Proiect 2D – Grafica pe calculator

-Flappy Bird Game-

Lung Alexandra

Grupa 352

Cuprins

1. Conceptul proiectului

Conceptul proiectului este dezvoltarea jocului "Flappy Bird" folosind OpenGL și C++. Acest joc implică controlul unei păsări care trebuie să evite coliziunile cu turnurile verzi, sărind. În funcție de numărul de turnuri depășite, fără a se lovi, se adaugă puncte la scorul final.

2. Descriere

a) Desenarea obiectelor

- background-ul este un dreptunghi pe care este aplicată o textură.
- pasărea (tot un dreptunghi centrat în origine) are aplicată o textură
- turnul (pipe-ul) este un dreptunghi cu centrul în origine

```
// bird's coordonate (actually a square)
//  Coordonate;                              Culori;        Coordonate de texturare;
 bird_xmin, bird_ymin , 0.0f, 1.0f,    1.0f, 1.0f, 1.0f,     0.0f, 0.0f,
 bird_xmax, bird_ymin , 0.0f, 1.0f,    1.0f, 1.0f, 1.0f,     1.0f, 0.0f,
 bird_xmax, bird_ymax , 0.0f, 1.0f,    1.0f, 1.0f, 1.0f,     1.0f, 1.0f,
 bird_xmin, bird_ymax , 0.0f, 1.0f,    1.0f, 1.0f, 1.0f,     0.0f, 1.0f,

//pipe coordonate

// Coordonate;                         Culori;            Coordonate de texturare;
pipe_xmin, pipe_ymin, 0.0f, 1.0f,      0.0f, 1.0f, 0.0f,     0.0f, 0.0f,
pipe_xmax, pipe_ymin, 0.0f, 1.0f,      0.0f, 1.0f, 0.0f,     1.0f, 0.0f,
pipe_xmax, pipe_ymax, 0.0f, 1.0f,      0.0f, 1.0f, 0.0f,     1.0f, 1.0f,
pipe_xmin, pipe_ymax, 0.0f, 1.0f,      0.0f, 1.0f, 0.0f,     0.0f, 1.0f,

//background
xMin, yMin, 0.0f, 1.0f,    0.0f, 1.0f, 0.0f,     0.0f, 0.0f,
xMax, yMin, 0.0f, 1.0f,    0.0f, 1.0f, 0.0f,     1.0f, 0.0f,
xMax, yMax, 0.0f, 1.0f,    0.0f, 1.0f, 0.0f,     1.0f, 1.0f,
xMin, yMax, 0.0f, 1.0f,    0.0f, 1.0f, 0.0f,     0.0f, 1.0f,
```

```
//  Indicii care determina ordinea de parcurgere a varfurilor;
GLuint Indices[] = {
    //indices for bird draw
       0,1,2,3,
    //indices for pipedraw
       4,5,6,7,
    //indices for background
       8,9,10,11,
};
```

Matricile aplicate pentru background:

- resizeMatrix este folosită pentru a aduce scena desenată la dimensiunea standard [-1,1]x[-1,1]

```
void DrawBackground(void) {
   glUseProgram(BackgroundProgramId);
   glBindTexture(GL_TEXTURE_2D, textures[2]);

   myMatrix = resizeMatrix;

   // Transmiterea variabilelor uniforme pentru MATRICEA DE TRANSFORMARE
   glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
   glUniform1f(glGetUniformLocation(BackgroundProgramId, "gametime"), gametime);

   // Draw background
   glDrawElements(GL_TRIANGLE_FAN, 4, GL_UNSIGNED_INT, (void*)(8 *
sizeof(GLuint)));
}
```

Matricile aplicate pentru bird:

- matrRot: înclină pasărea cu unghiul corespunzator între – π/2 și π/2 corespunzator, inițial valoarea unghiului fiind 0.
- matrScale: mărește dimensiunea dreptunghiului cu niște valori constante
- matrTranslate: poziționează pasărea spre stânga pe Ox iar pe Oy la distnața corespunzătoare calculată

```
void DrawBird(void) {
    glUseProgram(BirdProgramId);
    glBindTexture(GL_TEXTURE_2D, textures[0]);
    // Matrici pentru transformari;
    glm::mat4 matrRot = glm::rotate(glm::mat4(1.0f), rotationAngle,
glm::vec3(0.0, 0.0, 1.0));
    glm::mat4 matrScale = glm::scale(glm::mat4(1.f), glm::vec3(80, 80,
1.0));
    glm::mat4 matrTranslate = glm::translate(glm::mat4(1.f), glm::vec3(-300,
birdY, 1.0));

    myMatrix = resizeMatrix * matrTranslate * matrScale * matrRot;

    glUniform1f(glGetUniformLocation(BirdProgramId, "gametime"), gametime);

    // Transmiterea variabilelor uniforme pentru MATRICEA DE TRANSFORMARE
    glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);

    // Draw bird
    glDrawElements(GL_TRIANGLE_FAN, 4, GL_UNSIGNED_INT, 0);
}
```

Matricile aplicate pentru turnuri:

a) Pentru turnul inferior

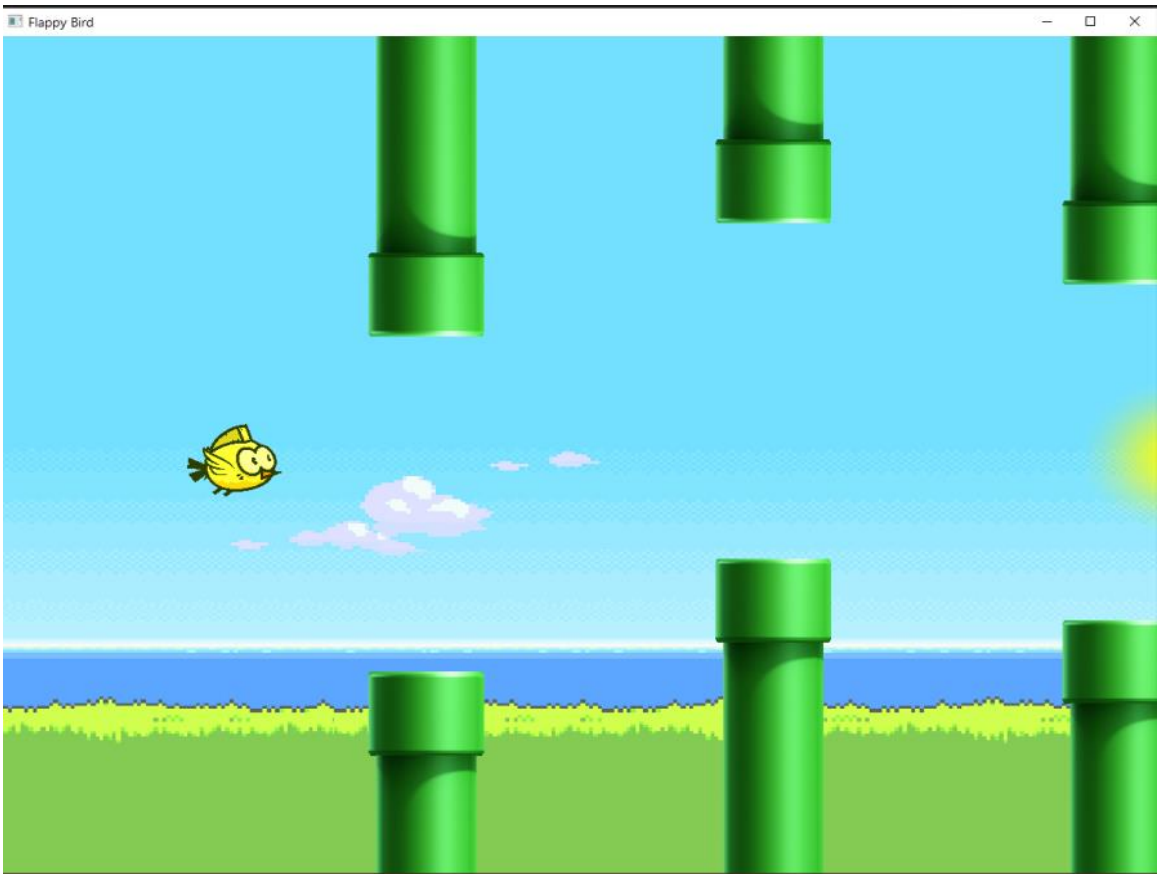
- matrScalePipe: scalează mărește dimensiunea dreptunghiului ce reprezintă turnul
- matrTranslatePipe: mișcă trunul în funcție de coordonata x a turnului (calculata cum se descrie la subpunctul c) și coordonata y egală cu yMin + pipe.y care reprezintă coordonata verticală absolută (poziția reală pe ecran). pipe.y care indică cât de sus sau de jos se află segmentul de conductă în raport cu yMin este calculat cu functia random

b) Pentru turnul superior

- matrScalePipe: scalează mărește dimensiunea dreptunghiului ce reprezintă turnul.Coordonata x este negativă pentru a da flip la textura aplicată.
- matrTranslatePipe: mișcă trunul în funcție de coordonata x a turnului (calculata cum se descrie la subpunctul c) și coordonata y egală cu yMax + pipe.y care reprezintă coordonata verticală absolută (poziția reală pe ecran). pipe.y care indică cât de sus sau de jos se află segmentul de conductă în raport cu yMax este calculat cu functia random
- matrRotPipe: ajută la rotirea dreptunghilui cu 180 de grade

```
void DrawPipeDown(const Pipe&pipe, bool upDown) {
    glUseProgram(PipeProgramId);
    glBindTexture(GL_TEXTURE_2D, textures[1]);

    if (upDown == true) {
       // matrici pentru pipe
       glm::mat4 matrScalePipe = glm::scale(glm::mat4(1.f), glm::vec3(200,
300, 1.0));
       glm::mat4 matrTranslatePipe = glm::translate(glm::mat4(1.f),
          glm::vec3(pipe.x, yMin + pipe.y, 1.0));

       myMatrix = matrTranslatePipe * matrScalePipe;
    }
    else {
       // matrici pentru pipe
       glm::mat4 matrRotPipe = glm::rotate(glm::mat4(1.0f), PI,
glm::vec3(0.0, 0.0, 1.0));
       // scale to a negative value on x to flip horizontal the texture
       glm::mat4 matrScalePipe = glm::scale(glm::mat4(1.f), glm::vec3(-200,
300, 1.0));
       glm::mat4 matrTranslatePipe = glm::translate(glm::mat4(1.f),
          glm::vec3(pipe.x, yMax + pipe.y, 1.0));

       myMatrix =  matrTranslatePipe * matrScalePipe * matrRotPipe;
    }
    // Transmiterea variabilei uniforme pentru TEXTURARE spre shaderul de
fragmente;
```

```
    glUniform1i(glGetUniformLocation(PipeProgramId, "pipeTexture"), 0);

    myMatrix = resizeMatrix * myMatrix;
    // Transmiterea variabilelor uniforme pentru MATRICEA DE TRANSFORMARE
    glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);

    // Draw pipe
    glDrawElements(GL_TRIANGLE_FAN, 4, GL_UNSIGNED_INT, (void*)(4 *
sizeof(GLuint)));

}
```



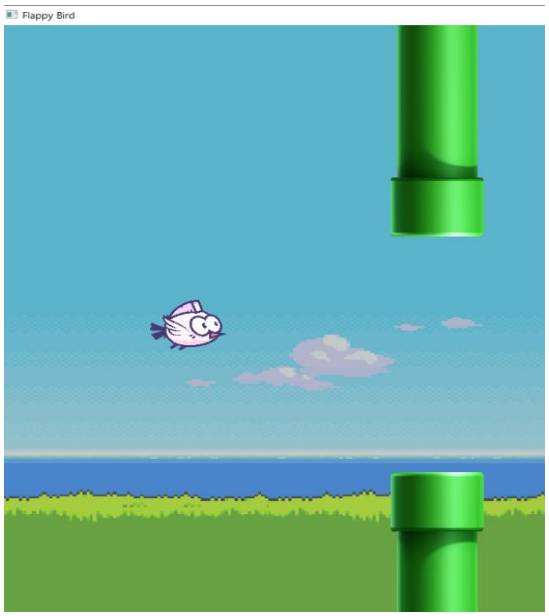
b) Mișcarea păsării

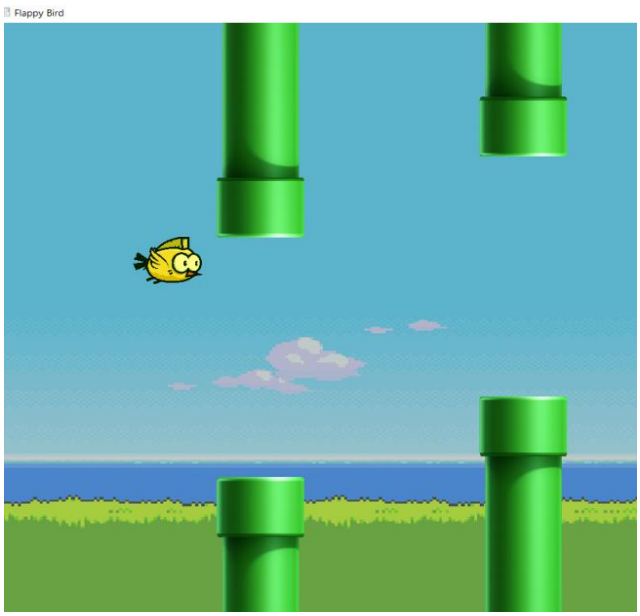
Pasărea se deplasează doar pe Oy. Aceasta este controlată prin apăsarea tastei space în funcția ProcessNormalKey definită astfel:

```
void ProcessNormalKey(unsigned char key, int x, int y) {
   if (key == ' ') {
     MoveUp();
   }
}
```

Funcția MoveUp() este folosită pentru a controla mișcarea în sus a păsării. Initial viteza păsării este 0, fiind stocată în variabila ‘birdVelocity’. Dacă viteza păsării este mai mica sau egală cu 0.5, adaug o valoare constantă de 0.3 ('jump_strength’) pentru a crește viteza, în caz contrat limitez viteza la 0.5.

```
void MoveUp(void) {
    if (birdVelocity > 0.5)
       birdVelocity = 0.5;
    else
       birdVelocity += jump_strength;
}
```



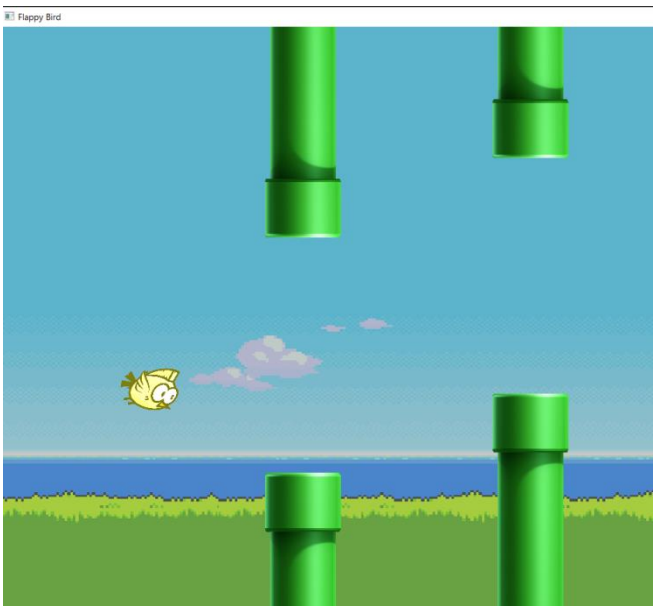


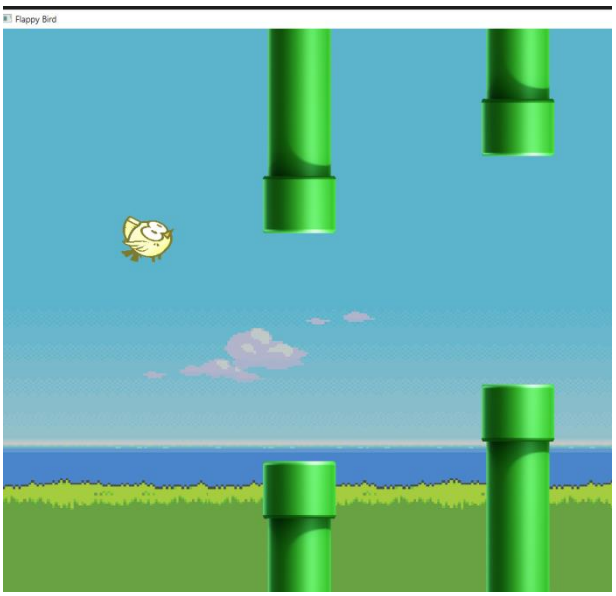
Funcția UpdateBird() este responsabila de actializarea poziției pe Oy a păsării astfel:

- am introdus o variabilă care simuleaza efectul gravitației asupra păsării
- birdY reprezintă poziția pe Oy a păsării
- dacă viteza păsarii devine mai mică de -1, aceasta este limitată la -1, aceasta prevenind căderea prea rapidă a acesteia
- calculez unghiul de înclinare al păsării (care trebuie să ia valori între $-\pi/2$ și $\pi/2$), unghiul fiind calculat astfel încât dacpă pasărea este în mișcare ascendentă să fie înclinată în sus,iar la mișcare descendentă să fie înclinată în jos.

```
void UpdateBird(void) {
    birdVelocity -= gravity;
    if (birdVelocity < -1)
       birdVelocity = -1;
    birdY += birdVelocity;
    rotationAngle = ((birdVelocity + 1) / 1.5f) * PI - PI / 2;
    if (collision()) {
       game_over();
    }
}
```

Dacă are loc o coliziune (pasărea lovește pământul sau se lovește de un turn) jocul se oprește și în consolă este afișat scorul final.





c) Mișcarea turnurilor

Pentru turnuri am creat o structură care va retine pozitia pe axa Ox, poziția cu care sunt deplasate pe Oy și o variabilă booleană care reține dacă turnul respectiv a fost depășit.

```
struct Pipe {
    float x;
    float y;
    bool passed;
};
```

Inițial sunt create 5 turnuri care le reținîntr-un vector distanța dintre ele fiind fixă (300.f)

```
std::vector<Pipe> pipes;
void initialisationPipes() {
    float x = 0;
    for (int i = 0; i < 5; ++i) {
       pipes.push_back(randomPipe(i * pipeOffset));
    }
}
```

Trunurile se deplasează spre stânga cu o viteză constantă dată de delta_t * pipeVelocity. Când un turn are coordonatele în afara spațiului vizibil acesta este șters și altul se adaugă la final.

```
void UpdatePipes(void) {
    BoundingBox bird = getBoundingBoxBird();
    for (Pipe& p : pipes) {
       p.x -= delta_t * pipeVelocity;
   // to verify if a pipe is passed we verify if p.x < bird x position min
       if (p.x < bird.x_left && p.passed != true) {
          p.passed = true;
          UpdateScore();
       }
    }
    if (pipes.empty()) {
       return;
    }
    Pipe leftmost = pipes.front();
    Pipe last = pipes.back();
    if( leftmost.x < xMin - 100){
       pipes.erase(pipes.begin());
       float next_x = last.x + pipeOffset;
       pipes.push_back(randomPipe(next_x));
    }
}
```

3. Originalitate

- Pasărea se deplasează realist rotindu-se în sus atunci când este apăsată tasta space, respectiv în jos atunci când nu mai este apasata tasta.
- Grafica proprie
  - pasărea are o textură diferită de jocul original și, în shader-ul „bird.frag”, se aplică o serie de transformări pe aceasta, culorile texturii fiind influențate de un factor de timp.

```
void main(void){
    vec4 tex_color = texture(birdTexture, tex_Coord);
    if (tex_color.a < 0.1)
     discard;
    out_Color = tex_color + 0.5 * vec4(sin(gametime*3 + 0.15), sin(gametime*3), sin(gametime*3 +
1.5), 0);
}
```

  - background-ul propriu, în funcție de timpul petrecut în joc fundalul este actualizat din shader astfel încât să creeze impresia că este zi, respectiv noapte.

```
void main(){
    vec4 tex_color = texture(backgroundTexture, tex_Coord);
    // Calculate the rotated position of the sun based on game time
    float rotationSpeed = 0.5;
    float rotationAngle = rotationSpeed * gametime;
    float yOffset = sin(rotationAngle)*0.7;
    float xOffset = cos(rotationAngle)*0.75;
    // Draw the sun
    vec2 sunCenter = vec2(0.5 + xOffset, 1 - yOffset);
    float sunRadius = 0.1;
    vec4 sunColor = vec4(1.0, 1.0, 0.0, 1.0);
    //Draw moon
    vec2 moonCenter= vec2(0.5 - xOffset, 1 + yOffset);
    float moonRadius = 0.08;
    vec4 moonColor = vec4(0.8, 0.8, 0.8, 1.0);
    vec2 shiftedMoonCenter = vec2(moonCenter.x - 0.05, moonCenter.y);
    float distanceToSun = length(tex_Coord - sunCenter);
    float distanceToMoon = length(tex_Coord - moonCenter);
    float distanceToShiftedMoon = length(tex_Coord - shiftedMoonCenter);

    float brightness = 1;
    if (sunCenter.y > yOffset){
        brightness = smoothstep(0, sunCenter.y - yOffset, 0.4);
        tex_color *= brightness;
    }
    if (distanceToSun < sunRadius) {
        out_Color = vec4(mix(sunColor, tex_color, smoothstep(0.05, sunRadius,
distanceToSun)));
    }
    else if (distanceToMoon < moonRadius && distanceToShiftedMoon > moonRadius){
            out_Color = vec4(mix(tex_color,moonColor, smoothstep(0.0,
moonRadius,distanceToMoon)));
    }
    else {
        out_Color = tex_color;
    }
}
```

Unghiul de rotație, rotationAngle, este calculat pe baza timpului de joc și vitezei de rotație. Soarele și luna se rotesc în jurul punctului de coordonate (0.5, 1) pe o elipsă cu razele 0.75, respectiv 0.7.Luna este reprezentată ca o semilună obținută astfel: moonCenter reprezintă centrul cercului care ar corespunde lunii, shiftedMoonCenter este calculat ca o versiune ușor deplasată a centrului lunii în stânga cu 0.05 pe axa Ox.

Variabilele distanceToSun, distanceToMoon și distanceToShiftedMoon calculează distanța de la fiecare pixel la centrul soarelui, lunii și a lunii deplasate.Variabila brightness întunecă textura fundalului în funcție de poziția soarelui pe ecran. Pentru a intuneca textura atunci cand soarele apune (se afla sub linia orizontală definită de yOffset) calculez un factor de atenuare cu ajutorul funcței smoothstep pentru a crea o tranziție lină de la textura complet iluminată, la cea întunecată.

Pentru desenarea acestora folosesc funcția smoothstep pentru a realiza o interpolare liniară, apoi functia mix pentru a amesteca culorile sorelui/lunii cu cea a texturii.

4. Cod

main.cpp

```
//  Biblioteci

#include <windows.h>        // Utilizarea functiilor de sistem Windows (crearea de
ferestre, manipularea fisierelor si directoarelor);
#include <stdlib.h>         //  Biblioteci necesare pentru citirea shaderelor;
#include <stdio.h>
#include <GL/glew.h>        //  Defineste 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"        // Biblioteca pentru texturare;
#include <iostream>
#include <vector>
```

```
#include <string>

//  Identificatorii obiectelor de tip OpenGL;
GLuint
VaoId,
VboId,
EboId,
BirdProgramId,
PipeProgramId,
BackgroundProgramId,
myMatrixLocation,
projLocation,
matrRotlLocation,
codColLocation;
GLuint
textures[3];
// Dimensiunile ferestrei de afisare;
GLfloat
winWidth = 1200, winHeight = 900;
// Variabile catre matricile de transformare;
glm::mat4
myMatrix, resizeMatrix;


// Variabila ce determina schimbarea culorii pixelilor in shader;
int codCol;

// Elemente pentru matricea de proiectie;
float  xMin = -500.f, xMax = 500, yMin = -500, yMax = 500;

float PI = 3.141592;

float
deltax = xMax - xMin, deltay = yMax - yMin, // lungimile laturilor dreptunghiului decupat
xcenter = (xMin + xMax) * 0.5, ycenter = (yMin + yMax) * 0.5; // centrul dreptunghiului
decupat

float pipe_xmin = -0.25f, pipe_xmax = 0.25f,
pipe_ymin = -1.0f, pipe_ymax = 1.f;

float pipeVelocity = 150, gametime = 0, delta_t = 0;
float last_time = 0;

float pipeOffset = 300; // the space between the pipes.

// Global variables
float birdY = 0.f; // Initial bird position
float birdVelocity = 0;

float bird_xmin = -0.5f, bird_xmax = 0.5f,
bird_ymin = -0.5f, bird_ymax = 0.5f;
float rotationAngle = 0;


const float gravity = 0.001f;
const float jump_strength = 0.5f;
int score = 0;
int maxScore = 0;

// Defines a bounding box.
```

```
struct BoundingBox {
   float x_left;
   float x_right;
   float y_down;
   float y_up;
};

bool intersect(const BoundingBox& a, const BoundingBox& b) {
   return (a.x_left < b.x_right && a.x_right > b.x_left && a.y_down <b.y_up && a.y_up >
b.y_down);
}

std::ostream& operator<<(std::ostream&o, const BoundingBox& b) {
   o << "(x: " << b.x_left << " - " << b.x_right << " y: " << b.y_down << " - " << b.y_up;
   return o;

}

struct Pipe {
   float x;
   float y;
   bool passed;
};

BoundingBox getBoundingBoxDown(const Pipe& pipe) {
   glm::mat4 matrScalePipe = glm::scale(glm::mat4(1.f), glm::vec3(200, 300, 1.0));
   glm::mat4 matrTranslatePipe = glm::translate(glm::mat4(1.f),
      glm::vec3(pipe.x, yMin + pipe.y, 1.0));
   glm::mat4 model = matrTranslatePipe * matrScalePipe;
   glm::vec4 up_left = (model * glm::vec4(pipe_xmin, pipe_ymax, 0, 1));
   glm::vec4 down_right = (model * glm::vec4(pipe_xmax, pipe_ymin, 0, 1));
   return BoundingBox{ up_left.x, down_right.x, down_right.y, up_left.y };
}

BoundingBox getBoundingBoxUp(const Pipe& pipe) {
   glm::mat4 matrRotPipe = glm::rotate(glm::mat4(1.0f), PI, glm::vec3(0.0, 0.0, 1.0));
   // scale to a negative value on x to flip horizontal the texture
   glm::mat4 matrScalePipe = glm::scale(glm::mat4(1.f), glm::vec3(200, 300, 1.0));
   glm::mat4 matrTranslatePipe = glm::translate(glm::mat4(1.f),
      glm::vec3(pipe.x, yMax + pipe.y, 1.0));
   glm::mat4 model = matrTranslatePipe * matrScalePipe * matrRotPipe;
   glm::vec4 up_left = (model * glm::vec4(pipe_xmax, pipe_ymin, 0, 1));
   glm::vec4 down_right = (model * glm::vec4(pipe_xmin, pipe_ymax, 0, 1));
   return BoundingBox{ up_left.x, down_right.x, down_right.y, up_left.y };
}

BoundingBox getBoundingBoxBird() {
   glm::mat4 matrScaleBird = glm::scale(glm::mat4(1.f), glm::vec3(80, 80, 1.0));
   glm::mat4 matrTranslateBird = glm::translate(glm::mat4(1.f), glm::vec3(-300, birdY,
1.0));
   glm::mat4 model = matrTranslateBird * matrScaleBird;
   glm::vec4 up_left = (model * glm::vec4(bird_xmin, bird_ymax, 0, 1));
   glm::vec4 down_right = (model * glm::vec4(bird_xmax, bird_ymin, 0, 1));
   return BoundingBox{ up_left.x, down_right.x , down_right.y , up_left.y };
}



void UpdateScore() {
   score += 1;
}
```

```
// Generate a random pipe at the given xcoordinate.
// The y coordinate will be randomly chosen between -100, 100
Pipe randomPipe(float xCoord) {
   //float y = rand() % (401) - 200;
   float y = rand() % (201) - 100;
   return Pipe{ xCoord, y, false};
}

std::vector<Pipe> pipes;
void initialisationPipes() {
   float x = 0;
   for (int i = 0; i < 5; ++i) {
      pipes.push_back(randomPipe(i * pipeOffset));
   }
}

void game_over(void) {
   if (score > maxScore) {
      maxScore = score; // Actualizați scorul maxim dacă scorul curent este mai mare
   }
   // Bird hit the ground
   std::cout << "Game Over. Your score: " << score << std::endl;
   score = 0; // Resetați scorul curent
   exit(0);
}

bool collision() {
   // First check pipe collision.
   BoundingBox bird = getBoundingBoxBird();

   /*std::cout << "Front pipe: " << pipes.front().x << " " << pipes.front().y <<std::endl;
   std:: cout << "Down: "<< frontDown << std::endl;
   std::cout << "Up: " << frontUp << std::endl;
   std::cout << "Bird: " << birdY << std::endl;
   std::cout << "Bounding box bird: " <<bird<< std::endl;*/

   for (const Pipe& p : pipes) {
      BoundingBox frontDown = getBoundingBoxDown(p);
      BoundingBox frontUp = getBoundingBoxUp(p);
      if (intersect(bird, frontDown))
         return true;
      if (intersect(bird, frontUp))
         return true;
   }

   // Then check ground collision.
   if (birdY < yMin) {
      return true;
   }
   return false; // Nicio coliziune detectată
}

// Updates the position and velocity of the bird.
// Also, handle collisions.
void UpdateBird(void) {
   birdVelocity -= gravity;
   if (birdVelocity < -1)
      birdVelocity = -1;
   birdY += birdVelocity;
   /*
```

```
       v -> (-1,0.5) => v+1 -> (0,1.5) => (v+1)/1.5 -> (0,1) => (v+1)/1.5 * pi -> (0,pi)
       => (v+1)/1.5*pi - pi/2 => (-pi/2,pi/2)
   */

   rotationAngle = ((birdVelocity + 1) / 1.5f) * PI - PI / 2;
   if (collision()) {
      game_over();
   }

}

// Updates the state of the pipes.
// Each frame, move the pipes to the left side.
// If the first pipe in the list exits the screen, delete it and add another one to the
end.
void UpdatePipes(void) {
   BoundingBox bird = getBoundingBoxBird();
   for (Pipe& p : pipes) {
      p.x -= delta_t * pipeVelocity;
      // to verify if a pipe is passed we verify if p.x < bird x position min
      if (p.x < bird.x_left && p.passed != true) {
         p.passed = true;
         UpdateScore();
      }
   }
   if (pipes.empty()) {
      return;
   }
   Pipe leftmost = pipes.front();
   Pipe last = pipes.back();

   if( leftmost.x < xMin - 100){
      pipes.erase(pipes.begin());

      float next_x = last.x + pipeOffset;
      pipes.push_back(randomPipe(next_x));
   }
}

 void Update(void) {
   //timpul curent in s
   float time = glutGet(GLUT_ELAPSED_TIME) * 0.001f;
   delta_t = time - last_time;
   last_time = time;
   UpdateBird();
   UpdatePipes();

   glutPostRedisplay();
}


void MoveUp(void) {
   if (birdVelocity > 0.5)
      birdVelocity = 0.5;
   else
      birdVelocity += jump_strength;
}


void ProcessNormalKey(unsigned char key, int x, int y) {
   if (key == ' ') {
```

```
      MoveUp();
   }
}


// Functia de incarcare a texturilor in program;
void LoadTexture(const char* photoPath, unsigned int textureId)
{
   // first parameter represents how many textures we want to generate
   glGenTextures(1, &textures[textureId]);
   glBindTexture(GL_TEXTURE_2D, textures[textureId]);
   // Desfasurarea imaginii pe orizonatala/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_REPEAT);

   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

   int width, height;
   unsigned char* image = SOIL_load_image(photoPath, &width, &height, 0, SOIL_LOAD_RGBA);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
image);
   glGenerateMipmap(GL_TEXTURE_2D);

   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)
{
   BirdProgramId = LoadShaders("bird.vert", "bird.frag");
   PipeProgramId = LoadShaders("pipe.vert", "pipe.frag");
   BackgroundProgramId = LoadShaders("background.vert", "background.frag");
}

//  Se initializeaza un Vertex Buffer Object (VBO) pentru tranferul datelor spre memoria
placii grafice (spre shadere);
//  In acesta se stocheaza date despre varfuri (coordonate, culori, indici, texturare
etc.);
void CreateVBO(void)
{

   // Atributele varfurilor -  COORDONATE, CULORI, COORDONATE DE TEXTURARE;
   GLfloat Vertices[] = {
      // bird's coordonate (actually a square)
      // Coordonate;                              Culori;        Coordonate de texturare;
       bird_xmin, bird_ymin , 0.0f, 1.0f,       1.0f, 1.0f, 1.0f,     0.0f, 0.0f,
       bird_xmax, bird_ymin , 0.0f, 1.0f,       1.0f, 1.0f, 1.0f,     1.0f, 0.0f,
       bird_xmax, bird_ymax , 0.0f, 1.0f,       1.0f, 1.0f, 1.0f,     1.0f, 1.0f,
       bird_xmin, bird_ymax , 0.0f, 1.0f,       1.0f, 1.0f, 1.0f,     0.0f, 1.0f,

      //pipe coordonate

      // Coordonate;                      Culori;             Coordonate de texturare;
      pipe_xmin, pipe_ymin, 0.0f, 1.0f,     0.0f, 1.0f, 0.0f,     0.0f, 0.0f,  // stanga
jos
      pipe_xmax, pipe_ymin, 0.0f, 1.0f,     0.0f, 1.0f, 0.0f,     1.0f, 0.0f,  // dreapta
```

```
jos
      pipe_xmax, pipe_ymax, 0.0f, 1.0f,     0.0f, 1.0f, 0.0f,     1.0f, 1.0f,  // dreapta
sus
      pipe_xmin, pipe_ymax, 0.0f, 1.0f,     0.0f, 1.0f, 0.0f,     0.0f, 1.0f,  // stanga
sus

      //background
      xMin, yMin, 0.0f, 1.0f,       0.0f, 1.0f, 0.0f,     0.0f, 0.0f,
      xMax, yMin, 0.0f, 1.0f,       0.0f, 1.0f, 0.0f,     1.0f, 0.0f,
      xMax, yMax, 0.0f, 1.0f,       0.0f, 1.0f, 0.0f,     1.0f, 1.0f,
      xMin, yMax, 0.0f, 1.0f,       0.0f, 1.0f, 0.0f,     0.0f, 1.0f,

   };

   // Indicii care determina ordinea de parcurgere a varfurilor;
   GLuint Indices[] = {
      //indices for bird draw
         0,1,2,3,
      //indices for pipedraw
         4,5,6,7,
      //indices for background
         8,9,10,11,
   };

   //  Transmiterea datelor prin buffere;

   //  Se creeaza / se leaga un VAO (Vertex Array Object) - util cand se utilizeaza mai
multe VBO;
   glGenVertexArrays(1, &VaoId);                                                    //
Generarea VAO si indexarea acestuia catre variabila VaoId;
   glBindVertexArray(VaoId);

   //  Se creeaza un buffer pentru VARFURI - COORDONATE, CULORI si TEXTURARE;
   glGenBuffers(1, &VboId);                                        //  Generarea bufferului
si indexarea acestuia catre variabila VboId;
   glBindBuffer(GL_ARRAY_BUFFER, VboId);                             //  Setarea tipului de
buffer - atributele varfurilor;
   glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices), Vertices, GL_STATIC_DRAW);

   // Se creeaza un buffer pentru INDICI;
   glGenBuffers(1, &EboId);                                          //  Generarea
bufferului si indexarea acestuia catre variabila EboId;
   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EboId);                          //  Setarea
tipului de buffer - atributele varfurilor;
   glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices), Indices, 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)));
}

//  Elimina obiectele de tip shader dupa rulare;
```

```
void DestroyShaders(void)
{
   glDeleteProgram(BirdProgramId);
   glDeleteProgram(PipeProgramId);
   glDeleteProgram(BackgroundProgramId);
}

//  Eliminarea obiectelor de tip VBO dupa rulare;
void DestroyVBO(void)
{
   //  Eliberarea atributelor din shadere (pozitie, culoare, texturare etc.);
   glDisableVertexAttribArray(2);
   glDisableVertexAttribArray(1);
   glDisableVertexAttribArray(0);

   //  Stergerea bufferelor pentru VARFURI (Coordonate, Culori, Textura), INDICI;
   glBindBuffer(GL_ARRAY_BUFFER, 0);
   glDeleteBuffers(1, &VboId);
   glDeleteBuffers(1, &EboId);

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

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

//  Setarea parametrilor necesari pentru fereastra de vizualizare;
void Initialize(void)
{
   glClearColor(1.0f, 1.0f, 1.0f, 1.0f);     //  Culoarea de fond a ecranului;
   CreateShaders();                      //  Initilizarea shaderelor;

   //  Trecerea datelor de randare spre bufferul folosit de shadere;
   CreateVBO();


   // Instantierea variabilelor uniforme pentru a "comunica" cu shaderele;
   myMatrixLocation = glGetUniformLocation(BirdProgramId, "myMatrix");
   myMatrixLocation = glGetUniformLocation(PipeProgramId, "myMatrix");

   resizeMatrix = glm::ortho(xMin, xMax, yMin, yMax);


   // Incarcarea texturii si legarea acesteia cu shaderul;
   glUseProgram(BirdProgramId);
   LoadTexture("bird.png",0);
   glBindTexture(GL_TEXTURE_2D, textures[0]);


   glUseProgram(PipeProgramId);
   LoadTexture("pipe.png",1);
   glBindTexture(GL_TEXTURE_2D, textures[1]);


   glUseProgram(BackgroundProgramId);
```

```
   LoadTexture("background.png", 2);
   glBindTexture(GL_TEXTURE_2D, textures[2]);

}
void DrawBackground(void) {
   glUseProgram(BackgroundProgramId);
   glBindTexture(GL_TEXTURE_2D, textures[2]);

   myMatrix = resizeMatrix;

   // Transmiterea variabilelor uniforme pentru MATRICEA DE TRANSFORMARE
   glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
   glUniform1f(glGetUniformLocation(BackgroundProgramId, "gametime"), gametime);

   // Draw background
   glDrawElements(GL_TRIANGLE_FAN, 4, GL_UNSIGNED_INT, (void*)(8 * sizeof(GLuint)));
}

void DrawBird(void) {
   glUseProgram(BirdProgramId);
   glBindTexture(GL_TEXTURE_2D, textures[0]);
   // Matrici pentru transformari;
   glm::mat4 matrRot = glm::rotate(glm::mat4(1.0f), rotationAngle, glm::vec3(0.0, 0.0,
1.0));
   glm::mat4 matrScale = glm::scale(glm::mat4(1.f), glm::vec3(80, 80, 1.0));
   glm::mat4 matrTranslate = glm::translate(glm::mat4(1.f), glm::vec3(-300, birdY, 1.0));

   myMatrix = resizeMatrix * matrTranslate * matrScale * matrRot;

   // Transmiterea variabilei uniforme pentru TEXTURARE spre shaderul de fragmente;
   glUniform1f(glGetUniformLocation(BirdProgramId, "gametime"), gametime);

   // Transmiterea variabilelor uniforme pentru MATRICEA DE TRANSFORMARE
   glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);

   // Draw bird
   glDrawElements(GL_TRIANGLE_FAN, 4, GL_UNSIGNED_INT, 0);
}

void DrawPipeDown(const Pipe&pipe, bool upDown) {
   glUseProgram(PipeProgramId);
   glBindTexture(GL_TEXTURE_2D, textures[1]);

   if (upDown == true) {
      // matrici pentru pipe
      glm::mat4 matrScalePipe = glm::scale(glm::mat4(1.f), glm::vec3(200, 300, 1.0));
      glm::mat4 matrTranslatePipe = glm::translate(glm::mat4(1.f),
         glm::vec3(pipe.x, yMin + pipe.y, 1.0));

      myMatrix = matrTranslatePipe * matrScalePipe;
   }
   else {
      // matrici pentru pipe
      glm::mat4 matrRotPipe = glm::rotate(glm::mat4(1.0f), PI, glm::vec3(0.0, 0.0, 1.0));
      // scale to a negative value on x to flip horizontal the texture
      glm::mat4 matrScalePipe = glm::scale(glm::mat4(1.f), glm::vec3(-200, 300, 1.0));
      glm::mat4 matrTranslatePipe = glm::translate(glm::mat4(1.f),
         glm::vec3(pipe.x, yMax + pipe.y, 1.0));

      myMatrix =  matrTranslatePipe * matrScalePipe * matrRotPipe;
   }
```

```
   // Transmiterea variabilei uniforme pentru TEXTURARE spre shaderul de fragmente;
   glUniform1i(glGetUniformLocation(PipeProgramId, "pipeTexture"), 0);

   myMatrix = resizeMatrix * myMatrix;
   // Transmiterea variabilelor uniforme pentru MATRICEA DE TRANSFORMARE
   glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);

   // Draw pipe
   glDrawElements(GL_TRIANGLE_FAN, 4, GL_UNSIGNED_INT, (void*)(4 * sizeof(GLuint)));

}


//  Functia de desenarea a graficii pe ecran;
void RenderFunction(void)
{
   glClear(GL_COLOR_BUFFER_BIT);        //  Se curata ecranul OpenGL pentru a fi desenat
noul continut;
   gametime += delta_t;

   DrawBackground();
   DrawBird();

   for (Pipe p : pipes) {
      //second argument represents if we draw the up pipe or down
      // 1 -> up
      // 0 -> down
      DrawPipeDown(p,false);
      DrawPipeDown(p,true);
   }
   glutSwapBuffers(); // Inlocuieste imaginea deseneata in fereastra cu cea randata;
   glFlush();       //  Asigura rularea tuturor comenzilor OpenGL apelate anterior;
}


// 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_DOUBLE | GLUT_RGB);              // Se folosesc 2 buffere
(unul pentru afisare si unul pentru randare => animatii cursive) si culori RGB;
   glutInitWindowSize(winWidth, winHeight);                 //  Dimensiunea ferestrei;
   glutInitWindowPosition(100, 100);                     //  Pozitia initiala a
ferestrei;
   glutCreateWindow("Flappy Bird");      // 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();

   Initialize();
   initialisationPipes();
      //  Setarea parametrilor necesari pentru fereastra de vizualizare;
   glutDisplayFunc(RenderFunction);      //  Desenarea scenei in fereastra;
   glutIdleFunc(Update);
```

```
   // Functii ce proceseaza inputul de la tastatura utilizatorului;
   glutKeyboardFunc(ProcessNormalKey);

   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;
}
```

bird.frag

```
#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 uniforme;
uniform sampler2D birdTexture;
uniform float gametime;

void main(void){

    vec4 tex_color = texture(birdTexture, tex_Coord);
    if (tex_color.a < 0.1)
        discard;
    out_Color = tex_color + 0.5 * vec4(sin(gametime*3 + 0.15), sin(gametime*3),
sin(gametime*3 + 1.5), 0);

}
```

bird.vert

```
//  Shaderul de varfuri / Vertex shader - afecteaza geometria scenei;
//

#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);

}
```

pipe.frag

```
#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 uniforme;
uniform sampler2D pipeTexture;

// Variabile pentru culori;
vec4 background = vec4(0.00, 0.74, 1.00, 1.0);

void main(void){

    vec4 tex_color = texture(pipeTexture, tex_Coord);
    if (tex_color.a < 0.1)
        discard;
    out_Color = tex_color;

}
```

pipe.vert

```
//  Shaderul de varfuri / Vertex shader - afecteaza geometria scenei;
//

#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);

}
```

background.frag

```
#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 uniforme;
uniform sampler2D backgroundTexture;
uniform float gametime;

void main(){

    vec4 tex_color = texture(backgroundTexture, tex_Coord);

    // Calculate the rotated position of the sun based on game time
    float rotationSpeed = 0.9;
    float rotationAngle = rotationSpeed * gametime;

    float yOffset = sin(rotationAngle)*0.7;
    float xOffset = cos(rotationAngle)*0.75;

    // Draw the sun
    vec2 sunCenter = vec2(0.5 + xOffset, 1 - yOffset);
    float sunRadius = 0.1;
    vec4 sunColor = vec4(1.0, 1.0, 0.0, 1.0);

    //Draw moon
    vec2 moonCenter= vec2(0.5 - xOffset, 1 + yOffset);
    float moonRadius = 0.08;
    vec4 moonColor = vec4(0.8, 0.8, 0.8, 1.0);

    vec2 shiftedMoonCenter = vec2(moonCenter.x - 0.05, moonCenter.y);

    float distanceToSun = length(tex_Coord - sunCenter);
    float distanceToMoon = length(tex_Coord - moonCenter);
    float distanceToShiftedMoon = length(tex_Coord - shiftedMoonCenter);

    float brightness = 1;
    if (sunCenter.y > yOffset){
        brightness = smoothstep(0, sunCenter.y - yOffset, 0.4);
        tex_color *= brightness;
    }
```

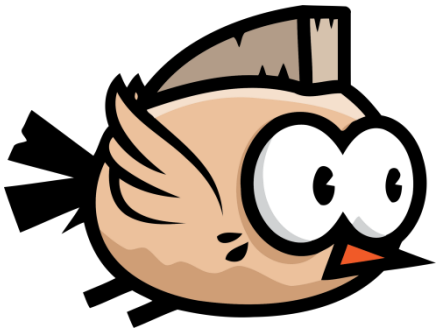
```
    if (distanceToSun < sunRadius) {
        out_Color = vec4(mix(sunColor, tex_color, smoothstep(0.0, sunRadius,
distanceToSun)));
    }
    else if (distanceToMoon < moonRadius && distanceToShiftedMoon > moonRadius){
            out_Color = vec4(mix(tex_color,moonColor, smoothstep(0.0,
moonRadius,distanceToMoon)));
    }
    else {
        out_Color = tex_color;
    }
}
```

background.vert

```
//  Shaderul de varfuri / Vertex shader - afecteaza geometria scenei;
//

#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);

}
```

Imagini folosite:

bird.png



pipe.png

background.png

5. Demo

https://github.com/Lung-Alexandra/opengl-projects/