Laborator 6

1 Obiective

Acest laborator vă prezintă noțiunile de bază privind conceptul de textură și utilizarea sa într-o scenă 3D a obiectelor. Acesta vă va oferi, de asemenea, un cadru de bază pentru încărcarea și includerea obiectelor 3D în proiectul OpenGL, prezentând apoi câteva notiuni despre animația obiectelor.

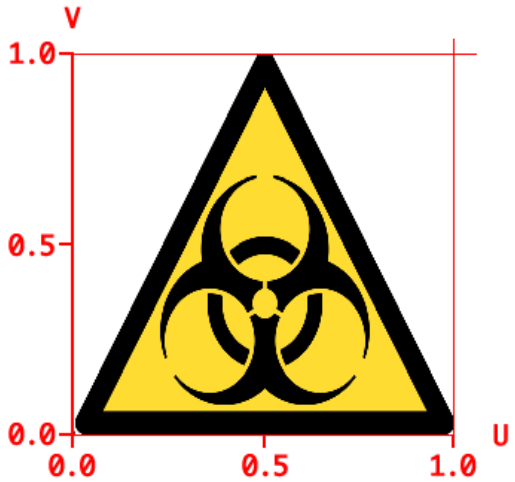
2 Fundament teoretic

Puteți să vă gândiți la o textură ca la o imagine 2D care poate fi aplicată pe un obiect 3D. Deși este posibil să existe texturi 1D și 3D, acestea sunt dincolo de sfera de interes a acestui laborator.

În general, texturile sunt aplicate pe obiecte 3D pentru a crește fotorealismul, adăugând detalii pe suprafețele lor. Spre exemplu, este posibil să se adauge detalii de culoare unui model prin aplicarea unor culori asociate fiecărui vârf (conform exemplului din laboratorul precedent), dar acest lucru ar fi foarte risipitor, atât în ceea ce privește volumul de date, cât și puterea de procesare necesară.

Texturile sunt în principiu hărți de pixeli 2D încărcate în memoria video, care pot fi apoi asociate unui obiect 3D dat. Asocierea se face prin atribuirea fiecărui vârf 3D unui punct din cadrul imaginii 2D. Spațiul de textura 2D este de obicei exprimat folosind numele parametrilor "u" și "v", spre deosebire de "x" și "y", pentru a evita confuzia cu valorile poziției vârfului.

Spre deosebire de imaginile normale ale căror coordonate sunt numere întregi care exprimă valoarea unui pixel dat, coordonatele UV ale unei texturi sunt valori în virgulă variabilă în intervalul [0, 1], unde punctul (0, 0) reprezintă colțul din stânga jos al imaginii, iar (1, 1) reprezintă colțul din dreapta sus. Aceste coordonate pot fi transformate în pixeli prin înmulțirea valorilor lor cu lățimea și înălțimea imaginii.



Pentru a utiliza o textură, trebuie să o încărcați mai întâi dintr-un fișier extern:

- Încărcați datele de imagine dintr-un fișier imagine (de preferință ".png" sau ".tex" - întotdeauna cu lățime și înălțime egale cu o putere de 2 - ex: 128, 256, 512 etc.)
- Generați un ID de textura - folosind funcția: "glGenTextures"
- Legați sau "activați" textura - folosind funcția "glBindTexture"
- Generați o textură din datele imaginii - utilizând funcția "glTexImage2D"
- Configurați parametrii de textură

Funcția de încărcare a texturii este implementată în cadrul "ReadTextureFromFile". Examinați această funcție. Pentru a aplica textura pe un obiect, este suficient să o legați (bind) înainte de a desena obiectul - folosind funcția "glBindTexture".

3 Tutorial

3.1 Aplicarea texturii

În scopul demonstrării utilizării texturilor, vom încerca mai întâi un exemplu simplu.

Să începem prin desenarea a două triunghiuri. Definiți datele de vârf - poziția și coordonatele texturii:

```
//vertex position and UV coordinates
GLfloat vertexData[] = {
        //  primul triunghi
        -5.0f, 0.0f, 0.0f,    0.0f, 0.0f,
        5.0f,0.0f, 0.0f,    1.0f, 0.0f,
        0.0f, 8.0f, 0.0f,    0.5f, 1.0f,
        //  al doilea triunghi
        0.1f, 8.0f, 0.0f,    0.0f, 0.0f,
        5.1f, 0.0f, 0.0f,    0.0f, 0.0f,
        10.1f, 8.0f, 0.0f,    0.0f, 0.0f
};
GLuint vertexIndices[] = {
        0,1,2,
        3,4,5
};
```

Încărcați datele triunghiului apelând funcția predefinită "loadTriangles ()" - înainte de începerea buclei de rasterizare:

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

        . . . . . .

        loadTriangleData();

        while (!glfwWindowShouldClose(glWindow)) {
                renderScene();
```

```
            glfwPollEvents();
            glfwSwapBuffers(glWindow);
        }
        ..........
}
```

În cele din urmă, desenați triunghiurile:

```
void renderScene()
{
        ..............

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

Construiți și executați aplicația. Ar trebui să vedeți două triunghiuri negre.

Pentru a aplica textura, trebuie să trimitem coordonatele texturii către Fragment Shader. Acest lucru înseamnă că ele trebuie să treacă mai întâi prin Vertex Shader:

```
#version 410 core

layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexNormal;
layout(location = 2) in vec2 textcoord;

out vec3 colour;
out vec2 passTexture;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
   colour = vertexNormal;
   passTexture = textcoord;
   gl_Position = projection * view * model * vec4(vertexPosition, 1.0);
}
```

În interiorul Fragment Shader, avem nevoie de acces la o unitate de textură, pe care o să o denumim "diffuseTexture". Funcția "texture" va returna o valoare din textura activă curentă, pe baza coordonatelor 2D UV pe care le primește ca argumente.

Fragment Shader ar trebui să arate așa:

```
#version 410 core

in vec3 colour;
```

```
in vec2 passTexture;

out vec4 fragmentColour;

uniform sampler2D diffuseTexture;

void main() {
  fragmentColour = texture(diffuseTexture, passTexture);
}
```

Asigurați-vă că încărcați textura din fișierul imagine:

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

        loadTriangleData();
        texture = ReadTextureFromFile("textures/hazard2.png");

        while (!glfwWindowShouldClose(glWindow)) {
                renderScene();

                glfwPollEvents();
                glfwSwapBuffers(glWindow);
        }
        . . . . . .
}
```

Acum activați textura înainte de a desena triunghiurile:

```
void renderScene()
{
        . . . . . .

        glActiveTexture(GL_TEXTURE0);
        glUniform1i(glGetUniformLocation(myCustomShader.shaderProgram, "diffuseTexture"), 0);
        glBindTexture(GL_TEXTURE_2D, texture);

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

Construiți și executați aplicația.

Modificați coordonatele UV ale texturii pentru al doilea triunghi, astfel încât aplicația dvs. să arate astfel:

3.2 Încărcarea obiectelor 3D

În scopul încărcării și desenării obiectelor 3D complexe, o opțiune pe care o aveți este să utilizați clasa “Model3D” furnizată în pachetul de resurse al acestui laborator. Această clasă ar trebui să poată încărca orice fel de obiect, de la obiecte simple la scene complexe, cu multiple texturi aplicate. După cum puteți vedea în interiorul "Model3D.hpp", acesta conține o listă de obiecte de tip „mesh” (fiecare reprezentând câte un obiect 3D) precum și colecția de texturi încărcate - necesare pentru acele obiecte.

Să vedem cum putem folosi clasa “Model3D” pentru a încărca un model dintr-un fișier ".obj". Modificați funcția principală. Ștergeți funcționalitatea adăugată anterior și adăugați:

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

	. . . . . .
	//send matrix data to shader
	GLint projLoc = glGetUniformLocation(myCustomShader.shaderProgram, "projection");
	glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));

	myModel.LoadModel("objects/stall.obj");

	while (!glfwWindowShouldClose(glWindow)) {
		renderScene();

		glfwPollEvents();
		glfwSwapBuffers(glWindow);
	}
	. . . . . .
}
```

Pentru a rasteriza modelul încărcat, modificați și adăugați următoarele linii în cadrul funcției "renderScene":

```
void renderScene()
{
	. . . . . .
	//create rotation matrix
	model = glm::rotate(model, glm::radians(angle), glm::vec3(0.0f, 1.0f, 0.0f));
	model = glm::rotate(model, glm::radians(180.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

```
        //send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
        myModel.Draw(myCustomShader);
}
```

Construiți și executați aplicația. După cum puteți vedea, **modelul nu este texturat**. Pentru a aplica o textură, trebuie să încărcați o textură dintr-un fișier extern:

```
int main(int argc, const char * argv[]) {
        ..................
        myModel.LoadModel("objects/stall.obj");
        texture = ReadTextureFromFile("textures/stall_texture.png");
        ..................  }
```

Activați textura înainte de a desena modelul:

```
void renderScene()
{
        ..................
        glActiveTexture(GL_TEXTURE0);
        glUniform1i(glGetUniformLocation(myCustomShader.shaderProgram, "diffuseTexture"), 0);
        glBindTexture(GL_TEXTURE_2D, texture);

        myModel.Draw(myCustomShader);
}
```

Rulați aplicația. Utilizați tastele "W", "A", "S" și "D" pentru navigare; "Q" și "E" pentru a roti modelul.

Amintiți-vă când am spus că un obiect Model3D poate reține atât obiectele „mesh” 3D cât și texturile lor. Cel mai adesea, atunci când descărcați obiecte 3D din surse online, o să primiți un fișier ".obj" și un ".mtl", împreună cu o colecție de imagini cu textură deja aplicată. Fișierul .mtl specifică datele de asociere ale texturii pentru obiectul dat. **În aceste cazuri, obiectul Model3D va citi obiectul 3D și va activa texturile asociate după cum este necesar - NU VA MAI FI NEVOIE CA DVS. SĂ ÎNCĂRCAȚI ȘI SĂ ACTIVAȚI TEXTURILE.**

Puteți găsi fișiere ".obj" la www.turbosquid.com, www.tf3dm.com și https://sketchfab.com/features/free-3d-models .

Să încercăm să adăugăm un fișier ".obj" cu texturile incluse. Modificați funcția principală pentru a încărca noul obiect:

```
int main(int argc, const char * argv[]) {
        ..................
        //argument 1 = obj file; argument 2 = textures folder
        myModel.LoadModel("objects/Farmhouse.obj", "textures/");
        //texture = ReadTextureFromFile("textures/stall_texture.png");

        while (!glfwWindowShouldClose(glWindow)) {
                renderScene();
```

```
            glfwPollEvents();
            glfwSwapBuffers(glWindow);
        }
        ..................
}
```

Comentați instrucțiunile de activare a texturii din renderScene. Acest obiect conține informații de mapare a texturii, prin urmare obiectul Model3D va încerca să încarce automat texturile:

```
void renderScene()
{
        ............
        //glActiveTexture(GL_TEXTURE0);
        //glUniform1i(glGetUniformLocation(myCustomShader.shaderProgram, "diffuseTexture"),0);
        //glBindTexture(GL_TEXTURE_2D, texture);

        myModel.Draw(myCustomShader);
}
```

Regulile generale de urmat la încărcarea fișierelor .obj de pe internet:

- Căutați fișierul .mtl - conține date despre materialele și texturile obiectului. În interiorul .obj ar trebui să aveți o referință la fișierul .mtl.
- În cadrul fișierului .mtl, **căutați referințele fișierelor de textură** => asigurați-vă ca acestea nu conțin si nume de folder, doar numele fișierului
- Asigurați-vă că texturile sunt în format ".png" sau ".tga" și au dimensiuni (lățime și înălțime) egale cu puterile de 2 (ex: 128, 256, 512 etc.).
- Încercați să păstrați proiectul curat și ordonat - păstrați obiectele 3D și texturile (împreună cu fișierul .mtl) în propriile lor foldere, separate.

3.3 Animația

Să adăugăm o transformare de translație obiectului nostru - translatăm pe axa Ox cu o cantitate egală cu "delta". Creșteți delta în interiorul funcției "renderScene".

```
float delta = 0;
void renderScene()
{
        ..............
        delta += 0.001;
        model = glm::translate(model, glm::vec3(delta, 0, 0));

        //create rotation matrix
        model = glm::rotate(model, glm::radians(angle), glm::vec3(0.0f, 1.0f, 0.0f));
        model = glm::rotate(model, glm::radians(180.0f), glm::vec3(0.0f, 1.0f, 0.0f));
        //send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

        glBindTexture(GL_TEXTURE_2D, texture);
```

```
        myModel.Draw(myCustomShader);
}
```

Rulați aplicația. Încercați apoi să desenați 500 de instanțe ale aceluiași obiect. Observați scăderea vitezei de animație:

```
void renderScene()
{
        ..............

        for (int i = 0; i < 500;i++)
                myModel.Draw(myCustomShader);
}
```

Această animație depinde de frecvența cu care se invocă funcția - adică de frecvența de refresh a aplicației. Așa cum ați putea să ghiciți, frecvența de refresh variază în funcție de resursele hardware sau de complexitatea scenelor. În mod ideal, am dori ca animația să funcționeze independent de capacitățile de refresh ale aplicației noastre.

Soluția este de a crește delta pe baza unei variabile de viteză date - "unități pe secundă". Măsurăm timpul dintre actualizările succesive și calculăm distanța pe care ar trebui să o deplaseze obiectul:

*Distance = speed * time*

```
float delta = 0;
float movementSpeed = 2; // units per second
void updateDelta(double elapsedSeconds) {
        delta = delta + movementSpeed * elapsedSeconds;
}
double lastTimeStamp = glfwGetTime();

void renderScene()
{
        ..............
        //initialize the view matrix
        glm::mat4 view = myCamera.getViewMatrix();
        //send matrix data to shader
        GLint viewLoc = glGetUniformLocation(myCustomShader.shaderProgram, "view");
        glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));

        // get current time
        double currentTimeStamp = glfwGetTime();
        updateDelta(currentTimeStamp - lastTimeStamp);
        lastTimeStamp = currentTimeStamp;
        model = glm::translate(model, glm::vec3(delta, 0, 0));
        ..............
}
```

Puteți încerca animația cu un număr tot mai mare de instanțe de obiecte, pentru a vedea dacă viteza de mișcare se modifică sau nu.

4 Documentație

Funcție (și legătură)	Descriere
glGenTextures	Generează nume de texturi
glBindTexture	Legați o textură numită într-o textură țintă
glTexImage2D	Specificați o imagine textura bidimensională
glTexParameter	Setați parametrii de textură

5 Temă

- Încărcați cel puțin 3 modele. Schimbați poziția unui obiect folosind tastele și animați un alt obiect.