Floating-Island-OpenGL

Ivan Emil Ovidiu & Beldi Darius Vlad
Grupa 342

https://github.com/Darius-Beldi/Floating-Island-OpenGL

1 Conceptul Proiectului

Un proiect care constă într-o scenă 3D ce reprezintă 3 insule plutitoare (2 mici și una mai mare).

Pe insula principală se află o casă, un copac mare și 2 lămpi de iluminat. Pe insulele mici se află copaci și o cruce mare.

Scopul nostru era să creăm o scenă horror, de aceea avem și puțină lumină.

2 Elementele incluse

Scena este realizată în Blender, apoi exportată în `.obj`. Iluminarea este făcută complet în OpenGL.

3 De ce este original?

Am încercat să creăm o atmosferă cât mai *spooky*, și credem că ne-a ieșit.

De asemenea, am extins clasa `objloader` cu funcția `loadmtl`:

```
bool loadMTL(const std::string& path, std::map<std::string, Material>&
    materials) {
    FILE* file = fopen(path.c_str(), "r");
    if (!file) return false;

    char lineHeader[128];
    Material mat;
    std::string currentMat;
    while (fscanf(file, "%s", lineHeader) != EOF) {
        if (strcmp(lineHeader, "newmtl") == 0) {
            if (!currentMat.empty()) {
                materials[currentMat] = mat;
            }
            char name[128];
            fscanf(file, "%s\n", name);
            currentMat = name;
            mat = Material();
            mat.name = currentMat;
        }
        else if (strcmp(lineHeader, "Kd") == 0) {
            fscanf(file, "%f %f %f\n", &mat.diffuseColor.r, &mat.
    diffuseColor.g, &mat.diffuseColor.b);
```

```
        }
        else {
            char buffer[1024];
            fgets(buffer, 1024, file);
        }
    }
    if (!currentMat.empty()) {
        materials[currentMat] = mat;
    }
    fclose(file);
    return true;
}
```

Aceasta permite programului să implementeze și fișierele `.mtl` din Blender.

4 Contribuții individuale

Am lucrat amândoi la acest proiect în egală măsură.

- **Darius** s-a ocupat de scena 3D.
- **Ovidiu** s-a ocupat de iluminare.

5 Coduri Sursă

5.1 main.cpp

```
#include <vector>
#include <stdio.h>
#include <string>
#include <cstring>
#include <iostream>
#include <stdlib.h>
#include <GL/glew.h>
#include <GL/freeglut.h>
#include "loadShaders.h"
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtx/transform.hpp"
#include "glm/gtc/type_ptr.hpp"
#include "objloader.hpp"

// OpenGL object identifiers
GLuint VaoId, VboId, ProgramId, nrVertLocation, myMatrixLocation,
   viewPosLocation, viewLocation, projLocation;
GLuint light1PosLocation, light1ColorLocation;
GLuint light2PosLocation, light2ColorLocation;
GLuint light3PosLocation, light3ColorLocation;

// Pi constant for mathematical calculations
float PI = 3.141592;

// Variable to store the number of vertices
int nrVertices;

```

```
// Vectors for vertices, texture coordinates, normals and colors
std::vector<glm::vec3> vertices;
std::vector<glm::vec2> uvs;
std::vector<glm::vec3> normals;
std::vector<glm::vec3> colors;

// Transformation matrices
glm::mat4 myMatrix;
glm::mat4 view;
glm::mat4 projection;

// View matrix elements
float refX = 0.0f, refY = 0.0f, refZ = 0.0f, obsX, obsY, obsZ, vX = 0.0f
    , vY = 0.0f, vZ = 1.0f;

// Spherical movement parameters
float alpha = 0.0f, beta = 0.0f, dist = 6.0f, incrAlpha1 = 0.01,
    incrAlpha2 = 0.01;

// Projection matrix parameters
float width = 800, height = 600, dNear = 4.f, fov = 60.f * PI / 180;

// Light 1 warm dawn illumination from the left lamp post
glm::vec3 light1Pos = glm::vec3(-0.5f, 1.0f, 2.98429f);
float light1Intensity = 0.8f;
glm::vec3 light1Color = glm::vec3(1.0f, 0.8f, 0.6f) * light1Intensity;

// Light 2  warm light from the right lamp post
glm::vec3 light2Pos = glm::vec3(1.4f, 0.462519f, 2.9411f);
float light2Intensity = 0.8f;
glm::vec3 light2Color = glm::vec3(1.0f, 0.8f, 0.6f) * light2Intensity;

// Light 3 soft pink-ish  light in the center front of the house
glm::vec3 light3Pos = glm::vec3(0.35f, 0.0f, 3.23623f);
float light3Intensity = 0.4f;  // Dimmer than the other two lights
glm::vec3 light3Color = glm::vec3(0.9f, 0.7f, 0.8f) * light3Intensity;
    // Soft pink glow

// Handles regular keyboard input
void processNormalKeys(unsigned char key, int x, int y)
{
    switch (key)
    {
    case '+':
        dist -= 0.25; // Move camera closer to the scene
        break;
    case '-':
        dist += 0.25; // Move camera farther from the scene
        break;
    }
    if (key == 27) // ESC key exits the program
        exit(0);
}

// Handles special keyboard keys (arrow keys)
void processSpecialKeys(int key, int xx, int yy)
{
    switch (key)
```

```
    {
    case GLUT_KEY_LEFT:
        beta -= 0.01; // Rotate camera left around the scene
        break;
    case GLUT_KEY_RIGHT:
        beta += 0.01; // Rotate camera right around the scene
        break;
    case GLUT_KEY_UP:
        alpha += incrAlpha1; // Move camera upward along the sphere
        if (abs(alpha - PI / 2) < 0.05)
        {
            incrAlpha1 = 0.f; // Stop at the top pole to prevent gimbal lock
        }
        else
        {
            incrAlpha1 = 0.01f;
        }
        break;
    case GLUT_KEY_DOWN:
        alpha -= incrAlpha2; // Move camera downward along the sphere
        if (abs(alpha + PI / 2) < 0.05)
        {
            incrAlpha2 = 0.f; // Stop at the bottom pole
        }
        else
        {
            incrAlpha2 = 0.01f;
        }
        break;
    }
}

// Initializes Vertex Buffer Object for transferring data to the GPU
void CreateVBO(void)
{
    // Generate and bind Vertex Array Object
    glGenVertexArrays(1, &VaoId);
    glBindVertexArray(VaoId);

    // Generate and bind Vertex Buffer Object
    glGenBuffers(1, &VboId);
    glBindBuffer(GL_ARRAY_BUFFER, VboId);

    // Allocate space for positions, normals and colors in a single buffer
    glBufferData(GL_ARRAY_BUFFER,
        vertices.size() * sizeof(glm::vec3) +
        normals.size() * sizeof(glm::vec3) +
        colors.size() * sizeof(glm::vec3),
        NULL, GL_STATIC_DRAW);

    // Copy data to buffer in separate sections
    glBufferSubData(GL_ARRAY_BUFFER, 0, vertices.size() * sizeof(glm::vec3), &vertices[0]);
    glBufferSubData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3),
        normals.size() * sizeof(glm::vec3), &normals[0]);
```

```
137     glBufferSubData(GL_ARRAY_BUFFER,
138         vertices.size() * sizeof(glm::vec3) + normals.size() * sizeof(
    glm::vec3),
139         colors.size() * sizeof(glm::vec3), &colors[0]);
140 
141     // Set up vertex attributes
142     glEnableVertexAttribArray(0); // Attribute 0 = position
143     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);
144 
145     glEnableVertexAttribArray(1); // Attribute 1 = normals
146     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
147         (GLvoid*)(vertices.size() * sizeof(glm::vec3)));
148 
149     glEnableVertexAttribArray(2); // Attribute 2 = color
150     glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 0,
151         (GLvoid*)(vertices.size() * sizeof(glm::vec3) +
152             normals.size() * sizeof(glm::vec3)));
153 }
154 
155 // Cleanup function to destroy VBO objects
156 void DestroyVBO(void)
157 {
158     glDisableVertexAttribArray(0);
159     glDisableVertexAttribArray(1);
160     glDisableVertexAttribArray(2);
161     glBindBuffer(GL_ARRAY_BUFFER, 0);
162     glBindVertexArray(0);
163     glDeleteVertexArrays(1, &VaoId);
164 }
165 
166 // Creates and compiles shader programs
167 void CreateShaders(void)
168 {
169     ProgramId = LoadShaders("Shader.vert", "Shader.frag");
170     glUseProgram(ProgramId);
171 }
172 
173 // Cleanup function to destroy shader programs
174 void DestroyShaders(void)
175 {
176     glDeleteProgram(ProgramId);
177 }
178 
179 // Main cleanup function called when program exits
180 void Cleanup(void)
181 {
182     DestroyShaders();
183     DestroyVBO();
184 };
185 
186 // Initialize rendering parameters and load 3D model
187 void Initialize(void)
188 {
189     // Set background color to match dawn/dusk atmosphere
190     glClearColor(0.15f, 0.10f, 0.12f, 1.0f);  // Dark purple-orange sky
191 
192     // Load the 3D model from OBJ file format
193     bool model = loadOBJ("Assets/Island.obj", vertices, uvs, normals,
```

```
    colors);
    nrVertices = vertices.size();

    // Create VBO and shader programs
    CreateVBO();
    CreateShaders();

    // Get uniform locations from shader program for later use
    nrVertLocation = glGetUniformLocation(ProgramId, "nrVertices");
    myMatrixLocation = glGetUniformLocation(ProgramId, "myMatrix");
    viewPosLocation = glGetUniformLocation(ProgramId, "viewPos");
    viewLocation = glGetUniformLocation(ProgramId, "view");
    projLocation = glGetUniformLocation(ProgramId, "projection");

    // Get uniform locations for all three light sources
    light1PosLocation = glGetUniformLocation(ProgramId, "light1Pos");
    light1ColorLocation = glGetUniformLocation(ProgramId, "light1Color")
    ;
    light2PosLocation = glGetUniformLocation(ProgramId, "light2Pos");
    light2ColorLocation = glGetUniformLocation(ProgramId, "light2Color")
    ;
    light3PosLocation = glGetUniformLocation(ProgramId, "light3Pos");
    light3ColorLocation = glGetUniformLocation(ProgramId, "light3Color")
    ;

    // Pass initial values to shaders
    glUniform1i(ProgramId, nrVertices);
}

// Main rendering function called every frame
void RenderFunction(void)
{
    // Clear color and depth buffers to prepare for new frame
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST); // Enable depth testing for proper 3D
    rendering

    // Set up model transformation matrix (rotations to orient the model
    correctly)
    myMatrix = glm::rotate(glm::mat4(1.0f), PI / 2, glm::vec3(0.0, 1.0,
    0.0)) *
        glm::rotate(glm::mat4(1.0f), PI / 2, glm::vec3(0.0, 0.0, 1.0));
    glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);

    // Calculate observer position using spherical coordinates
    obsX = refX + dist * cos(alpha) * cos(beta);
    obsY = refY + dist * cos(alpha) * sin(beta);
    obsZ = refZ + dist * sin(alpha);

    // Define view matrix vectors
    glm::vec3 obs = glm::vec3(obsX, obsY, obsZ);        // Observer
    position
    glm::vec3 pctRef = glm::vec3(refX, refY, refZ);     // Point to look
    at (origin)
    glm::vec3 vert = glm::vec3(vX, vY, vZ);             // Up vector

    // Send observer position to shader for lighting calculations
    glUniform3f(viewPosLocation, obsX, obsY, obsZ);
```

```
243
244     // Create and send view matrix to shader
245     view = glm::lookAt(obs, pctRef, vert);
246     glUniformMatrix4fv(viewLocation, 1, GL_FALSE, &view[0][0]);
247
248     // Set up perspective projection with infinite far plane
249     projection = glm::infinitePerspective(GLfloat(fov), GLfloat(width) /
        GLfloat(height), dNear);
250     glUniformMatrix4fv(projLocation, 1, GL_FALSE, &projection[0][0]);
251
252     // Send light parameters to shaders for all three light sources
253     glUniform3f(light1PosLocation, light1Pos.x, light1Pos.y, light1Pos.z
        );
254     glUniform3f(light1ColorLocation, light1Color.r, light1Color.g,
        light1Color.b);
255
256     glUniform3f(light2PosLocation, light2Pos.x, light2Pos.y, light2Pos.z
        );
257     glUniform3f(light2ColorLocation, light2Color.r, light2Color.g,
        light2Color.b);
258
259     glUniform3f(light3PosLocation, light3Pos.x, light3Pos.y, light3Pos.z
        );
260     glUniform3f(light3ColorLocation, light3Color.r, light3Color.g,
        light3Color.b);
261
262     // Bind VAO and draw the 3D model
263     glBindVertexArray(VaoId);
264     glEnableVertexAttribArray(0); // Enable position attribute
265     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);
266     glDrawArrays(GL_TRIANGLES, 0, vertices.size()); // Render all
        triangles
267
268     // Swap buffers for double buffering
269     glutSwapBuffers();
270     glFlush();
271 }
272
273 int main(int argc, char* argv[])
274 {
275     glutInit(&argc, argv);
276     glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
277     glutInitWindowPosition(100, 100);
278     glutInitWindowSize(1200, 900);
279     glutCreateWindow("Haunted house");
280
281     glewInit();
282     Initialize();
283
284     //register callback functions
285     glutIdleFunc(RenderFunction);
286     glutDisplayFunc(RenderFunction);
287     glutKeyboardFunc(processNormalKeys);
288     glutSpecialFunc(processSpecialKeys);
289     glutCloseFunc(Cleanup);
290
291     glutMainLoop();
292 }
```

5.2 objloader.hpp

```cpp
#pragma once
#include <vector>
#include <string>
#include "glm/glm.hpp"
#include <map>

struct Material {
    std::string name;
    glm::vec3 diffuseColor = glm::vec3(0.8f, 0.8f, 0.8f); // Kd
};

bool loadOBJ(
    const char* path,
    std::vector<glm::vec3>& out_vertices,
    std::vector<glm::vec2>& out_uvs,
    std::vector<glm::vec3>& out_normals,
    std::vector<glm::vec3>& out_colors // nou!
);


bool loadAssImp(
    const char * path,
    std::vector<unsigned short> & indices,
    std::vector<glm::vec3> & vertices,
    std::vector<glm::vec2> & uvs,
    std::vector<glm::vec3> & normals
);
```

5.3 objloader.cpp

```cpp
// Adaptat dupa http://www.opengl-tutorial.org/
#ifdef _MSC_VER
#define _CRT_SECURE_NO_WARNINGS
#endif

#include <vector>
#include <stdio.h>
#include <string>
#include <cstring>
#include <iostream>
#include "glm/glm.hpp"

#include "objloader.hpp"

bool loadMTL(const std::string& path, std::map<std::string, Material>&
    materials) {
    FILE* file = fopen(path.c_str(), "r");
    if (!file) return false;

    char lineHeader[128];
    Material mat;
    std::string currentMat;
    while (fscanf(file, "%s", lineHeader) != EOF) {
        if (strcmp(lineHeader, "newmtl") == 0) {
            if (!currentMat.empty()) {
```

```
                materials[currentMat] = mat;
            }
            char name[128];
            fscanf(file, "%s\n", name);
            currentMat = name;
            mat = Material();
            mat.name = currentMat;
        }
        else if (strcmp(lineHeader, "Kd") == 0) {
            fscanf(file, "%f %f %f\n", &mat.diffuseColor.r, &mat.
    diffuseColor.g, &mat.diffuseColor.b);
        }
        else {
            char buffer[1024];
            fgets(buffer, 1024, file);
        }
    }
    if (!currentMat.empty()) {
        materials[currentMat] = mat;
    }
    fclose(file);
    return true;
}

bool loadOBJ(
    const char* path,
    std::vector<glm::vec3>& out_vertices,
    std::vector<glm::vec2>& out_uvs,
    std::vector<glm::vec3>& out_normals,
    std::vector<glm::vec3>& out_colors // nou!
)
{
    printf("Loading OBJ file %s...\n", path);

    std::vector<unsigned int> vertexIndices, uvIndices, normalIndices,
    materialIndices;
    std::vector<glm::vec3> temp_vertices;
    std::vector<glm::vec2> temp_uvs;
    std::vector<glm::vec3> temp_normals;
    std::vector<std::string> faceMaterials;

    std::map<std::string, Material> materials;
    std::string currentMaterial = "";
    std::string mtlFile = "";

    // Determin  directorul fi ierului OBJ
    std::string objPath(path);
    std::string directory = objPath.substr(0, objPath.find_last_of("/\\"
    ) + 1);

    FILE* file = fopen(path, "r");
    if (file == NULL) {
        printf("Impossible to open the file !\n");
        return false;
    }

    char lineHeader[128];
    while (fscanf(file, "%s", lineHeader) != EOF) {
```

```
80          if (strcmp(lineHeader, "v") == 0) {
81              glm::vec3 vertex;
82              fscanf(file, "%f %f %f\n", &vertex.x, &vertex.y, &vertex.z);
83              temp_vertices.push_back(vertex);
84          }
85          else if (strcmp(lineHeader, "vt") == 0) {
86              glm::vec2 uv;
87              fscanf(file, "%f %f\n", &uv.x, &uv.y);
88              uv.y = -uv.y;
89              temp_uvs.push_back(uv);
90          }
91          else if (strcmp(lineHeader, "vn") == 0) {
92              glm::vec3 normal;
93              fscanf(file, "%f %f %f\n", &normal.x, &normal.y, &normal.z);
94              temp_normals.push_back(normal);
95          }
96          else if (strcmp(lineHeader, "f") == 0) {
97              unsigned int vertexIndex[3], uvIndex[3], normalIndex[3];
98              int matches = fscanf(file, "%d/%d/%d %d/%d/%d %d/%d/%d\n",
99                  &vertexIndex[0], &uvIndex[0], &normalIndex[0],
100                 &vertexIndex[1], &uvIndex[1], &normalIndex[1],
101                 &vertexIndex[2], &uvIndex[2], &normalIndex[2]);
102             if (matches != 9) {
103                 printf("File can't be read by our simple parser :-( Try
    exporting with other options\n");
104                 fclose(file);
105                 return false;
106             }
107             for (int i = 0; i < 3; ++i) {
108                 vertexIndices.push_back(vertexIndex[i]);
109                 uvIndices.push_back(uvIndex[i]);
110                 normalIndices.push_back(normalIndex[i]);
111                 faceMaterials.push_back(currentMaterial);
112             }
113         }
114         else if (strcmp(lineHeader, "mtllib") == 0) {
115             char mtlName[128];
116             fscanf(file, "%s\n", mtlName);
117             mtlFile = directory + mtlName;
118             loadMTL(mtlFile, materials);
119         }
120         else if (strcmp(lineHeader, "usemtl") == 0) {
121             char matName[128];
122             fscanf(file, "%s\n", matName);
123             currentMaterial = matName;
124         }
125         else {
126             char buffer[1024];
127             fgets(buffer, 1024, file);
128         }
129     }
130     fclose(file);
131
132     // Asambleaz  datele finale, inclusiv culoarea
133     for (unsigned int i = 0; i < vertexIndices.size(); i++) {
134         unsigned int vertexIndex = vertexIndices[i];
135         unsigned int uvIndex = uvIndices[i];
136         unsigned int normalIndex = normalIndices[i];
```

```
137
138         glm::vec3 vertex = temp_vertices[vertexIndex - 1];
139         glm::vec2 uv = temp_uvs[uvIndex - 1];
140         glm::vec3 normal = temp_normals[normalIndex - 1];
141
142         out_vertices.push_back(vertex);
143         out_uvs.push_back(uv);
144         out_normals.push_back(normal);
145
146         // Adaug  culoarea materialului curent
147         glm::vec3 color(0.8f, 0.8f, 0.8f);
148         if (!faceMaterials[i].empty() && materials.count(faceMaterials[i
    ])) {
149             color = materials[faceMaterials[i]].diffuseColor;
150         }
151         out_colors.push_back(color);
152     }
153     return true;
154 }
```

5.4 Shader.frag

```
1  #version 330 core
2
3  // Inputs from vertex shader (interpolated per fragment)
4  in vec3 fragPos;       // Fragment position in world space
5  in vec3 fragNormal;    // Surface normal vector
6  in vec3 fragColor;     // Material color
7
8  // Output color
9  out vec4 FragColor;
10
11 // Uniforms - same for all fragments
12 uniform vec3 viewPos;      // Camera position
13
14 // Three light sources
15 uniform vec3 light1Pos;
16 uniform vec3 light1Color;
17 uniform vec3 light2Pos;
18 uniform vec3 light2Color;
19 uniform vec3 light3Pos;
20 uniform vec3 light3Color;
21
22 //calculates lighting from one point light using Phong model
23 vec3 calculateLight(vec3 lightPos, vec3 lightColor, vec3 normal, vec3
    fragPosition, vec3 viewDirection)
24 {
25     //DIFFUSE LIGHTING
26     //makes surfaces brighter when facing the light
27     vec3 lightDir = normalize(lightPos - fragPosition);  // Direction to
     light
28     float diff = max(dot(normal, lightDir), 0.0);        // How much
    surface faces light (0-1)
29     vec3 diffuse = diff * lightColor;                    // Apply light
    color
30
31
```

```
    //SPECULAR LIGHTING
    //creates shiny highlights on surfaces
    float specularStrength = 0.1;                          // How shiny
   the material is
    vec3 reflectDir = reflect(-lightDir, normal);          // Reflection
   direction
    float spec = pow(max(dot(viewDirection, reflectDir), 0.0), 32);  //
   Shininess
    vec3 specular = specularStrength * spec * lightColor;


    //ATTENUATION
    //light weakens with distance
    float distance = length(lightPos - fragPosition);
    float attenuation = 1.0 / (1.0 + 0.5 * distance + 0.3 * (distance *
   distance));

    //applying distance falloff to both diffuse and specular
    diffuse *= attenuation;
    specular *= attenuation;

    return diffuse + specular;
}

void main()
{
    //AMBIENT LIGHTING
    //the base lighting that illuminates everything equally
    float ambientStrength = 0.03;                    // How bright the
   ambient light is
    vec3 ambient = ambientStrength * vec3(0.9, 0.7, 0.8);  // Ambient
   color (soft purple-pink)


    // Normalize vectors for calculations
    vec3 norm = normalize(fragNormal);                // Ensure normal is
    unit length
    vec3 viewDir = normalize(viewPos - fragPos);      // Direction from
   fragment to camera


    //Calculate contribution from each light source
    vec3 light1 = calculateLight(light1Pos, light1Color, norm, fragPos,
   viewDir);
    vec3 light2 = calculateLight(light2Pos, light2Color, norm, fragPos,
   viewDir);
    vec3 light3 = calculateLight(light3Pos, light3Color, norm, fragPos,
   viewDir);


    //Combining all lighting components
    vec3 totalLight = ambient + light1 + light2 + light3;


    //multiplying lighting by material color to get final result
    vec3 result = totalLight * fragColor;


```

```
    //Output final color (RGB + alpha= 1.0 for opaque)
    FragColor = vec4(result, 1.0);
}
```

5.5 Shader.vert

```
#version 330 core

layout(location = 0) in vec3 in_position;
layout(location = 1) in vec3 in_normal;
layout(location = 2) in vec3 in_color;

out vec3 fragPos;
out vec3 fragNormal;
out vec3 fragColor;

uniform mat4 myMatrix;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    // Transform position to world space
    vec4 worldPos = myMatrix * vec4(in_position, 1.0);
    fragPos = worldPos.xyz;

    // Transform normal to world space (using normal matrix)
    fragNormal = mat3(transpose(inverse(myMatrix))) * in_normal;

    // Pass color to fragment shader
    fragColor = in_color;

    // Final position
    gl_Position = projection * view * worldPos;
}
```