Laborator 5

1 Obiective

Acest laborator prezintă noțiunile cheie privind transformările 3D. Acesta va acoperi modelul de bază (traducerea, scala, rotația), transformările camerei și proiecției.

2 Definirea punctelor 2D și 3D

Un punct 2D este definit într-un sistem de coordonate omogen prin (x*w, y*w, w). Pentru simplitate, în sistemele bidimensionale, parametrul w este setat la 1. Prin urmare, definiția punctului este (x, y, 1).

O altă reprezentare pentru acest punct este următoarea: $P = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$.

În mod similar, în 3D definim punctele ca (x*w, y*w, z*w, w) și reprezentăm punctul ca vector coloană:

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3 Transformări

După ce au fost procesate de Vertex Shader, se preconizează că nodurile care urmează a fi redate vor avea coordonatele lor încadrate în intervalul [-1, +1] - denumit și Normalized Device Coordinates (NDC). Pentru a obține vârfurile în această etapă, ele trebuie să sufere o serie de transformări, ajungând din Spațiul Obiectului Local (Local Object Space) în Spațiul de Global de Coordonate al Scenei (World Coordinate System), apoi în pe Spațiul Coordonatelor de Vizualizare (View Coordinates Space) și, în final, în Spațiul de Clipping (Clipping Space).

3.1 Transformări de model

Atunci când definiți un obiect, vârf cu vârf, faceți acest lucru relativ la un sistem de coordonate locale. De cele mai multe ori, pentru comoditate, definiți obiectele centrate în originea acestui sistem - aceasta este o practică standard pentru majoritatea instrumentelor de modelare a obiectelor. Când plasați obiectul într-o anumită scenă (lume), trebuie să îl poziționați în locul potrivit și poate să îl re-dimensionați pentru a-l încadra în interiorul scenei.

Transformările (translație, rotație, scalare) necesare pentru poziționarea modelului într-o scenă se mai numesc și transformări de model. Ele transformă vârfurile obiectului din spațiul local în spațiul global.

3.1.1 Translație

Transformarea de translație este folosită pentru a muta un obiect (punct) cu un anumit deplasament.

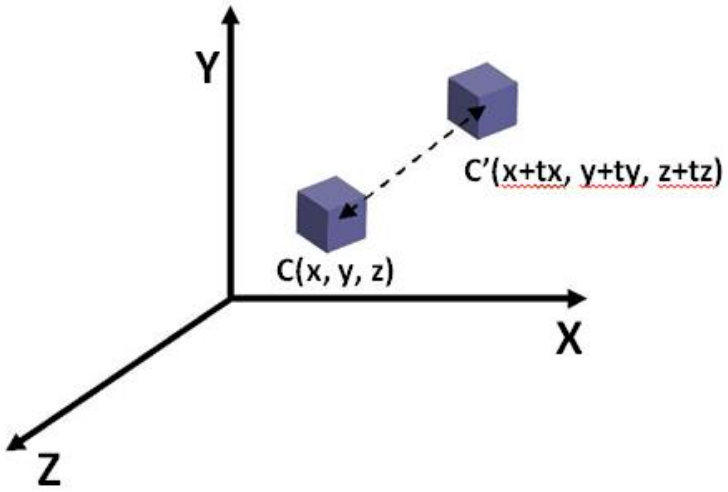
Matricea de translație este:

$$T = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matricea de translație inversă este:

$$T' = \begin{bmatrix} 1 & 0 & 0 & -Tx \\ 0 & 1 & 0 & -Ty \\ 0 & 0 & 1 & -Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Tx, Ty și Tz reprezintă factorii de translație pe axele x, y și z.



Puteți genera o matrice de translație utilizând:

```
glm::translate(glm::mat4 init_matrix, glm::vec3(float Tx, float Ty, float Tz))
```

3.1.2 Scalare

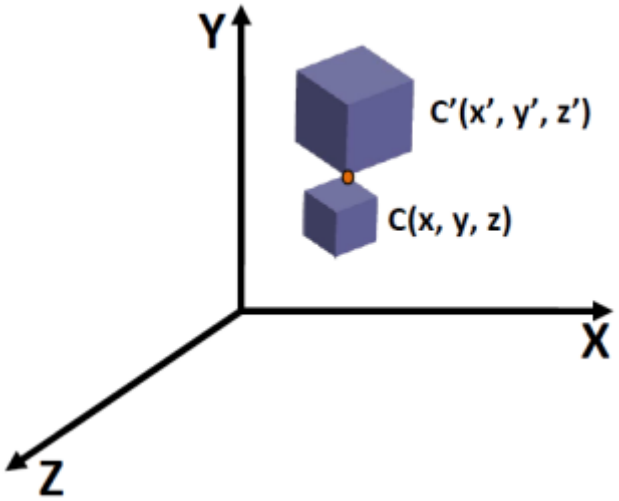
Transformarea de scalare mărește sau reduce dimensiunile unui obiect. Transformarea este *relativă la origine*.

Matricea de scalare este:

$$S = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matricea inversă de scalare este:

$$S' = \begin{bmatrix} 1/Sx & 0 & 0 & 0 \\ 0 & 1/Sy & 0 & 0 \\ 0 & 0 & 1/Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Sx, Sy and Sz reprezintă factorii de scalare pe axele x, y și z. Dacă Sx, Sy și Sz sunt egali, transformarea de scalare este uniformă. Dacă Sx, Sy și Sz nu sunt egali, transformarea de scalare este neuniformă

Dacă setați factorii de scalare la +/- 1, atunci puteți reflecta forma originală.

Puteți genera o matrice de scalare folosind:

```
glm::scale(glm::mat4 init_matrix, glm::vec3(float Sx, float Sy, float Sz))
```

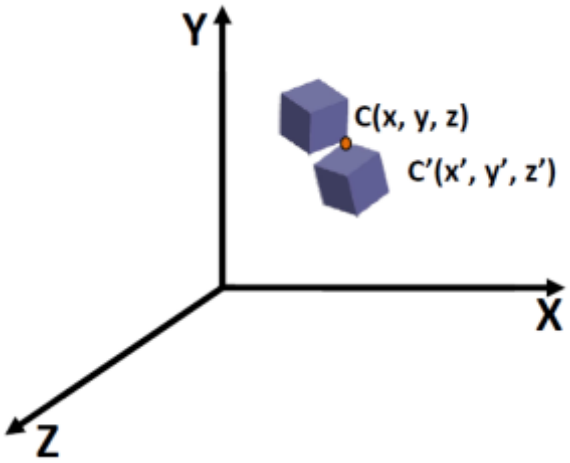
3.1.3 Rotație

Această transformare rotește un obiect cu un unghi dat. Această transformare este, de asemenea, relativă la origine.

Pentru un punct 2D, matricile de rotație și rotație inversă sunt următoarele:

$$R = \begin{bmatrix} \cos\propto & -\sin\propto & 0 \\ \sin\propto & \cos\propto & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

În spațiul 3D, specificăm rotația în mod independent pe axele x, y și z. Rotația în jurul axei z este similară cu rotația din 2D (coordonatele z rămân neschimbate).



$$R_z = \begin{bmatrix} \cos\propto & -\sin\propto & 0 & 0 \\ \sin\propto & \cos\propto & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\propto & -\sin\propto & 0 \\ 0 & \sin\propto & \cos\propto & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} \cos\propto & 0 & \sin\propto & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\propto & 0 & \cos\propto & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Pentru a schimba centrul de rotație - adică pentru a face o rotație în jurul unui punct dat în locul originii - trebuie să facem o serie de trei transformări:

- Translatați noul punct central în origine
- Realizați rotația
- Translația inversă a punctului

Din punct de vedere matematic, aceasta ar însemna: $\mathbf{P_{new} = T_{inverse} * R * T * P}$

Puteți genera o matrice de rotație folosind:

```
glm::rotate(glm::mat4 init_matrix, float angle, glm::vec3(float x, float y, float z))
```

- x, y și z reprezintă vectorul 3D (axa) în jurul căruia este efectuată rotația
- unghiul de rotație este specificat în grade, nu în radiani.

3.2 Transformări ale camerei

După ce scena a fost configurată, trebuie să definiți poziția și direcția de vizionare a observatorului (camera). Acest lucru este făcut pentru a trasa grafic scena din perspectiva observatorului. Dacă până acum coordonatele vârfurilor erau relative la originea scenei, acum ele vor fi exprimate în funcție de

poziția camerei. Trebuie să efectuăm o serie de rotații și translații pentru a aduce originea scenei în poziția camerei. Aceasta se numește transformarea din Spațiul Global în Spațiul Coordonatelor de Vizualizare.

Seria de transformări necesare pentru a converti scena în Spațiul de Vizualizare este cuprinsă într-o matrice, numită matricea cameră sau matricea de vizualizare. Se construiește ținând cont de poziția camerei, de orientarea acesteia și de vectorul "UP" specificat de utilizator.

Matricea de vizualizare poate fi construită utilizând următoarea funcție GLM:

```
glm::lookAt(glm::vec3 camPosition, glm::vec3 viewPoint, glm::vec3 UP)
```

unde:

- camPosition – un vec3 specificând coordonatele 3D ale poziției camerei;
- viewPoint – un vec3 specificând coordonatele 3D ale punctului însprecare este îndreptată camera. Rețineți că vectorul definit ca viewPoint - camPosition specifică direcția de vizualizare;
- UP – vectorul “UP” – specifică direcția axei Y și este de obicei egală cu (0,1,0);

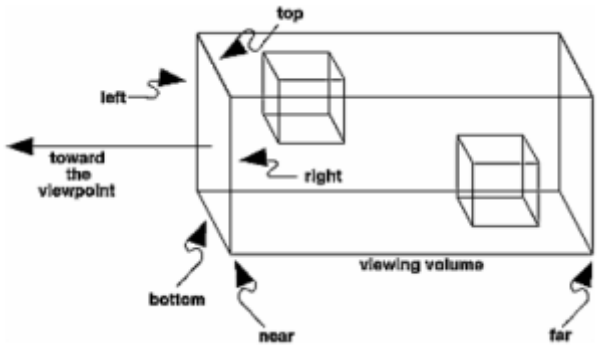
3.3 Transformări de proiecţie

Aceste transformări sunt necesare pentru a crea o reprezentare 2D dintr-o porțiune a scenei 3D de obiecte definite în cadrul proiectului dvs. "Porțiunea" scenei care contribuie la reprezentarea 2D obținută este delimitată de un volum de vizualizare. Aceasta, la rândul său, depinde de parametrii cum ar fi poziția spectatorului, câmpul vizual și direcția vectorului de vizualizare. În termeni simpli, vă puteți gândi la o proiecție ca la umbra pe care un obiect o aruncă pe un plan sau o suprafață 2D.

Deși există un număr de tipuri diferite de proiecții, cele mai frecvent folosite sunt proiecțiile ortografice și de perspectivă.

3.3.1 Proiecţie ortografică

O transformare de proiecție ortografică definește volumul de vizionare ca formă cubică, folosind patru parametri: lățimea, înălțimea, un plan apropiat și un plan îndepărtat. Fiecare vârf 3D va fi reprezentat ca o poziție determinată de coordonatele X și Y ale acestuia.



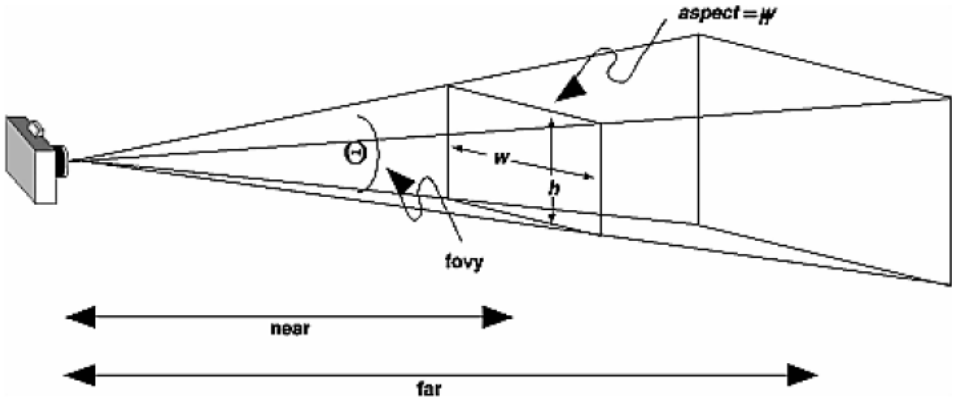
Puteți crea o matrice de proiecție ortografică utilizând funcția `glm::ortho`, văzută mai jos:

```
glm::ortho(GLint left, GLint right, GLint bottom, GLint top, GLfloat near, GLfloat far)
```

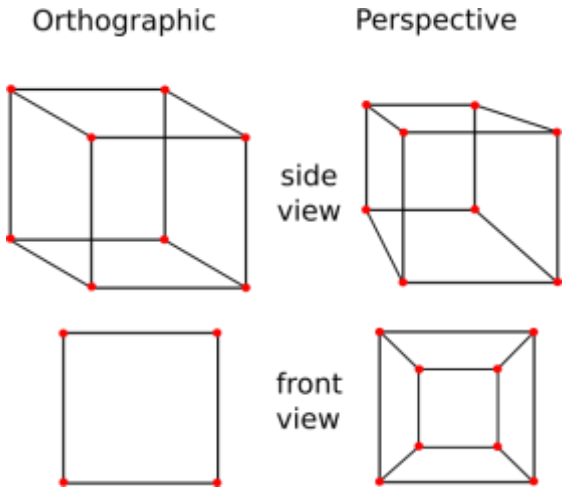
Fiecare parametru al acestei funcții este o variabilă ce definește limitele X, Y și Z ale volumului de vizualizare.

3.3.2 Proiecție perspectivă

Transformarea de perspectivă definește volumul de vizionare sub formă de frustum, pornind de la poziția camerei și extinzându-se spre exterior, de-a lungul axei Z.



În acest caz, toate cele trei coordonate ale vârfului afectează coordonatele 2D ale proiecției sale. Rezultatul net este că, pe măsură ce un obiect este mai departe de vizualizator, cu atât mai mic va apărea. Acest lucru îi va da privitorului o senzație de distanță în ceea ce privește obiectele din scenă.



Puteți crea o matrice de transformare perspectivă utilizând următoarea funcție GLM:

```
glm::perspective(GLfloat fov, GLfloat aspect, GLfloat near, GLfloat far);
```

unde:

- fov – unghiul care definește cât de mare este frustumul de vizionare (de obicei 45 de grade)
- aspect – definește raportul de aspect și este egal cu lățimea ferestrei de vizualizare împărțită la înălțimea acesteia
- near & far – coordonatele Z ale planurilor apropiate și îndepărtate, care sunt capetele frustumului vizual

4 Tutorial

Trebuie să creăm matricele de transformare pentru fiecare din cele trei etape:

- Transformări de model
- Transformări de vizualizare/cameră
- Transformări de proiecție

După definirea matricelor, le vom folosi pentru a transforma fiecare vârf al scenei. Pentru a face acest lucru, vom aplica cele trei transformări în interiorul Vertex Shader-ului. După aplicarea transformărilor, vârful rezultat va arăta astfel:

$$P_{final} = M_{projection} * M_{view} * M_{model} * P_{initial}$$

Rețineți că ordinea în care transformările sunt aplicate la vârf este contrariul ordinii în care apar în formula de mai sus. Pentru a obține ordinea corectă în care sunt aplicate la vârf, trebuie să citiți transformările de la dreapta la stânga.

4.1 Transformări de model

Începeți prin descărcarea aplicației de pe site-ul web al laboratorului. Construiți și executați aplicația.

Ar trebui să vezi un pătrat purpuriu. Acesta este de fapt un cub multi-colorat, vazut din fata. Pentru a-l percepe ca pe un cub, vom adăuga mai întâi o transformare de rotație.

Așa cum am menționat mai devreme, transformările de modelare ar trebui să fie aplicate pe vârfuri în cadrul **Vertex Shader-ului**. În acest scop, îi vom trimite matricea de transformare a modelului ca variabilă uniformă:

```
#version 400


layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColour;


out vec3 colour;
```

```
uniform mat4 model;

void main() {
    colour = vertexColour;
    gl_Position = model * vec4(vertexPosition, 1.0);
}
```

Trebuie să trimitem matricea modelului din programul principal către vertex shader. Declarați matricea model și locația sa ca variabile globale:

```
GLuint verticesVBO;
GLuint verticesEBO;
GLuint objectVAO;

gps::Shader myCustomShader;
glm::mat4 model;
GLint  modelLoc;
```

Creați și trimiteți matricea model:

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

                . . . . . .

        model = glm::mat4(1.0f);
        modelLoc = glGetUniformLocation(myCustomShader.shaderProgram, "model");
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

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

                glfwPollEvents();
                glfwSwapBuffers(glWindow);
```

```
        }
        //close GL context and any other GLFW resources
        glfwTerminate();

        return 0;
}
```

Să adăugăm o transformare de rotație în interiorul "renderScene ()". Aceasta se va activa la apăsarea tastei "R":

```
float angle=0;
void renderScene()
{
        . . . . . .

        model = glm::mat4(1.0f);

        if (glfwGetKey(glWindow, GLFW_KEY_R)) {
            angle += 0.0002f;
        }

        // create rotation matrix
        model = glm::rotate(model, angle, glm::vec3(0, 1, 0));
        // send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

        myCustomShader.useShaderProgram();
        glBindVertexArray(objectVAO);
        glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
}
```

Încercați rotația apăsând tasta "R".

4.2 Transformări de cameră și proiecție

Următorul pas este să aplicăm transformările de vizualizare și de proiecție pe scena noastră de obiecte. Aceasta va permite o reprezentare mai realistă și mai controlabilă a scenei.

Începeți prin adăugarea a încă două matrici în interiorul vertex shader:

```
#version 400

layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColour;

out vec3 colour;

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

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

Acum trebuie să definim, să inițializăm și să trimitem cele două matrice la vertex shader. Adăugați următoarele:

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

        . . . .

        model = glm::mat4(1.0f);
        modelLoc = glGetUniformLocation(myCustomShader.shaderProgram, "model");
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

        // initialize the view matrix
```

```
        glm::mat4 view = glm::lookAt(glm::vec3(0.0f, 0.0f, 5.0f), glm::vec3(0.0f, 0.0f, -10.0f), glm::vec3(0.0f, 1.0f,
0.0f));
        // send matrix data to shader
        GLint viewLoc = glGetUniformLocation(myCustomShader.shaderProgram, "view");
        glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));

    // initialize the projection matrix
        glm::mat4 projection = glm::perspective(70.0f, (float)glWindowWidth / (float)glWindowHeight, 0.1f, 1000.0f);
        // send matrix data to shader
        GLint projLoc = glGetUniformLocation(myCustomShader.shaderProgram, "projection");
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));

        . . . .
}
```

4.3 Adăugarea transformărilor

Desenați un al doilea cub și plasați-l în partea stângă a primului. În mod implicit, un obiect va fi întotdeauna centrat la origine. Pentru a evita acest lucru, efectuăm o translație înainte de a desena cel de-al doilea cub:

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

        // create rotation matrix
        model = glm::rotate(model, angle, glm::vec3(0, 1, 0));
        // send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

        myCustomShader.useShaderProgram();
        glBindVertexArray(objectVAO);
        glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
```

```
        // create a translation matrix
        model = glm::translate(model, glm::vec3(2, 0, 0));
        // send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
        // draw the second cube
        myCustomShader.useShaderProgram();
        glBindVertexArray(objectVAO);
        glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
}
```

Cubul va fi desenat tot în origine, dar transformarea de translație tocmai a mutat originea cu două unități la dreapta.

Efectuați o rotație ținând apăsată tasta "R". Observați cum se aplică transformarea pe întreaga scenă.

Acum, să aplicăm o rotație pe cel de-al doilea cub:

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

        // create rotation matrix
        model = glm::rotate(model, 0.3f, glm::vec3(0, 1, 0));
        // send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

        // create a translation matrix
        model = glm::translate(model, glm::vec3(2, 0, 0));
        // send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
        // draw the second cube
        myCustomShader.useShaderProgram();
        glBindVertexArray(objectVAO);
        glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
}
```

5 Temă

1. Controlați rotația celui de-al doilea cub folosind tasta "T".
2. Modificați rotația celui de-al doilea cub rotindu-l în jurul centrului său - punctul (2, 0, 0).
   * Sfat: utilizați secvența de operații $\mathbf{T_{inverse}}$ *** R * T**
3. Implementați clasa Camera (pornire de la Camera.hpp prezentă pe site-ul web). Controlați camera utilizând intrările de la tastatură și mouse. **Sugestie:** Înregistrați câteva funcții „callback” pentru a procesa intrările de la tastatură și mouse:

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

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


...


glfwSetKeyCallback(glWindow, keyboardCallback);

glfwSetCursorPosCallback(glWindow, mouseCallback);

glfwSetInputMode(glWindow, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```