Documentatie Proiect-1 Grafica pe Calculator

Membri echipa:

- Capatina Razvan Nicolae (grupa 352)
- Luculescu Teodor (grupa 351)

Tema aleasa:

Stol de pasari

Scurta descriere:

Proiectul afiseaza un stol de pasari, fiecare pasare avand in mod aleatoriu generat o traiectorie eliptica pe care se deplaseaza (sens trigonometric sau al acelor de ceasornic, directie aleasa in mod aleatoriu pentru fiecare pasare in parte). Utilizatorul poate lansa pietre inspre stolul de pasari, piatra initial este semi transparenta, iar odata lansata devine opaca si respecta anumita ecuatii care ii definesc scalarea si viteza de deplasare pe axa OY. Pe fundal avem o textura cu un cer, asupra caruia aplicam mix-uri si transparente, incat sa simulam un ciclu zi-noapte.

Transformari folosite:

- Scalarare
- Rotatie
- Translatie

Codul contine un singleton numit Renderer responsabil pentru incarcarea shader-elor si pentru apelul de desenat grafica pe ecran. Renderer-ul, de asemenea, tine stocate niste primitive (un poligon cu 10 laturi folosit pe post de piatra, un patrat care e folosit pentru a desena cerul si o alta figura similara cu o clepsidra orizontala folosita pentru pasari. Aceste 3 figuri geometrice sunt tinute in Renderer centrate in (0, 0) si in coordonate normalizate (au dimensiunea 1x1), astfel incat atunci cand o entitate doreste sa fie desenata trebuie sa apeleze metoda draw() din Renderer si doar sa ofere numele primitivei, numele texturii, scalarea sa, unghiul de rotatie si pozitia sa.

Proiectul include de asemenea alte calcule vectoriale (dot product, testul de orientare, folosite pentru a altera unghiul si traiectoria pe care pasarile se deplaseaza).

Alta formula folosita este ecuatia miscarii constant accelerate.

Originalitate proiect:

Proiectul imbina notiuni din fizica (mecanica newtoniana) pentru a putea simula deplasari ale obiectelor.

Proiectul a fost implementat intr-o maniera Object-Oriented, folosind design patterns, permitand o scalabilitate si alterare usoara a logicii aplicatiei.

Contributii membrii echipa:

Capatina Razvan Nicolae:

- Ciclu zi-noapte
- Texturi
- Rezolvarea ecuatiilor utilizate pentru definirea miscarii constant accelerate
- Implementarea singleton-urilor Renderer si TextureManager
- Utilitarul RandomGenerator folosit pentru generarea unor valori aleatorii folosind o distributie uniforma

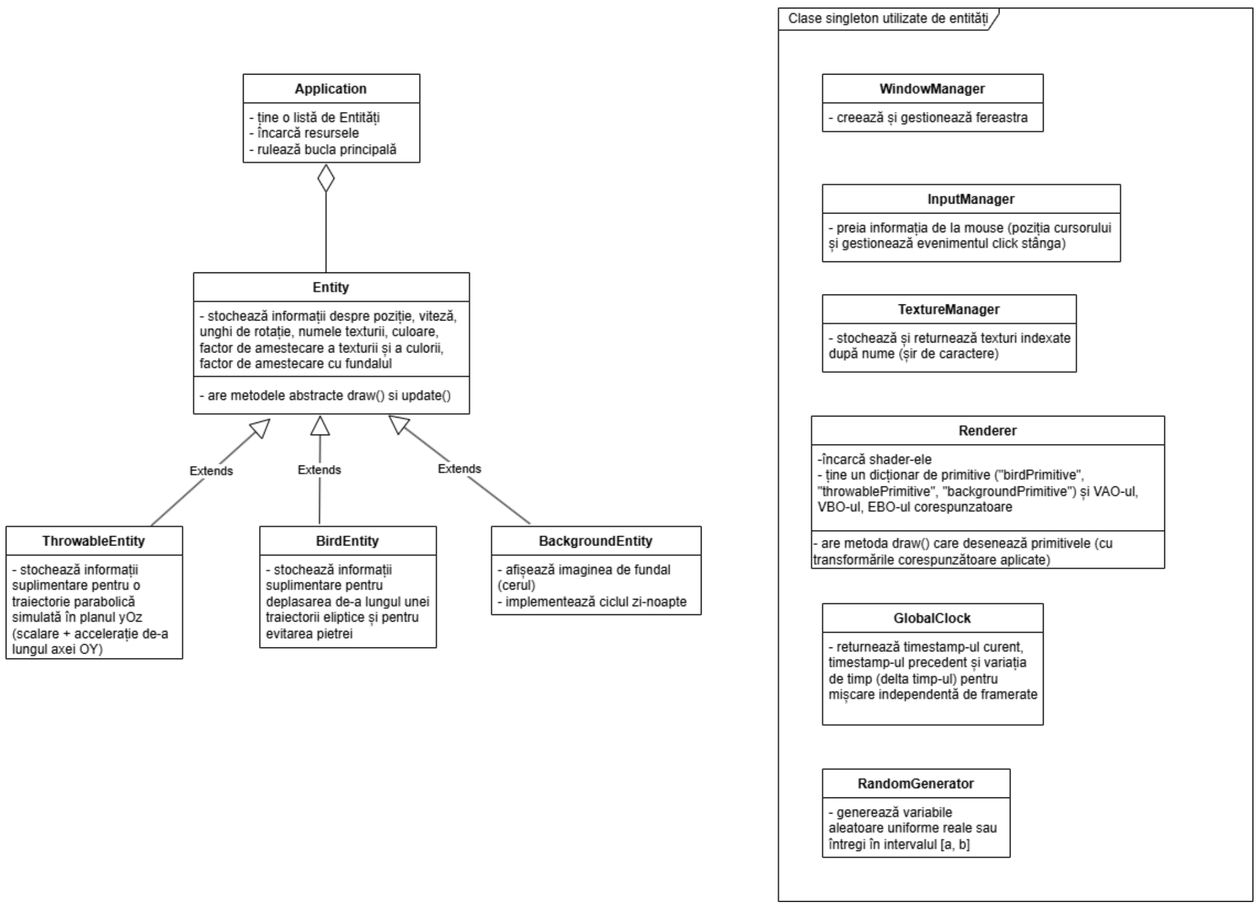
Luculescu Teodor:

- Implementarea ecuatiei eliptice pentru traiectoriile pasarilor
- Implementarea ierarhiei de entitati
- Singleton-ul de GlobalClock, folosit pentru a nu avea efecte/miscari dependente de framerate-ul aplicatiei
- Documentatie

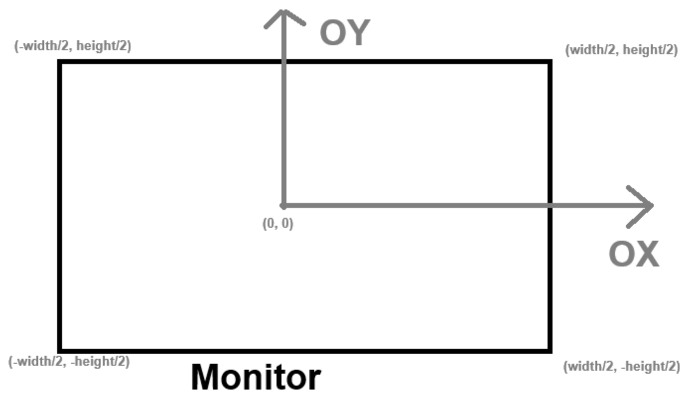
Celelalte implementari au fost realizate sincron si cu o contributie din partea ambilor membri.

Fragmente interesante ale proiectului:

Ierarhia de clase:

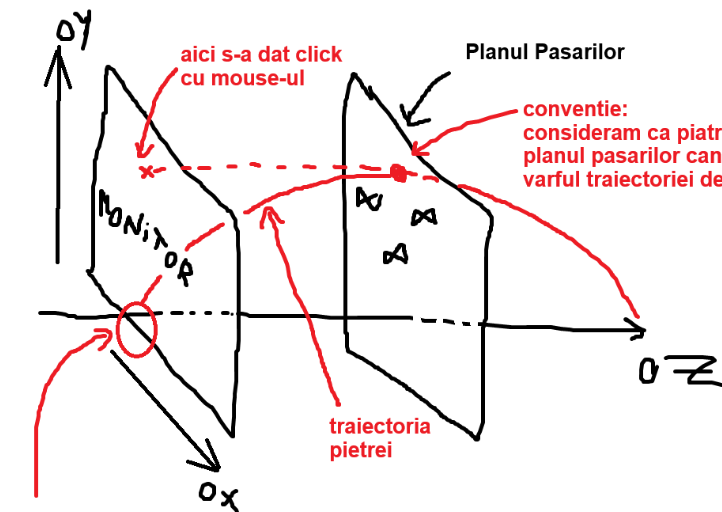


Sistem de coordonate ales:



-folosim o matrice ortogonala centrata in mijlocul ferestrei,
avand dimensiunile width si height

Miscarea pietrei:



conventie: coordonata y unde s-a dat click cu mouse-ul este aceeasi cu y-ul unde se va afla piatra cand intersecteaza planul pasarilor

conventie: o pasare poate fi lovita de piatra doar atunci cand aceasta se afla in planul pasarilor

Presupunem ca piatra se afla la pozitia initiala (atunci cand este aruncata) ($x0$, $y0$).

Conventie: durata intreaga de deplasare a pietrei se cunoaste si o notam cu d.

Daca piatra a fost lansata la momentul $t0 = 0$, atunci stim ca varful parabolei va fi atins pentru momentul de timp t' = d / 2

=> v0 (viteza initiala, cea de lansare) = g * t', unde g este acceleratia gravitationala (piatra are viteza 0 dupa t’ timp, aflandu-se in varful parabolei)

In ecuatia de mai sus necunoscutele sunt v0 si g.

Stim de asemenea ca vrem sa ajunge la coordonata yMouse atunci cand intersectam planul pasarilor (conform ecuatiei miscarii rectilinii si uniform accelerate)

=> yMouse = y0 + v0 * t' - [g * (t')^2] / 2

yMouse se cunoaste, y0 se cunoaste (pozitia initiala a pietrei) => avem cea de a doua ecuatiei cu cele 2 necunoscute v0 si g.

Inlocuim in a doua ecuatie pe v0 cu g * t’ si avem:

yMouse – y0 = g * t’ * t’ - (g * t’ * t’ / 2)

yMouse – y0 = (g * t’ * t’) * (1 / 2)

2 * (yMouse – y0) = g * (d ^ 2 / 4) => aici cunoastem toate variabilele si putem afla pe g

g = 8 * (yMouse – y0) / (d * d)

Si atunci:

v0 = 4 * (yMouse – y0) / d

Daca scoatem g in functie de v0 rezulta:

g = 2 * v0 / d

Scalarea pietrei:

Conventie: presupunem ca piatra are o dimensiune initiala s0 si dorim ca dupa un timp d (durata intreaga de lansare, valoare pe care o stim) dimensiunea pietrei sa devina 0.

Vrem totusi o transformare mai complexa decat una liniara, asa ca ne-am folosit de urmatoarea formula patratica:

sCrt = max(0, 1 - (t / d) * (t / d)) * s0, unde t este timpul scurs de cand s-a lansat piatra, d este durata intreaga a unei lansari, iar sCrt este dimensiunea curenta a pietrei.

In felul acesta dimensiunea devine 0 dupa un timp d, iar transformarea este una parabolica.

Implementarea traiectoriei ThrowableEntity (metoda update()):

```
void ThrowableEntity::update()
```

```
{
if (this->status == ThrowableEntity::Status::IN_HAND)
{
this->posCenterX = InputManager::get().getCurrentMouseX();
```

- Cat timp bila nu e in aer, se deplaseaza mereu pe axa OX la X-ul mouse-ului

```
bool leftMouseButtonUp = InputManager::get().getLeftMouseButtonUp();
if (leftMouseButtonUp)
{
this->targetPosX = InputManager::get().getCurrentMouseX();
this->targetPosY = InputManager::get().getCurrentMouseY();

this->status = ThrowableEntity::Status::IN_AIR;
this->rotationDirection = 2 * RandomGenerator::randomUniformInt(0, 1) - 1;
this->backgroundBlendFactor = 1.0f;
this->launchTime = GlobalClock::get().getCurrentTime();

this->currentInitialLaunchSpeed = 4.0f * (InputManager::get().getCurrentMouseY() -
this->initialPosY) / this->launchDuration;
this->speed.y = this->currentInitialLaunchSpeed;
```

- La eliberarea butonului stanga, i se seteaza o viteza initiala
- Aici se poate observa formula v0 = 4 * (yMouse – y0) / d

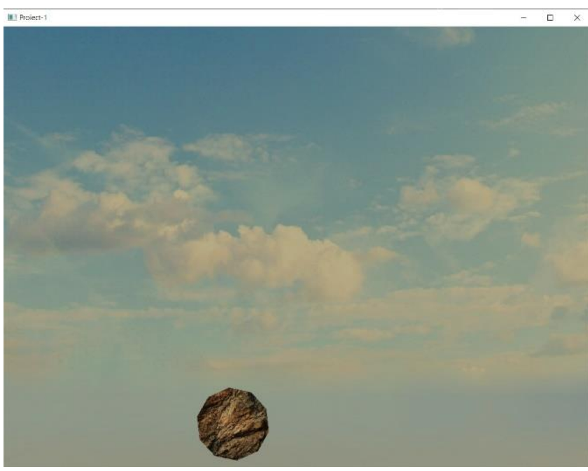
```
}
}
else if (this->status == ThrowableEntity::Status::IN_AIR)
{
if (GlobalClock::get().getCurrentTime() - this->launchTime > this->launchDuration)
{
this->status = ThrowableEntity::Status::IN_HAND;
this->backgroundBlendFactor = 0.5f;
this->posCenterX = this->initialPosX;
this->posCenterY = this->initialPosY;
this->radius = this->initialRadius;

this->currentInitialLaunchSpeed = 0.0f;

this->speed.y = 0.0f;
}
else
{
float timeSinceLaunch = GlobalClock::get().getCurrentTime() - this->launchTime;

this->radius = this->initialRadius *
std::max((1.0f - (timeSinceLaunch / this->launchDuration) * (timeSinceLaunch /
this->launchDuration)), 0.0f);
```

- Cat timp se deplaseaza in aer, raza i se micsoreaza conform parabolei 1 – X^2 (pe intervalul [0, 1])
- Aici se poate observa formula sCrt = s0 * max(1 - (t / d) * (t / d), 0)

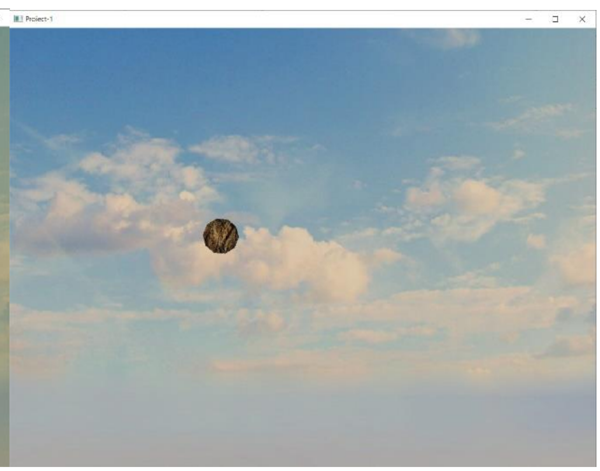
```
this->speed.y -= 2.0f * (this->currentInitialLaunchSpeed / this->launchDuration) *
GlobalClock::get().getDeltaTime();
```

- Asupra vitezei se aplica acceleratia –2 * viteza_initiala / durata_zborului
- Aici se poate observa si formula g = 2 * v0 / d
- GlobalClock::get().getDeltaTime() ne ofera timpul intre frame-ul curent si frame-ul anterior in milisecunde, astfel incat simularile ce au loc in aplicatie sa nu fie dependente de framerate-ul la care ruleaza aplicatia

```
this->posCenterY += this->speed.y * GlobalClock::get().getDeltaTime();

this->rotateAngle = 500.0f * this->rotationDirection *
GlobalClock::get().getCurrentTime();
}
}
else
{
std::cout << "Error: ThrowableEntity: update(): invalid status" << std::endl;
}


this->initialRadius = WindowManager::get().getWindowWidth() / 16.0f;
}
```



t



t + α

Implementarea traiectoriei BirdEntity(metoda update()):

```
void BirdEntity::update()
{
this->currentWidth = this->width;
this->currentHeight = this->height;

if (this->status == BirdEntity::Status::FALLING)
{
this->speed.y -= this->gravity * GlobalClock::get().getDeltaTime();

this->posCenterX += this->speed.x * GlobalClock::get().getDeltaTime();
this->posCenterY += this->speed.y * GlobalClock::get().getDeltaTime();

this->rotateAngle = 100.0f * this->rotationDirectionFalling *
GlobalClock::get().getCurrentTime();
}

if (this->status == BirdEntity::Status::FLYING)
{
if (ThrowableEntity::get().getStatus() == ThrowableEntity::Status::IN_AIR)
{
float timeSinceLaunch = GlobalClock::get().getCurrentTime() -
ThrowableEntity::get().getLaunchTime();

if (std::abs((ThrowableEntity::get().getLaunchDuration() / 2.0f) - timeSinceLaunch)
< ThrowableEntity::get().getLaunchDuration() * this->collisionEpsilon
&& this->isInCollisionWithThrowable())
{
this->status = BirdEntity::Status::FALLING;
// this->speed.y = 0.0f;
// this->speed.x = 0.0f;
}
else // bila e in aer, nu stim unde, dar pasarea nu a fost lovita
{
GLfloat targetPosX = ThrowableEntity::get().getTargetPosX();
GLfloat targetPosY = ThrowableEntity::get().getTargetPosY();

float distanceTargetX = targetPosX - this->posCenterX;
float distanceTargetY = targetPosY - this->posCenterY;
```

- Se construieste vectorul de pozitie a pietrei fata de pasare

```
float crossProductSign = glm::sign(distanceTargetX * this->speed.y - this->speed.x
* distanceTargetY);
```

- Calculam produsul vectorial dintre vectorul de pozitie si viteza pasarii si luam doar semnul rezultatului

```
float scaleSign = -1.0f * crossProductSign * this->rotationDirectionFlying;

// scaleSign = -1 = micsorare elipsa, scaleSign = 1 = marire elipsa
// -1 = micsoram ellipseScale
// 1 = marim ellipseScale
```

```
this->ellipseScale += scaleSign * this->ellipseSpeedScale * this->speedScalar *
GlobalClock::get().getDeltaTime();
```

- In functie de semnul produsului vectorial (adica daca piatra se afla in stanga sau in dreapta pasarii) alegem sa marim sau sa micsoram elipsa pe care se deplaseaza pasarea respectiva (produsul vectorial depinde de sensul in care se deplasa pasarea: trigonometric sau invers-trigonometric)
- Observatie: pasarea isi micsoreaza/mareste elipsa pe care se deplaseaza proportional cu scalarul vitezei de deplasare
- GlobalClock::get().getDeltaTime() ne ofera timpul intre frame-ul curent si frame-ul anterior in milisecunde, astfel incat simularile ce au loc in aplicatie sa nu fie dependente de framerate-ul la care ruleaza aplicatia

```
}
}
else // bila nu e in aer
{
if (this->ellipseScale > 1.0f)
this->ellipseScale = std::max(1.0f, this->ellipseScale - this->ellipseSpeedScale *
this->speedScalar * GlobalClock::get().getDeltaTime());
else
this->ellipseScale = std::min(1.0f, this->ellipseScale + this->ellipseSpeedScale *
this->speedScalar * GlobalClock::get().getDeltaTime());
}
}

if (this->status == BirdEntity::Status::FLYING)
{
this->posCenterX = this->centerXEllipse + this->ellipseScale * this->aEllipse *
glm::cos(glm::radians(this->rotationDirectionFlying * (this->timeOffset + this-
>speedScalar * GlobalClock::get().getCurrentTime())));
this->posCenterY = this->centerYEllipse + this->ellipseScale * this->bEllipse *
glm::sin(glm::radians(this->rotationDirectionFlying * (this->timeOffset + this-
>speedScalar * GlobalClock::get().getCurrentTime())));
```

- Deplasam pasarea pe ecuatia elipsei de centru (centerXEllipse, centerYEllipse), cu razele (aEllipse, bEllipse)

```
float dX = this->posCenterX - this->centerXEllipse;
float dY = this->posCenterY - this->centerYEllipse;

float dXPerpendicular = -dY * this->rotationDirectionFlying + dX * (this-
>ellipseScale - 1.0f);
float dYPerpendicular = dX * this->rotationDirectionFlying + dY * (this-
>ellipseScale - 1.0f);
```

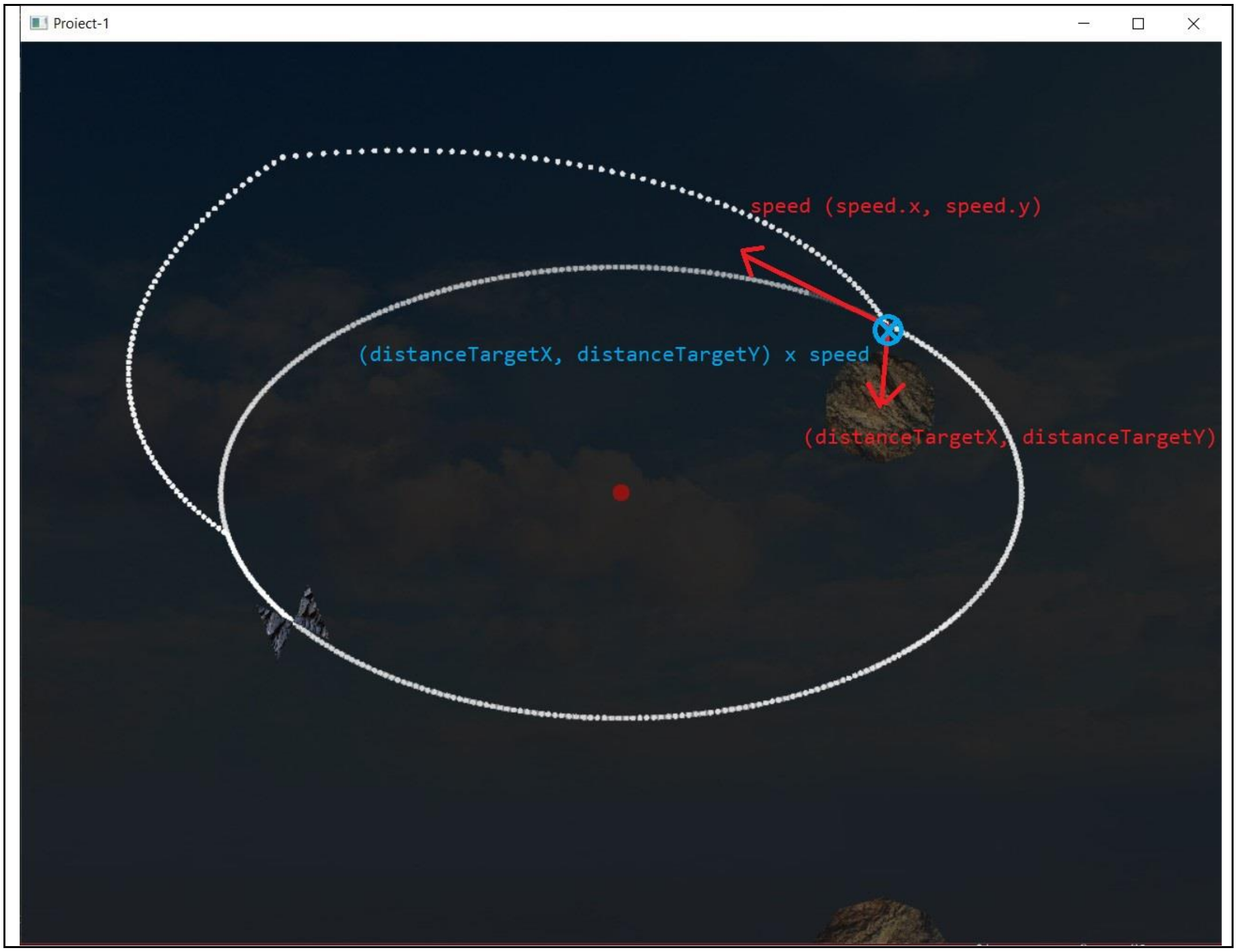
- Calculam vectorul de pozitie a pasarii fata de centrul elipsei si vectorul tangent la elipsa (perpendicular pe vectorul de pozitie), luand in considerare si sensul de rotatie (luarea in considerare a sensului de rotatie se face inmultind componentele vectorului perpendicular cu variabila rotationDirectionFlying (care are ca domeniu de definitie doar –1 sau 1), deoarece un vector (a, b) are 2 perpendiculare: (b, -a) si (-b, a) si trebuie sa o alegem pe cea cu acelasi sens ca directia de deplasare.

- dX * (ellipseScale – 1.0) si dY * (ellipseScale – 1.0) sunt folosite pentru a lua in calcul si daca in acel moment de timp pasarea este in proces de micsorare/marire a elipsei pe care se deplaseaza pentru a evita o lovitura din partea pietrei. Aceste 2 valori se adauga la vectorul de deplasare, pentru ca ele reprezinta o compunere de viteze intre viteza tangenta pe elipsa si cea de apropiere/departare de centrul elipsei.

```
float normPerpendicular = glm::sqrt(dXPerpendicular * dXPerpendicular +
dYPerpendicular * dYPerpendicular);
float dXNormalizedPerpendicular = dXPerpendicular / normPerpendicular;
float dYNormalizedPerpendicular = dYPerpendicular / normPerpendicular;

this->rotateAngle = (glm::degrees((glm::acos(1.0f * dXNormalizedPerpendicular +
0.0f * dYNormalizedPerpendicular)))) * glm::sign(dYNormalizedPerpendicular) +
90.0f;
```

- Normalizam (ca sa nu se altereze si modulul vitezei de-a lungul elipsei) si gasim unghiul de rotatie a pasarii in juril propriului centru astfel incat sa priveasca mereu inainte, pe directia de deplasare pe elipsa. Apoi realizam produsul scalar intre directia normalizata si versorul (1, 0) (pe axa OX), astfel incat sa aflam cosinusul unghiului dintre axa OX si directia de deplasare.

```
this->speed.x = this->speedScalar * dXNormalizedPerpendicular;
this->speed.y = this->speedScalar * dYNormalizedPerpendicular;

this->currentWidth = this->width + 20.0f * glm::sin(glm::radians(10.0f * this-
>timeOffset + 10.0f * this->speedScalar * GlobalClock::get().getCurrentTime())); //
INFO: valori hardcodate aici
}

if (GlobalClock::get().getCurrentTime() - this->lastTimeAddedStoredPosition > this-
>timeBetweenStoredPositions)
{
this->storedPositions[totalNumPositions++] = glm::vec2(this->posCenterX, this-
>posCenterY);
if (this->storedPositions.size() > this->MAX_STORED_POSITIONS)
this->storedPositions.erase(this->storedPositions.begin());

this->lastTimeAddedStoredPosition = GlobalClock::get().getCurrentTime();
}
}
```



Cod Shader-e:

Vertex Shader

```
#version 330 core

layout(location = 0) in vec2 vertexCoord;
layout(location = 1) in vec2 inTextureCoord;

uniform mat4 transformationMatrix;

out vec2 textureCoord;

void main()
{
    textureCoord = vec2(inTextureCoord.x, 1.0f - inTextureCoord.y);
// Flip pe axa OY
```

```
    gl_Position = transformationMatrix * vec4(vertexCoord, 0.0f,
1.0f);
}
```

Fragment Shader

```
#version 330 core

in vec2 textureCoord;

uniform sampler2D textureSampler2D;
uniform vec4 color;
uniform float textureBlendFactor;
uniform float backgroundBlendFactor;

out vec4 fragmentColor;

void main()
{
    vec4 textureColor = texture(textureSampler2D, textureCoord);

    if (textureColor.a == 0.0f) // canalul alpha rgba
discard;

    fragmentColor = mix(textureColor, color, textureBlendFactor);
    fragmentColor.a = backgroundBlendFactor;
}
```

Singleton-ul Renderer:

Renderer.h:

```
#pragma once

#include <string>
#include <vector>
#include <map>
#include <tuple>
```

```
#include <GL/glew.h>
#include <GL/freeglut.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

class Renderer
{
private:
Renderer();
~Renderer();
Renderer(const Renderer& other) = delete;
Renderer& operator= (const Renderer& other) = delete;
Renderer(const Renderer&& other) = delete;
Renderer& operator= (const Renderer&& other) = delete;

const std::string VERTEX_SHADER_PATH;
const std::string FRAGMENT_SHADER_PATH;

std::string readShader(const std::string& shaderPath);
const std::string vertexShaderSource;
const std::string fragmentShaderSource;

GLuint shaderProgram;

std::map<std::string, std::tuple<GLuint, GLuint, GLuint, std::vector<GLfloat>,
std::vector<GLuint>>> primitives;

GLint transformationMatrixLocation;
GLint textureSampler2DLocation;
GLint colorLocation;
GLint textureBlendFactorLocation;
GLint backgroundBlendFactorLocation;

void populateBirdPrimitive(std::vector<GLfloat>& coordinates, std::vector<GLuint>&
indices);
const int NUM_EDGES_THROWABLE_PRIMITIVE;
void populateThrowablePrimitive(std::vector<GLfloat>& coordinates,
std::vector<GLuint>& indices);
void populateBackgroundPrimitive(std::vector<GLfloat>& coordinates,
std::vector<GLuint>& indices);
```

```
void generatePrimitiveBuffers(const std::string& primitiveName);

public:
static Renderer& get();

void draw(GLfloat posCenterX, GLfloat posCenterY, float width, float height, GLfloat
rotateAngle, const std::string& primitiveName, const std::string& textureName2D,
glm::vec3 color, float textureBlendFactor, float backgroundBlendFactor);

inline const std::string& getVertexShaderPath() const { return this-
>VERTEX_SHADER_PATH; }
inline const std::string& getFragmentShaderPath() const { return this-
>FRAGMENT_SHADER_PATH; }

void releaseResources();
};
```

Renderer.cpp:

```
#include "Renderer.h"

#include <iostream>
#include <fstream>
#include <sstream>

#include "../WindowManager/WindowManager.h"
#include "../TextureManager/TextureManager.h"

Renderer::Renderer()
: VERTEX_SHADER_PATH("Shaders/vertexShader.txt"),
FRAGMENT_SHADER_PATH("Shaders/fragmentShader.txt")
, vertexShaderSource(this->readShader(VERTEX_SHADER_PATH)),
fragmentShaderSource(this->readShader(FRAGMENT_SHADER_PATH))
, NUM_EDGES_THROWABLE_PRIMITIVE(10)
{
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);

const char* vertexShaderSource = this->vertexShaderSource.c_str();
glShaderSource(vertexShader, 1, &vertexShaderSource, nullptr);
glCompileShader(vertexShader);

const char* fragmentShaderSource = this->fragmentShaderSource.c_str();
```

```
glShaderSource(fragmentShader, 1, &fragmentShaderSource, nullptr);
glCompileShader(fragmentShader);

this->shaderProgram = glCreateProgram();
glAttachShader(this->shaderProgram, vertexShader);
glAttachShader(this->shaderProgram, fragmentShader);
glLinkProgram(this->shaderProgram);

glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);

glUseProgram(this->shaderProgram);

this->transformationMatrixLocation = glGetUniformLocation(this->shaderProgram,
"transformationMatrix");
this->textureSampler2DLocation = glGetUniformLocation(this->shaderProgram,
"textureSampler2D");
// glActiveTexture(GL_TEXTURE0); // este scrisa in metoda draw()
glUniform1i(this->textureSampler2DLocation, 0);
this->colorLocation = glGetUniformLocation(this->shaderProgram, "color");
this->textureBlendFactorLocation = glGetUniformLocation(this->shaderProgram,
"textureBlendFactor");
this->backgroundBlendFactorLocation = glGetUniformLocation(this->shaderProgram,
"backgroundBlendFactor");

this->generatePrimitiveBuffers("birdPrimitive");
this->generatePrimitiveBuffers("throwablePrimitive");
this->generatePrimitiveBuffers("backgroundPrimitive");

// Teoretic ar fi enabled pt fiecare apel in parte, apoi glDisable(GL_BLEND);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
}

void Renderer::populateBirdPrimitive(std::vector<GLfloat>& coordinates,
std::vector<GLuint>& indices)
{
coordinates.clear();
indices.clear();

coordinates.push_back(0.0f);
coordinates.push_back(0.0f);
coordinates.push_back(0.5f);
coordinates.push_back(0.5f);
```

```
coordinates.push_back(-0.5f);
coordinates.push_back(-0.5f);
coordinates.push_back(0.0f);
coordinates.push_back(0.0f);

coordinates.push_back(-0.5f);
coordinates.push_back(0.5f);
coordinates.push_back(0.0f);
coordinates.push_back(1.0f);

coordinates.push_back(0.5f);
coordinates.push_back(-0.5f);
coordinates.push_back(1.0f);
coordinates.push_back(0.0f);

coordinates.push_back(0.5f);
coordinates.push_back(0.5f);
coordinates.push_back(1.0f);
coordinates.push_back(1.0f);

indices.push_back(0);
indices.push_back(1);
indices.push_back(2);

indices.push_back(0);
indices.push_back(4);
indices.push_back(3);
}

void Renderer::populateThrowablePrimitive(std::vector<GLfloat>& coordinates,
std::vector<GLuint>& indices)
{
coordinates.clear();
indices.clear();

coordinates.push_back(0.0f);
coordinates.push_back(0.0f);
coordinates.push_back(0.5f);
coordinates.push_back(0.5f);

coordinates.push_back(0.5f);
coordinates.push_back(0.0f);
coordinates.push_back(1.0f);
```

```
coordinates.push_back(0.5f);

for (int i = 1; i <= this->NUM_EDGES_THROWABLE_PRIMITIVE; i++)
{
GLfloat angle = 2.0f * glm::pi<float>() * i / this->NUM_EDGES_THROWABLE_PRIMITIVE;

coordinates.push_back(0.5f * glm::cos(angle));
coordinates.push_back(0.5f * glm::sin(angle));
coordinates.push_back(0.5f + 0.5f * glm::cos(angle));
coordinates.push_back(0.5f + 0.5f * glm::sin(angle));

int indexCurent = coordinates.size() / 4 - 1;

indices.push_back(0);
indices.push_back(indexCurent - 1);
indices.push_back(indexCurent);
}
}

void Renderer::populateBackgroundPrimitive(std::vector<GLfloat>& coordinates,
std::vector<GLuint>& indices)
{
coordinates.clear();
indices.clear();

coordinates.push_back(-0.5f);
coordinates.push_back(-0.5f);
coordinates.push_back(0.0f);
coordinates.push_back(0.0f);

coordinates.push_back(0.5f);
coordinates.push_back(-0.5f);
coordinates.push_back(1.0f);
coordinates.push_back(0.0f);

coordinates.push_back(-0.5f);
coordinates.push_back(0.5f);
coordinates.push_back(0.0f);
coordinates.push_back(1.0f);

coordinates.push_back(0.5f);
coordinates.push_back(0.5f);
coordinates.push_back(1.0f);
coordinates.push_back(1.0f);
```

```
indices.push_back(0);
indices.push_back(1);
indices.push_back(2);

indices.push_back(2);
indices.push_back(1);
indices.push_back(3);
}

void Renderer::generatePrimitiveBuffers(const std::string& primitiveName)
{
if (this->primitives.find(primitiveName) != this->primitives.end())
{
std::cout << "Warning: Primitive with name " << primitiveName << " already exists" <<
std::endl;
}

this->primitives[primitiveName] = std::tuple<GLuint, GLuint, GLuint,
std::vector<GLfloat>, std::vector<GLuint>>();
GLuint* VAO = &std::get<0>(this->primitives[primitiveName]);
GLuint* VBO = &std::get<1>(this->primitives[primitiveName]);
GLuint* EBO = &std::get<2>(this->primitives[primitiveName]);
std::vector<GLfloat>* coordinates = &std::get<3>(this->primitives[primitiveName]);
std::vector<GLuint>* indices = &std::get<4>(this->primitives[primitiveName]);

if (primitiveName == "birdPrimitive")
{
this->populateBirdPrimitive(*coordinates, *indices);
}
else if (primitiveName == "throwablePrimitive")
{
this->populateThrowablePrimitive(*coordinates, *indices);
}
else if (primitiveName == "backgroundPrimitive")
{
this->populateBackgroundPrimitive(*coordinates, *indices);
}
else
{
std::cout << "Error: Primitive with name " << primitiveName << " not found" << std::endl;
return;
}
```

```
glGenVertexArrays(1, VAO);
glBindVertexArray(*VAO);

glGenBuffers(1, VBO);
glBindBuffer(GL_ARRAY_BUFFER, *VBO);
glBufferData(GL_ARRAY_BUFFER, coordinates->size() * sizeof(GLfloat), coordinates-
>data(), GL_STATIC_DRAW);

glGenBuffers(1, EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, *EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices->size() * sizeof(GLuint), indices-
>data(), GL_STATIC_DRAW);

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

glBindVertexArray(0);
}

Renderer::~Renderer()
{

}

void Renderer::releaseResources()
{
glDisableVertexAttribArray(1);
glDisableVertexAttribArray(0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);

for (auto& primitive : this->primitives)
{
glDeleteBuffers(1, &std::get<2>(primitive.second));
glDeleteBuffers(1, &std::get<1>(primitive.second));
glDeleteVertexArrays(1, &std::get<0>(primitive.second));
}

glUseProgram(0);
```

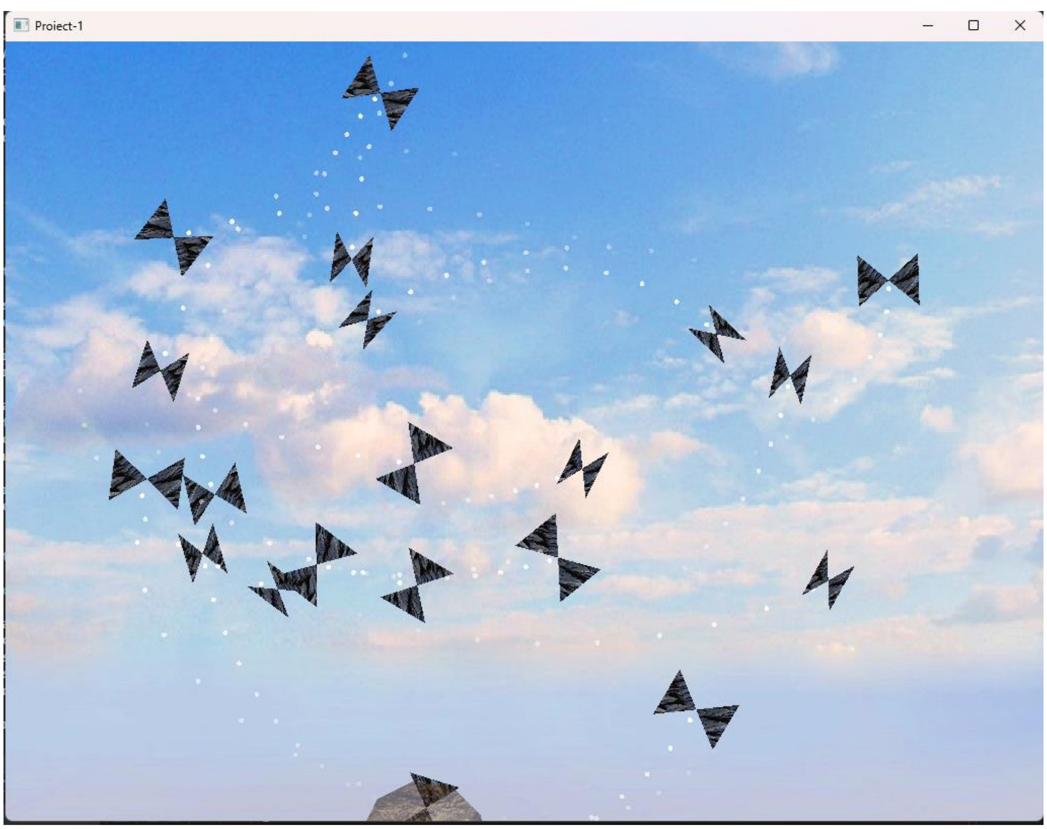
```
glDeleteProgram(this->shaderProgram);
}

Renderer& Renderer::get()
{
static Renderer instance;
return instance;
}

void Renderer::draw(GLfloat posCenterX, GLfloat posCenterY, float width, float height, GLfloat rotateAngle, const std::string& primitiveName, const std::string& textureName2D, glm::vec3 color, float textureBlendFactor, float backgroundBlendFactor)
{
if (this->primitives.find(primitiveName) == this->primitives.end())
{
std::cout << "Error: Primitive with name " << primitiveName << " not found" << std::endl;
return;
}

auto& primitive = this->primitives[primitiveName];

glActiveTexture(GL_TEXTURE0); // Din cauza ca shader-ul contine o singura textura, am putea sa mutam aceasta linie in constructor-ul clasei
glBindTexture(GL_TEXTURE_2D, TextureManager::get().getTexture(textureName2D));

// glUseProgram(this->shaderProgram); // Nu schimbam shader-ul, deoarece avem doar unul

glm::mat4 transformationMatrix =
glm::ortho(0.0f, (GLfloat)WindowManager::get().getWindowWidth(), 0.0f, (GLfloat)WindowManager::get().getWindowHeight())
* glm::translate(glm::mat4(1.0f), glm::vec3(posCenterX, posCenterY, 0.0f))
* glm::rotate(glm::mat4(1.0f), glm::radians(rotateAngle), glm::vec3(0.0f, 0.0f, 1.0f))
* glm::scale(glm::mat4(1.0f), glm::vec3(width, height, 1.0f));

glUniformMatrix4fv(this->transformationMatrixLocation, 1, GL_FALSE, glm::value_ptr(transformationMatrix));

glUniform4f(this->colorLocation, color.x, color.y, color.z, 1.0f);
glUniform1f(this->textureBlendFactorLocation, textureBlendFactor);
glUniform1f(this->backgroundBlendFactorLocation, backgroundBlendFactor);

glBindVertexArray(std::get<0>(primitive));
```

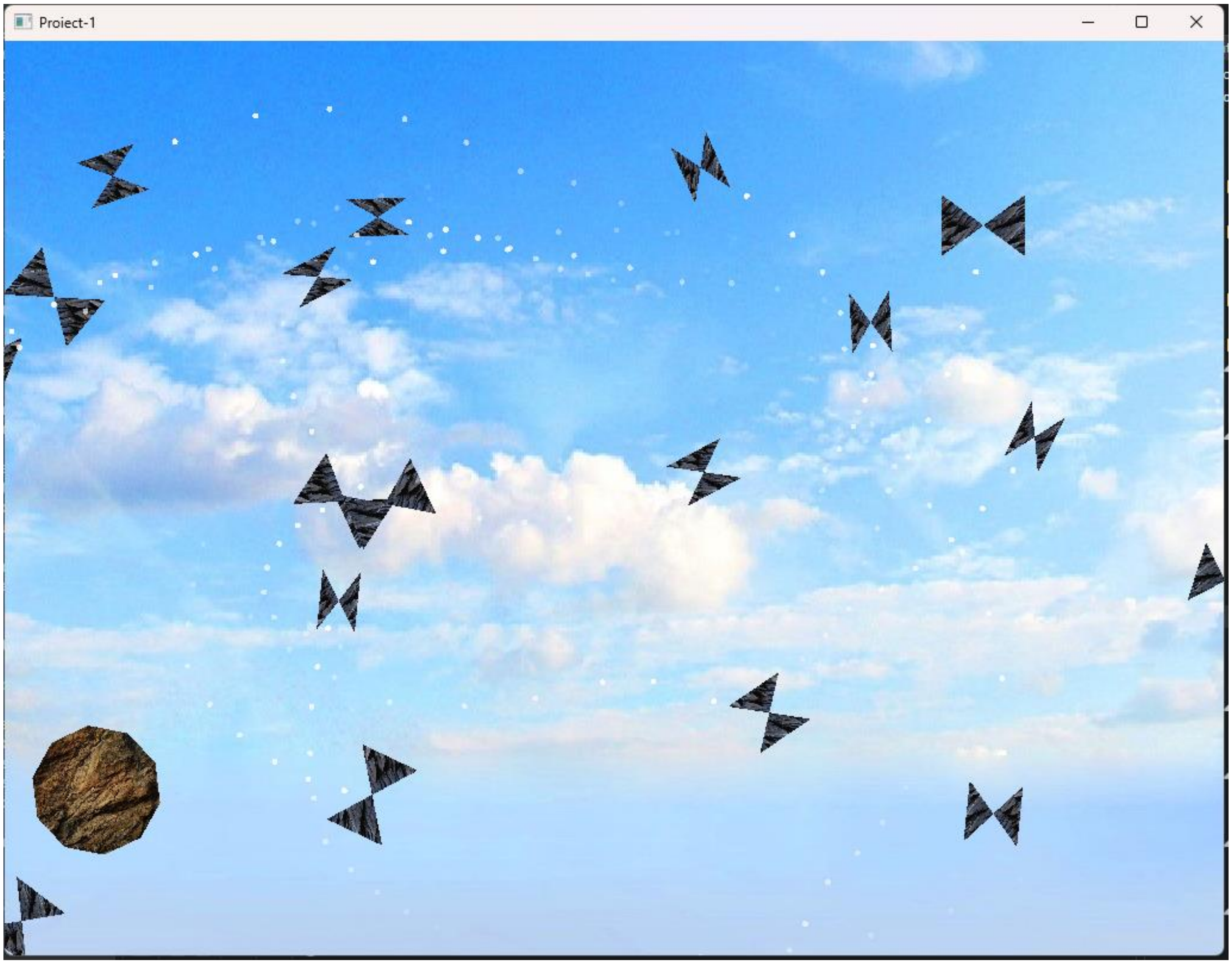
```
glDrawElements(GL_TRIANGLES, std::get<4>(primitive).size(), GL_UNSIGNED_INT, 0);
glBindVertexArray(0);

glBindTexture(GL_TEXTURE_2D, 0);
}

std::string Renderer::readShader(const std::string& shaderPath)
{
std::ifstream inputFileStream(shaderPath);
if (!inputFileStream.is_open())
{
std::cout << "Error: Could not open shader file from: " << shaderPath << std::endl;
}

std::stringstream stringStream;
stringStream << inputFileStream.rdbuf();

// std::cout << "Shader file from " << shaderPath << " content: " << stringStream.str() <<
std::endl;

return stringStream.str();
}
```

Link Repository GitHub:

Razvan48/Proiect-1-Grafica-pe-Calculator-GC: Proiect 1 Grafica pe Calculator (GC) Anul 3, Semestrul 1, Facultatea de Matematica si Informatica, Universitatea din Bucuresti

Exemple de rulare:





Resurse utilizate:

- https://learnopengl.com/
- https://en.wikipedia.org/wiki/Dot_product
- https://en.wikipedia.org/wiki/Cross_product
- https://lectii-virtuale.ro/teorie/miscarea-rectilinie-uniform-variata-ecuatia-de-miscare-si-relatia-galilei