

Cuprins

1. Introducere	3
2. Prezentarea generală a aplicației	4
3. Împărțirea aplicației	9
4. Cerințe minime de sistem.....	13
5. Bibliografie	14

1. Introducere

Aplicația Terra Nova este destinată curioșilor, care vor să-și îmbunătățească cultura generală prin explorarea unor locuri nemaiauzite până acum. Terra Nova pune la dispoziția privitorului șansa de a accesa harta lumii și de a-și alege propria destinație dorită. Aplicația se adresează persoanelor de orice vîrstă, întrucât este ușor de folosit și dispune de o multitudine de atracții interesante. Am ales să realizez acest atestat sub această temă, deoarece am o pasiune pentru călătorit și niciodată nu mă satur să aflu informații noi despre lucrurile pe care le-am vizitat sau urmează să le vizitez.

Aplicația se deschide cu pagina principală unde, după încărcare, se poate apăsa tasta Enter pentru a continua la următoarea fereastră. Am ales un design cât mai retro, vintage, cu paletă de culori care să susțină ideea designului propus. Odată ajuns în a doua parte a atestatului, utilizatorului îi se va deschide harta lumii, fiecare continent cu câte o busolă. Apăsând pe busolă, harta se mărește pe locul dorit. Majoritatea țărilor vor avea la rîndul lor o mică busolă care, odată apăsată, îl va direcționa la o fereastră cu informații despre locul ales.

Pentru a ieși din fereastra cu informații, se apasă click dreapta pe mouse, iar pentru a ieși din modul de zoom, se utilizează scroll-ul de la mouse pentru a da înapoi. Aceste comenzi se pot realiza de un număr nelimitat de ori, aplicația fiind construită în aşa fel încât utilizatorul să aibă acces la toate țările cu informații de câte ori dorește.

2. Prezentarea generală a aplicației

Aplicația a fost realizată în Visual Studio 2017 Community Edition, utilizând limbajul de programare C++ 11. Acesta a fost ales datorită flexibilității oferite în legătură cu tehniciile de programare, dar și a compatibilității între sistemele de operare. Pentru reprezentarea vizuală a interfeței a fost folosită librăria grafică OpenGL, care la rândul ei poate fi accesată de pe orice platformă. Pentru procesele intermediare afișării graficii în cadrul aplicației au fost utilizate următoarele librării externe:

- Lua (limbaj de programare abstract care se ocupă cu generarea soluției)
- Stb-Image (librărie scrisă în C++ responsabilă cu încărcarea unei imagini externe în aplicație)
- Gml (librărie folosită cu scopul de a ușura și de a optimiza calculul matematic necesar aplicației)
- Premake5 (aplicație responsabilă cu compilarea și executarea fișierelor scrise în Lua)

Principalele caracteristici folosite din OpenGL au fost shaderele și texturile.

Shaderele sunt programe care sunt compilate și executate pe placa video și sunt responsabile de culoare fiecărui pixel. În cadrul aplicației acestea au fost reprezentate prin clase, clasele fiind responsabile de încărcarea, ștergerea și folosirea shaderului.

```
class shader
{
public:
    shader() = default;
    shader(const std::string& vertex_shader_path, const std::string& fragment_shader_path);
    ~shader();

    void bind();

    void set_uniform_1i(const std::string& uniform_name, int value);
    void set_uniform_1f(const std::string& uniform_name, float value);
    void set_uniform_mat4(const std::string& uniform_name, const glm::mat4& value);

private:
    unsigned int shader_id;

private:
    std::string read_shader(const std::string& shader_path);
    unsigned int compile_shader(unsigned int shader_type, const std::string& shader_path);
};
```

Încărcarea shaderelor din fișierele de tip text se realizează cu ajutorul librăriei fstream. Textul din interiorul fișierelor este inițial stocat în variabile de tip std::string, iar apoi este trimis către OpenGL pentru a fi compilat.

```
std::string shader::read_shader(const std::string& shader_path)
{
    std::string shader_source;
    std::ifstream shader_file(shader_path);

    std::string current_line;
    while (std::getline(shader_file, current_line))
        shader_source += current_line + '\n';

    shader_file.close();
    return shader_source;
}
```

```
unsigned int shader::compile_shader(unsigned int shader_type, const std::string& shader_path)
{
    unsigned int shader_program = glCreateShader(shader_type);

    std::string shader_source_code = read_shader(shader_path);
    const char* shader_source = shader_source_code.c_str();
    glShaderSource(shader_program, 1, &shader_source, 0);
    glCompileShader(shader_program);

    int result;
    glGetShaderiv(shader_program, GL_COMPILE_STATUS, &result);

    if (!result)
    {
        int lenght;
        glGetShaderiv(shader_program, GL_INFO_LOG_LENGTH, &lenght);

        char message[512];
        glGetShaderInfoLog(shader_program, lenght, &lenght, message);
        std::cout << message << std::endl;

        glDeleteShader(shader_program);

        return 0;
    }

    return shader_program;
}
```

Texturile sunt o colecție de informații legate de o imagine în formatul jpg, png, etc. În reprezentarea texturilor, mecanismul de utilizare este întocmai ca la shadere.

```
class texture
{
public:
    // constructor- atunci cand fac un nou obiect de tipu "texture" se cheama constructorul declarat aici dar definit in "Texture.cpp"
    texture(const std::string& filepath);
    // deconstructor - cand se distrughe un obiect de tipu asta se cheama
    ~texture();

    void bind(int slot = 0) const;

private:
    // variabile specifice unei texturi
    unsigned int texture_id;
    std::string texture_filepath;

    int width;
    int height;
    int bits_per_pixel;

    unsigned int format = 0;
    unsigned int internal_format = 0;
};
```

Pentru a ușura mecanismul de afișare a texturilor și calculul din spatele afișării a fost creată clasa statică „renderer”.

```
class renderer
{
public:
    static void init();
    static void close();
    static void clear(float r, float g, float b);

    static void draw(const texture& tex, glm::vec2 pos, glm::vec2 size, float px = 1.0f, float py = 1.0f);
    static void set_camera(const camera& cam);

private:
    static vertex vertex_elements;

    static shader* shader_program;
    static glm::mat4 renderer_camera;

private:
    // constructorul si destrucotrul sunt functii private pentru ca nu vreau sa creez obiecte de tipul "renderer"
    renderer() = default;
    ~renderer() = default;
};
```

- funcția „init” ia locul constructorului, iar în ea sunt inițializate toate variabilele care mai apoi vor fi folosite pe parcursul executării programului.
- funcția „close” eliberează memoria prin ștergerea variabilelor alocate la pornirea aplicației.
- funcția „clear”, cu cei 3 parametrii ai săi r,g,b, reprezentând cele 3 culori primare din programarea grafică, are scopul de a curăța ecranul și de a-l colora cu culoarea introdusă prin intermediul parametrilor

- funcția „draw” are 5 parametrii: primul parametru reprezintă textura pe care vrem să o afișăm; al doilea parametru coordonatele la care vrem să afișăm, iar al treilea dimensiunea texturii. Ultimii doi parametrii sunt optionali, cu ajutorul lor este posibilă afișarea parțială a texturii, tehnică folosită în animația de la început, unde se deschide harta.

```
void renderer::draw(const texture& tex, glm::vec2 pos, glm::vec2 size, float px, float py)
{
    glm::mat4 model = glm::translate(glm::mat4(1.0f), glm::vec3(pos, 0.0f)) * glm::scale(glm::mat4(1.0f), glm::vec3(size, 1.0f));

    glBindVertexArray(vertex_elements.vertex_array);
    glBindBuffer(GL_ARRAY_BUFFER, vertex_elements.vertex_buffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vertex_elements.index_buffer);

    shader_program->bind();

    tex.bind();
    shader_program->set_uniform_1i("tex", 0);
    shader_program->set_uniform_mat4("ortho_matrix", renderer_camera);
    shader_program->set_uniform_mat4("model_matrix", model);

    shader_program->set_uniform_1f("u_px", px);
    shader_program->set_uniform_1f("u_py", py);

    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
}
```

Prima etapă a afișării texturii este formarea matricei care reprezintă poziția și dimensiunea texturii. După urmează conectarea seturilor de date necesare afișării, iar apoi trimiterea tuturor datelor către shader. Ultima etapă este afișarea celor 2 triunghiuri care formează textura.

-funcția „set_camera” obține matricea camerei primită ca parametru și o folosește pentru afișările viitoare

De exemplu, pentru a afișa textura de pe ecranul principal și textura care indică faptul că aplicația se încarcă este de ajuns să:

```
renderer::draw(main_menu, { screen_width / 2.0f, screen_height / 2.0f }, { screen_width, screen_height });
renderer::draw/loading, { screen_width / 2.0f, screen_height / 2.0f - 100.0f }, { 300.0f, 50.0f });
```

Clasa renderer folosește un singur shader pentru a afișa tot conținutul aplicației. Acest shader este compus din două părți:

- vertex shader (acesta se ocupă de poziționarea corectă a colțurilor triunghiului pe ecran și de trimiterea datelor primite prin vertex buffer către a doua parte a shaderului)

```

#version 330 core

layout(location = 0) in vec2 vertex_position;
layout(location = 1) in vec2 texture_coordinate;

out vec2 tex_coord;
uniform mat4 ortho_matrix;
uniform mat4 model_matrix;

uniform float u_px;
uniform float u_py;

void main()
{
    gl_Position = ortho_matrix * model_matrix * vec4(vertex_position, 0.0, 1.0);
    tex_coord.x = texture_coordinate.x * u_px;
    tex_coord.y = texture_coordinate.y * u_py;
}

```

- fragment shader (acesta se ocupă cu colorarea pixelilor din interiorul triunghiului delimitat de către cele 3 colțuri calculate în faza anterioară a shaderului)

```

#version 330 core

out vec4 pixel_color;

in vec2 tex_coord;
uniform sampler2D tex;

void main()
{
    pixel_color = texture(tex, tex_coord);
}

```

Aplicația este constituită din trei părți:

1. Ecran principal
2. Ecran de navigare
3. Ecran de afișare a informațiilor

Acste trei ecrane sunt reprezentate prin intermediul unei enumerații și a unei variabile:

```
application_state app_state = application_state::main_menu;
```

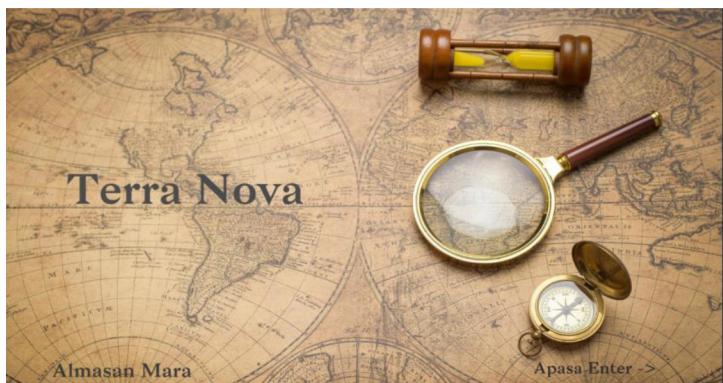
```

enum class application_state
{
    main_menu,
    navigation,
    information_tab,
};

```

3. Împărțirea aplicației

3.1. Ecran principal



Ecranul principal reprezintă prima interacțiune a utilizatorului cu aplicația. Acesta oferă privitorului ocazia de a-și contura o primă impresie legată de tema propusă. Din punct de vedere tehnic acesta are și scopul de a încărca toate resursele necesare.

Acest ecran relaționează numai cu tasta Enter care face trecerea la următoarea pagină.

```
if (input::is_key_pressed(GLFW_KEY_ENTER))
    app_state = application_state::navigation;
```

3.2. Ecran de navigare

Ecranul de navigare este partea principală a aplicației, fiind formată din două componente:

1. Camera

Această componentă a fost creată cu scopul de a oferi o experiență fluidă. Ea se poate apropia de obiectivul turistic selectat sau de centrul hărții. Prin mișcarea mouse-ului ea se poate deplasa numai în interiorul hărții. Pentru a nu permite camerei să iasă din cadrul principal, au fost approximate coordonatele celor 4 colțuri ale sale, acestea aflându-se într-un sistem de coordonate diferit fata de cel al hărții și s-au verificat cu marginile mediului în care se poate deplasa:

```

//calculez colturile camerei de la pozitia camerei + - cat pot sa vad din harta( nivelul de zoom)
int right = res_x / 2.0f + position.x + (res_x / 2.0f / zoom_level);
int left = res_x / 2.0f + position.x - (res_x / 2.0f / zoom_level);
int top = res_y / 2.0f + position.y + (res_y / 2.0f / zoom_level);
int bottom = res_y / 2.0f + position.y - (res_y / 2.0f / zoom_level);

//verific colturile cu marginea hartii(in caz caiese din harta repositionez camera)
if (right > res_x)
    position.x = res_x / 2.0f - (res_x / 2.0f / zoom_level);
if (left < 0)
    position.x = -res_x / 2.0f + (res_x / 2.0f / zoom_level);
//verific stanga-dreapta, sus-jos
if (top > res_y)
    position.y = res_y / 2.0f - (res_y / 2.0f / zoom_level);
if (bottom < 0)
    position.y = -res_y / 2.0f + (res_y / 2.0f / zoom_level);

```

1. Harta

La deschiderea aplicației unul dintre primele lucruri pe care aceasta le face este de a încărca din fișiere datele utilizate pentru a afișa harta într-un mod corect, fiind poziționate sub rezoluția dorită de aplicație:

coordonata x	coordonata y	numele fișierului de tip png care corespunde locației
--------------	--------------	-------------------------------------------------------

Exemplu:

```

0.489955 0.56286 val_doricia_italia
0.281433 0.357398 valea_sacra_peru
0.542469 0.637739 veliky_novgorod_rusia
0.582387 0.543221 vulcanul_garasu_azerbaijan
0.71435 0.455871 wat_pao_tailand

```

La crearea aplicației a fost utilizată rezoluția de 1366 x 768, pentru a evita poziționarea incorectă a locațiilor, coordonatele acestora au fost împărțite la rezoluția folosită, pentru a fi mai apoi înmulțite cu rezoluția dorită.

```

void read_locations_from_file(const std::string& filepath, std::vector<std::pair<pinpoint, std::string>>& pp)
{
    std::ifstream f(filepath.c_str());
    if (f.good())
    {
        int size;
        f >> size;
        pp.resize(size);

        for (int i = 0; i < size; i++)
        {
            f >> pp[i].first.position.x >> pp[i].first.position.y >> pp[i].second;
            pp[i].first.position.x *= screen_width;
            pp[i].first.position.y *= screen_height;
        }
    }
}

```

Aceste calcule se aplică și pentru continente. După această etapă, urmează încărcarea imaginilor specifice fiecărei locații.

```
// aici incarc imaginile specifice fiecarii locatii
for(const auto& location : locations)
    location_textures.push_back(new texture("Resources/Textures/Locatii/" + location.second + ".png"));
```

Afișare imaginilor se face prin selectarea și apăsarea busolelor de către utilizator.



După un anumit nivel de zoom al camerei se face trecerea de la afișarea continentelor la afișarea țărilor cu atracțiile turistice specifice. Apăsarea unei busole redirecționează utilizatorul către pagina cu informații despre locația aleasă.

Pentru a ușura preluarea evenimentelor realizate la nivelul tastaturii și mouse-ului a fost creată clasa statică „input”.

```
class input
{
public:
    static void init(GLFWwindow* wnd);

    static bool is_key_pressed(int code);
    static bool is_mouse_key_pressed(int code);
    static glm::vec2 get_mouse_position();

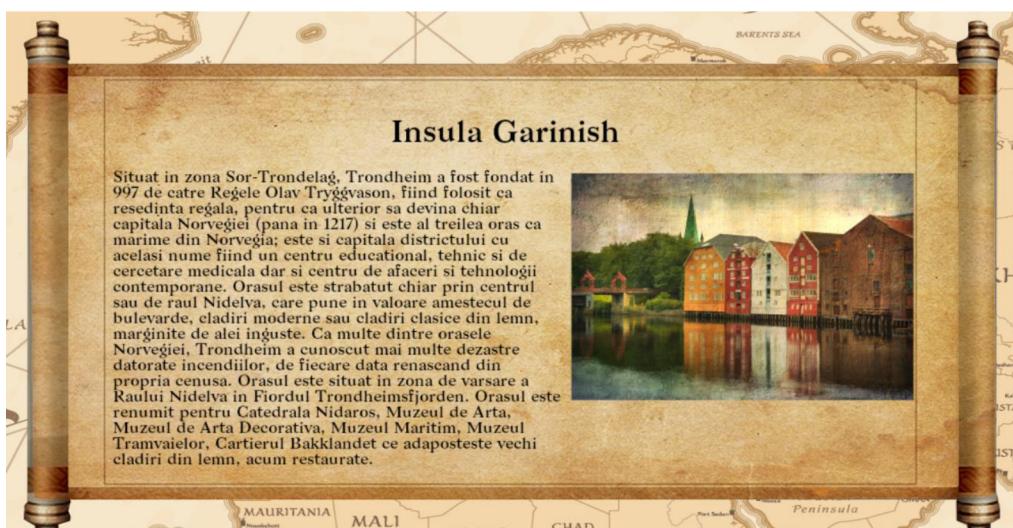
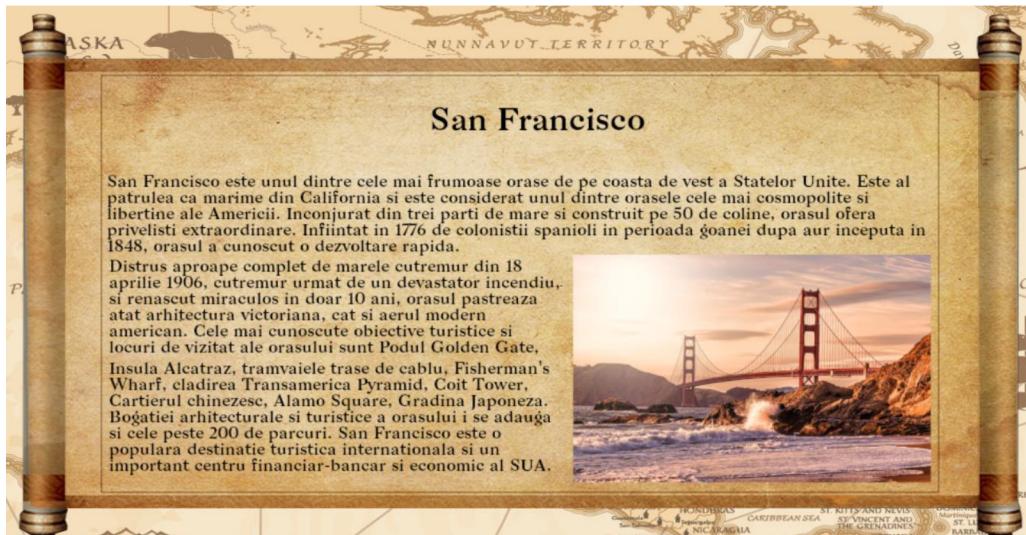
    static int scroll_state;

private:
    static GLFWwindow* window;
};
```

3.3 Ecran de afişare a informaţiilor

Acest ecran se deschide cu animaţia de desfacere a pergamentului care a fost realizată prin manipularea poziţiilor capetelor acestuia şi afişarea parţială a texturii dintre. Texturile au fost încărcate pe baza numelui locaţiei atribuit fiecărei busole, după cum am menţionat şi mai sus.

Informaţiile sunt sub formă de poze, realizate în aplicaţia GIMP.



4. Cerințe minime de sistem

Sistem de operare: Windows, Linux, Vista

Procesor: Intel HD Family

Placa video: integrată sau dedicată

Memorie RAM: 256 Mb

Audio: nu e necesar

Display: 1366 x 768 sau mai mult

Dispozitiv de conectare: Computer

Conexiune la internet: nu este necesară

5. Bibliografie

<https://learnopengl.com/Introduction>

<https://docs.microsoft.com/en-us/cpp/cpp/?view=msvc-170>

<https://www.cplusplus.com/doc/tutorial/>

<https://docs.g1/>

<https://www.eturia.ro/>

<https://www.youtube.com/watch?v=45MIykWJ-C4>

<http://www.opengl-tutorial.org/>

<https://riptutorial.com/opengl>

<https://riptutorial.com/opengl>

<https://ogldev.org/index.html>

<https://blogs.oregonstate.edu/learnfromscratch/>

<https://ocw.cs.pub.ro/courses/spg/laboratoare/00>