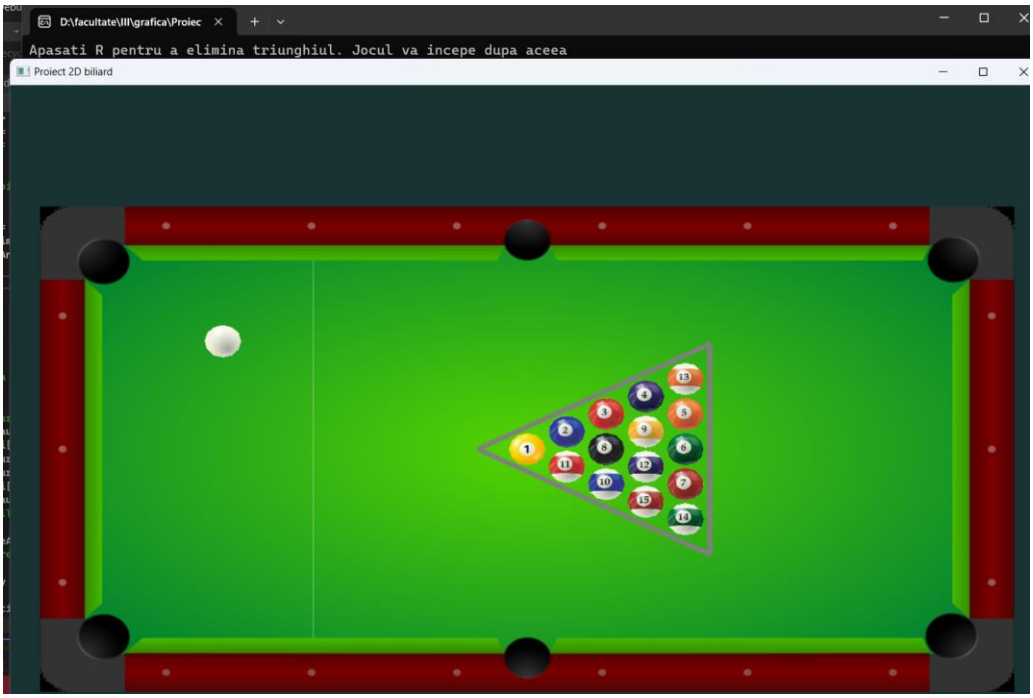
Tema Proiectului: Joc Biliard 2 D

Petre-Soldan Adela grupa 331

Conceptul proiectului:

Proiectul simuleaza un biliard cu 16 bile, una fiind alba. Fiecare jucator loveste bila alba cu tacul o singura data, apoi e rândul celuilalt. Eliminarea bilei albe duce la pierderea jocului. Jucatorul ‘A’ este cel care incepe si are asociate bilele 0-7, iar jucatorul B 9-15. Castiga cel care nu mai are pe masa nicio bila asociata. Informatiile despre bile sunt retinute in vectorul de struct vBile. Pentru miscare, fiecare bila are o viteza pe axele x si y. La fiecare randare se aduna la coordonate viteza inmultita cu un coeficient subunitar, pentru a simula forta de frecare. Cand modulul vitezei pe o axa e mult prea mic vitezei i se atribuie valoarea 0.0 pentru ca altfel ar dura foarte mult pana s-ar opri complet.

Transformarile folosite sunt **rotatia, translatia si scalarea.** La inceput bilele sunt asezate intr-un triunghi, mai putin cea alba. Bilele sunt **translatate** in pozitia curenta la fiecare randare in funcţie de coordonatele x si y. Coordonatele centrului bilei reprezinta coordonatele bilei.



Pornirea jocului:

Jocul incepe cand se apasa tasta R, care va activa cu GlutIdleFunc functia care se ocupa de animatia eliminarii triunghiului prin 3 tranzitii consecutive: mai intai se aplica o **scalare** treptata pe x si y pana la o anumita valoare pentru a da impresia ca triunghiul se ridica. Va fi modificata treptat si grosimea segmentelor. Apoi se creeaza un efect de asteptare incrementand o variabila pana la o anumita valoare. Apoi urmeaza **translatia** pe axa Y care aduce triunghiul in afara planului vizibil. Functia apeleaza

dupa fiecare schimbare GlPostRedisplay() pentru a forta redesenarea scenei si a crea efectul de miscare. Retinem la care dintre cele 3 etape suntem prin variabile bool.

```
void EliminareTriunghi()
{
    if (jocInceput == true)
        return;

    if (marireTriunghi == true)
    {
        scalareCurentaTriunghi += deltaMarireTriunghi;
        triunghiLineWidth += deltaLineWidth;
        if (scalareCurentaTriunghi >= maximMarireTriunghi)
        {
            asteptare = true;
            marireTriunghi = false;
        }
    }

    if (asteptare == true)
    {
        timpCurent += deltaAsteptare;
        if (timpCurent > timpFinal)
        {
            asteptare = false;
            translatareTriunghi= true;
        }
    }

    if (translatareTriunghi == true)
    {
        translatieCurentaTriunghi += deltaTanslatieTriunghi;
        if (translatieCurentaTriunghi > yMax + 11 * razaBile) // verificam daca triunghiul mai e vizibil pe ecran, daca nu, jocul incepe
        {
            jocInceput = true;
            translatareTriunghi = false;
            glutIdleFunc(NULL); // asa nu se va mai apela mereu functia de eliminare triunghi daca nu mai e cazul
            lovireBile = true;
            std::cout << "Incepe jocul!\n";
            std::cout << "E randul lui " << jucatorCurent << '\n';
            std::cout << "Apasati X pentru a lansa bila alba!\n";
            xTac = xInitialMingeAlba - razaBile - 5.0f; // punem tacul orientat spre bila alba
            yTac = yInitialMingeAlba;
        }
    }

    glutPostRedisplay();
}
```

```
// TRIUNGHIUL:
if (jocInceput == false) // triunghiul se afla undeva pe ecran
{
    glBindVertexArray(VaoId1);
    glLineWidth(triunghiLineWidth);
    codCol = 0;
    glUniform1i(codColLocation, codCol);

    scalareTriunghi = glm::scale(glm::mat4(1.0f), glm::vec3(scalareCurentaTriunghi, scalareCurentaTriunghi, 0.0));
    translatieTriunghi = glm::translate(glm::mat4(1.0f), glm::vec3(0.0, translatieCurentaTriunghi, 0.0));
    myMatrix = resizeMatrix * translatieTriunghi * translatieTriunghiDinOrigine * scalareTriunghi * translatieTriunghiOrigine;
    glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
    indexAnterior = 4;
    glDrawElements(GL_LINE_STRIP, 4, GL_UNSIGNED_INT, (void*)(indexAnterior * sizeof(GLuint)));
}
```

Se aplica mereu matricile de scalare si translatie, dar initial variabila pentru scalare e 1.0 si pentru translatie 0.0, deci nu vor avea efect pana nu se apasa R. Pentru ca triunghiul nu are centrul in origine mai este nevoie de alte 2 matrici de translatie: inainte sa il scalam il translatam in origine si dupa scalare in pozitia initiala (cele 2 translatii au loc doar pe axa x pentru ca triunghiul e pozitionat simetric fata de Ox). Vertices pentru triunghi:

```
// triunghi:
 -2.7 * razaBile, 0.0f, 0.0f, 1.0f,
 10 * razaBile, -3 * distantaInitialaCentre, 0.0f, 1.0f,
 10 * razaBile,  3 * distantaInitialaCentre, 0.0f, 1.0f,
```

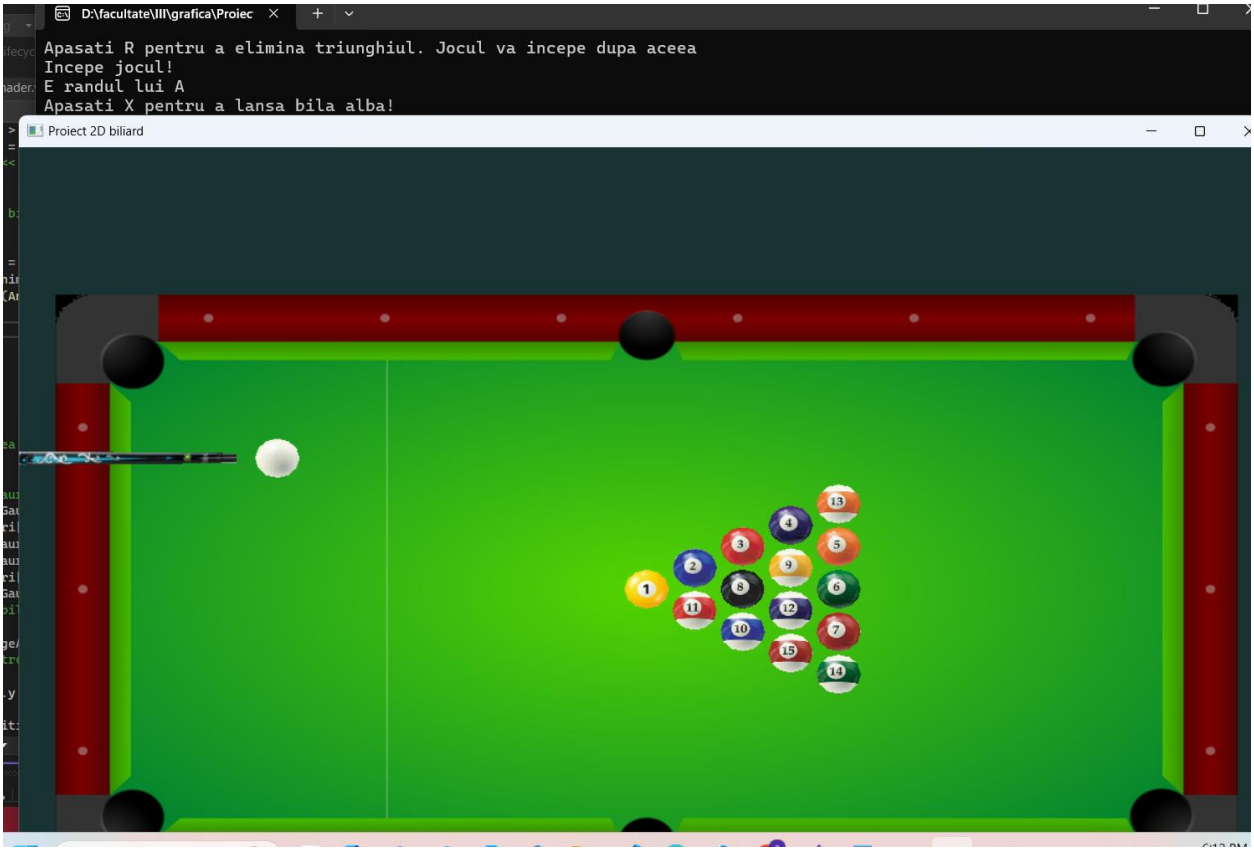
Deci scadem si apoi adunam la loc 3.65 * razaBile.

Dupa eliminarea triunghiului:

Jocul incepe (jocInceput = true) si apare in imagine tacul (este pe ecran doar cand e rândul cuiva sa loveasca, nu si in timpul miscarii bilelor). Pentru tac retinem mereu coordonatele x si y in raport cu originea (varful sa fie in origine, nu sa aibă centrul in origine), deci mereu folosim in render matricea de **translatie** cu x si y.

Deplasarea tacului:

Cand vine rândul cuiva sa loveasca (lovireBile == true), tacul este plasat pe coordonata y a bilei albe si cu un x putin mai mic decat al bilei.



Inainte de a lovi, jucatorul poate deplasa tacul in fata si spate (tastele A D, care cresc sau scad x-ul, de y neavand nevoie pentru ca tacul nerotit e asezat orizontal) sau sa il **roteasca** in jurul bilei albe(Q, E). Cu cat tacul e mai departe de bila cu atât lovitura este mai puternică. Pentru a fi rotit in jurul bilei il translatam cu o matrice care duce bila in origine, apoi rotim si ne intoarcem cu matricea de translatie de coordonate opuse.

```
void ProcessNormalKeys(unsigned char key, int x, int y)
{
    switch (key) {
        case 'r':
            if (marireTriunghi == false)
            {
                marireTriunghi = true;
                glutIdleFunc(EliminareTriunghi);
            }
            break;
        case 'a': // tacul se departeaza de bila, isi pastreaza orientarea
            if (lovireBile)
            {
                xTac += deltaPozTac * glm::sign(xTac - vBile[0].x); // ne ocupam doar de axa x pentru ca initial tacul este pe axa x si e rotit apoi
            }
            break;
        case 'd': // tacul se apropie de bila, dar nu trece de ea
            if (lovireBile)
            {
                // ne orientam mereu dupa unghiul dintre deltaPozTac si xTac:
                if (abs(xTac - vBile[0].x) >= razaBile + deltaPozTac + 0.5f) // verificam daca ne-am apropiat prea mult
                {
                    xTac -= deltaPozTac * glm::sign(xTac - vBile[0].x);
                }
            }
            break;
        case 'e': // tacul e rotit anti trigonometric
            if (lovireBile)
            {
                unghiTac -= deltaUnghiTac;
                if (unghiTac < 0)
                    unghiTac = 2 * PI;
                //std::cout << unghiTac / PI << '\n';
            }
            break;
        case 'q': // tacul e rotit trigonometirc
            if (lovireBile)
            {
                unghiTac += deltaUnghiTac;
                if (unghiTac > 2 * PI)
                    unghiTac = 0.0;
                //std::cout << unghiTac / PI << '\n';
            }
            break;
        case 'x': // lovirea bilei, mai intai o animatie in care tacul se apropie de bila, apoi se va misca bila
            if (lovireBile)
            {
                miscareSpate = true;
                deltaPozTacAnimatieFata = fabs(xTac - vBile[0].x) * 0.009f; // viteza cu care se apropie tacul va diferi in functie de distanta pana la minge
                glutIdleFunc(AnimatieTacSiLovireBila);
            }
            break;
    }
```

```
// TACUL:
if (lovireBile == true && stopJoc == false) // tacul apare doar daca este randul cuiva sa loveasca bila alba
{
    glBindVertexArray(VaoId1);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, textureTac);
    glUniform1i(glGetUniformLocation(ProgramId, "myTexture"), 0);

    codCol = 1;
    glUniform1i(codColLocation, codCol);
    translatieTacFataDeBila = glm::translate(glm::mat4(1.0f), glm::vec3(xTac, yTac, 0.0)); // pune tacul in fata bilei albe
    translatieBilaAlbaInOrigine = glm::translate(glm::mat4(1.0f), glm::vec3(-vBile[0].x, -vBile[0].y, 0.0)); // ca sa putem roti tacul in jurul bilei albe
    translatieBilaAlbaInapoi = glm::translate(glm::mat4(1.0f), glm::vec3(vBile[0].x, vBile[0].y, 0.0));
    rotireTac = glm::rotate(glm::mat4(1.0f), unghiTac, glm::vec3(0.0, 0.0, 1.0));
    myMatrix = resizeMatrix * translatieBilaAlbaInapoi * rotireTac * translatieBilaAlbaInOrigine * translatieTacFataDeBila;
    glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);

    indexAnterior = 8;
    glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_INT, (void*)(indexAnterior * sizeof(GLuint)));
}
```

Lovirea bilei:

Dupa aceea, tasta X declanseaza lovitura. Se activeaza cu GlutIdleFunc functia AnimatieTacSiLovireBila() care simuleaza prin randari repetate tacul departand-se putin de bila si apoi apropiindu-se pana atinge bila (distanta == razaBila), similar cu animatiile de eliminare a triunghiului. Apropierea de bila se face cu o viteza direct proportionala cu distanta dintre tac si bila, delta-ul fiind inmultit cu distanta. Apoi lovireBile = false si tacul dispare.

Inainte de a se activa functia care se ocupa de miscare si coliziuni, se calculeaza viteza nedescompusa a bilei direct proportional cu distanta. Pentru a o descompune folosim sin si cos(unghiRotire), pentru ca tacul cu un unghi de rotire cu valori intre (0, pi/2) s-ar afla de fapt in cadranul 3. In acest caz bila ar trebui sa aibă ambele viteze pozitive, adică valori din cadranul 1. Semnul vitezelor e calculat corect si pentru restul cadranelor.

Miscarea bilelor - functia Miscare():

Functia de miscare e activa cat timp există bile cu viteza nenula sau bile pentru care e activa animatia de cadere in gaura, apoi e dezactivata cu GlutIdleFunc(NULL), apare din nou tacul (plasat langa

bila alba), vine rândul celuilalt jucator (lovireBile e true din nou) si se afişează in consola cate bile au fost eliminate pana acum din ambele categorii.

```
bile pline:0 bile dungate:1.
Randul lui A
bile pline:0 bile dungate:1.
Randul lui B
bile pline:0 bile dungate:1.
Randul lui A
gaura 2
bile pline:0 bile dungate:2.
Randul lui B
```

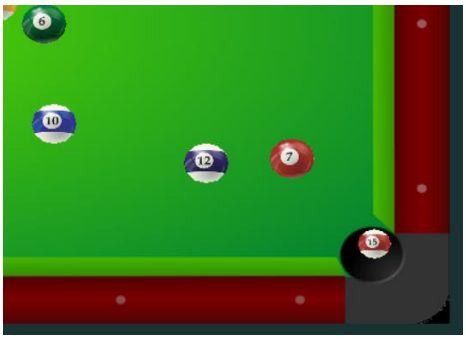
Ciocniri:

Ciocnirea intre bile aplica conservarea impulsului pentru corpuri cu mase egale. Fiecare bila primeste vitezele celeilalte. Conditia pentru coliziune e ca distanta dintre bile sa fie <= 2R. Functia de miscare verifica si coliziunile cu marginile mesei. Pentru coliziuni cu marginile orizontale se inverseaza vY, altfel vX. Masa are doar 4 vertices cu o textura (codCol = 1 pentru textura), coordonatele pana la care se pot deplasa bilele pe masa au fost aproximate.

Eliminarea bilelor:

Pentru fiecare gaura retin coordonatele aproximative ale centrului. Pentru a vedea daca o bila cade intr-o gaura calculam distanta dintre centrul gaurii si cel al bilei.

Pentru fiecare bila retinem daca a inceput animatia de cadere si daca a fost eliminata deja. Cand incepe caderea, vitezele bilei devin nule si bila ia coordonatele centrului gaurii (translatie). Fara translatie bilele nu s-ar afla complet in gaura. Apoi se aplica o **scalare** treptata de micsorare pana la 0.5f pentru a crea efectul de cadere, ceea ce ar putea constitui un aspect de originalitate. Apoi bila este marcata ca eliminata si nu mai e desenata. Pentru fiecare bila retin coeficientul curent de scalare (initial este 1.0 la toate).



```
// BILELE:
glBindVertexArray(VaoId2);
codCol = 1;
glUniform1i(codColLocation, codCol);
for (int i = 0; i < nrBile; i++)
{
    if (vBile[i].eliminata == false)
    {
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, textureBile[i]);
        glUniform1i(glGetUniformLocation(ProgramId, "myTexture"), 0);
        if (vBile[i].incepereEliminare == true)
            miscorareBilaCareCade = glm::scale(glm::mat4(1.0f), glm::vec3(vBile[i].scalareCurentaPentruMicsorare, vBile[i].scalareCurentaPentruMicsorare, 0.0));
        else
            miscorareBilaCareCade = glm::scale(glm::mat4(1.0f), glm::vec3(1.0f, 1.0f, 0.0));
        translatieBila = glm::translate(glm::mat4(1.0f), glm::vec3(vBile[i].x, vBile[i].y, 0.0));
        myMatrix = resizeMatrix * translatieBila * miscorareBilaCareCade;
        glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
        glDrawElements(GL_POLYGON, 12, GL_UNSIGNED_INT, (void*)(0));
    }
}
```

In functia de miscare parcurg toate bilele care nu sunt marcate ca eliminate:

- pentru fiecare bila care nu a cazut inca intr-o gaura verific daca la randarea curenta a cazut

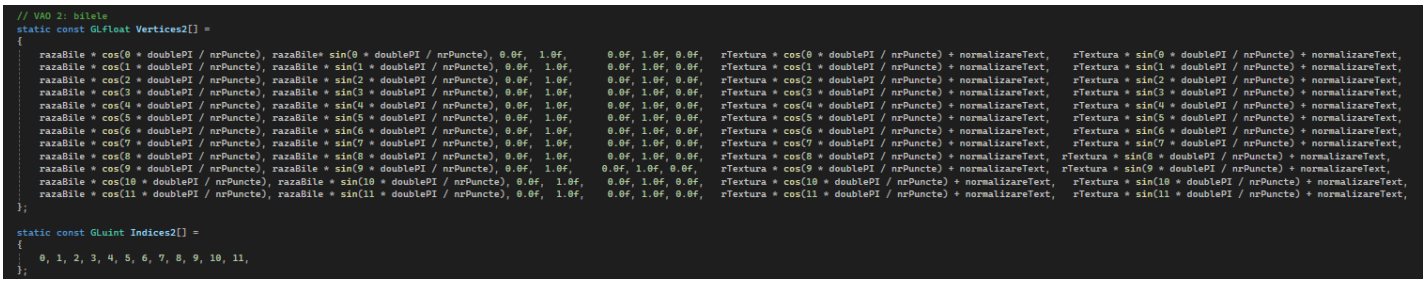
- pentru bilele care cad continui animatia scazand coeficientul de scalare cu o constanta (randarea dupa fiecare rulare a functiei de miscare va crea efectul animatiei)

```
    if (vBile[i].incepereEliminare == false) // daca bila nu a cazut intr o gaura pana acum
    {
        // verificam daca a cazut intr-o gaura:
        for (int j = 0; j < 6; j++) // verificam distanta de la centrul gaurii la centrul bilei. Daca e <= razaGaura bila va cadea in gaura
        {
            if (sqrt((vBile[i].x - vGauri[j].x) * (vBile[i].x - vGauri[j].x) + (vBile[i].y - vGauri[j].y) * (vBile[i].y - vGauri[j].y)) <= razaGauri)
            {
                vBile[i].vX = vBile[i].vY = 0.0f;
                std::cout << "gaura " << j << '\n';
                vBile[i].x = vGauri[j].x;
                vBile[i].y = vGauri[j].y;
                vBile[i].incepereEliminare = true;
                vBile[i].scalareCurentaPentruMicsorare = 1.0f;
                if (i < 9 && i >0)
                {
                    bilePlineEliminate += 1;
                }
                else if (i > 9)
                {
                    bileDungateEliminate += 1;
                }
                else if (i == 0)
                {
                    stopJoc = true;
                    if (jucatorCurent == 'A') castigator = 'B';
                    else castigator = 'A';
                    std::cout << "Eliminarea bilei albe este interzisa. Catiga " << castigator << '\n';
                }
                if (bileDungateEliminate == 7) {
                    std::cout << "Au fost eliminate toate bilele dungate. Catiga B\n";
                    stopJoc = true;
                }
                else if (bilePlineEliminate == 7) {
                    std::cout << "Au fost eliminate toate bilele pline. Catiga A\n";
                    stopJoc = true;
                }
            }
        }
    }

    if (vBile[i].incepereEliminare == true) // continuam simularea caderii pentru bilele care cad
    {
        bileCareCad += 1;
        if (vBile[i].scalareCurentaPentruMicsorare - deltaMicsorareBila < 0.5)
        {
            vBile[i].eliminata = true;
            vBile[i].incepereEliminare = false;
        }
        else
        {
            vBile[i].scalareCurentaPentruMicsorare -= deltaMicsorareBila;
        }
    }
}
```

Textura bilelor:

Bilele sunt poligoane cu 12 puncte pe un cerc de raza 6.0 cu centrul in origine. Pentru a obţine coordonatele de textura trebuie sa cream un cerc de raza ½ cu centrul in (½, ½). Deci raza cercului ar fi ½ si apoi la toate coordonatele adunam ½. Avem un vector de texturi pentru bile astfel incat bilei i ii corespunde textura i.



```
// VAO 2: bilele
static const GLfloat Vertices2[] =
{
    razaBile * cos(0 * doublePI / nrPuncte), razaBile* sin(0 * doublePI / nrPuncte), 0.0f,  1.0f,      0.0f, 1.0f, 0.0f,   rTextura * cos(0 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(0 * doublePI / nrPuncte) + normalizareText,
    razaBile * cos(1 * doublePI / nrPuncte), razaBile * sin(1 * doublePI / nrPuncte), 0.0f,  1.0f,      0.0f, 1.0f, 0.0f,   rTextura * cos(1 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(1 * doublePI / nrPuncte) + normalizareText,
    razaBile * cos(2 * doublePI / nrPuncte), razaBile * sin(2 * doublePI / nrPuncte), 0.0f,  1.0f,      0.0f, 1.0f, 0.0f,   rTextura * cos(2 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(2 * doublePI / nrPuncte) + normalizareText,
    razaBile * cos(3 * doublePI / nrPuncte), razaBile * sin(3 * doublePI / nrPuncte), 0.0f,  1.0f,      0.0f, 1.0f, 0.0f,   rTextura * cos(3 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(3 * doublePI / nrPuncte) + normalizareText,
    razaBile * cos(4 * doublePI / nrPuncte), razaBile * sin(4 * doublePI / nrPuncte), 0.0f,  1.0f,      0.0f, 1.0f, 0.0f,   rTextura * cos(4 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(4 * doublePI / nrPuncte) + normalizareText,
    razaBile * cos(5 * doublePI / nrPuncte), razaBile * sin(5 * doublePI / nrPuncte), 0.0f,  1.0f,      0.0f, 1.0f, 0.0f,   rTextura * cos(5 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(5 * doublePI / nrPuncte) + normalizareText,
    razaBile * cos(6 * doublePI / nrPuncte), razaBile * sin(6 * doublePI / nrPuncte), 0.0f,  1.0f,      0.0f, 1.0f, 0.0f,   rTextura * cos(6 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(6 * doublePI / nrPuncte) + normalizareText,
    razaBile * cos(7 * doublePI / nrPuncte), razaBile * sin(7 * doublePI / nrPuncte), 0.0f,  1.0f,      0.0f, 1.0f, 0.0f,   rTextura * cos(7 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(7 * doublePI / nrPuncte) + normalizareText,
    razaBile * cos(8 * doublePI / nrPuncte), razaBile * sin(8 * doublePI / nrPuncte), 0.0f,  1.0f,      0.0f, 1.0f, 0.0f,   rTextura * cos(8 * doublePI / nrPuncte) + normalizareText,  rTextura * sin(8 * doublePI / nrPuncte) + normalizareText,
    razaBile * cos(9 * doublePI / nrPuncte), razaBile * sin(9 * doublePI / nrPuncte), 0.0f,  1.0f,     0.0f, 1.0f, 0.0f,    rTextura * cos(9 * doublePI / nrPuncte) + normalizareText,  rTextura * sin(9 * doublePI / nrPuncte) + normalizareText,
    razaBile * cos(10 * doublePI / nrPuncte), razaBile * sin(10 * doublePI / nrPuncte), 0.0f,  1.0f,    0.0f, 1.0f, 0.0f,   rTextura * cos(10 * doublePI / nrPuncte) + normalizareText,   rTextura * sin(10 * doublePI / nrPuncte) + normalizareText,
    razaBile * cos(11 * doublePI / nrPuncte), razaBile * sin(11 * doublePI / nrPuncte), 0.0f,  1.0f,    0.0f, 1.0f, 0.0f,   rTextura * cos(11 * doublePI / nrPuncte) + normalizareText,   rTextura * sin(11 * doublePI / nrPuncte) + normalizareText,
};

static const GLuint Indices2[] =
{
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
};
```

Bibliografie:

https://www.fizichim.ro/docs/fizica/clasa9/capitolul4-teoreme-de-variatie-si-legi-de-conservare-in-mecanica/IV-8-ciocniri/IV-8-2-ciocniri-perfect-elastice/

https://www.opengl.org/resources/libraries/glut/spec3/node63.html

https://learnopengl.com/

Codul sursa:

Vertex shader:

```
#version 330 core


//  Variabile de intrare (dinspre programul principal);

layout (location = 0) in vec4 in_Position;     //  Se preia din buffer de pe prima pozitie (0) atributul care contine coordonatele;

layout (location = 1) in vec4 in_Color;       //  Se preia din buffer de pe a doua pozitie (1) atributul care contine culoarea;

layout (location=2) in vec2 texCoord;         //  Se preia din buffer de pe a treia pozitie (2) atributul care contine textura;


//  Variabile de iesire;

out vec4 gl_Position;   //  Transmite pozitia actualizata spre programul principal;

out vec4 ex_Color;      //  Transmite culoarea (de modificat in Shader.frag);

out vec2 tex_Coord;     //  Transmite textura (de modificat in Shader.frag);


//  Variabile uniforme;

uniform mat4 myMatrix;


void main(void)
```

```
 {
  gl_Position = myMatrix*in_Position;
  ex_Color = in_Color;
  tex_Coord = vec2(texCoord.x, 1-texCoord.y);
 }
```

Fragment shader

```
#version 330 core

//		Variabile de intrare (dinspre Shader.vert);
in vec4 ex_Color;
in vec2 tex_Coord;			//		Coordonata de texturare;

//		Variabile de iesire	(spre programul principal);
out vec4 out_Color;			//		Culoarea actualizata;

// Variabile uniform:
uniform int codColShader;
uniform sampler2D myTexture;

void main(void)
 {
		switch (codColShader)
		{
			case 0:
			  out_Color = ex_Color;
			  break;
			case 1:
				 out_Color = texture(myTexture, tex_Coord);
				break;
			case 2:
				out_Color=vec4 (1.0, 0.0, 0.0, 0.0);
				break;
			default:
```

```
                break;
        };
}
```

Main.cpp

```
#include <windows.h>
#include <stdlib.h>        //  Biblioteci necesare pentru citirea shaderelor;
#include <stdio.h>
#include <GL/glew.h>       //  Definește prototipurile functiilor OpenGL si constantele necesare pentru programarea OpenGL moderna;
#include <GL/freeglut.h>   //       Include functii pentru:
                                                                                //              - gestionarea ferestrelor si evenimentelor de tastatura si mouse,
                                                                                //  - desenarea de primitive grafice precum dreptunghiuri, cercuri sau linii,
                                                                                //  - crearea de meniuri si submeniuri;
#include "loadShaders.h"           //              Fisierul care face legatura intre program si shadere;
#include "glm/glm.hpp"             //              Bibloteci utilizate pentru transformari grafice;
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtx/transform.hpp"
#include "glm/gtc/type_ptr.hpp"
#include "SOIL.h"
#include <iostream>
#include <chrono>
#include <math.h>
#include <glm/gtx/log_base.hpp>


//  Identificatorii obiectelor de tip OpenGL;
GLuint
        VaoId1, VaoId2,                 // Identificatori pentru cele doua VAO;
        VboId1, VboId2,                 // Fiecare VAO va contine propriile buffere pt. coordonate, culori, indici;
        EboId1, EboId2,
        TextVboId1, TextVboId2,
        ProgramId,
        myMatrixLocation,
        codColLocation;
GLuint
```

```
        textureMasa, textureTac, textureBile[20];
GLfloat
winWidth = 1200, winHeight = 940;
glm::mat4
myMatrix, resizeMatrix, translatieBila, scalareTriunghi, translatieTriunghi, translatieTriunghiOrigine, translatieTriunghiDinOrigine,
translatieTacFataDeBila, translatieTac, rotireTac, translatieBilaAlbaInOrigine, translatieBilaAlbaInapoi,
miscorareBilaCareCade;
//      Variabile pentru proiectia ortogonala;
float xMin = -170.f, xMax = 170.f, yMin = -135.f, yMax = 135.f;
// marginile in care se incadreaza bilele:
float xMasaMin = -146.0f, xMasaMax = 135.0f, yMasaMin = -76.0, yMasaMax = 63.0f;
// render si fizica jocului:
float PI = 3.141592, doublePI = 2 * PI;
float razaBile = 6.0f, coeficientFrecare = 0.999, masaBila = 1.0, distantaInitialaCentre = 2 * razaBile + 1, rTextura = 0.5f, normalizareText = 0.5f,
razaGauri = 6.5f;
float coeficientLansareBilaAlba = 0.13f;
float masaTac = 2.0, xTac = 0.0f, yTac = 0.0f, unghiTac = 0.0f, deltaPozTac = 0.9f, deltaUnghiTac = PI / 50;
float xInitialTac = 0.0, yInitialTac = 0; // varful tacului e in origine
float xInitialMingeAlba = -100, yInitialMingeAlba = 40;
int codCol, indexAnterior;
const int nrPuncte = 12, nrBile = 16;
// pentru simularea caderii in gaura:
float deltaMicsorareBila = 0.0008;

// pentru ridicarea triunghiului:
// se ridica prin apasarea tastei R, dupa aceea va incepe jocul
bool marireTriunghi = false, asteptare = false, translatareTriunghi = false, jocInceput = false;
// etape: rulare cod, marireTriunghi, asteprare, translatareTriunghi, jocInceput;
float deltaMarireTriunghi = 0.008, scalareCurentaTriunghi = 1.0, maximMarireTriunghi = 1.5, deltaLineWidth = 0.05f, triunghiLineWidth = 6.0f;
float deltaTanslatieTriunghi = 0.2, translatieCurentaTriunghi = 0.0;
float deltaAsteptare = 0.05, timpFinal = 10.0, timpCurent = 0.0f;

//float deltaMarireTriunghi = 100.0, scalareCurentaTriunghi = 1.0, maximMarireTriunghi = 1.5;
//float deltaTanslatieTriunghi = 100.0, translatieCurentaTriunghi = 0.0;
//float deltaAsteptare = 100.0, timpFinal = 10.0, timpCurent = 0.0f;
```

```
//animatia de miscare a tacului la lovire:
float miscareInSpate = 10.0f, miscareInSpateCurent = 0.0f, deltaPozTacAnimatieSpate = 0.05f;
float deltaPozTacAnimatieFata;
bool miscareFata = false, miscareSpate = false;
// scor:
int scorJucatorA = 0, scorJucatorB = 0;
char jucatorCurent = 'A';
int tipBilaA = 1, tipBilaB = 2; // jucatorul A trebuie sa eliminse bilele 0-7 si jucatorul B bilele 9-15
int bilePlineEliminate = 0, bileDungateEliminate = 0;
bool lovireBile = false, miscareBile = false, stopJoc = false;
char castigator = NULL;
// variabile pentru bile:
struct bila {
	float x, y;
	float vX, vY;
	bool eliminata;
	bool incepereEliminare;
	float scalareCurentaPentruMicsorare;
} vBile[nrBile + 1];

struct gaura {
	float x, y;
} vGauri[7];

void EliminareTriunghi()
{
	if (jocInceput == true)
		return;

	if (marireTriunghi == true)
	{
		scalareCurentaTriunghi += deltaMarireTriunghi;
		triunghiLineWidth += deltaLineWidth;
		if (scalareCurentaTriunghi >= maximMarireTriunghi)
```

```
		{
			asteptare = true;
			marireTriunghi = false;
		}
	}

	if (asteptare == true)
	{
		timpCurent += deltaAsteptare;
		if (timpCurent > timpFinal)
		{
			asteptare = false;
			translatareTriunghi= true;
		}
	}

	if (translatareTriunghi == true)
	{
		translatieCurentaTriunghi += deltaTanslatieTriunghi;
		if (translatieCurentaTriunghi > yMax + 11 * razaBile) // verificam daca triunghiul mai e vizibil pe ecran, daca nu, jocul
incepe
		{
			jocInceput = true;
			translatareTriunghi = false;
			glutIdleFunc(NULL); // asa nu se va mai apela mereu functia de eliminare triunghi daca nu mai e cazul
			lovireBile = true;
			std::cout << "Incepe jocul!\n";
			std::cout << "E randul lui " << jucatorCurent << '\n';
			std::cout << "Apasati X pentru a lansa bila alba!\n";
			xTac = xInitialMingeAlba - razaBile - 5.0f; // punem tacul orientat spre bila alba
			yTac = yInitialMingeAlba;
		}
	}

	glutPostRedisplay();
```

```
}

void Miscare()
{
	int vitezeNenule = 0;
	int bileCareCad = 0;

	for (int i = 0; i < nrBile; i++)
	{
		if (vBile[i].eliminata == true)
		{
			continue;
		}
		vBile[i].x += vBile[i].vX * 0.18;
		vBile[i].y += vBile[i].vY * 0.18;

		vBile[i].vX *= coeficientFrecare; // vitezele scad in functie de frecare
		vBile[i].vY *= coeficientFrecare;

		if (fabs(vBile[i].vX) < 0.01)
		{
			vBile[i].vX = 0.0;
		}
		if (fabs(vBile[i].vY) < 0.01)
		{
			vBile[i].vY = 0.0;
		}


		if (vBile[i].incepereEliminare == false) // daca bila nu a cazut intr o gaura pana acum
		{
			// verificam daca a cazut intr-o gaura:
			for (int j = 0; j < 6; j++) // verificam distanta de la centrul gaurii la centrul bilei. Daca e <= razaGaura bila va
cadea in gaura
			{
```

```
                                        if (sqrt((vBile[i].x - vGauri[j].x) * (vBile[i].x - vGauri[j].x) + (vBile[i].y - vGauri[j].y) * (vBile[i].y -
vGauri[j].y)) <= razaGauri)
                                        {
                                                vBile[i].vX = vBile[i].vY = 0.0f;
                                                std::cout << "gaura " << j << '\n';
                                                vBile[i].x = vGauri[j].x;
                                                vBile[i].y = vGauri[j].y;
                                                vBile[i].incepereEliminare = true;
                                                vBile[i].scalareCurentaPentruMicsorare = 1.0f;
                                                if (i < 9 && i >0)
                                                {
                                                        bilePlineEliminate += 1;
                                                }
                                                else if (i > 9)
                                                {
                                                        bileDungateEliminate += 1;
                                                }
                                                else if (i == 0)
                                                {
                                                        stopJoc = true;
                                                        if (jucatorCurent == 'A') castigator = 'B';
                                                        else castigator = 'A';
                                                        std::cout << "Eliminarea bilei albe este interzisa. Catiga " << castigator << '\n';
                                                }
                                                if (bileDungateEliminate == 7) {
                                                        std::cout << "Au fost eliminate toate bilele dungate. Catiga B\n";
                                                        stopJoc = true;
                                                }
                                                else if (bilePlineEliminate == 7) {
                                                        std::cout << "Au fost eliminate toate bilele pline. Catiga A\n";
                                                        stopJoc = true;
                                                }
                                        }
                                }
                        }
```

```
		if (vBile[i].incepereEliminare == true) // continuam simularea caderii pentru bilele care cad
		{
			bileCareCad += 1;
			if (vBile[i].scalareCurentaPentruMicsorare - deltaMicsorareBila < 0.5)
			{
				vBile[i].eliminata = true;
				vBile[i].incepereEliminare = false;
			}
			else
			{
				vBile[i].scalareCurentaPentruMicsorare -= deltaMicsorareBila;
			}
		}
	}

	for (int i = 0; i < nrBile; i++)
	{
		if (fabs(vBile[i].vX) != 0.0 || fabs(vBile[i].vY) != 0.0)
		{
			vitezeNenule += 1;
		}
	}

	if (vitezeNenule == 0 && bileCareCad == 0) // daca nu se mai misca nicio bila pe masa si nu mai e nicio bila pt care simulam caderea
in gaura
	{
		glutIdleFunc(NULL);
		miscareBile = false;
		lovireBile = true;
		if (jucatorCurent == 'A')
		{
			jucatorCurent = 'B';
		}
		else
```

```
                {
                        jucatorCurent = 'A';
                }
                if (stopJoc == false)
                {
                        std::cout << "bile pline:" << bilePlineEliminate << " bile dungate:" << bileDungateEliminate << ".\nRandul lui "
<< jucatorCurent << '\n';
                        xTac = vBile[0].x - razaBile - 10.0f; // tacul il punem iar in fata bilei albe
                        yTac = vBile[0].y;

                }
        }

        // verificam coliziuni:
        for (int i = 0; i < nrBile; i++)
        {
                if (vBile[i].incepereEliminare == true || vBile[i].eliminata == true)
                {
                        continue;
                }
                for (int j = i + 1; j < nrBile; j++) // coliziuni cu alte bile
                {
                        float distantaBile = sqrt((vBile[i].x - vBile[j].x) * (vBile[i].x - vBile[j].x) + (vBile[i].y - vBile[j].y) * (vBile[i].y -
vBile[j].y));
                        if (distantaBile <= 2 * razaBile) // avem coliziune
                        {
                                // ciocnirea pe axa x:
                                float vi = vBile[i].vX, vj = vBile[j].vX;
                                vBile[i].vX = vj;
                                vBile[j].vX = vi;
                                // ciocnirea pe axa y:
                                vi = vBile[i].vY, vj = vBile[j].vY;
                                vBile[i].vY = vj;
                                vBile[j].vY = vi;
                        }
                }
```

```
                // coliziuni cu coltirile mesei:
                if (vBile[i].x - razaBile <= xMasaMin) // latura din stanga
                {
                        vBile[i].vX *= -1.0;
                }
                if (vBile[i].x - razaBile >= xMasaMax) // latura din dreapta
                {
                        vBile[i].vX *= -1.0;
                }
                if (vBile[i].y - razaBile <= yMasaMin) // latura de sus
                {
                        vBile[i].vY *= -1.0;
                }
                if (vBile[i].y - razaBile >= yMasaMax) // latura de jos
                {
                        vBile[i].vY *= -1.0;
                }
        }
        glutPostRedisplay();
}

void AnimatieTacSiLovireBila()
{
        if (miscareSpate)
        {
                //std::cout << miscareInSpateCurent << ' ';
                miscareInSpateCurent += deltaPozTacAnimatieSpate * glm::sign(xTac - vBile[0].x);
                //std::cout << miscareInSpateCurent << '\n';
                xTac += deltaPozTacAnimatieSpate * glm::sign(xTac - vBile[0].x);
                if (fabs(miscareInSpateCurent) >= miscareInSpate)
                {
                        miscareSpate = false;
                        miscareFata = true;
                        miscareInSpateCurent = 0.0f;
```

```
                }
                glutPostRedisplay();
        }
        else if (miscareFata)
        {
                float deltaCuSemn = - glm::sign(xTac - vBile[0].x) * deltaPozTacAnimatieFata; // cu mins ca ne apropiem de bila
                if (fabs(xTac + deltaCuSemn - vBile[0].x) <= razaBile) // daca tacul s ar apropia prea mult incepe miscarea bilei acum
                {
                        miscareFata = false;
                }
                else
                {
                        xTac += deltaCuSemn;
                }
                glutPostRedisplay();
        }
        else if (miscareSpate == false && miscareFata == false) // s-a terminat animatia pentru tac, bila alba primeste impuls si incepe
miscarea
        {
                lovireBile = false;
                // trebuie sa dam viteza bilei albe
                // ne vom folosi de unghiTac, nu putem calcula unghiul in functie de cat de rotit este tacul pentru ca rotirea e facuta de
glm::rotate;
                // pentru noi tacul e mereu orizontal pe axa x cu yTac = yBilaAlba. Nu ii stim coordonatele adevrate dupa rotire.
                /* daca le am fi stiut:
                float distantaBilaVarfTac = sqrt((vBile[0].x - xTac) * (vBile[0].x - xTac) + (vBile[0].y - yTac) * (vBile[0].y - yTac));
                float vitezaNedescompusa = distantaBilaVarfTac * coeficientLansareBilaAlba;
                vBile[0].vX = vitezaNedescompusa * -1.0 * (xTac - vBile[0].x) / distantaBilaVarfTac;
                vBile[0].vY = vitezaNedescompusa * -1.0 * (yTac - vBile[0].y) / distantaBilaVarfTac;
                */

                float distantaBilaVarfTac = abs(vBile[0].x - xTac); // distanta nu tine cont si de y
                float vitezaNedescompusa = coeficientLansareBilaAlba * distantaBilaVarfTac;
                if (vitezaNedescompusa > 0.8f)
                {
                        vitezaNedescompusa = 4.0f * vitezaNedescompusa;
```

```
                }

                //if (unghiTac >= 0 && unghiTac <= PI / 2) // pentru unghiul tacului este cadranul 1, tacul arata ca in cadranul 3, deci viteza va fi orientata spre cadranul 1
                //{
                //      vBile[0].vX = vitezaNedescompusa * cos(unghiTac);
                //      vBile[0].vY = vitezaNedescompusa * sin(unghiTac);
                //}
                //else if (unghiTac >= PI / 2 && unghiTac <= PI)
                //{
                //      vBile[0].vX = -1.0 * vitezaNedescompusa * cos(PI - unghiTac);
                //      vBile[0].vY = vitezaNedescompusa * sin(PI - unghiTac);
                //}
                //else if (unghiTac >= PI && unghiTac <= 3.0 * PI / 2.0)
                //{
                //      vBile[0].vX = -1.0 * vitezaNedescompusa * cos(unghiTac - PI);
                //      vBile[0].vY = -1.0 * vitezaNedescompusa * sin(unghiTac - PI);
                //}
                //else
                //{
                //      vBile[0].vX = vitezaNedescompusa * cos(2 * PI - unghiTac);
                //      vBile[0].vY = -1.0 * vitezaNedescompusa * sin(2 * PI - unghiTac);
                //}
                vBile[0].vX = vitezaNedescompusa * cos(unghiTac);
                vBile[0].vY = vitezaNedescompusa * sin(unghiTac);
                //std::cout << "vx=" << vBile[0].vX << " vy=" << vBile[0].vY << '\n';
                glutIdleFunc(Miscare);
        }

}

void ProcessNormalKeys(unsigned char key, int x, int y)
{
        switch (key) {
                case 'r':
```

```
			if (marireTriunghi == false)
			{
				marireTriunghi = true;
				glutIdleFunc(EliminareTriunghi);
			}
			break;
		case 'a': // tacul se departeaza de bila, isi pastreaza orientarea
			if (lovireBile)
			{
				xTac += deltaPozTac * glm::sign(xTac - vBile[0].x); // ne ocupam doar de axa x pentru ca initial tacul este pe axa x si e rotit apoi
			}
			break;
		case 'd': // tacul se apropie de bila, dar nu trece de ea
			if (lovireBile)
			{
				// ne orientam mereu dupa unghiul dintre deltaPozTac si xTac:
				if (abs(xTac - vBile[0].x) >= razaBile + deltaPozTac + 0.5f) // verificam daca ne-am apropiat prea mult
				{
					xTac -= deltaPozTac * glm::sign(xTac - vBile[0].x);
				}
			}
			break;
		case 'e': // tacul e rotit anti trigonometric
			if (lovireBile)
			{
				unghiTac -= deltaUnghiTac;
				if (unghiTac < 0)
					unghiTac = 2 * PI;
				//std::cout << unghiTac / PI << '\n';
			}
			break;
		case 'q': // tacul e rotit trigonometirc
			if (lovireBile)
			{
```

```
                                        unghiTac += deltaUnghiTac;
                                        if (unghiTac > 2 * PI)
                                                unghiTac = 0.0;
                                        //std::cout << unghiTac / PI << '\n';
                                }
                                break;
                        case 'x': // lovirea bilei, mai intai o animatie in care tacul se apropie de bila, apoi se va misca bila
                                if (lovireBile)
                                {
                                        miscareSpate = true;
                                        deltaPozTacAnimatieFata = fabs(xTac - vBile[0].x) * 0.009f; // viteza cu care se apropie tacul va diferi
in functie de distanta pana la minge
                                        glutIdleFunc(AnimatieTacSiLovireBila);
                                }
                                break;

        }
        if (key == 27)
                exit(0);
        glutPostRedisplay();
}
// Functii pentru desfasurarea jocului:
void InitializeGame()
{
        // intializam centrele gaurilor:
        vGauri[0].x = -140.0f, vGauri[0].y = -69.0f; // stanga jos
        vGauri[1].x = 0.0f, vGauri[1].y = -76.0f; // mijloc jos
        vGauri[2].x = 140.0f, vGauri[2].y = -66.0f; // dreapta jos
        vGauri[3].x = 140.0f, vGauri[3].y = 68.0f; // dreapta sus
        vGauri[4].x = 0.0f, vGauri[4].y = 74.0f; // mijloc sus
        vGauri[5].x = -140.0f, vGauri[5].y = 68.0f; // stanga sus
        // initalizam pozitiile bilelor (pozitia unei bile inseamna unde este centrul ei):
        // bila alba:
        vBile[0].x = xInitialMingeAlba, vBile[0].y = yInitialMingeAlba;
        // distanta initiala dintre centrele bilelor vecine este 13 (2*r + 1):
```

```
		// linia 1:
		vBile[1].x = 0, vBile[1].y = 0;
		// linia 2:
		vBile[11].x = distantaInitialaCentre, vBile[11].y = -distantaInitialaCentre / 2;
		vBile[2].x = distantaInitialaCentre, vBile[2].y = distantaInitialaCentre / 2;
		// linia 3:
		vBile[10].x = 2 * distantaInitialaCentre, vBile[10].y = -distantaInitialaCentre;
		vBile[8].x = 2 * distantaInitialaCentre, vBile[8].y = 0;
		vBile[3].x = 2 * distantaInitialaCentre, vBile[3].y = distantaInitialaCentre;
		// linia 4:
		vBile[12].x = 3 * distantaInitialaCentre, vBile[12].y = -distantaInitialaCentre / 2;
		vBile[9].x = 3 * distantaInitialaCentre, vBile[9].y = distantaInitialaCentre / 2;
		vBile[15].x = 3 * distantaInitialaCentre, vBile[15].y = -3 * distantaInitialaCentre / 2;
		vBile[4].x = 3 * distantaInitialaCentre, vBile[4].y = 3 * distantaInitialaCentre / 2;
		// linia 5;
		vBile[6].x = 4 * distantaInitialaCentre, vBile[6].y = 0;
		vBile[5].x = 4 * distantaInitialaCentre, vBile[5].y = distantaInitialaCentre;
		vBile[7].x = 4 * distantaInitialaCentre, vBile[7].y = -distantaInitialaCentre;
		vBile[13].x = 4 * distantaInitialaCentre, vBile[13].y = 2 * distantaInitialaCentre;
		vBile[14].x = 4 * distantaInitialaCentre, vBile[14].y = -2 * distantaInitialaCentre;

		std::cout << "Apasati R pentru a elimina triunghiul. Jocul va incepe dupa aceea\n";
}
//		Functia de incarcare a texturilor in program;
void LoadTexture(const char* texturePath, GLuint &texture)
{
		// Generarea unui obiect textura si legarea acestuia;
		glGenTextures(1, &texture);
		glBindTexture(GL_TEXTURE_2D, texture);

		//		Desfasurarea imaginii pe orizontala/verticala in functie de parametrii de texturare;
		glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
		glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

		// Modul in care structura de texeli este aplicata pe cea de pixeli;
```

```
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

	// Incarcarea texturii si transferul datelor in obiectul textura;
	int width, height;
	unsigned char* image = SOIL_load_image(texturePath, &width, &height, 0, SOIL_LOAD_RGB);
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
	glGenerateMipmap(GL_TEXTURE_2D);

	// Eliberarea resurselor
	SOIL_free_image_data(image);
	glBindTexture(GL_TEXTURE_2D, 0);
}

// Crearea si compilarea obiectelor de tip shader;
//	Trebuie sa fie in acelasi director cu proiectul actual;
// Shaderul de varfuri / Vertex shader - afecteaza geometria scenei;
// Shaderul de fragment / Fragment shader - afecteaza culoarea pixelilor;
void CreateShaders(void)
{
	ProgramId = LoadShaders("shader1.vert", "shader1.frag");
	glUseProgram(ProgramId);
}

// Se initializeaza Vertex Buffer Objects (VBOs) pentru tranferul datelor spre memoria placii grafice (spre shadere);
// In acestea se stocheaza date despre varfuri (coordonate, culori, indici, texturare etc.);

// Sunt create VAO-urile
void CreateVAOs(void)
{
	// pentru VAO1: masa biliard, tac, triunghi
	static const GLfloat Vertices1[] = {
		//	Coordonate;	Culori;	Coordonate de texturare;
		-160.0f, -90.0f, 0.0f, 1.0f,	1.0f, 0.0f, 0.0f,	0.0f, 0.0f,
```

```
                    160.0f, -90.0f, 0.0f, 1.0f,   0.0f, 1.0f, 0.0f,      1.0f, 0.0f,
                    160.0f,  90.0f, 0.0f, 1.0f,   1.0f, 1.0f, 0.0f,      1.0f, 1.0f,
                   -160.0f,  90.0f, 0.0f, 1.0f,   0.0f, 1.0f, 1.0f,      0.0f, 1.0f,
                   // triunghi:
                    -2.7 * razaBile, 0.0f, 0.0f, 1.0f,                  0.5f, 0.5f, 0.5f,            0.0f, 0.0f,
                    10 * razaBile, -3 * distantaInitialaCentre, 0.0f, 1.0f,   0.5f, 0.5f, 0.5f,              1.0f, 0.0f,
                    10 * razaBile,  3 * distantaInitialaCentre, 0.0f, 1.0f,   0.5f, 0.5f, 0.5f,              1.0f, 1.0f,
                   // tac:
                   xInitialTac -140.0f, -3.0f, 0.0f, 1.0f,        1.0f, 0.0f, 0.0f,        0.0f, 0.0f,
                   xInitialTac, -1.5f, 0.0f, 1.0f,              1.0f, 0.0f, 0.0f,              1.0f, 0.0f,
                   xInitialTac, 1.5f, 0.0f, 1.0f,                              1.0f, 0.0f, 0.0f,            1.0f, 1.0f,
                   xInitialTac - 140.0f, 3.0f, 0.0f, 1.0f,        1.0f, 0.0f, 0.0f,        0.0f, 1.0f,
        };


        //          Indicii care determina ordinea de parcurgere a varfurilor;
        static const GLuint Indices1[] = {
                0, 1, 2, 3, // masa
                4, 5, 6, 4, // triunghi
                7, 8, 9, 10, // tac
        };


        // VAO 2: bilele
        static const GLfloat Vertices2[] =
        {
                razaBile * cos(0 * doublePI / nrPuncte), razaBile* sin(0 * doublePI / nrPuncte), 0.0f,  1.0f,     0.0f, 1.0f, 0.0f,   rTextura *
cos(0 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(0 * doublePI / nrPuncte) + normalizareText,

                razaBile * cos(1 * doublePI / nrPuncte), razaBile * sin(1 * doublePI / nrPuncte), 0.0f,  1.0f,     0.0f, 1.0f, 0.0f,   rTextura *
cos(1 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(1 * doublePI / nrPuncte) + normalizareText,

                razaBile * cos(2 * doublePI / nrPuncte), razaBile * sin(2 * doublePI / nrPuncte), 0.0f,  1.0f,     0.0f, 1.0f, 0.0f,   rTextura *
cos(2 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(2 * doublePI / nrPuncte) + normalizareText,

                razaBile * cos(3 * doublePI / nrPuncte), razaBile * sin(3 * doublePI / nrPuncte), 0.0f,  1.0f,     0.0f, 1.0f, 0.0f,   rTextura *
cos(3 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(3 * doublePI / nrPuncte) + normalizareText,

                razaBile * cos(4 * doublePI / nrPuncte), razaBile * sin(4 * doublePI / nrPuncte), 0.0f,  1.0f,     0.0f, 1.0f, 0.0f,   rTextura *
cos(4 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(4 * doublePI / nrPuncte) + normalizareText,

                razaBile * cos(5 * doublePI / nrPuncte), razaBile * sin(5 * doublePI / nrPuncte), 0.0f,  1.0f,     0.0f, 1.0f, 0.0f,   rTextura *
cos(5 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(5 * doublePI / nrPuncte) + normalizareText,

                razaBile * cos(6 * doublePI / nrPuncte), razaBile * sin(6 * doublePI / nrPuncte), 0.0f,  1.0f,     0.0f, 1.0f, 0.0f,   rTextura *
cos(6 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(6 * doublePI / nrPuncte) + normalizareText,
```

```
		razaBile * cos(7 * doublePI / nrPuncte), razaBile * sin(7 * doublePI / nrPuncte), 0.0f,  1.0f,     0.0f, 1.0f, 0.0f,   rTextura * cos(7 * doublePI / nrPuncte) + normalizareText,    rTextura * sin(7 * doublePI / nrPuncte) + normalizareText,

		razaBile * cos(8 * doublePI / nrPuncte), razaBile * sin(8 * doublePI / nrPuncte), 0.0f,  1.0f,     0.0f, 1.0f, 0.0f,   rTextura * cos(8 * doublePI / nrPuncte) + normalizareText,            rTextura * sin(8 * doublePI / nrPuncte) + normalizareText,

		razaBile * cos(9 * doublePI / nrPuncte), razaBile * sin(9 * doublePI / nrPuncte), 0.0f,  1.0f,     0.0f, 1.0f, 0.0f,    rTextura * cos(9 * doublePI / nrPuncte) + normalizareText,            rTextura * sin(9 * doublePI / nrPuncte) + normalizareText,

		razaBile * cos(10 * doublePI / nrPuncte), razaBile * sin(10 * doublePI / nrPuncte), 0.0f,  1.0f,    0.0f, 1.0f, 0.0f,   rTextura * cos(10 * doublePI / nrPuncte) + normalizareText,   rTextura * sin(10 * doublePI / nrPuncte) + normalizareText,

		razaBile * cos(11 * doublePI / nrPuncte), razaBile * sin(11 * doublePI / nrPuncte), 0.0f,  1.0f,    0.0f, 1.0f, 0.0f,   rTextura * cos(11 * doublePI / nrPuncte) + normalizareText,   rTextura * sin(11 * doublePI / nrPuncte) + normalizareText,

	};


	static const GLuint Indices2[] =

	{

		0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,

	};


	// PRIMUL VAO:

	glGenVertexArrays(1, &VaoId1);

	glBindVertexArray(VaoId1);


	//  Se creeaza un buffer pentru VARFURI - COORDONATE, CULORI si TEXTURARE;

	glGenBuffers(1, &VboId1);
				//  Generarea bufferului si indexarea acestuia catre variabila VboId;

	glBindBuffer(GL_ARRAY_BUFFER, VboId1);
	//  Setarea tipului de buffer - atributele varfurilor;

	glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices1), Vertices1, GL_STATIC_DRAW);


	//		Se creeaza un buffer pentru INDICI;

	glGenBuffers(1, &EboId1);
					//  Generarea bufferului si indexarea acestuia catre variabila EboId;

	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EboId1);
	//  Setarea tipului de buffer - atributele varfurilor;

	glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices1), Indices1, GL_STATIC_DRAW);


	//		Se activeaza lucrul cu atribute;

	//  Se asociaza atributul (0 = coordonate) pentru shader;

	glEnableVertexAttribArray(0);

	glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)0);
```

```
	// Se asociaza atributul (1 = culoare) pentru shader;
	glEnableVertexAttribArray(1);
	glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)(4 * sizeof(GLfloat)));
	// Se asociaza atributul (2 = texturare) pentru shader;
	glEnableVertexAttribArray(2);
	glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)(7 * sizeof(GLfloat)));



	// AL DOILEA VAO:
	glGenVertexArrays(1, &VaoId2);
	glBindVertexArray(VaoId2);


	glGenBuffers(1, &VboId2);
				// Generarea bufferului si indexarea acestuia catre variabila VboId;
	glBindBuffer(GL_ARRAY_BUFFER, VboId2);
	// Setarea tipului de buffer - atributele varfurilor;
	glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices2), Vertices2, GL_STATIC_DRAW);


	glGenBuffers(1, &EboId2);
					// Generarea bufferului si indexarea acestuia catre variabila EboId;
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EboId2);
	// Setarea tipului de buffer - atributele varfurilor;
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices2), Indices2, GL_STATIC_DRAW);


	glEnableVertexAttribArray(0);
	glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)0);
	glEnableVertexAttribArray(1);
	glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)(4 * sizeof(GLfloat)));
	glEnableVertexAttribArray(2);
	glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)(7 * sizeof(GLfloat)));
}


// Elimina obiectele de tip shader dupa rulare;
void DestroyShaders(void)
{
	glDeleteProgram(ProgramId);
}
```

```
void DestroyVBO(void)
{
	// Eliberarea atributelor din shadere (pozitie, culoare, texturare etc.);
	glDisableVertexAttribArray(1);
	glDisableVertexAttribArray(0);

	// Stergerea bufferelor pentru VARFURI(Coordonate + Culori), INDICI;
	glBindBuffer(GL_ARRAY_BUFFER, 0);
	glDeleteBuffers(1, &VboId1);
	glDeleteBuffers(1, &EboId1);
	glDeleteBuffers(1, &VboId2);
	glDeleteBuffers(1, &EboId2);

	// Eliberaea obiectelor de tip VAO;
	glBindVertexArray(0);
	glDeleteVertexArrays(1, &VaoId1);
	glDeleteVertexArrays(1, &VaoId2);
}

// Functia de eliberare a resurselor alocate de program;
void Cleanup(void)
{
	DestroyShaders();
	DestroyVBO();
}

void LoadTextures()
{
	LoadTexture("masa_biliard.png", textureMasa);
	LoadTexture("tac1.png", textureTac);
	for (int i = 0; i < nrBile; i++)
	{
		char numeFisierBila[20];
		sprintf_s(numeFisierBila, sizeof(numeFisierBila), "bila_%d.png", i);
```

```
                LoadTexture(numeFisierBila, textureBile[i]);
        }
}
// Setarea parametrilor necesari pentru fereastra de vizualizare;
void Initialize(void)
{
        glClearColor(0.1f, 0.2f, 0.2f, 1.0f);           // Culoarea de fond a ecranului;
        // Trecerea datelor de randare spre bufferul folosit de shadere;
        CreateVAOs();
        // initializarea pozitiilor elementelor jocului (bile, tac, triunghi):
        InitializeGame();
        // Initializarea shaderelor;
        CreateShaders();
        // Incarcam texturile:
        LoadTextures();
        //      Instantierea variabilelor uniforme pentru a "comunica" cu shaderele;
        myMatrixLocation = glGetUniformLocation(ProgramId, "myMatrix");
        codColLocation = glGetUniformLocation(ProgramId, "codColShader");
        //      Dreptunghiul "decupat";
        resizeMatrix = glm::ortho(xMin, xMax, yMin, yMax);
        // pt triunghi:
        translatieTriunghiOrigine = glm::translate(glm::mat4(1.0f), glm::vec3(-3.65 * razaBile, 0.0, 0.0));
        translatieTriunghiDinOrigine = glm::translate(glm::mat4(1.0f), glm::vec3(3.65 * razaBile, 0.0, 0.0));
}

// Functia de desenarea a graficii pe ecran;
void RenderFunction(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glPointSize(20.0);

        // MASA DE BILIARD:
        glBindVertexArray(VaoId1);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, textureMasa);
```

```
glUniform1i(glGetUniformLocation(ProgramId, "myTexture"), 0);


codCol = 1;
glUniform1i(codColLocation, codCol);
myMatrix = resizeMatrix;
glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_INT, (void*)(0));


// TRIUNGHIUL:
if (jocInceput == false) // triunghiul se afla undeva pe ecran
{
	glBindVertexArray(VaoId1);
	glLineWidth(triunghiLineWidth);
	codCol = 0;
	glUniform1i(codColLocation, codCol);


	scalareTriunghi = glm::scale(glm::mat4(1.0f), glm::vec3(scalareCurentaTriunghi, scalareCurentaTriunghi, 0.0));
	translatieTriunghi = glm::translate(glm::mat4(1.0f), glm::vec3(0.0, translatieCurentaTriunghi, 0.0));
	myMatrix = resizeMatrix * translatieTriunghi * translatieTriunghiDinOrigine * scalareTriunghi * translatieTriunghiOrigine;
	glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
	indexAnterior = 4;
	glDrawElements(GL_LINE_STRIP, 4, GL_UNSIGNED_INT, (void*)(indexAnterior * sizeof(GLuint)));
}


// BILELE:
glBindVertexArray(VaoId2);
codCol = 1;
glUniform1i(codColLocation, codCol);
for (int i = 0; i < nrBile; i++)
{
	if (vBile[i].eliminata == false)
	{
		glActiveTexture(GL_TEXTURE0);
		glBindTexture(GL_TEXTURE_2D, textureBile[i]);
		glUniform1i(glGetUniformLocation(ProgramId, "myTexture"), 0);
```

```
                                if (vBile[i].incepereEliminare == true)
                                        miscorareBilaCareCade = glm::scale(glm::mat4(1.0f),
glm::vec3(vBile[i].scalareCurentaPentruMicsorare, vBile[i].scalareCurentaPentruMicsorare, 0.0));
                                else
                                        miscorareBilaCareCade = glm::scale(glm::mat4(1.0f), glm::vec3(1.0f, 1.0f, 0.0));
                                translatieBila = glm::translate(glm::mat4(1.0f), glm::vec3(vBile[i].x, vBile[i].y, 0.0));
                                myMatrix = resizeMatrix * translatieBila * miscorareBilaCareCade;
                                glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
                                glDrawElements(GL_POLYGON, 12, GL_UNSIGNED_INT, (void*)(0));
                        }
                }

                // TACUL:
                if (lovireBile == true && stopJoc == false) // tacul apare doar daca este randul cuiva sa loveasca bila alba
                {
                        glBindVertexArray(VaoId1);
                        glActiveTexture(GL_TEXTURE0);
                        glBindTexture(GL_TEXTURE_2D, textureTac);
                        glUniform1i(glGetUniformLocation(ProgramId, "myTexture"), 0);

                        codCol = 1;
                        glUniform1i(codColLocation, codCol);
                        translatieTacFataDeBila = glm::translate(glm::mat4(1.0f), glm::vec3(xTac, yTac, 0.0)); // pune tacul in fata bilei albe
                        translatieBilaAlbaInOrigine = glm::translate(glm::mat4(1.0f), glm::vec3(-vBile[0].x, -vBile[0].y, 0.0)); // ca sa putem roti
tacul in jurul bilei albe
                        translatieBilaAlbaInapoi = glm::translate(glm::mat4(1.0f), glm::vec3(vBile[0].x, vBile[0].y, 0.0));
                        rotireTac = glm::rotate(glm::mat4(1.0f), unghiTac, glm::vec3(0.0, 0.0, 1.0));
                        myMatrix = resizeMatrix * translatieBilaAlbaInapoi * rotireTac * translatieBilaAlbaInOrigine * translatieTacFataDeBila;
                        glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);

                        indexAnterior = 8;
                        glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_INT, (void*)(indexAnterior * sizeof(GLuint)));
                }

                glutSwapBuffers();
                glFlush();
```

```
}

//	Punctul de intrare in program, se ruleaza rutina OpenGL;
int main(int argc, char* argv[])
{
	// Se initializeaza GLUT si contextul OpenGL si se configureaza fereastra si modul de afisare;

	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);	//	Modul de afisare al ferestrei, se foloseste un singur buffer de afisare si culori RGB;
	glutInitWindowSize(winWidth, winHeight);	// Dimensiunea ferestrei;
	glutInitWindowPosition(320, 0);	// Pozitia initiala a ferestrei;
	glutCreateWindow("Proiect 2D biliard");	//	Creeaza fereastra de vizualizare, indicand numele acesteia;

	//	Se initializeaza GLEW si se verifica suportul de extensii OpenGL modern disponibile pe sistemul gazda;
	// Trebuie initializat inainte de desenare;

	glewInit();
	GLenum err = glewInit();
	if (err != GLEW_OK) {
		std::cerr << "GLEW init error: " << glewGetErrorString(err) << std::endl;
		return 0;
	}

	Initialize();	// Setarea parametrilor necesari pentru fereastra de vizualizare;
	glutDisplayFunc(RenderFunction);	// Desenarea scenei in fereastra;
	glutKeyboardFunc(ProcessNormalKeys);	//	Functii ce proceseaza inputul de la tastatura utilizatorului;
	glutCloseFunc(Cleanup);	// Eliberarea resurselor alocate de program;

	// Bucla principala de procesare a evenimentelor GLUT (functiile care incep cu glut: glutInit etc.) este pornita;
	// Prelucreaza evenimentele si deseneaza fereastra OpenGL pana cand utilizatorul o inchide;

	glutMainLoop();

	return 0;
```

```
}
```