

*Proiect*

*Sisteme de prelucrare grafică*

*Nume :* Podaru Bogdan-Mihai

*Facultatea* : Automatic*ă* si Calculatoare

*Specializarea* : Calculatoare

*Grupa* : 30238

*Data :* 12-01-2020

***1 . Cuprins :***

* Prezentarea temei
* Scenariul
* Detalii de implementare
* Prezentarea interfeței grafice utilizator / manual de utilizare
* Concluzii și dezvoltări ulterioare
* Referințe

***2. Prezentarea temei***

Tema presupune realizarea unei aplicații grafice interactive prin următoarele : selectarea obiectelor ce vor alcătui scena, selectarea texturilor ce se vor mapa pe obiecte, realizarea efectului de lumină prin intermediul a 2 surse diferite si cu ajutorul diferiților algoritmi, realizarea efectului de umbră, de ceață , animații ale obiectelor, ale scenei, deplasări in scenă cu ajutorul mouse-ului si a tastaturii. Proiectul consolideaza cunostiințele dobandite pe parcursul laboratoarelor. Acesta dezvoltă si latura artistică deoarece pentru asamblarea scenei s-a folosit Blender 2.7 si 2.8.

***3. Scenariul***

Propun următorul scenariu : Intrucat stiința se dezvoltă foarte mult si oamenii devin tot mai insetați de “necunoscut”, o echipa de cercetare antrenată corespunzător astronauților, a plecat intr-o misiune pe Marte, scopul lor : prelevarea datelor si mostrelor din zonele cu ceață de pe Marte .

Personajul principal, de asemenea un astronaut are capabilitatea de a se deplasa mai ușor în aer datorită lipsei de gravitație. Pentru comunicarea cu stația de la NASA de acasă au transportat de asemenea un mic satelit. Nava cu care au sosit a fost andocată corespunzător în schela demontabilă, care se află în navă. Pentru că misiunea lor durează 10 ani echipa de cercetare a construit 6 baze spațiale pentru oxigen și adăpost.

***Obiecte (+texturi):***

* *nava spatial*ă
* *schela metalic*ă
* *6 baze pentru stațiune*
* *rover pentru deplasare*
* *panouri solare pentru alimentare cu energie electric*ă
* *antene radio*
* *satelit pentru comunicare*
* *telescop*
* *nave masive transport*
* *hal*ă *reparații*
* *skybox Mars*

***Functionalit***ă***ti:***

* *efecte de lumin*ă
* *efecte de ceață*
* *deplasarea cu ajutorul mouse-ului și a tastaturii*
* *animație satelit*

***4. Detalii de implementare***

***4.1. funcții și algoritmi***

***4.2. modelul grafic***

***4.3. structuri de date***

***4.4 ierarhia de clase***

***4.1.funcții și algoritmi***

*Modulul Camera.cpp conține metodele folosite pentru alegerea și schimbarea poziției obiectivului. Metoda „move” primește ca și parametri direcția și viteza de deplasare și translateaz*ă *pe aceast*ă *direcție.*

***void Camera::move(MOVE\_DIRECTION direction, float speed)***

{

switch (direction) {

case MOVE\_FORWARD:

cameraPosition += cameraDirection \* speed;

break;

case MOVE\_BACKWARD:

cameraPosition -= cameraDirection \* speed;

break;

case MOVE\_RIGHT:

cameraPosition += cameraRightDirection \* speed;

break;

case MOVE\_LEFT:

cameraPosition -= cameraRightDirection \* speed;

break;

}

***}***

*Pentru rotirea pe 2 axe se folosește metoda „rotate”.*

***void Camera::rotate(float pitch, float yaw)***

{

glm::vec3 front;

front.x = cos(glm::radians(yaw)) \* cos(glm::radians(pitch));

front.y = sin(glm::radians(pitch));

front.z = sin(glm::radians(yaw)) \* cos(glm::radians(pitch));

cameraDirection = glm::normalize(front);

cameraRightDirection = glm::normalize(glm::cross(this->cameraDirection, glm::vec3(0.0f, 1.0f, 0.0f)));

***}***

*Metodele „keyboardCallback” și „mouseCallback” regleaza interacțiunile utilizatorului cu tastatura, respectiv mouse-ul .*

*void* ***keyboardCallback(GLFWwindow\* window, int key, int scancode, int action, int mode)***

{

if (key == GLFW\_KEY\_ESCAPE && action == GLFW\_PRESS)

glfwSetWindowShouldClose(window, GL\_TRUE);

if (key >= 0 && key < 1024)

{

if (action == GLFW\_PRESS)

pressedKeys[key] = true;

else if (action == GLFW\_RELEASE)

pressedKeys[key] = false;

}

if (glfwGetKey(glWindow, GLFW\_KEY\_F))

{

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

}

if (glfwGetKey(glWindow, GLFW\_KEY\_G))

{

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL);

}

if (glfwGetKey(glWindow, GLFW\_KEY\_H))

{

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_POINT);

}

if (glfwGetKey(glWindow, GLFW\_KEY\_SPACE))

{

toggle\_fog = 1-toggle\_fog;

}

***}***

*void* ***mouseCallback(GLFWwindow \* window, double xpos, double ypos)***

{

if (canControl) {

if (firstMouse)

{

lastX = xpos;

lastY = ypos;

firstMouse = false;

}

float xoffset = xpos - lastX;

float yoffset =lastY - ypos; //reversed since y-coordinates go from bottom to top

lastX = xpos;

lastY = ypos;

float sensitivity = 0.1f; // change this value to your liking

xoffset \*= sensitivity;

yoffset \*= sensitivity;

yaw += xoffset;

pitch += yoffset;

// make sure that when pitch is out of bounds, screen doesn't get flipped

if (pitch > 89.0f)

pitch = 89.0f;

if (pitch < -89.0f)

pitch = -89.0f;

myCamera.rotate(pitch, yaw);

}

***}***

*Acțiunea fiec*ă*rei taste este definit*ă *în metoda „processMovement”:*

*void processMovement()*

{

if (pressedKeys[GLFW\_KEY\_Q]) {

angle += 1.0f;

if (angle > 360.0f)

angle -= 360.0f;

}

if (pressedKeys[GLFW\_KEY\_E]) {

angle -= 1.0f;

if (angle < 0.0f)

angle += 360.0f;

}

if (pressedKeys[GLFW\_KEY\_W]) {

myCamera.move(gps::MOVE\_FORWARD, cameraSpeed);

}

if (pressedKeys[GLFW\_KEY\_S]) {

myCamera.move(gps::MOVE\_BACKWARD, cameraSpeed);

}

if (pressedKeys[GLFW\_KEY\_A]) {

myCamera.move(gps::MOVE\_LEFT, cameraSpeed);

}

if (pressedKeys[GLFW\_KEY\_D]) {

myCamera.move(gps::MOVE\_RIGHT, cameraSpeed);

}

if (pressedKeys[GLFW\_KEY\_J]) {

lightAngle += 0.3f;

if (lightAngle > 360.0f)

lightAngle -= 360.0f;

glm::vec3 lightDirTr = glm::vec3(glm::rotate(glm::mat4(1.0f), glm::radians(lightAngle), glm::vec3(0.0f, 1.0f, 0.0f)) \* glm::vec4(lightDir, 1.0f));

myCustomShader.useShaderProgram();

glUniform3fv(lightDirLoc, 1, glm::value\_ptr(lightDirTr));

}

if (pressedKeys[GLFW\_KEY\_L]) {

lightAngle -= 0.3f;

if (lightAngle < 0.0f)

lightAngle += 360.0f;

glm::vec3 lightDirTr = glm::vec3(glm::rotate(glm::mat4(1.0f), glm::radians(lightAngle), glm::vec3(0.0f, 1.0f, 0.0f)) \* glm::vec4(lightDir, 1.0f));

myCustomShader.useShaderProgram();

glUniform3fv(lightDirLoc, 1, glm::value\_ptr(lightDirTr));

}

*}*

*Crearea ferestrei la dimensiune fullscreen și callback-ul la redimensionarea acesteia.*

*bool initOpenGLWindow()*

{

if (!glfwInit()) {

fprintf(stderr, "ERROR: could not start GLFW3\n");

return false;

}

const GLFWvidmode\* mode = glfwGetVideoMode(glfwGetPrimaryMonitor());

//for Mac OS X

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 1);

glfwWindowHint(GLFW\_OPENGL\_FORWARD\_COMPAT, GL\_TRUE);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

glfwWindowHint(GLFW\_SAMPLES, 4);

glWindow = glfwCreateWindow(mode->width, mode->height, "My Title", glfwGetPrimaryMonitor(), NULL);

if (!glWindow) {

fprintf(stderr, "ERROR: could not open window with GLFW3\n");

glfwTerminate();

return false;

}

glfwSetWindowSizeCallback(glWindow, windowResizeCallback);

glfwMakeContextCurrent(glWindow);

// start GLEW extension handler

glewExperimental = GL\_TRUE;

glewInit();

// get version info

const GLubyte\* renderer = glGetString(GL\_RENDERER); // get renderer string

const GLubyte\* version = glGetString(GL\_VERSION); // version as a string

printf("Renderer: %s\n", renderer);

printf("OpenGL version supported %s\n", version);

//for RETINA display

glfwGetFramebufferSize(glWindow, &retina\_width, &retina\_height);

glfwSetKeyCallback(glWindow, keyboardCallback);

glfwSetCursorPosCallback(glWindow, mouseCallback);

return true;

*}*

*void initOpenGLState()*

{

glClearColor(0.3f, 0.3f, 0.3f, 1.0f);

glViewport(0, 0, retina\_width, retina\_height);

glEnable(GL\_FRAMEBUFFER\_SRGB);

glEnable(GL\_DEPTH\_TEST); // enable depth-testing

glDepthFunc(GL\_LESS); // depth-testing interprets a smaller value as "closer"

glEnable(GL\_CULL\_FACE); // cull face

glCullFace(GL\_BACK); // cull back face

glFrontFace(GL\_CCW); // GL\_CCW for counter clock-wise

*}*

*Tot în acest modul are loc și inițializarea framebuffer-ului, iar apoi generarea, configurarea și atașarea texturilor la acesta.*

*void initFBOs()*

{

//generate FBO ID

glGenFramebuffers(1, &shadowMapFBO);

//create depth texture for FBO

glGenTextures(1, &depthMapTexture);

glBindTexture(GL\_TEXTURE\_2D, depthMapTexture);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_DEPTH\_COMPONENT,

SHADOW\_WIDTH, SHADOW\_HEIGHT, 0, GL\_DEPTH\_COMPONENT, GL\_FLOAT, NULL);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

//attach texture to FBO

glBindFramebuffer(GL\_FRAMEBUFFER, shadowMapFBO);

glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, GL\_TEXTURE\_2D, depthMapTexture, 0);

glDrawBuffer(GL\_NONE);

glReadBuffer(GL\_NONE);

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

//skybox texture

glGenTextures(1, &textureID);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, textureID);

*}*

*De asemenea, sunt inițializate și încărcate obiectele(scena,cubul și satelitul).*

*void initModels()*

{

myModel = gps::Model3D("objects/hope/Scene.obj", "objects/hope/");

lightCube = gps::Model3D("objects/cube/cube.obj", "objects/cube/");

satelite = gps::Model3D("objects/Sat/Satellite.obj", "objects/Sat/");

}

void initShaders()

{

myCustomShader.loadShader("shaders/shaderStart.vert", "shaders/shaderStart.frag");

lightShader.loadShader("shaders/lightCube.vert", "shaders/lightCube.frag");

depthMapShader.loadShader("shaders/simpleDepthMap.vert", "shaders/simpleDepthMap.frag");

*}*

*Are loc setarea direcției luminii, precum și a tipului acesteia.*

*void initUniforms()*

{

myCustomShader.useShaderProgram();

modelLoc = glGetUniformLocation(myCustomShader.shaderProgram, "model");

viewLoc = glGetUniformLocation(myCustomShader.shaderProgram, "view");

normalMatrixLoc = glGetUniformLocation(myCustomShader.shaderProgram, "normalMatrix");

lightDirMatrixLoc = glGetUniformLocation(myCustomShader.shaderProgram, "lightDirMatrix");

projection = glm::perspective(glm::radians(45.0f), (float)retina\_width / (float)retina\_height, 0.1f, 1000.0f);

projectionLoc = glGetUniformLocation(myCustomShader.shaderProgram, "projection");

glUniformMatrix4fv(projectionLoc, 1, GL\_FALSE, glm::value\_ptr(projection));

//set the light direction (direction towards the light)

lightDir = glm::vec3(0.0f, 1.0f, 2.0f);

lightDirLoc = glGetUniformLocation(myCustomShader.shaderProgram, "lightDir");

glUniform3fv(lightDirLoc, 1, glm::value\_ptr(lightDir));

//set light color

lightColor = glm::vec3(1.0f, 1.0f, 1.0f); //white light

lightColorLoc = glGetUniformLocation(myCustomShader.shaderProgram, "lightColor");

glUniform3fv(lightColorLoc, 1, glm::value\_ptr(lightColor));

lightShader.useShaderProgram();

glUniformMatrix4fv(glGetUniformLocation(lightShader.shaderProgram, "projection"), 1, GL\_FALSE, glm::value\_ptr(projection));

*}*

*Gestionarea compunerii scenei are loc în metoda „renderScene”, unde prin intermediul metodelor enumerate se încarcă obiectele și se gestionează poziția inițială a acestora.*

*void renderScene()*

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

if(canControl)

processMovement();

//render the scene to the depth buffer (first pass)

depthMapShader.useShaderProgram();

glUniformMatrix4fv(glGetUniformLocation(depthMapShader.shaderProgram, "lightSpaceTrMatrix"),

1,

GL\_FALSE,

glm::value\_ptr(computeLightSpaceTrMatrix()));

glViewport(0, 0, SHADOW\_WIDTH, SHADOW\_HEIGHT);

glBindFramebuffer(GL\_FRAMEBUFFER, shadowMapFBO);

glClear(GL\_DEPTH\_BUFFER\_BIT);

//create model matrix for nanosuit

model = glm::rotate(glm::mat4(1.0f), glm::radians(angle), glm::vec3(0, 1, 0));

//send model matrix to shader

glUniformMatrix4fv(glGetUniformLocation(depthMapShader.shaderProgram, "model"),

1,

GL\_FALSE,

glm::value\_ptr(model));

myModel.Draw(depthMapShader);

//create model matrix for ground

model = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, -1.0f, 0.0f));

//send model matrix to shader

glUniformMatrix4fv(glGetUniformLocation(depthMapShader.shaderProgram, "model"),

1,

GL\_FALSE,

glm::value\_ptr(model));

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

//render the scene (second pass)

myCustomShader.useShaderProgram();

if (toggle\_fog == 1)

{

glUniform1i(glGetUniformLocation(myCustomShader.shaderProgram, "setFog"), 1);

}

else

{

glUniform1i(glGetUniformLocation(myCustomShader.shaderProgram, "setFog"), 0);

}

//send lightSpace matrix to shader

glUniformMatrix4fv(glGetUniformLocation(myCustomShader.shaderProgram, "lightSpaceTrMatrix"),

1,

GL\_FALSE,

glm::value\_ptr(computeLightSpaceTrMatrix()));

//send view matrix to shader

view = myCamera.getViewMatrix();

glUniformMatrix4fv(glGetUniformLocation(myCustomShader.shaderProgram, "view"),

1,

GL\_FALSE,

glm::value\_ptr(view));

//compute light direction transformation matrix

lightDirMatrix = glm::mat3(glm::inverseTranspose(view));

//send lightDir matrix data to shader

glUniformMatrix3fv(lightDirMatrixLoc, 1, GL\_FALSE, glm::value\_ptr(lightDirMatrix));

glViewport(0, 0, retina\_width, retina\_height);

myCustomShader.useShaderProgram();

//bind the depth map

glActiveTexture(GL\_TEXTURE3);

glBindTexture(GL\_TEXTURE\_2D, depthMapTexture);

glUniform1i(glGetUniformLocation(myCustomShader.shaderProgram, "shadowMap"), 3);

//create model matrix for nanosuit

model = glm::rotate(glm::mat4(1.0f), glm::radians(angle), glm::vec3(0, 1, 0));

//send model matrix data to shader

glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(model));

//compute normal matrix

normalMatrix = glm::mat3(glm::inverseTranspose(view \* model));

//send normal matrix data to shader

glUniformMatrix3fv(normalMatrixLoc, 1, GL\_FALSE, glm::value\_ptr(normalMatrix));

myModel.Draw(myCustomShader);

//satelite

//bind the depth map

glActiveTexture(GL\_TEXTURE3);

glBindTexture(GL\_TEXTURE\_2D, depthMapTexture);

glUniform1i(glGetUniformLocation(myCustomShader.shaderProgram, "shadowMap"), 3);

sateliteAngle += 0.5f;

if (sateliteAngle > 360.0f)

sateliteAngle -= 360.0f;

//create model matrix for nanosuit

model = glm::rotate(glm::mat4(1.0f), glm::radians(sateliteAngle), glm::vec3(0.15, 1, 0));

//send model matrix data to shader

glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(model));

//compute normal matrix

normalMatrix = glm::mat3(glm::inverseTranspose(view \* model));

//send normal matrix data to shader

glUniformMatrix3fv(normalMatrixLoc, 1, GL\_FALSE, glm::value\_ptr(normalMatrix));

satelite.Draw(myCustomShader);

//create model matrix for ground

model = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, -1.0f, 0.0f));

//send model matrix data to shader

glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(model));

//create normal matrix

normalMatrix = glm::mat3(glm::inverseTranspose(view \* model));

//send normal matrix data to shader

glUniformMatrix3fv(normalMatrixLoc, 1, GL\_FALSE, glm::value\_ptr(normalMatrix));

//draw a white cube around the light

lightShader.useShaderProgram();

glUniformMatrix4fv(glGetUniformLocation(lightShader.shaderProgram, "view"), 1, GL\_FALSE, glm::value\_ptr(view));

model = glm::rotate(glm::mat4(1.0f), glm::radians(lightAngle), glm::vec3(0.0f, 1.0f, 0.0f));

model = glm::translate(model, 1.0f \* lightDir);

model = glm::scale(model, glm::vec3(0.05f, 0.05f, 0.05f));

glUniformMatrix4fv(glGetUniformLocation(lightShader.shaderProgram, "model"), 1, GL\_FALSE, glm::value\_ptr(model));

lightCube.Draw(lightShader);

lightShader.useShaderProgram();

glUniformMatrix4fv(glGetUniformLocation(lightShader.shaderProgram, "view"), 1, GL\_FALSE, glm::value\_ptr(view));

model = glm::rotate(glm::mat4(1.0f), -glm::radians(sateliteAngle), glm::vec3(0.0f, 1.0f, 0.0f));

model = glm::translate(model, 2.0f \* lightDir);

model = glm::scale(model, glm::vec3(0.05f, 0.05f, 0.05f));

glUniformMatrix4fv(glGetUniformLocation(lightShader.shaderProgram, "model"), 1, GL\_FALSE, glm::value\_ptr(model));

lightCube.Draw(lightShader);

lightShader.useShaderProgram();

glUniformMatrix4fv(glGetUniformLocation(lightShader.shaderProgram, "view"), 1, GL\_FALSE, glm::value\_ptr(view));

model = glm::rotate(glm::mat4(1.0f), glm::radians(sateliteAngle), glm::vec3(1.0f, 0.0f, 0.0f));

model = glm::translate(model, 2.0f \* lightDir );

model = glm::scale(model, glm::vec3(0.05f, 0.05f, 0.05f));

glUniformMatrix4fv(glGetUniformLocation(lightShader.shaderProgram, "model"), 1, GL\_FALSE, glm::value\_ptr(model));

lightCube.Draw(lightShader);

double seconds\_since\_start = difftime(time(0), start);

if (seconds\_since\_start < 6) {

angle += 1.064f;

myCamera.move(gps::MOVE\_FORWARD, 3 \* cameraSpeed);

}

else if (seconds\_since\_start < 10) {

myCamera.move(gps::MOVE\_FORWARD, 1.80f \* cameraSpeed);

}

else {

canControl = true;

}

}

*Metoda „main” gestioneaz*ă *ordinea incărcării, iar mai apoi asigură interacțiunea corectă cu acestea.*

*int main(int argc, const char\* argv[]) {*

initOpenGLWindow();

initOpenGLState();

initFBOs();

std::vector<const GLchar\*> faces;

faces.push\_back("skybox/4/rt.tga");

faces.push\_back("skybox/4/lf.tga");

faces.push\_back("skybox/4/up.tga");

faces.push\_back("skybox/4/dn.tga");

faces.push\_back("skybox/4/bk.tga");

faces.push\_back("skybox/4/ft.tga");

mySkyBox.Load(faces);

skyboxShader.loadShader("shaders/skyboxShader.vert", "shaders/skyboxShader.frag");

skyboxShader.useShaderProgram();

view = myCamera.getViewMatrix();

glUniformMatrix4fv(glGetUniformLocation(skyboxShader.shaderProgram, "view"), 1, GL\_FALSE, glm::value\_ptr(view));

projection = glm::perspective(glm::radians(45.0f), (float)retina\_width / (float)retina\_height, 0.1f, 1000.0f);

glUniformMatrix4fv(glGetUniformLocation(skyboxShader.shaderProgram, "projection"), 1, GL\_FALSE, glm::value\_ptr(projection));

initModels();

initShaders();

initUniforms();

glCheckError();

glfwSetInputMode(glWindow, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED);

start = time(0);

while (!glfwWindowShouldClose(glWindow)) {

renderScene();

mySkyBox.Draw(skyboxShader, view, projection);

glfwPollEvents();

glfwSwapBuffers(glWindow);

}

//close GL context and any other GLFW resources

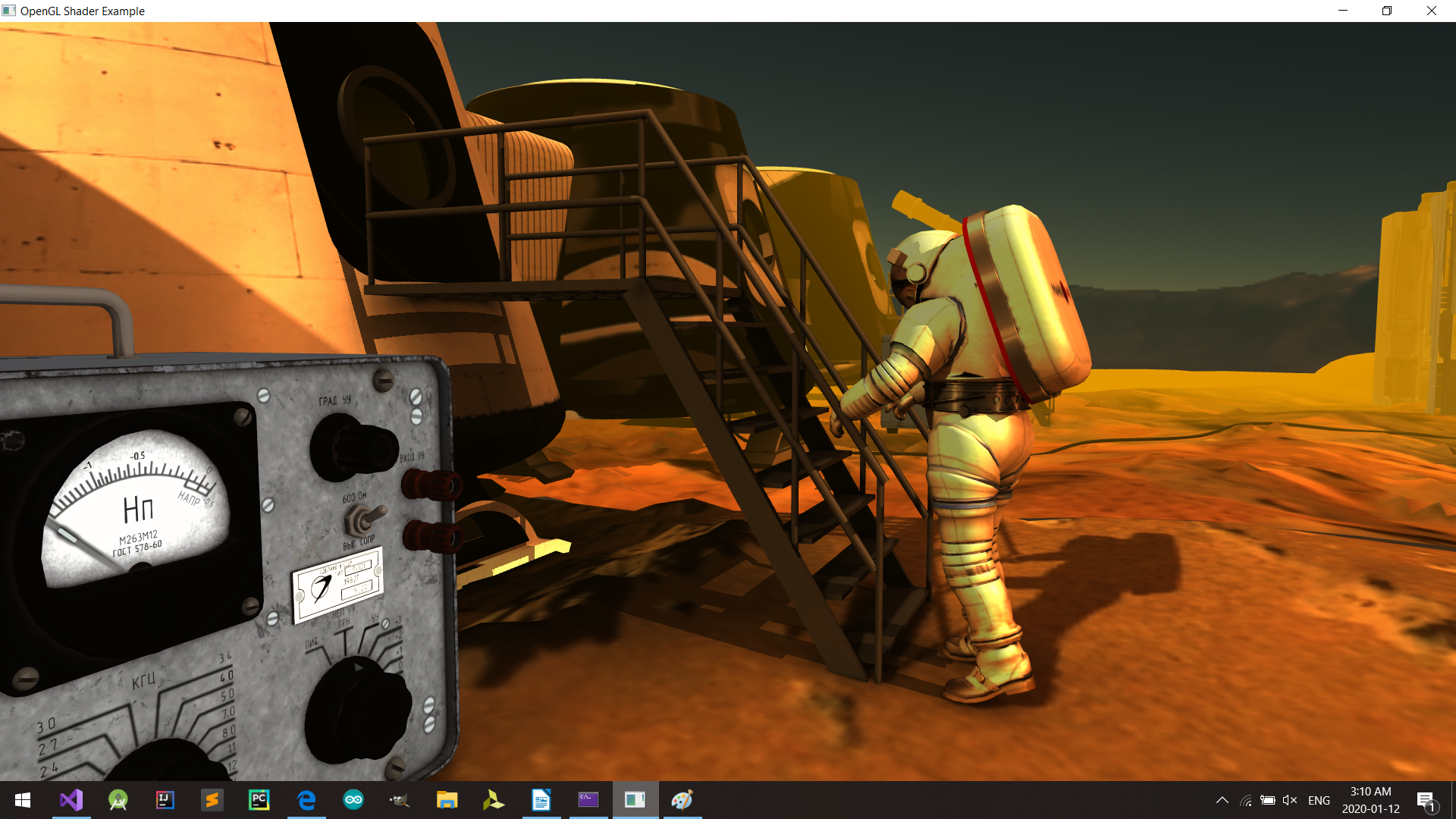
glfwTerminate();

return 0;

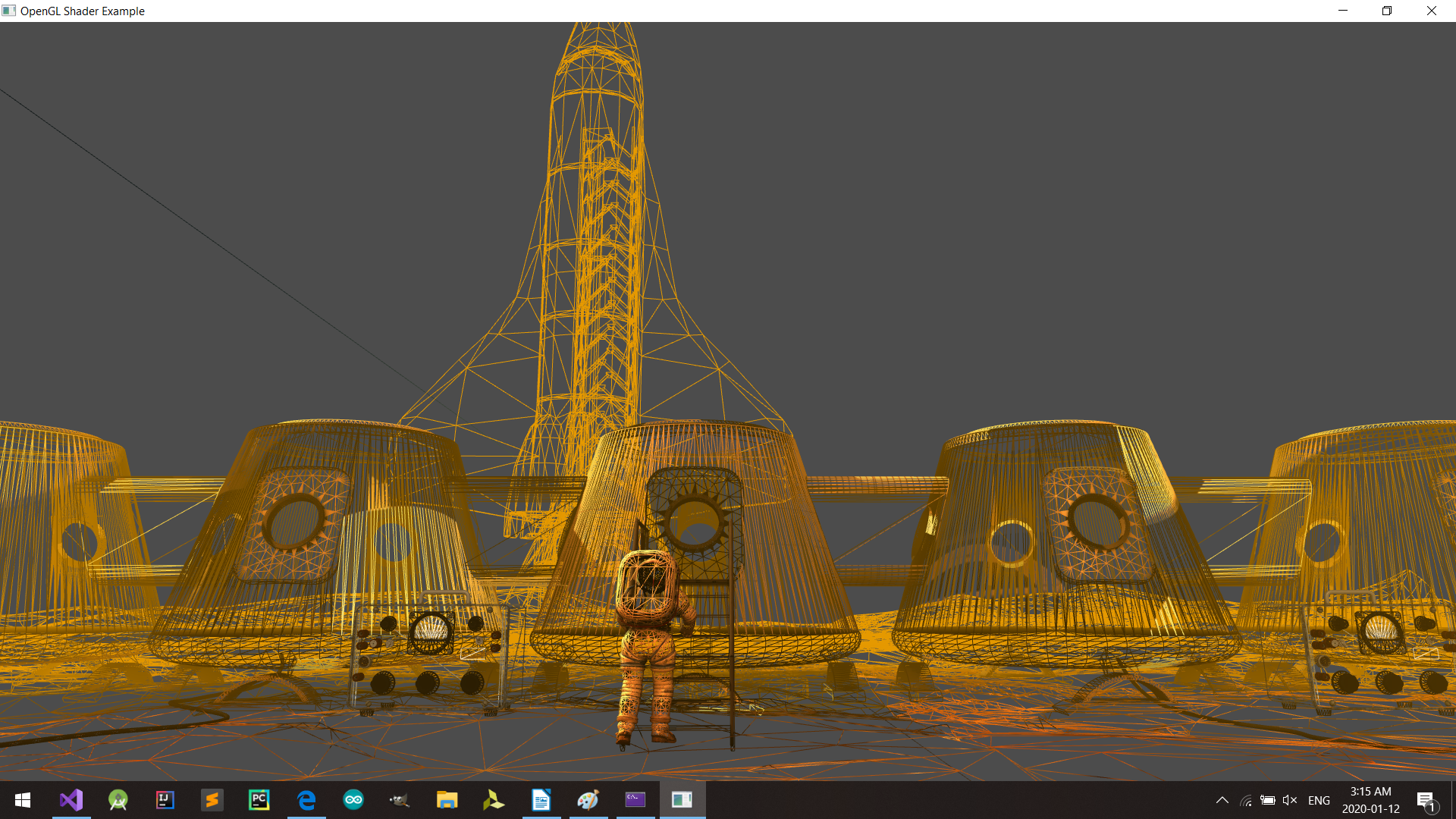
*}*

***5. Prezentarea interfeței grafice utilizator / manual de utilizare***

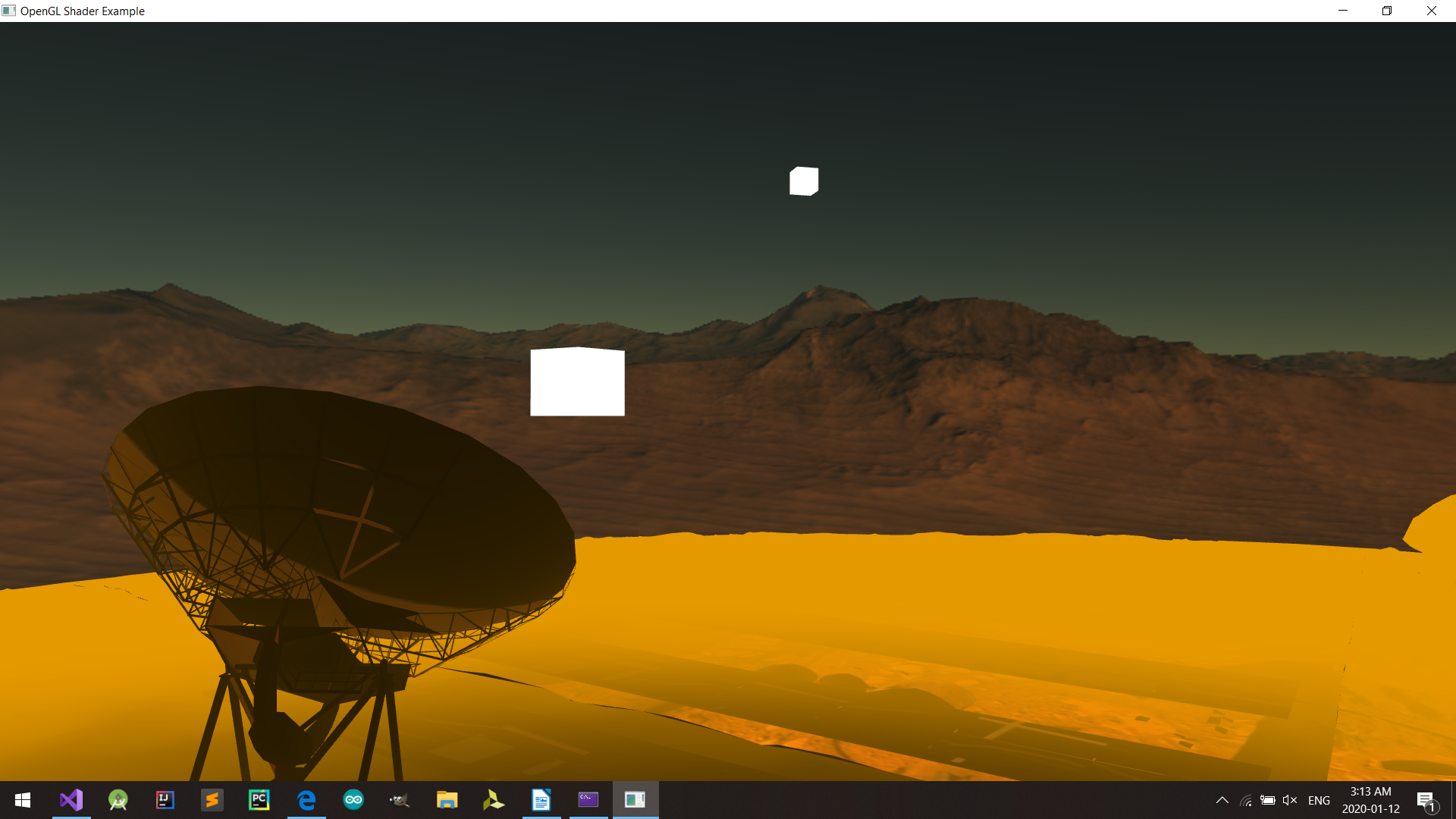
*Deplasarea in scena se face ajutorul mouse-ului si a tastaturii (W,A,S,D) . Un exemplu:*

**

*Modul wireframe, line, egde :*

**

*Exemple de lumini ( totodata se poate observa si ceața : variază invers cu pătratul distanței ) :*

**

***6. Concluzii și dezvoltări ulterioare***

*Proiectul este sursa perfectă pentru ințelegerea pipeline-ului grafic, a mecanismului de randare, a conceptelor des intalnite in grafică cum sunt shaders, lumini, camera, matrice model, proiecție, vizualizare, animații, texturare, umbre.*

*Pentru dezolvoltare propun : utilizarea unui sistem de particule pentru reactoarele rachetei si o animație cu lansarea acesteia, atasarea unui first-person character pentru cameră, crearea unui vizor pentru o casă de astronaut (cu ajutorul barrel distortion ) pentru efectul prezentării scenei din interiorul unei căsti de astronaut, imbunătățirea texturilor , importarea mai multor obiecte si crearea mai multor animații.*

***7. Referințe***

* ***Lumini :*** <https://learnopengl.com/Lighting/Multiple-lights>
* ***Camera :*** <https://learnopengl.com/Getting-started/Camera>
* ***Laborator :*** <https://moodle.cs.utcluj.ro/>