
```
                    self.grid_cells[1][1].configure(text="You",
bg=c.BACKGROUND_COLOR_CELL_EMPTY)
                    self.grid_cells[1][2].configure(text="Win!",
bg=c.BACKGROUND_COLOR_CELL_EMPTY)
                elif logic.game_state(self.matrix) == 'lose':
                    self.grid_cells[1][1].configure(text="You",
bg=c.BACKGROUND_COLOR_CELL_EMPTY)
                    self.grid_cells[1][2].configure(text="Lose!",
bg=c.BACKGROUND_COLOR_CELL_EMPTY)


game_grid = GameGrid()
```

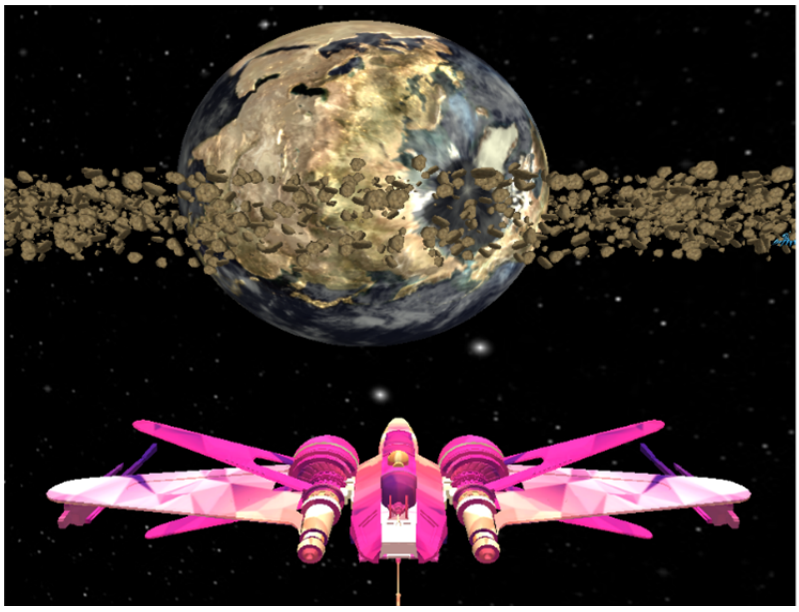
Space_Combat

Overview

This is a C++ game based on OpenGL.

Double Click !Game.exe to play the game.





Instructions:

Keyboard:

```
Key "Esc": exit
Key "UP": Move the spacecraft forward (With an acceleration process).
Key "DOWN": Move the spacecraft backwad (With an acceleration process).
Key "LEFT": Move the spacecraft to the left.
Key "RIGHT": Move the spacecraft to the right.
Key "Q": Make the spacecraft turn left (Similar to mouse control).
Key "E": Make the spacecraft turn right (Similar to mouse control).
```

```
Key "LEFT_SHIFT": Use the booster (Increase the maximum velocity andacceleration).
Key "W": Increase the brightness of directional light.
Key "S": Reduce the brightness of directional light.
```

Mouse:

```
Change camera position: Press and hold the left button then move the mouse.
```

Manipulation:

```
Features:
There are multiple objects rendered in the scene, including the planet, the
spacecraft, the vehicle, the asteroid ring, the spacestation and the ufo.

The planet is mapped by combining 3 textures, including the earth surface, the
earth cloud and the earth illumination.
(Reference: https://www.cgtrader.com/free-3d-models/space/planet/earth-and-
international-space-station)
Also, the earth normal is mapped onto it. It does self-rotation all the time.

The spacecraft could be controlled with the instructions below, to make it more
realistic, I've add the velocity and acceleration when moving forward and
backward, so it takes some time for it to reach its max speed. You can hold left
shift to increase the maximum velocity and acceleration.

The camera viewpoint is always behind and a little bit above the spacecraft.

The vehicle and the ufo are quite similar, the vehicle is given in the package and
the ufo is found online.
(Reference: https://www.cgtrader.com/free-3d-models/space/spaceship/free-
flying-saucer)
They are always doing self-rotation and are randomly jumping within a certain
horizontal area at a certain time.
When the spacecraft touches them, they will change their rotation speed and
texture color forever, and they stay put until the spacecraft moves away.

The asteroid ring is waiting to be added by teammate.

The spacestation is a static object loaded online.
(Reference: https://www.cgtrader.com/free-3d-models/space/other/overseer-
station-element-one--3)

There is a skybox showing the universe.

There are two light sources, one is directional light with sunlight (yellow)
color, one is point light with ocean (blue) color.
The directional light could be controlled with the instructions above.
```

Code Display

main game file

```
// main.cpp

#include "Dependencies/glew/glew.h"
#include "Dependencies/GLFW/glfw3.h"
#include "Dependencies/glm/glm.hpp"
#include "Dependencies/glm/gtc/matrix_transform.hpp"
#include "Dependencies/stb_image/stb_image.h"

#include "Shader.h"
//#include "Texture.h"

#include <iostream>
#include <fstream>
#include <vector>
#include <map>

GLuint programID[5];

float current_time;
float last_time;
float delta_time;
bool press_UP = false;
bool press_DOWN = false;
bool press_LEFT = false;
bool press_RIGHT = false;
bool press_Q = false;
bool press_E = false;
bool press_SHIFT = false;

float x_pos = 0.0f;
float y_pos = 0.0f;
float z_pos = 25.0f;
float spacecraft_rot = 180.0f;
float spacecraft_vel = 0.0f;
float spacecraft_max_vel = 5.0f;
float spacecraft_acc = 2.5f;
float spacecraft_brake_acc = 5.0f;
float spacecraft_dodge_vel = 2.5f;

float vehicle_pos_x = 0.0f;
float vehicle_pos_z = 0.0f;
float ufo_pos_x = 0.0f;
float ufo_pos_z = 0.0f;
float vehicle_rot_speed = 60.0f;
float ufo_rot_speed = 60.0f;
float jump_min_radius = 5.0f;
float jump_radius = 5.0f;
```

```
float jump_time = 3.0f;
float jump_time_ufo = 5.0f;
float wait_time = 0.0f;
float wait_time_ufo = 0.0f;

unsigned int rock_amount = 3000;
float rock_radius = 5.0f;
float rock_offset = 1.0f;

bool firstMouse = true;
bool change_view = false;
double lastX = 0.0;
double lastY = 0.0;
double yaw = 0.0;
double pitch = 0.0;
double xoffset;
double yoffset;
double sensitivity = 0.2;
float cam_radius = 3.0f;
glm::vec3 front;
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, 1.0f);

int planet_t_index = 0;
int spacecraft_t_index = 0;
float lightIntensity_R = 1.2f;
float lightIntensity_G = 1.2f;
float lightIntensity_B = 1.0f;
float ambient_R = 0.5f;
float ambient_G = 0.5f;
float ambient_B = 0.5f;

// screen setting
const int SCR_WIDTH = 800;
const int SCR_HEIGHT = 600;

// struct for storing the obj file
struct Vertex {
    glm::vec3 position;
    glm::vec2 uv;
    glm::vec3 normal;
};

struct Model {
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;
};



Model loadOBJ(const char* objPath)
{
    // function to load the obj file
    // Note: this simple function cannot load all obj files.
```

```
    struct V {
        // struct for identify if a vertex has showed up
        unsigned int index_position, index_uv, index_normal;
        bool operator == (const V& v) const {
            return index_position == v.index_position && index_uv == v.index_uv &&
index_normal == v.index_normal;
        }
        bool operator < (const V& v) const {
            return (index_position < v.index_position) ||
                (index_position == v.index_position && index_uv < v.index_uv) ||
                (index_position == v.index_position && index_uv == v.index_uv &&
index_normal < v.index_normal);
        }
    };

    std::vector<glm::vec3> temp_positions;
    std::vector<glm::vec2> temp_uvs;
    std::vector<glm::vec3> temp_normals;

    std::map<V, unsigned int> temp_vertices;

    Model model;
    unsigned int num_vertices = 0;

    std::cout << "\nLoading OBJ file " << objPath << "..." << std::endl;

    std::ifstream file;
    file.open(objPath);

    // Check for Error
    if (file.fail()) {
        std::cerr << "Impossible to open the file! Do you use the right path? See
Tutorial 6 for details" << std::endl;
        exit(1);
    }

    while (!file.eof()) {
        // process the object file
        char lineHeader[128];
        file >> lineHeader;

        if (strcmp(lineHeader, "v") == 0) {
            // geometric vertices
            glm::vec3 position;
            file >> position.x >> position.y >> position.z;
            temp_positions.push_back(position);
        }
        else if (strcmp(lineHeader, "vt") == 0) {
            // texture coordinates
            glm::vec2 uv;
            file >> uv.x >> uv.y;
            temp_uvs.push_back(uv);
        }
        else if (strcmp(lineHeader, "vn") == 0) {
```

```
            // vertex normals
            glm::vec3 normal;
            file >> normal.x >> normal.y >> normal.z;
            temp_normals.push_back(normal);
        }
        else if (strcmp(lineHeader, "f") == 0) {
            // Face elements
            V vertices[3];
            for (int i = 0; i < 3; i++) {
                char ch;
                file >> vertices[i].index_position >> ch >> vertices[i].index_uv
>> ch >> vertices[i].index_normal;
            }

            // Check if there are more than three vertices in one face.
            std::string redundency;
            std::getline(file, redundency);
            /*if (redundency.length() >= 5) {
                std::cerr << "There may exist some errors while load the obj file.
Error content: [" << redundency << " ]" << std::endl;
                std::cerr << "Please note that we only support the faces drawing
with triangles. There are more than three vertices in one face." << std::endl;
                std::cerr << "Your obj file can't be read properly by our simple
parser :-( Try exporting with other options." << std::endl;
                exit(1);
            }*/

            for (int i = 0; i < 3; i++) {
                if (temp_vertices.find(vertices[i]) == temp_vertices.end()) {
                    // the vertex never shows before
                    Vertex vertex;
                    vertex.position = temp_positions[vertices[i].index_position -
1];
                    vertex.uv = temp_uvs[vertices[i].index_uv - 1];
                    vertex.normal = temp_normals[vertices[i].index_normal - 1];

                    model.vertices.push_back(vertex);
                    model.indices.push_back(num_vertices);
                    temp_vertices[vertices[i]] = num_vertices;
                    num_vertices += 1;
                }
                else {
                    // reuse the existing vertex
                    unsigned int index = temp_vertices[vertices[i]];
                    model.indices.push_back(index);
                }
            } // for
        } // else if
        else {
            // it's not a vertex, texture coordinate, normal or face
            char stupidBuffer[1024];
            file.getline(stupidBuffer, 1024);
        }
    }
```

```
    file.close();

    std::cout << "There are " << num_vertices << " vertices in the obj file.\n" <<
std::endl;
    return model;
}


void get_OpenGL_info()
{
    // OpenGL information
    const GLubyte* name = glGetString(GL_VENDOR);
    const GLubyte* renderer = glGetString(GL_RENDERER);
    const GLubyte* glversion = glGetString(GL_VERSION);
    std::cout << "OpenGL company: " << name << std::endl;
    std::cout << "Renderer name: " << renderer << std::endl;
    std::cout << "OpenGL version: " << glversion << std::endl;
}

GLuint vao[8];
GLuint vbo[8];
GLuint ebo[8];
GLuint skyboxvao;
GLuint skyboxvbo;
Model obj[8];
Model planet;
Model spacecraft;
Model vehicle;
Model rock;
Model station;
Model ufo;
GLuint skybox_t;
GLuint textures[8];
GLuint planet_t[4];
GLuint spacecraft_t[4];
GLuint vehicle_t[4];
GLuint rock_t[4];
GLuint station_t[4];
GLuint ufo_t[4];
int vehicle_t_index = 0;
int ufo_t_index = 0;

GLuint loadTexture(const char* texturePath)
{
    stbi_set_flip_vertically_on_load(true);
    int Width, Height, BPP;
    unsigned char* data = stbi_load(texturePath, &Width, &Height, &BPP, 0);

    GLenum format = 3;
    switch (BPP) {
    case 1: format = GL_RED; break;
    case 3: format = GL_RGB; break;
    case 4: format = GL_RGBA; break;
    }
```

```
    if(!data) {
        std::cout << "Failed to load texture: " << texturePath << std::endl;
        exit(1);
    }

    GLuint textureID = 0;

    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);

    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_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, Width, Height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
    stbi_image_free(data);
    std::cout << "Load " << texturePath << " successfully!" << std::endl;
    glBindTexture(GL_TEXTURE_2D, 0);
    return textureID;
}

GLuint loadskybox() {

    GLfloat skybox[] =
    {
        -1.0f,  1.0f, -1.0f,
        -1.0f, -1.0f, -1.0f,
         1.0f, -1.0f, -1.0f,
         1.0f, -1.0f, -1.0f,
         1.0f,  1.0f, -1.0f,
        -1.0f,  1.0f, -1.0f,

        -1.0f, -1.0f,  1.0f,
        -1.0f, -1.0f, -1.0f,
        -1.0f,  1.0f, -1.0f,
        -1.0f,  1.0f, -1.0f,
        -1.0f,  1.0f,  1.0f,
        -1.0f, -1.0f,  1.0f,

         1.0f, -1.0f, -1.0f,
         1.0f, -1.0f,  1.0f,
         1.0f,  1.0f,  1.0f,
         1.0f,  1.0f,  1.0f,
         1.0f,  1.0f, -1.0f,
         1.0f, -1.0f, -1.0f,

        -1.0f, -1.0f,  1.0f,
        -1.0f,  1.0f,  1.0f,
         1.0f,  1.0f,  1.0f,
```

```
         1.0f,  1.0f,  1.0f,
         1.0f, -1.0f,  1.0f,
        -1.0f, -1.0f,  1.0f,

        -1.0f,  1.0f, -1.0f,
         1.0f,  1.0f, -1.0f,
         1.0f,  1.0f,  1.0f,
         1.0f,  1.0f,  1.0f,
        -1.0f,  1.0f,  1.0f,
        -1.0f,  1.0f, -1.0f,

        -1.0f, -1.0f, -1.0f,
        -1.0f, -1.0f,  1.0f,
         1.0f, -1.0f, -1.0f,
         1.0f, -1.0f, -1.0f,
        -1.0f, -1.0f,  1.0f,
         1.0f, -1.0f,  1.0f
    };

    /*-1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        -1.0f, 1.0f, -1.0f,
        1.0f, 1.0f, -1.0f,
        -1.0f, -1.0f, 1.0f,
        1.0f, -1.0f, 1.0f,
        -1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f*/

    //skybox_t[0] = loadTexture("materials/texture/earthTexture.bmp");

    glGenVertexArrays(1, &skyboxvao);
    glBindVertexArray(skyboxvao);

    glGenBuffers(1, &skyboxvbo);
    glBindBuffer(GL_ARRAY_BUFFER, skyboxvbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(skybox), &skybox, GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);

    std::vector <const GLchar*> skybox_faces;
    skybox_faces.push_back("materials/skybox/right.bmp");
    skybox_faces.push_back("materials/skybox/left.bmp");
    skybox_faces.push_back("materials/skybox/bottom.bmp");
    skybox_faces.push_back("materials/skybox/top.bmp");
    skybox_faces.push_back("materials/skybox/back.bmp");
    skybox_faces.push_back("materials/skybox/front.bmp");

    GLuint textureID;
    glGenTextures(1, &textureID);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
    int width, height, BPP;
    unsigned char* data;
```

```
    for (int i = 0; i < skybox_faces.size(); i++)
    {
        data = stbi_load(skybox_faces[i], &width, &height, &BPP, 0);
        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, height,
0, GL_RGB, GL_UNSIGNED_BYTE, data);
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    return textureID;
}

void loadplanet() {

    planet = loadOBJ("materials/object/planet.obj");
    planet_t[0] = loadTexture("materials/texture/earthTexture_1.jpg");
    planet_t[1] = loadTexture("materials/texture/earthNormal.bmp");
    planet_t[2] = loadTexture("materials/texture/earthCloud.jpg");
    planet_t[3] = loadTexture("materials/texture/earthIllumination.jpg");

    glGenVertexArrays(1, &vao[0]);
    glBindVertexArray(vao[0]);

    glGenBuffers(1, &vbo[0]);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glBufferData(GL_ARRAY_BUFFER, planet.vertices.size() * sizeof(Vertex),
&planet.vertices[0], GL_STATIC_DRAW);

    glGenBuffers(1, &ebo[0]);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[0]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, planet.indices.size() * sizeof(unsigned
int), &planet.indices[0], GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, position));
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, uv));
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, normal));

}

void loadspacecraft() {

    spacecraft = loadOBJ("materials/object/spacecraft.obj");
    spacecraft_t[0] = loadTexture("materials/texture/spacecraftTexture.bmp");
    //spacecraft_t[1] = loadTexture("resources/spacecraft/spacecraft_02.jpg");
```

```
    glGenVertexArrays(1, &vao[1]);
    glBindVertexArray(vao[1]);

    glGenBuffers(1, &vbo[1]);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glBufferData(GL_ARRAY_BUFFER, spacecraft.vertices.size() * sizeof(Vertex),
&spacecraft.vertices[0], GL_STATIC_DRAW);

    glGenBuffers(1, &ebo[1]);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[1]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, spacecraft.indices.size() *
sizeof(unsigned int), &spacecraft.indices[0], GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, position));
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, uv));
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, normal));

}

void loadvehicle() {

    vehicle = loadOBJ("materials/object/craft.obj");
    vehicle_t[0] = loadTexture("materials/texture/vehicleTexture.bmp");
    vehicle_t[1] = loadTexture("materials/texture/spacecraftTexture.bmp");

    glGenVertexArrays(1, &vao[2]);
    glBindVertexArray(vao[2]);

    glGenBuffers(1, &vbo[2]);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
    glBufferData(GL_ARRAY_BUFFER, vehicle.vertices.size() * sizeof(Vertex),
&vehicle.vertices[0], GL_STATIC_DRAW);

    glGenBuffers(1, &ebo[2]);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[2]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, vehicle.indices.size() * sizeof(unsigned
int), &vehicle.indices[0], GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, position));
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, uv));
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, normal));
```

```
}

void loadrock() {

    rock = loadOBJ("materials/object/rock.obj");
    rock_t[0] = loadTexture("materials/texture/rockTexture.bmp");

    glm::mat4* modelMatrices;
    modelMatrices = new glm::mat4[rock_amount];
    srand(glfwGetTime()); // initialize random seed
    for (unsigned int i = 0; i < rock_amount; i++)
    {
        glm::mat4 model = glm::mat4(1.0f);
        // 1. translation: displace along circle with 'radius' in range [-offset,
offset]
        float angle = (float)i / (float)rock_amount * 360.0f;
        float displacement = (rand() % (int)(2 * rock_offset * 100)) / 100.0f -
rock_offset;
        float x = sin(angle) * rock_radius + displacement;
        displacement = (rand() % (int)(2 * rock_offset * 100)) / 100.0f -
rock_offset;
        float y = displacement * 0.4f + .75f; // keep height of field smaller
compared to width of x and z
        displacement = (rand() % (int)(2 * rock_offset * 100)) / 100.0f -
rock_offset;
        float z = cos(angle) * rock_radius + displacement;
        model = glm::translate(model, glm::vec3(x, y, z));

        // 2. scale: scale between 0.005 and 0.025f
        float scale = (rand() % 20) / 500.0f + 0.005f;
        model = glm::scale(model, glm::vec3(scale));

        // 3. rotation: add random rotation around a (semi)randomly picked
rotation axis vector
        float rotAngle = (rand() % 360);
        model = glm::rotate(model, rotAngle, glm::vec3(0.4f, 0.6f, 0.8f));

        // 4. now add to list of matrices
        modelMatrices[i] = model;
    }

    glGenVertexArrays(1, &vao[3]);
    glBindVertexArray(vao[3]);

    glGenBuffers(1, &vbo[3]);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[3]);
    glBufferData(GL_ARRAY_BUFFER, rock.vertices.size() * sizeof(Vertex),
&rock.vertices[0], GL_STATIC_DRAW);

    glGenBuffers(1, &ebo[3]);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[3]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, rock.indices.size() * sizeof(unsigned
int), &rock.indices[0], GL_STATIC_DRAW);
```

```
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, position));
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, uv));
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, normal));

    glGenBuffers(1, &vbo[4]);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[4]);
    glBufferData(GL_ARRAY_BUFFER, rock_amount * sizeof(glm::mat4),
&modelMatrices[0], GL_STATIC_DRAW);

    // set attribute pointers for matrix (4 times vec4)
    glEnableVertexAttribArray(3);
    glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)0);
    glEnableVertexAttribArray(4);
    glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)
(sizeof(glm::vec4)));
    glEnableVertexAttribArray(5);
    glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)(2 *
sizeof(glm::vec4)));
    glEnableVertexAttribArray(6);
    glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)(3 *
sizeof(glm::vec4)));

    glVertexAttribDivisor(3, 1);
    glVertexAttribDivisor(4, 1);
    glVertexAttribDivisor(5, 1);
    glVertexAttribDivisor(6, 1);

}

void loadstation() {

    station = loadOBJ("materials/object/station.obj");
    station_t[0] = loadTexture("materials/texture/stationTexture.bmp");

    glGenVertexArrays(1, &vao[4]);
    glBindVertexArray(vao[4]);

    glGenBuffers(1, &vbo[4]);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[4]);
    glBufferData(GL_ARRAY_BUFFER, station.vertices.size() * sizeof(Vertex),
&station.vertices[0], GL_STATIC_DRAW);

    glGenBuffers(1, &ebo[4]);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[4]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, station.indices.size() * sizeof(unsigned
int), &station.indices[0], GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
```

```
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, position));
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, uv));
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, normal));

}

void loadufo() {

    ufo = loadOBJ("materials/object/ufo.obj");
    ufo_t[0] = loadTexture("materials/texture/ufoTexture.png");
    ufo_t[1] = loadTexture("materials/texture/spacecraftTexture.bmp");

    glGenVertexArrays(1, &vao[5]);
    glBindVertexArray(vao[5]);

    glGenBuffers(1, &vbo[5]);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[5]);
    glBufferData(GL_ARRAY_BUFFER, ufo.vertices.size() * sizeof(Vertex),
&ufo.vertices[0], GL_STATIC_DRAW);

    glGenBuffers(1, &ebo[5]);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[5]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, ufo.indices.size() * sizeof(unsigned
int), &ufo.indices[0], GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, position));
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, uv));
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, normal));

}


void sendDataToOpenGL()
{
    loadplanet();
    loadspacecraft();
    loadvehicle();
    loadrock();
    loadstation();
    loadufo();
    skybox_t = loadskybox();
}
```

```
bool checkStatus(
    GLuint objectID,
    PFNGLGETSHADERIVPROC objectPropertyGetterFunc,
    PFNGLGETSHADERINFOLOGPROC getInfoLogFunc,
    GLenum statusType)
{
    GLint status;
    objectPropertyGetterFunc(objectID, statusType, &status);
    if (status != GL_TRUE)
    {
        GLint infoLogLength;
        objectPropertyGetterFunc(objectID, GL_INFO_LOG_LENGTH, &infoLogLength);
        GLchar* buffer = new GLchar[infoLogLength];

        GLsizei bufferSize;
        getInfoLogFunc(objectID, infoLogLength, &bufferSize, buffer);
        std::cout << buffer << std::endl;

        delete[] buffer;
        return false;
    }
    return true;
}

bool checkShaderStatus(GLuint shaderID) {
    return checkStatus(shaderID, glGetShaderiv, glGetShaderInfoLog,
GL_COMPILE_STATUS);
}

bool checkProgramStatus(GLuint programID) {
    return checkStatus(programID, glGetProgramiv, glGetProgramInfoLog,
GL_LINK_STATUS);
}

std::string readShaderCode(const char* fileName) {
    std::ifstream meInput(fileName);
    if (!meInput.good()) {
        std::cout << "File failed to load ... " << fileName << std::endl;
        exit(1);
    }
    return std::string(
        std::istreambuf_iterator<char>(meInput),
        std::istreambuf_iterator<char>()
    );
}

void installShaders() {
    GLuint vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);

    const GLchar* adapter[1];
    //adapter[0] = vertexShaderCode;
    std::string temp = readShaderCode("VertexShaderCode.glsl");
    adapter[0] = temp.c_str();
```

```
    glShaderSource(vertexShaderID, 1, adapter, 0);
    //adapter[0] = fragmentShaderCode;
    temp = readShaderCode("FragmentShaderCode.glsl");
    adapter[0] = temp.c_str();
    glShaderSource(fragmentShaderID, 1, adapter, 0);

    glCompileShader(vertexShaderID);
    glCompileShader(fragmentShaderID);

    if (!checkShaderStatus(vertexShaderID) ||
!checkShaderStatus(fragmentShaderID))
        return;

    programID[0] = glCreateProgram();
    glAttachShader(programID[0], vertexShaderID);
    glAttachShader(programID[0], fragmentShaderID);
    glLinkProgram(programID[0]);

    if (!checkProgramStatus(programID[0]))
        return;
    glUseProgram(programID[0]);

}

void installShaders_skybox()
{
    GLuint vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);

    const GLchar* adapter[1];
    //adapter[0] = vertexShaderCode;
    std::string temp = readShaderCode("VertexShaderCode_skybox.glsl");
    adapter[0] = temp.c_str();
    glShaderSource(vertexShaderID, 1, adapter, 0);
    //adapter[0] = fragmentShaderCode;
    temp = readShaderCode("FragmentShaderCode_skybox.glsl");
    adapter[0] = temp.c_str();
    glShaderSource(fragmentShaderID, 1, adapter, 0);

    glCompileShader(vertexShaderID);
    glCompileShader(fragmentShaderID);

    if (!checkShaderStatus(vertexShaderID) ||
!checkShaderStatus(fragmentShaderID))
        return;

    programID[1] = glCreateProgram();
    glAttachShader(programID[1], vertexShaderID);
    glAttachShader(programID[1], fragmentShaderID);
    glLinkProgram(programID[1]);

    if (!checkProgramStatus(programID[1]))
        return;
}
```

```
void installShaders_planet()
{
    GLuint vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);

    const GLchar* adapter[1];
    //adapter[0] = vertexShaderCode;
    std::string temp = readShaderCode("VertexShaderCode.glsl");
    adapter[0] = temp.c_str();
    glShaderSource(vertexShaderID, 1, adapter, 0);
    //adapter[0] = fragmentShaderCode;
    temp = readShaderCode("FragmentShaderCode_planet.glsl");
    adapter[0] = temp.c_str();
    glShaderSource(fragmentShaderID, 1, adapter, 0);

    glCompileShader(vertexShaderID);
    glCompileShader(fragmentShaderID);

    if (!checkShaderStatus(vertexShaderID) ||
!checkShaderStatus(fragmentShaderID))
        return;

    programID[2] = glCreateProgram();
    glAttachShader(programID[2], vertexShaderID);
    glAttachShader(programID[2], fragmentShaderID);
    glLinkProgram(programID[2]);

    if (!checkProgramStatus(programID[2]))
        return;
    //glUseProgram(programID[2]);
}

void installShaders_rocks()
{
    GLuint vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);

    const GLchar* adapter[1];
    //adapter[0] = vertexShaderCode;
    std::string temp = readShaderCode("VertexShaderCode_rocks.glsl");
    adapter[0] = temp.c_str();
    glShaderSource(vertexShaderID, 1, adapter, 0);
    //adapter[0] = fragmentShaderCode;
    temp = readShaderCode("FragmentShaderCode_rocks.glsl");
    adapter[0] = temp.c_str();
    glShaderSource(fragmentShaderID, 1, adapter, 0);

    glCompileShader(vertexShaderID);
    glCompileShader(fragmentShaderID);

    if (!checkShaderStatus(vertexShaderID) ||
!checkShaderStatus(fragmentShaderID))
        return;
```

```
    programID[3] = glCreateProgram();
    glAttachShader(programID[3], vertexShaderID);
    glAttachShader(programID[3], fragmentShaderID);
    glLinkProgram(programID[3]);

    if (!checkProgramStatus(programID[3]))
        return;
}

void initializedGL(void) //run only once
{
    if (glewInit() != GLEW_OK) {
        std::cout << "GLEW not OK." << std::endl;
    }

    get_OpenGL_info();
    sendDataToOpenGL();
    installShaders();
    installShaders_planet();
    installShaders_skybox();
    installShaders_rocks();

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
}

void setUniform(int ID_index)
{
    glUseProgram(programID[ID_index]);
    glm::vec3 cameraPos = glm::vec3(x_pos - 1.35f *
sin(glm::radians(spacecraft_rot)), y_pos + 0.8f, z_pos - 1.35f *
cos(glm::radians(spacecraft_rot)));
    glm::vec3 cameraTar = glm::vec3(x_pos, y_pos + 0.5f, z_pos);
    glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
    glm::mat4 viewMatrix = glm::lookAt(cameraPos, cameraTar, cameraUp);
    GLint viewMatrixUniformLocation =
        glGetUniformLocation(programID[ID_index], "viewMatrix");
    glUniformMatrix4fv(viewMatrixUniformLocation, 1,
        GL_FALSE, &viewMatrix[0][0]);

    glm::mat4 projectionMatrix = glm::perspective(glm::radians(60.0f), 1.0f, 1.0f,
100.0f);
    GLint projectionMatrixUniformLocation =
        glGetUniformLocation(programID[ID_index], "projectionMatrix");
    glUniformMatrix4fv(projectionMatrixUniformLocation, 1,
        GL_FALSE, &projectionMatrix[0][0]);

    GLint lightIntensityUniformLocation =
glGetUniformLocation(programID[ID_index], "lightIntensity");
    glm::vec3 lightIntensity(lightIntensity_R, lightIntensity_G,
lightIntensity_B);
    glUniform3fv(lightIntensityUniformLocation, 1, &lightIntensity[0]);
```

```
    glm::vec3 intrinsic_dirLight[4];
    intrinsic_dirLight[0] = glm::vec3(0.0f, -10.0f, 0.0f);
    glUniform3fv(glGetUniformLocation(programID[ID_index], "dirLight.direction"),
1, &intrinsic_dirLight[0][0]);
    intrinsic_dirLight[1] = glm::vec3(0.5f, 0.5f, 0.5f);
    glUniform3fv(glGetUniformLocation(programID[ID_index], "dirLight.ambient"), 1,
&intrinsic_dirLight[1][0]);
    intrinsic_dirLight[2] = glm::vec3(0.5f, 0.5f, 0.5f);
    glUniform3fv(glGetUniformLocation(programID[ID_index], "dirLight.diffuse"), 1,
&intrinsic_dirLight[2][0]);
    intrinsic_dirLight[3] = glm::vec3(1.0f, 1.0f, 1.0f);
    glUniform3fv(glGetUniformLocation(programID[ID_index], "dirLight.specular"),
1, &intrinsic_dirLight[3][0]);

    glm::vec3 intrinsic_pointLight[5];
    intrinsic_pointLight[0] = glm::vec3(2.0f, 6.0f, 2.0f);
    glUniform3fv(glGetUniformLocation(programID[ID_index], "pointLight.position"),
1, &intrinsic_pointLight[0][0]);
    intrinsic_pointLight[1] = glm::vec3(0.2f, 0.2f, 0.2f);
    glUniform3fv(glGetUniformLocation(programID[ID_index], "pointLight.ambient"),
1, &intrinsic_pointLight[1][0]);
    intrinsic_pointLight[2] = glm::vec3(0.5f, 0.5f, 0.5f);
    glUniform3fv(glGetUniformLocation(programID[ID_index], "pointLight.diffuse"),
1, &intrinsic_pointLight[2][0]);
    intrinsic_pointLight[3] = glm::vec3(1.0f, 1.0f, 1.0f);
    glUniform3fv(glGetUniformLocation(programID[ID_index], "pointLight.specular"),
1, &intrinsic_pointLight[3][0]);
    intrinsic_pointLight[4] = glm::vec3(1.0f, 0.7f, 1.8f);
    glUniform3fv(glGetUniformLocation(programID[ID_index],
"pointLight.attenuation"), 1, &intrinsic_pointLight[4][0]);

    GLint ambientLightUniformLocation = glGetUniformLocation(programID[ID_index],
"ambientLight");
    glm::vec3 ambientLight(ambient_R, ambient_G, ambient_B);
    glUniform3fv(ambientLightUniformLocation, 1, &ambientLight[0]);

    GLint lightPositionUniformLocation = glGetUniformLocation(programID[ID_index],
"lightPositionWorld");
    glm::vec3 lightPosition(2.0f, 10.0f, -6.0f);
    glUniform3fv(lightPositionUniformLocation, 1, &lightPosition[0]);

    GLint eyePositionUniformLocation = glGetUniformLocation(programID[ID_index],
"eyePositionWorld");
    glm::vec3 eyePosition(cameraPos.x, cameraPos.y, cameraPos.z);
    glUniform3fv(eyePositionUniformLocation, 1, &eyePosition[0]);
}

void setUniform_skybox()
{
    glUseProgram(programID[1]);
    glm::vec3 cameraPos = glm::vec3(x_pos - 3.0f *
sin(glm::radians(spacecraft_rot)), y_pos + 0.5f, z_pos - 3.0f *
cos(glm::radians(spacecraft_rot)));
    glm::vec3 cameraTar = glm::vec3(x_pos, y_pos, z_pos);
```

```
    glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
    glm::mat4 viewMatrix = glm::lookAt(cameraPos, cameraTar, cameraUp);
    viewMatrix = glm::mat4(glm::mat3(viewMatrix));
    GLint viewMatrixUniformLocation =
        glGetUniformLocation(programID[1], "viewMatrix");
    glUniformMatrix4fv(viewMatrixUniformLocation, 1,
        GL_FALSE, &viewMatrix[0][0]);

    glm::mat4 projectionMatrix = glm::perspective(glm::radians(45.0f), 1.0f, 1.0f,
100.0f);
    GLint projectionMatrixUniformLocation =
        glGetUniformLocation(programID[1], "projectionMatrix");
    glUniformMatrix4fv(projectionMatrixUniformLocation, 1,
        GL_FALSE, &projectionMatrix[0][0]);
}

void spacecraftControl()
{
    if (press_SHIFT)
    {
        spacecraft_max_vel = 10.0f;
        spacecraft_acc = 10.0f;
    }
    else
    {
        spacecraft_max_vel = 5.0f;
        spacecraft_acc = 2.5f;
    }
    if (press_UP)
    {
        if (spacecraft_vel <= spacecraft_max_vel)
        {
            spacecraft_vel += spacecraft_acc * delta_time;
        }
    }
    else
    {
        if (spacecraft_vel > 0.0f)
        {
            spacecraft_vel -= spacecraft_brake_acc * delta_time;
        }
    }
    if (press_DOWN)
    {
        if (spacecraft_vel >= -spacecraft_max_vel)
        {
            spacecraft_vel -= spacecraft_acc * delta_time;
        }
    }
    else
    {
        if (spacecraft_vel < 0.0f)
        {
            spacecraft_vel += spacecraft_brake_acc * delta_time;
```

```
        }
    }
    x_pos += spacecraft_vel * delta_time * sin(glm::radians(spacecraft_rot));
    z_pos += spacecraft_vel * delta_time * cos(glm::radians(spacecraft_rot));
    if (press_LEFT)
    {
        x_pos += spacecraft_dodge_vel * delta_time *
cos(glm::radians(spacecraft_rot));
        z_pos -= spacecraft_dodge_vel * delta_time *
sin(glm::radians(spacecraft_rot));
    }
    if (press_RIGHT)
    {
        x_pos -= spacecraft_dodge_vel * delta_time *
cos(glm::radians(spacecraft_rot));
        z_pos += spacecraft_dodge_vel * delta_time *
sin(glm::radians(spacecraft_rot));
    }
    if (press_Q)
    {
        spacecraft_rot += 180.0f * delta_time;
    }
    if (press_E)
    {
        spacecraft_rot -= 180.0f * delta_time;
    }
}

void paintGL(void) //always run
{
    glClearColor(0.5f, 0.5f, 0.5f, 0.5f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    current_time = glfwGetTime();
    delta_time = current_time - last_time;
    last_time = current_time;
    wait_time += delta_time;
    wait_time_ufo += delta_time;

    //Draw skybox
    glDepthMask(GL_FALSE);
    setUniform_skybox();

    glm::mat4 modelTransformMatrix_skybox = glm::mat4(1.0f);
    modelTransformMatrix_skybox = glm::scale(modelTransformMatrix_skybox,
glm::vec3(2.0f, 2.0f, 2.0f));
    GLint modelTransformMatrixUniformLocation_skybox =
        glGetUniformLocation(programID[1], "modelTransformMatrix");
    glUniformMatrix4fv(modelTransformMatrixUniformLocation_skybox, 1,
        GL_FALSE, &modelTransformMatrix_skybox[0][0]);
    glBindVertexArray(skyboxvao);
    GLuint TextureID_skybox = glGetUniformLocation(programID[1],
"myTextureSampler0");
    glActiveTexture(GL_TEXTURE0);
```

```
    glBindTexture(GL_TEXTURE_CUBE_MAP, skybox_t);
    glUniform1i(TextureID_skybox, 0);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glDepthMask(GL_TRUE);


    //Transform matrix for planet
    setUniform(2);
    glm::mat4 modelTransformMatrix = glm::mat4(1.0f);
    modelTransformMatrix = glm::rotate(modelTransformMatrix,
glm::radians(float(glfwGetTime() * 25.0f)), glm::vec3(0.0f, 1.0f, 0.0f));
    GLint modelTransformMatrixUniformLocation =
        glGetUniformLocation(programID[2], "modelTransformMatrix");
    glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1,
        GL_FALSE, &modelTransformMatrix[0][0]);

    glBindVertexArray(vao[0]);
    GLuint TextureID_0 = glGetUniformLocation(programID[2], "myTextureSampler0");
    GLuint TextureID_1 = glGetUniformLocation(programID[2], "myTextureSampler1");
    GLuint TextureID_2 = glGetUniformLocation(programID[2], "myTextureSampler2");
    GLuint TextureID_3 = glGetUniformLocation(programID[2], "myTextureSampler3");
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, planet_t[0]);
    glUniform1i(TextureID_0, 0);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, planet_t[1]);
    glUniform1i(TextureID_1, 1);
    glActiveTexture(GL_TEXTURE2);
    glBindTexture(GL_TEXTURE_2D, planet_t[2]);
    glUniform1i(TextureID_2, 2);
    glActiveTexture(GL_TEXTURE3);
    glBindTexture(GL_TEXTURE_2D, planet_t[3]);
    glUniform1i(TextureID_3, 3);
    glDrawElements(GL_TRIANGLES, planet.indices.size(), GL_UNSIGNED_INT, 0);


    //Transform matrix for rock
    setUniform(3);
    modelTransformMatrix = glm::mat4(1.0f);
    modelTransformMatrix = glm::rotate(modelTransformMatrix,
glm::radians(float(glfwGetTime() * 5.0f)), glm::vec3(0.0f, 1.0f, 0.0f));

    modelTransformMatrixUniformLocation =
        glGetUniformLocation(programID[3], "modelTransformMatrix");
    glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1,
        GL_FALSE, &modelTransformMatrix[0][0]);

    glBindVertexArray(vao[3]);
    GLuint TextureID_rocks = glGetUniformLocation(programID[3],
"myTextureSampler0");
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, rock_t[0]);
    glUniform1i(TextureID_rocks, 0);
    glDrawElementsInstanced(GL_TRIANGLES, rock.indices.size(), GL_UNSIGNED_INT, 0,
```

```
rock_amount);


    //Transform matrix for spacecraft
    setUniform(0);
    spacecraftControl();

    modelTransformMatrix = glm::mat4(1.0f);
    modelTransformMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(x_pos, y_pos,
z_pos));
    modelTransformMatrix = glm::scale(modelTransformMatrix, glm::vec3(0.001f,
0.001f, 0.001f));
    modelTransformMatrix = glm::rotate(modelTransformMatrix,
glm::radians(spacecraft_rot), glm::vec3(0.0f, 1.0f, 0.0f));
    modelTransformMatrixUniformLocation =
        glGetUniformLocation(programID[0], "modelTransformMatrix");
    glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1,
        GL_FALSE, &modelTransformMatrix[0][0]);

    glBindVertexArray(vao[1]);
    GLuint TextureID = glGetUniformLocation(programID[0], "myTextureSampler0");
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, spacecraft_t[spacecraft_t_index]);
    glUniform1i(TextureID, 0);
    glDrawElements(GL_TRIANGLES, spacecraft.indices.size(), GL_UNSIGNED_INT, 0);


    //Transform matrix for vehicle
    if (wait_time >= jump_time)
    {
        vehicle_pos_x = float(rand() / float(RAND_MAX / (2.0f * jump_radius))) -
jump_radius;
        vehicle_pos_z = float(rand() / float(RAND_MAX / (2.0f * jump_radius))) -
jump_radius;
        wait_time = 0.0f;
        if (vehicle_pos_x >= 0)
        {
            vehicle_pos_x += jump_min_radius;
        }
        else
        {
            vehicle_pos_x -= jump_min_radius;
        }
        if (vehicle_pos_z >= 0)
        {
            vehicle_pos_z += jump_min_radius;
        }
        else
        {
            vehicle_pos_z -= jump_min_radius;
        }
    }
    if (abs(x_pos - vehicle_pos_x) < 1.0f && abs(z_pos - vehicle_pos_z) < 1.0f)
    {
```

```
        vehicle_rot_speed = 120.0f;
        wait_time = 0.0f;
        vehicle_t_index = 1;
    }
    modelTransformMatrix = glm::mat4(1.0f);
    modelTransformMatrix = glm::translate(glm::mat4(1.0f),
glm::vec3(vehicle_pos_x, 0.0f, vehicle_pos_z));;
    modelTransformMatrix = glm::scale(modelTransformMatrix, glm::vec3(0.05f,
0.05f, 0.05f));
    modelTransformMatrix = glm::rotate(modelTransformMatrix,
glm::radians(float(glfwGetTime() * vehicle_rot_speed)), glm::vec3(0.0f, 1.0f,
0.0f));
    modelTransformMatrixUniformLocation =
        glGetUniformLocation(programID[0], "modelTransformMatrix");
    glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1,
        GL_FALSE, &modelTransformMatrix[0][0]);

    glBindVertexArray(vao[2]);
    TextureID = glGetUniformLocation(programID[0], "myTextureSampler0");
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, vehicle_t[vehicle_t_index]);
    glUniform1i(TextureID, 0);
    glDrawElements(GL_TRIANGLES, vehicle.indices.size(), GL_UNSIGNED_INT, 0);


    //Transform matrix for station
    modelTransformMatrix = glm::mat4(1.0f);
    modelTransformMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-5.0f, 0.0f,
5.0f));;
    modelTransformMatrix = glm::scale(modelTransformMatrix, glm::vec3(0.005f,
0.005f, 0.005f));
    modelTransformMatrixUniformLocation =
        glGetUniformLocation(programID[0], "modelTransformMatrix");
    glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1,
        GL_FALSE, &modelTransformMatrix[0][0]);

    glBindVertexArray(vao[4]);
    TextureID = glGetUniformLocation(programID[0], "myTextureSampler0");
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, station_t[0]);
    glUniform1i(TextureID, 0);
    glDrawElements(GL_TRIANGLES, station.indices.size(), GL_UNSIGNED_INT, 0);


    //Transform matrix for ufo
    if (wait_time_ufo >= jump_time_ufo)
    {
        ufo_pos_x = float(rand() / float(RAND_MAX / (2.0f * jump_radius))) -
jump_radius;
        ufo_pos_z = float(rand() / float(RAND_MAX / (2.0f * jump_radius))) -
jump_radius;
        wait_time_ufo = 0.0f;
        if (ufo_pos_x >= 0)
        {
```

```
            ufo_pos_x += jump_min_radius;
        }
        else
        {
            ufo_pos_x -= jump_min_radius;
        }
        if (ufo_pos_z >= 0)
        {
            ufo_pos_z += jump_min_radius;
        }
        else
        {
            ufo_pos_z -= jump_min_radius;
        }
    }
    if (abs(x_pos - ufo_pos_x) < 1.0f && abs(z_pos - ufo_pos_z) < 1.0f)
    {
        ufo_rot_speed = 120.0f;
        wait_time_ufo = 0.0f;
        ufo_t_index = 1;
    }
    modelTransformMatrix = glm::mat4(1.0f);
    modelTransformMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(ufo_pos_x,
0.0f, ufo_pos_z));;
    modelTransformMatrix = glm::scale(modelTransformMatrix, glm::vec3(0.01f,
0.01f, 0.01f));
    modelTransformMatrix = glm::rotate(modelTransformMatrix,
glm::radians(float(glfwGetTime() * ufo_rot_speed)), glm::vec3(0.0f, 1.0f, 0.0f));
    modelTransformMatrixUniformLocation =
        glGetUniformLocation(programID[0], "modelTransformMatrix");
    glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1,
        GL_FALSE, &modelTransformMatrix[0][0]);

    glBindVertexArray(vao[5]);
    TextureID = glGetUniformLocation(programID[0], "myTextureSampler0");
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, ufo_t[ufo_t_index]);
    glUniform1i(TextureID, 0);
    glDrawElements(GL_TRIANGLES, ufo.indices.size(), GL_UNSIGNED_INT, 0);

    glBindVertexArray(0);
}

void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}

void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
    // Sets the mouse-button callback for the current window.
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS) {
        change_view = true;
        firstMouse = true;
```

```
    }
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_RELEASE) {
        change_view = false;
    }
}

void cursor_position_callback(GLFWwindow* window, double xpos, double ypos)
{
    // Sets the cursor position callback for the current window
    if (change_view)
    {
        if (firstMouse)
        {
            lastX = xpos;
            lastY = ypos;
            firstMouse = false;
        }

        xoffset = xpos - lastX;
        yoffset = lastY - ypos;
        lastX = xpos;
        lastY = ypos;

        xoffset *= sensitivity;
        yoffset *= sensitivity;

        yaw -= xoffset;
        pitch -= yoffset;

        if (pitch > 89.0f)
            pitch = 89.0f;
        if (pitch < -89.0f)
            pitch = -89.0f;

        front.x = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
        front.y = sin(glm::radians(pitch));
        front.z = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
        //front.y = 0;
        cameraFront = glm::normalize(front);

        spacecraft_rot -= xoffset;
        //std::cout << cameraFront.x << cameraFront.y << cameraFront.z <<
std::endl;
    }
}

void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    // Sets the scoll callback for the current window.
    /*if (cam_radius >= 1.0 && cam_radius <= 30.0)
    {
        cam_radius -= yoffset;
    }
    if (cam_radius < 1.0)
```

```
    {
        cam_radius = 1.0;
    }
    if (cam_radius > 30.0)
    {
        cam_radius = 30.0;
    }*/
}

void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    // Sets the Keyboard callback for the current window.
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    if (key == GLFW_KEY_W && action == GLFW_PRESS) {
        lightIntensity_R += 0.1f;
        lightIntensity_G += 0.1f;
        lightIntensity_B += 0.1f;
        //ambient_R += 0.1f;
        //ambient_G += 0.1f;
        //ambient_B += 0.1f;
    }
    if (key == GLFW_KEY_S && action == GLFW_PRESS) {
        lightIntensity_R -= 0.1f;
        lightIntensity_G -= 0.1f;
        lightIntensity_B -= 0.1f;
        //ambient_R -= 0.1f;
        //ambient_G -= 0.1f;
        //ambient_B -= 0.1f;
    }
    if (key == GLFW_KEY_UP && action == GLFW_PRESS) {
        press_UP = true;
    }
    if (key == GLFW_KEY_UP && action == GLFW_RELEASE) {
        press_UP = false;
    }
    if (key == GLFW_KEY_DOWN && action == GLFW_PRESS) {
        press_DOWN = true;
    }
    if (key == GLFW_KEY_DOWN && action == GLFW_RELEASE) {
        press_DOWN = false;
    }
    if (key == GLFW_KEY_LEFT && action == GLFW_PRESS) {
        press_LEFT = true;
    }
    if (key == GLFW_KEY_LEFT && action == GLFW_RELEASE) {
        press_LEFT = false;
    }
    if (key == GLFW_KEY_RIGHT && action == GLFW_PRESS) {
        press_RIGHT = true;
    }
    if (key == GLFW_KEY_RIGHT && action == GLFW_RELEASE) {
        press_RIGHT = false;
    }
```

```
    if (key == GLFW_KEY_Q && action == GLFW_PRESS) {
        press_Q = true;
    }
    if (key == GLFW_KEY_Q && action == GLFW_RELEASE) {
        press_Q = false;
    }
    if (key == GLFW_KEY_E && action == GLFW_PRESS) {
        press_E = true;
    }
    if (key == GLFW_KEY_E && action == GLFW_RELEASE) {
        press_E = false;
    }
    if (key == GLFW_KEY_LEFT_SHIFT && action == GLFW_PRESS) {
        press_SHIFT = true;
    }
    if (key == GLFW_KEY_LEFT_SHIFT && action == GLFW_RELEASE) {
        press_SHIFT = false;
    }
}


int main(int argc, char* argv[])
{
    GLFWwindow* window;

    //Initialize random seed
    srand(0);

    /* Initialize the glfw */
    if (!glfwInit()) {
        std::cout << "Failed to initialize GLFW" << std::endl;
        return -1;
    }
    /* glfw: configure; necessary for MAC */
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "CSCI3260 Project", NULL,
NULL);
    if (!window) {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }

    /* Make the window's context current */
    glfwMakeContextCurrent(window);
```

```
    /*register callback functions*/
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
    glfwSetKeyCallback(window, key_callback);
    glfwSetScrollCallback(window, scroll_callback);
    glfwSetCursorPosCallback(window, cursor_position_callback);
    glfwSetMouseButtonCallback(window, mouse_button_callback);

    initializedGL();

    while (!glfwWindowShouldClose(window)) {
        /* Render here */
        paintGL();

        /* Swap front and back buffers */
        glfwSwapBuffers(window);

        /* Poll for and process events */
        glfwPollEvents();
    }

    glfwTerminate();

    return 0;
}
```

shader code

```
// shader.cpp

#include "Shader.h"
#include "Dependencies/glm/gtc/type_ptr.hpp"

#include <fstream>

void Shader::setupShader(const char* vertexPath, const char* fragmentPath)
{
    // similar to the installShaders() in the assignment 1
    unsigned int vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    unsigned int fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);

    const GLchar* vCode;
    std::string temp = readShaderCode(vertexPath);
    vCode = temp.c_str();
    glShaderSource(vertexShaderID, 1, &vCode, NULL);

    const GLchar* fCode;
    temp = readShaderCode(fragmentPath);
    fCode = temp.c_str();
    glShaderSource(fragmentShaderID, 1, &fCode, NULL);

    glCompileShader(vertexShaderID);
```

```
    glCompileShader(fragmentShaderID);

    if (!checkShaderStatus(vertexShaderID) ||
!checkShaderStatus(fragmentShaderID))
        return;

    ID = glCreateProgram();

    glAttachShader(ID, vertexShaderID);
    glAttachShader(ID, fragmentShaderID);
    glLinkProgram(ID);

    if (!checkProgramStatus(ID))
        return;

    glDeleteShader(vertexShaderID);
    glDeleteShader(fragmentShaderID);

    glUseProgram(0);
}


void Shader::use() const
{
    glUseProgram(ID);
}

void Shader::setMat4(const std::string& name, glm::mat4& value) const
{
    unsigned int transformLoc = glGetUniformLocation(ID, name.c_str());
    glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(value));
}

void Shader::setVec4(const std::string& name, glm::vec4 value) const
{
    glUniform4fv(glGetUniformLocation(ID, name.c_str()), 1, &value[0]);
}

void Shader::setVec3(const std::string& name, glm::vec3 value) const
{
    glUniform3fv(glGetUniformLocation(ID, name.c_str()), 1, &value[0]);
}

void Shader::setVec3(const std::string& name, float v1, float v2, float v3) const
{
    glUniform3f(glGetUniformLocation(ID, name.c_str()), v1, v2, v3);
}

void Shader::setFloat(const std::string& name, float value) const
{
    glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
}

void Shader::setInt(const std::string& name, int value) const
```

```
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), value);
}

std::string Shader::readShaderCode(const char* fileName) const
{
    std::ifstream myInput(fileName);
    if (!myInput.good())
    {
        std::cout << "File failed to load..." << fileName << std::endl;
        exit(1);
    }
    return std::string(
        std::istreambuf_iterator<char>(myInput),
        std::istreambuf_iterator<char>()
    );
}

bool Shader::checkShaderStatus(GLuint shaderID) const
{
    return checkStatus(shaderID, glGetShaderiv, glGetShaderInfoLog,
GL_COMPILE_STATUS);
}

bool Shader::checkProgramStatus(GLuint programID) const
{
    return checkStatus(programID, glGetProgramiv, glGetProgramInfoLog,
GL_LINK_STATUS);
}

bool Shader::checkStatus(GLuint objectID, PFNGLGETSHADERIVPROC
objectPropertyGetterFunc, PFNGLGETSHADERINFOLOGPROC getInfoLogFunc, GLenum
statusType) const
{
    GLint status;
    objectPropertyGetterFunc(objectID, statusType, &status);
    if (status != GL_TRUE)
    {
        GLint infoLogLength;
        objectPropertyGetterFunc(objectID, GL_INFO_LOG_LENGTH, &infoLogLength);
        GLchar* buffer = new GLchar[infoLogLength];

        GLsizei bufferSize;
        getInfoLogFunc(objectID, infoLogLength, &bufferSize, buffer);
        std::cout << buffer << std::endl;

        delete[] buffer;
        return false;
    }
    return true;
}
```

texture code

```
// texture.cpp

#include "Texture.h"

#include "Dependencies/glew/glew.h"
#define STB_IMAGE_IMPLEMENTATION
#include "Dependencies/stb_image/stb_image.h"

#include <iostream>

void Texture::setupTexture(const char* texturePath)
{
    // tell stb_image.h to flip loaded texture's on the y-axis.
    stbi_set_flip_vertically_on_load(true);
    // load the texture data into "data"
    unsigned char* data = stbi_load(texturePath, &Width, &Height, &BPP, 0);
    GLenum format=3;
    switch (BPP) {
        case 1: format = GL_RED; break;
        case 3: format = GL_RGB; break;
        case 4: format = GL_RGBA; break;
    }

    glGenTextures(1, &ID);
    glBindTexture(GL_TEXTURE_2D, ID);

    // set the texture wrapping parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // set texture filtering parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    //glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);

    if (data) {
        glTexImage2D(GL_TEXTURE_2D, 0, format, Width, Height, 0, format,
GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
        stbi_image_free(data);
    }
    else {
        std::cout << "Failed to load texture: " << texturePath << std::endl;
        exit(1);
    }

    std::cout << "Load " << texturePath << " successfully!" << std::endl;
    //glBindTexture(GL_TEXTURE_2D, 0);
}
```

```
void Texture::bind(unsigned int slot = 0) const
{
    glActiveTexture(GL_TEXTURE0 + slot);
    glBindTexture(GL_TEXTURE_2D, ID);
}

void Texture::unbind() const
{
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Mimic_Pokémon

Overview

This is a Unity game project.

Run game.exe to play the game.

```
Use "WASD" to control the movement of the player
Use "space" jump
Use mouse left click to throw the axe to attact
Use mouse right click to throw the Poké Ball to catch creatures
```

Code Display

C# scripts for main character

```
// player.cs
using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;

public class Player : MonoBehaviour
{
    [Tooltip("移动速度")]
    public float moveSpeed = 5;
    [Tooltip("跳跃初速度")]
    public float jumpSpeed = 8;
    [Tooltip("转身速度，数字越大转身越快")]
    public float rotateSpeed = 360;
    [Tooltip("摄像机物体，必填。移动方向是根据摄像机位置计算的")]
    public Transform transCam;

    [Tooltip("子弹预制体，子弹必须具有Rigidbody刚体组件")]
    public Rigidbody prefabShell;

    [Tooltip("发射子弹的初始速度")]
    public float fireForce = 15;

    Animator animator;
    Rigidbody rigid;

    // 发射子弹的起始位置
    public Transform firePos;

    // 是否接触地面
    bool isGrounded = false;

    // 转身量，前进量。根据输入和摄像机角度计算得出
    float turnAmount, forwardAmount;

    bool jump = false;
    float nextFireTime = 0;

    void Start()
    {
        animator = GetComponentInChildren<Animator>();
        rigid = GetComponent<Rigidbody>();
    }

    void Update()
    {
        InputMove();
        Action();
```

```
        animator.SetBool("IsGround", isGrounded);
        animator.SetFloat("Speed", forwardAmount);
    }

    void InputMove()
    {
        float h = 0;
        float v = 0;

        h = Input.GetAxis("Horizontal");
        v = Input.GetAxis("Vertical");

        // 根据摄像机位置，对move变换
        Vector3 forward = (transform.position - transCam.position).normalized;
        Vector3 right = Quaternion.Euler(0, 90, 0) * forward;

        Vector3 move = v * Vector3.ProjectOnPlane(forward, Vector3.up) + h *
Vector3.ProjectOnPlane(right, Vector3.up);

        if (move.magnitude > 1f)
        {
            move.Normalize();
        }

        move = transform.InverseTransformDirection(move);
        move = Vector3.ProjectOnPlane(move, Vector3.up);
        turnAmount = Mathf.Atan2(move.x, move.z);
        forwardAmount = move.z;

        if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
        {
            jump = true;
        }
    }

    private void FixedUpdate()
    {
        isGrounded = CheckGround();

        float verticalSpeed = rigid.velocity.y;
        if (jump)
        {
            verticalSpeed = jumpSpeed;
            jump = false;
        }

        transform.Rotate(0, turnAmount * rotateSpeed * Time.deltaTime, 0);

        rigid.velocity = transform.forward * forwardAmount * moveSpeed +
transform.up * verticalSpeed;
    }
```

```
    bool CheckGround()
    {
        Vector3 p = transform.position - new Vector3(0, 0.03f, 0);
        bool ground = Physics.CheckBox(p, new Vector3(0.08f, 0.08f, 0.08f),
Quaternion.identity, LayerMask.GetMask("Ground"));
        //Debug.DrawLine(p, p + new Vector3(0, -0.05f, 0), Color.red);
        //Debug.Log("ground: " + ground);
        return ground;
    }


    void Action()
    {
        if (Input.GetButtonDown("Fire1"))
        {
            if (Time.time > nextFireTime)
            {
                Fire();
                nextFireTime = Time.time + 0.8f;
            }
        }
    }

    void Fire()
    {
        Rigidbody shellInstance = Instantiate(prefabShell, firePos.position,
firePos.rotation);

        shellInstance.velocity = fireForce * firePos.forward;
        shellInstance.transform.Rotate(0, 180, 0);
        shellInstance.angularVelocity = new Vector3(0, 0, -20);
        //float Ang = 10;
        //Ang += Time.deltaTime * 100;
        //shellInstance.transform.Rotate(new Vector3(0, Ang, 0), Space.Self);

        animator.SetTrigger("Attack");
    }

    public void Die()
    {
        gameObject.SetActive(false);
    }

    public void Revive()
    {
        gameObject.SetActive(true);
        SendMessage("ReviveHP");
    }
}
```

This is the end of the code demonstration.

Feel free to access my github repositry to have hand on experiences with my projects.
https://github.com/ynCAOr06/USC_Appication_Game_demo