Grafică pe calculator – Proiect 2D

Stol de păsări – Simulare 2D

Popovici Antonia-Adelina

Grupa 331

Introducere

Prezentarea proiectului

Proiectul își propune să simuleze comportamentul unui stol de păsări într-o scenă 2D. Păsările se mișcă într-o anumită direcție, păstrând o anumită distanță între ele. Utilizatorul controlează stolul de păsări cu ajutorul tastaturii (WASD). Bazat pe conceptul *Boids*, proiectul explorează interacțiunile dintre păsări pentru a recrea modele complexe de comportament colectiv.

Teoria *Boids*

Termenul *Boids* este un joc de cuvinte derivat din *birds*. A fost folosit pentru prima dată în contextul informaticii grafice și al simulărilor pentru a descrie modelele de comportament colectiv al stolurilor de păsări. Acest concept a fost introdus de Craig Reynolds în anii 1980 pentru a explora comportamentul emergent al unui grup de entități simple, cum ar fi păsările, care interacționează într-un mediu comun.

Reguli pentru simularea interacțiunilor între păsări

- **Separare**: evitarea vecinilor (nu există conflicte între vecini)
- **Aliniere**: alinierea cu orientarea medie a grupului local (traiectoria principală a stolului)
- **Coeziune**: stolul se deplasează către poziția medie a grupului local (rămâne compact)

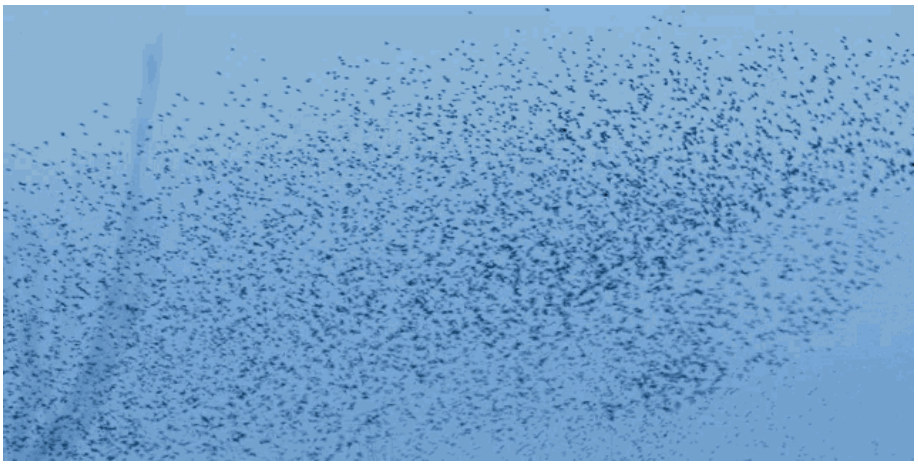

Figura 1. Exemplu de comportament al unui stol de păsări - *https://tenor.com/view/starlings-starling-birds-bird-flock-of-birds-gif-20940639*

Elemente de originalitate

Copac format din trunchi (triunghi) și coroană (hexagon)

Trunchiul, definit în funcția `trunkVBO`, este format dintr-o serie de coordonate ale vârfurilor și un vector de tipuri de obiecte pentru a stabili culoarea (negru). Utilizând o matrice de translație, am plasat trunchiul în stânga scenei.

Coroana copacului, implementată în funcția `crownVBO`, este reprezentată printr-un hexagon cu `n` vârfuri. Am calculat coordonatele coroanei în jurul unui punct specific `(0x, 0y, 0.0f)`. I-am atribuit culoarea roșie cu ajutorul vectorului de obiecte. Indicele `indices` este utilizat pentru a defini ordinea de desenare a triunghiurilor care compun coroana. Aceasta a fost, de asemenea, plasată la stânga scenei folosind o matrice de translație.

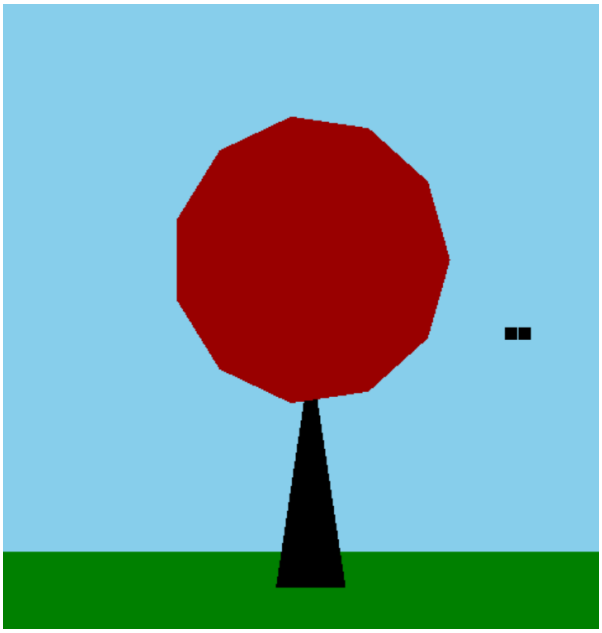

Figura 2. Copac format din triunghi și hexagon

Controlul păsărilor cu ajutorul tastaturii[1]

Păsările, reprezentate de două triunghiuri lipte între ele printr-o muchie comună, zboară într-o anumită direcție. Am folosit două funcții pentru controlul stolului de la tastatură: `keyboardP` și `keyboardR`.

[1] Menționez că forma și deplasarea păsărilor au fost modificate în urma discuțiilor din cadrul laboratorului.

Prima funcție reacționează la apăsarea tastelor, setând variabile booleane în funcție de tasta apăsată ('d', 'a', 'w', 's') și incluzând opțiunea de a încheia programul cu tasta 'q'. A doua funcție resetează variabilele corespunzătoare atunci când tastele sunt eliberate. Aceste funcții ajută la controlul direcției stolului de păsări într-un mod dinamic, adăugând un nivel de interactivitate simulării. Păsările se rotesc în direcția tastelor.

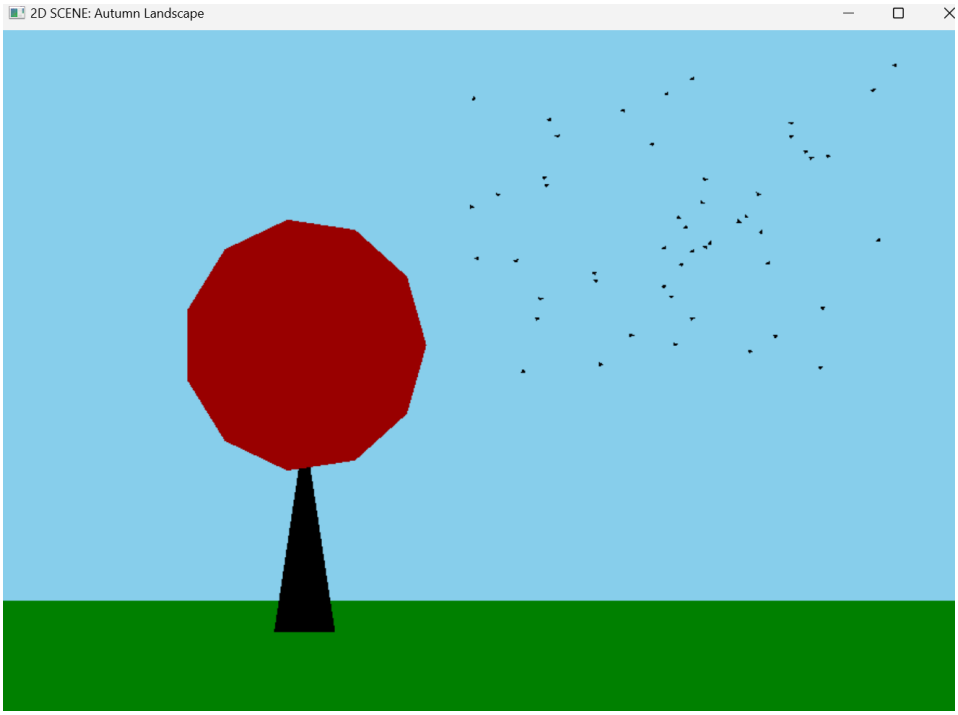

Figura 3. Stolul de păsări

Transformări incluse

Translație: Coroana copacului este plasată în spațiul 2D prin aplicarea unei translații. Este mutată coroana de la poziția sa originală la o nouă poziție specificată de coordonatele de translație (în acest caz, deplasarea la stânga cu -10.0f unități pe axa Ox). Similar pentru trunchi.

Rotație: `atan2(boids[i].vy, boids[i].vx)` calculează unghiul (în radiani) între axa x și direcția în care se mișcă pasărea; `atan2` returnează un unghi în funcție de coordonatele x și y. Rotim matricea pe axa care indică direcția în care pasărea se mișcă. `myMatrix = translationMatrix * rotationMatrix * resizeMatrix` combină matricele pentru a obține transformarea finală. Ordinea acestor operații contează. Astfel, fiecare pasăre va fi rotită în funcție de direcția sa de mișcare.

Matricea de proiecție (*Resize*): Matricea de proiecție este creată folosind funcția `glm::ortho`, definind astfel un volum ortografic de vizualizare în spațiul 2D. Acest volum este delimitat de coordonatele `xMin`, `xMax`, `yMin` și `yMax`.

Implementarea funcțiilor în C++

Clasa `Boid`

```
class Boid {
public:
    float px, py;  // position
    float vx, vy;  // velocity

    Boid(float _x, float _y) : px(_x), py(_y), vx(0.0), vy(0.0) {}
};

std::vector<Boid> boids;
```

Un *boid* are o poziție reprezentată de coordonatele `px`, `py` și o viteză reprezentată de componentele `vx`, `vy`. Constructorul clasei `Boid` primește coordonatele inițiale `_x`, `_y` ale *boidului* și inițializează vitezele cu 0.

De asemenea, este declarat un vector de obiecte de tip `Boid` numit `boids`, care va stoca mai multe instanțe ale acestei clase pentru a reprezenta un grup de *boids* într-o simulare. Fiecare *boid* are propriile sale date de poziție și viteză în cadrul vectorului `boids`.

Funcțiile `keyboardP` și `keyboardR` (descrise mai sus)

```
void keyboardP(unsigned char key, int x, int y) {
    switch (key) {
    case 'd':
        right = true;
        break;
    case 'a':
        left = true;
        break;
    case 'w':
        up = true;
        break;
    case 's':
        down = true;
        break;
    case 'q':
        exit(0);
        break;
    }
}
```

```
void keyboardR(unsigned char key, int x, int y) {
    switch (key) {
    case 'd':
        right = false;
        break;
    case 'a':
        left = false;
        break;
    case 'w':
        up = false;
        break;
    case 's':
        down = false;
        break;
    }
}
```

Le apelăm în `updateBoids`.

Funcția `updateBoids`

Pentru fiecare *boid* aplicăm cele trei reguli:

- **Separare**

```
// separation - boids don't get too close
float sepX = 0.0;
float sepY = 0.0;
for (const Boid& other : boids) {
    if (&boid != &other) {
        float dx = other.px - boid.px;
        float dy = other.py - boid.py;
        float d = std::sqrt(dx * dx + dy * dy);
        if (d < separationDistance) {
            sepX -= dx / d;
            sepY -= dy / d;
        }
    }
}
```

  - se inițializează componente pentru separare `sepX` și `sepY` cu 0 (vector de separare)
  - pentru fiecare *boid* se calculează diferența de poziție dintre *boidul* curent și vecin
  - se calculează distanța euclidiană dintre cele două poziții; dacă distanța este mai mica decât `separationDistance`, atunci componentele de separare sunt ajustate în funcție de poziția *boidului* vecin, astfel încât *boidul* curent să se îndepărteze de acesta

- **Aliniere**

```
// alignment - boids align their velocities with neighboring boids
float avgVx = 0.0;
float avgVy = 0.0;
int count = 0;
for (const Boid& other : boids) {
    if (&boid != &other) {
        float dx = other.px - boid.px;
        float dy = other.py - boid.py;
        float d = std::sqrt(dx * dx + dy * dy);
        if (d < alignmentDistance) {
            avgVx += other.vx;
            avgVy += other.vy;
            count++;
        }
    }
}
if (count > 0) {
    avgVx /= count;
    avgVy /= count;
}
```

  - se inițializează variabile pentru media vitezei `avgVx` și `avgVy` și un contor cu 0

- similar cu **separarea**, se calculează diferența dintre *boidul* curent și *boidul* vecin și distanța euclidiană dintre cele două poziții; dacă distanța este mai mică decât `alignmentDistance`, atunci se adaugă viteza *boidului* vecin la media vitezei și se incrementează contorul
- în final, media vitezei este calculată prin împărțirea sumei vitezelor la numărul de vecini

- **Coeziune**

```
// cohesion - boids move towards the center of mass of neighbors
float avgX = 0.0;
float avgY = 0.0;
count = 0;
for (const Boid& other : boids) {
    if (&boid != &other) {
        float dx = other.px - boid.px;
        float dy = other.py - boid.py;
        float d = std::sqrt(dx * dx + dy * dy);
        if (d < cohesionDistance) {
            avgX += other.px;
            avgY += other.py;
            count++;
        }
    }
}
if (count > 0) {
    avgX /= count;
    avgY /= count;
}
```

- se inițializează variabile pentru media vitezei `avgVx` și `avgVy` și un contor cu 0
- similar cu **separarea** și **alinierea**, se calculează diferența dintre *boidul* curent și *boidul* vecin și distanța euclidiană dintre cele două poziții; dacă distanța este mai mică decât `cohesionDistance`, atunci se adaugă poziția *boidului* vecin la media pozițiilor și se incrementează contorul
- la final, media pozițiilor este calculată prin împărțirea sumei pozițiilor la numărul de vecini

După ce aplicăm cele trei reguli, actualizăm vitezele și pozițiile păsărilor aplicând limitări de viteză și gestionând coliziunile cu limitele scenei.

Funcțiile `grassVBO`, `trunkVBO`, `crownVBO`, `birdVBO`

Aceste funcții sunt responsabile pentru generarea *Vertex Buffer Objects* pentru diferite elemente din scenă: iarbă, trunchiul copacului și coroana copacului. Aceste VBO-uri conțin informații despre pozițiile și culorile elementelor și sunt utilizate pentru a le afișa în cadrul scenei 2D. În funcție de tipul de obiect (`objectTypes[i]`), shaderul va alege o culoare specifică. De exemplu, tipul 0 este asociat cu negru, tipul 1 cu roșu, iar tipul 2 cu verde.

În cadrul funcției `trunkVBO` am aplicat translație pe vârfurile trunchiului copacului. Am mutat trunchiul 10.0f unități la stânga aplicând transformarea de translație asupra vârfurilor prin înmulțirea matricei de translație (`translationMatrix`) cu vectorul de coordonate (`vertex`).

```
// translation matrix - moves the trunk to the left
glm::mat4 translationMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-10.0f, 0.0f, 0.0f));

for (int i = 0; i < sizeof(trunkVertices) / sizeof(trunkVertices[0]); i += 4) {
    glm::vec4 vertex(trunkVertices[i], trunkVertices[i + 1], trunkVertices[i + 2], trunkVertices[i + 3]);
    vertex = translationMatrix * vertex;
    trunkVertices[i] = vertex.x;
    trunkVertices[i + 1] = vertex.y;
    trunkVertices[i + 2] = vertex.z;
    trunkVertices[i + 3] = vertex.w;
}
```

Similar, am mutat coroana copacului la stânga.

```
void calculateCrownVertices(float* crownVertices, int n, float r1, const glm::vec3& translation) {
    glm::mat4 translationMatrix = glm::translate(glm::mat4(1.0f), translation);

    crownVertices[0] = Ox;
    crownVertices[1] = Oy;
    crownVertices[2] = 0.0f;
    crownVertices[3] = 1.0f;

    // drawing the crown using triangles (=> hexagon of n vertices)
    for (int k = 1; k <= n; k++) {
        float angle = (2 * M_PI * k) / n;
        glm::vec4 vertex(Ox + r1 * cos(angle), Oy + r1 * sin(angle), 0.0f, 1.0f);
        vertex = translationMatrix * vertex;

        crownVertices[4 * k] = vertex.x;
        crownVertices[4 * k + 1] = vertex.y;
        crownVertices[4 * k + 2] = vertex.z;
        crownVertices[4 * k + 3] = vertex.w;
    }
}
```

În cadrul funcției `birdVBO`, se definesc coordonatele vârfurilor pentru păsare și aripile sale.

```
float wingspan = 1.0f;

GLfloat birdVertices[] = {
    boid.px, boid.py, 0.0f, 1.0f,
    boid.px - wingspan / 2, boid.py, 0.0f, 1.0f,
    boid.px + wingspan / 2, boid.py, 0.0f, 1.0f,

    boid.px, boid.py, 0.0f, 1.0f,
    boid.px - wingspan / 2, boid.py - wingspan / 2, 0.0f, 1.0f,
    boid.px + wingspan / 2, boid.py - wingspan / 2, 0.0f, 1.0f,
};
```

Bibliografie

- Laboratoarele din cadrul cursului *Grafică pe calculator*
- *https://betterprogramming.pub/mastering-flock-simulation-with-boids-c-opengl-and-imgui-5a3ddd9cb958*
- *https://en.wikipedia.org/wiki/Boids*

Anexe

main.cpp

```
# include <GL/glew.h>
# include <GL/freeglut.h>
# include "loadShaders.h"
# include "glm/gtx/transform.hpp"
# include "glm/gtc/type_ptr.hpp"
# include <vector>
# include "cmath"
# define M_PI           3.14159265358979323846
# define MIN_Y                    -20.0f

// important variables
GLuint
EboId,
birdVaoId,
birdVboId,
GrassVboId,
GrassVaoId,
TrunkVboId,
TrunkVaoId,
CrownVaoId,
CrownVboId,
ProgramId,
myMatrixLocation;

glm::mat4
myMatrix, resizeMatrix;

const float separationDistance = .02;
const float alignmentDistance = 0.6;
const float cohesionDistance = .7;
float xMin = -80, xMax = 80.f, yMin = -60.f, yMax = 60.f;
float Ox = -20.f;
float Oy = 10.275f;
int windowWidth = 900;
int windowHeight = 900;
bool right = false;
bool left = false;
bool up = false;
bool down = false;


class Boid {
public:
	float px, py;  // position
	float vx, vy;  // velocity

	Boid(float _x, float _y) : px(_x), py(_y), vx(0.0), vy(0.0) {}
};

std::vector<Boid> boids;
```

```
void shaders(void)
{
	ProgramId = loadShaders("shader.vert", "shader.frag");
	glUseProgram(ProgramId);
}


// functions for keyboard: press and release keys
void keyboardP(unsigned char key, int x, int y) {
	switch (key) {
	case 'd':
		right = true;
		break;
	case 'a':
		left = true;
		break;
	case 'w':
		up = true;
		break;
	case 's':
		down = true;
		break;
	case 'q':
		exit(0);
		break;
	}
}

void keyboardR(unsigned char key, int x, int y) {
	switch (key) {
	case 'd':
		right = false;
		break;
	case 'a':
		left = false;
		break;
	case 'w':
		up = false;
		break;
	case 's':
		down = false;
		break;
	}
}


// functions for boids
void initBoids() {
	for (int i = 0; i < 50; i++) {
		float x = static_cast<float>(rand()) / RAND_MAX;
		float y = static_cast<float>(rand()) / RAND_MAX;
		boids.push_back(Boid(x, y));
	}
}
std::vector<float> rotationAngles;

void updateBoids(int value) {
```

```
for (Boid& boid : boids) {
	// separation - boids don't get too close
	float sepX = 0.0;
	float sepY = 0.0;
	for (const Boid& other : boids) {
		if (&boid != &other) {
			float dx = other.px - boid.px;
			float dy = other.py - boid.py;
			float d = std::sqrt(dx * dx + dy * dy);
			if (d < separationDistance) {
				sepX -= dx / d;
				sepY -= dy / d;
			}
		}
	}

	// alignment - boids align their velocities with neighboring boids
	float avgVx = 0.0;
	float avgVy = 0.0;
	int count = 0;
	for (const Boid& other : boids) {
		if (&boid != &other) {
			float dx = other.px - boid.px;
			float dy = other.py - boid.py;
			float d = std::sqrt(dx * dx + dy * dy);
			if (d < alignmentDistance) {
				avgVx += other.vx;
				avgVy += other.vy;
				count++;
			}
		}
	}
	if (count > 0) {
		avgVx /= count;
		avgVy /= count;
	}

	// cohesion - boids move towards the center of mass of neighbors
	float avgX = 0.0;
	float avgY = 0.0;
	count = 0;
	for (const Boid& other : boids) {
		if (&boid != &other) {
			float dx = other.px - boid.px;
			float dy = other.py - boid.py;
			float d = std::sqrt(dx * dx + dy * dy);
			if (d < cohesionDistance) {
				avgX += other.px;
				avgY += other.py;
				count++;
			}
		}
	}
	if (count > 0) {
		avgX /= count;
		avgY /= count;
	}
```

```
                // update velocity with speed limit
                boid.vx += sepX + 0.1 * avgVx + 0.01 * (avgX - boid.px);
                boid.vy += sepY + 0.1 * avgVy + 0.01 * (avgY - boid.py);

                float speedLimit = 0.001f;
                float speed = std::sqrt(boid.vx * boid.vx + boid.vy * boid.vy);
                if (speed > speedLimit) {
                        boid.vx = (boid.vx / speed) * speedLimit;
                        boid.vy = (boid.vy / speed) * speedLimit;
                }

                // update position with the grass as limit
                boid.px += boid.vx;
                boid.py += boid.vy;

                if (boid.py < MIN_Y) {
                        boid.py = MIN_Y;
                        boid.vy = -boid.vy; // reverse y velocity to bounce off the bottom
boundary
                }

                float defaultSpeed = 0.1f;
                for (Boid& boid : boids) {
                        if (!right && !left && !up && !down) {
                                boid.vx += defaultSpeed;
                                boid.vy += 0.0;
                        }
                }
        }
        if (right) {
                // 'd' - move to the right
                for (Boid& boid : boids) {
                        boid.vx += .01;
                }
        }
        else if (left) {
                // 'a' - move to the left
                for (Boid& boid : boids) {
                        boid.vx -= .01;
                }
        }
        if (up) {
                // 'w' - move up
                for (Boid& boid : boids) {
                        boid.vy += .01;
                }
        }
        else if (down) {
                // 's' - move down
                for (Boid& boid : boids) {
                        boid.vy -= .01;
                }
        }

        glutPostRedisplay();

        glutTimerFunc(16, updateBoids, 0);
```

```
}
void grassVBO(void)
{
	static const GLfloat grassVertices[] = {
		-100.0f, -50.0f, 0.0f, 1.0f,
		100.0f, -50.0f, 0.0f, 1.0f,
		100.0f, -30.f, 0.0f, 1.0f,
		-100.0f, -30.f, 0.0f, 1.0f,
	};

	int objectTypes[4];
	for (int i = 0; i < 4; i++) {
		objectTypes[i] = 2; // type 2 => green color
	}

	glGenVertexArrays(1, &GrassVaoId);
	glBindVertexArray(GrassVaoId);


	glGenBuffers(1, &GrassVboId);
	glBindBuffer(GL_ARRAY_BUFFER, GrassVboId);
	glBufferData(GL_ARRAY_BUFFER, sizeof(grassVertices) + sizeof(objectTypes),
nullptr, GL_STATIC_DRAW);
	glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(grassVertices), grassVertices);
	glBufferSubData(GL_ARRAY_BUFFER, sizeof(grassVertices), sizeof(objectTypes),
objectTypes);
	glEnableVertexAttribArray(0);
	glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, nullptr);
	glEnableVertexAttribArray(1);
	glVertexAttribIPointer(1, 1, GL_INT, 0, (const GLvoid*)sizeof(grassVertices));
	glEnableVertexAttribArray(2);
	glVertexAttribIPointer(2, 1, GL_INT, 0, (const GLvoid*)sizeof(grassVertices));
}

void trunkVBO(void)
{
	static  GLfloat trunkVertices[] = {
		-20.0f, -3.0f, 0.0f, 1.0f,
		-25.0f, -35.0f, 0.0f, 1.0f,
		-15.0f, -35.f, 0.0f, 1.0f,
	};

	int objectTypes[3];
	for (int i = 0; i < 3; i++) {
		objectTypes[i] = 0; // type 0 => black color
	}

	// translation matrix - moves the trunk to the left
	glm::mat4 translationMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-10.0f,
0.0f, 0.0f));

	for (int i = 0; i < sizeof(trunkVertices) / sizeof(trunkVertices[0]); i += 4) {
		glm::vec4 vertex(trunkVertices[i], trunkVertices[i + 1], trunkVertices[i +
2], trunkVertices[i + 3]);
		vertex = translationMatrix * vertex;
		trunkVertices[i] = vertex.x;
		trunkVertices[i + 1] = vertex.y;
		trunkVertices[i + 2] = vertex.z;
```

```
			trunkVertices[i + 3] = vertex.w;
		}

		glGenVertexArrays(1, &TrunkVaoId);
		glBindVertexArray(TrunkVaoId);

		glGenBuffers(1, &TrunkVboId);
		glBindBuffer(GL_ARRAY_BUFFER, TrunkVboId);

		glBufferData(GL_ARRAY_BUFFER, sizeof(trunkVertices) + sizeof(objectTypes),
nullptr, GL_STATIC_DRAW);
		glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(trunkVertices), trunkVertices);
		glBufferSubData(GL_ARRAY_BUFFER, sizeof(trunkVertices), sizeof(objectTypes),
objectTypes);
		glEnableVertexAttribArray(0);
		glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, nullptr);
		glEnableVertexAttribArray(1);
		glVertexAttribIPointer(1, 1, GL_INT, 0, (const GLvoid*)sizeof(trunkVertices));
		glEnableVertexAttribArray(2);
		glVertexAttribIPointer(2, 1, GL_INT, 0, (const GLvoid*)sizeof(trunkVertices));
}

void calculateCrownVertices(float* crownVertices, int n, float r1, const glm::vec3&
translation) {
		glm::mat4 translationMatrix = glm::translate(glm::mat4(1.0f), translation);

		crownVertices[0] = Ox;
		crownVertices[1] = Oy;
		crownVertices[2] = 0.0f;
		crownVertices[3] = 1.0f;

		// drawing the crown using triangles (=> hexagon of n vertices)
		for (int k = 1; k <= n; k++) {
			float angle = (2 * M_PI * k) / n;
			glm::vec4 vertex(Ox + r1 * cos(angle), Oy + r1 * sin(angle), 0.0f, 1.0f);
			vertex = translationMatrix * vertex;

			crownVertices[4 * k] = vertex.x;
			crownVertices[4 * k + 1] = vertex.y;
			crownVertices[4 * k + 2] = vertex.z;
			crownVertices[4 * k + 3] = vertex.w;
		}
}

void crownVBO(void)
{
		static const int n = 11; // if modified, the indices[] should be modified too
		static const float r1 = 20.0f;
		glm::vec3 translation(-10.0f, 0.0f, 0.0f);

		GLfloat crownVertices[5 * n + 1];

		int objectTypes[5 * n + 1];
		for (int i = 0; i < 5 * n + 1; i++) {
			objectTypes[i] = 1; // type 1 => red color
		}

		GLuint indices[] = {
```

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

        calculateCrownVertices(crownVertices, n, r1, translation);

        glGenVertexArrays(1, &CrownVaoId);
        glBindVertexArray(CrownVaoId);

        glGenBuffers(1, &CrownVboId);
        glBindBuffer(GL_ARRAY_BUFFER, CrownVboId);

        glBufferData(GL_ARRAY_BUFFER, sizeof(crownVertices) + sizeof(objectTypes),
nullptr, GL_STATIC_DRAW);
        glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(crownVertices), crownVertices);
        glBufferSubData(GL_ARRAY_BUFFER, sizeof(crownVertices), sizeof(objectTypes),
objectTypes);

        glGenBuffers(1, &EboId);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EboId);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, nullptr);
        glEnableVertexAttribArray(1);
        glVertexAttribIPointer(1, 1, GL_INT, 0, (const GLvoid*)sizeof(crownVertices));
        glEnableVertexAttribArray(2);
        glVertexAttribIPointer(2, 1, GL_INT, 0, (const GLvoid*)sizeof(crownVertices));
}

void destroyShaders(void)
{
        glDeleteProgram(ProgramId);
}

void birdVBO(Boid boid)
{
        float wingspan = 1.0f;

        GLfloat birdVertices[] = {
                boid.px, boid.py, 0.0f, 1.0f,
                boid.px - wingspan / 2, boid.py, 0.0f, 1.0f,
                boid.px + wingspan / 2, boid.py, 0.0f, 1.0f,

                boid.px, boid.py, 0.0f, 1.0f,
                boid.px - wingspan / 2, boid.py - wingspan / 2, 0.0f, 1.0f,
                boid.px + wingspan / 2, boid.py - wingspan / 2, 0.0f, 1.0f,
        };
```

```
	int objectTypes[6];
	for (int i = 0; i < 6; i++) {
		objectTypes[i] = 0; // type 0 => color for birds
	}

	glGenVertexArrays(1, &birdVaoId);
	glBindVertexArray(birdVaoId);

	glGenBuffers(1, &birdVboId);
	glBindBuffer(GL_ARRAY_BUFFER, birdVboId);
	glBufferData(GL_ARRAY_BUFFER, sizeof(birdVertices) + sizeof(objectTypes), nullptr,
GL_DYNAMIC_DRAW);
	glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(birdVertices), birdVertices);
	glBufferSubData(GL_ARRAY_BUFFER, sizeof(birdVertices), sizeof(objectTypes),
objectTypes);

	glEnableVertexAttribArray(0);
	glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, nullptr);
	glEnableVertexAttribArray(1);
	glVertexAttribIPointer(1, 1, GL_INT, 0, (const GLvoid*)sizeof(birdVertices));
	glEnableVertexAttribArray(2);
	glVertexAttribIPointer(2, 1, GL_INT, 0, (const GLvoid*)sizeof(birdVertices));
}

void destroyVBO(void)
{
	glDisableVertexAttribArray(2);

	glBindBuffer(GL_ARRAY_BUFFER, 0);
	glDeleteBuffers(1, &CrownVboId);
	glDeleteBuffers(1, &EboId);

	glBindVertexArray(0);
	glDeleteVertexArrays(1, &CrownVaoId);
	glBindVertexArray(0);
	glDeleteVertexArrays(1, &birdVaoId);

	glBindVertexArray(0);
	glDeleteVertexArrays(1, &GrassVaoId);
	glBindVertexArray(0);
	glDeleteVertexArrays(1, &TrunkVaoId);
}

void cleanup(void)
{
	destroyShaders();
	destroyVBO();
}


void initialize(void)
{
	glClearColor(0.529f, 0.808f, 0.922f, 1.0f); // blue sky

	grassVBO();
	trunkVBO();
	crownVBO();
	shaders();
```

```
	initBoids();

	myMatrixLocation = glGetUniformLocation(ProgramId, "myMatrix");
	resizeMatrix = glm::ortho(xMin, xMax, yMin, yMax);

}


void render(void)
{
	glClear(GL_COLOR_BUFFER_BIT);

	// birds
	for (size_t i = 0; i < boids.size(); ++i) {
		float rotationAngle = atan2(boids[i].vy, boids[i].vx);
		glm::mat4 rotationMatrix = glm::rotate(glm::mat4(1.0f),
glm::degrees(rotationAngle), glm::vec3(0.0f, 0.0f, 1.0f));
		glm::mat4 translationMatrix = glm::translate(glm::mat4(1.0f),
glm::vec3(boids[i].px, boids[i].py, 0.0f));
		myMatrix = translationMatrix * rotationMatrix * resizeMatrix;

		glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);

		birdVBO(boids[i]);
		glBindVertexArray(birdVaoId);
		glDrawArrays(GL_TRIANGLES, 0, 6);
	}

	// grass
	glBindVertexArray(GrassVaoId);
	myMatrix = resizeMatrix;
	glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
	glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

	// trunk
	glBindVertexArray(TrunkVaoId);
	myMatrix = resizeMatrix;
	glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
	glDrawArrays(GL_TRIANGLES, 0, 3);

	// crown
	glBindVertexArray(CrownVaoId);
	myMatrix = resizeMatrix;
	glUniformMatrix4fv(myMatrixLocation, 1, GL_FALSE, &myMatrix[0][0]);
	glDrawElements(GL_TRIANGLES, 33, GL_UNSIGNED_INT, (void*)(0));

	glFlush();
}

int main(int argc, char* argv[])
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
	glutInitWindowPosition(400, -100);
	glutInitWindowSize(900, 900);
	glutCreateWindow("2D SCENE: Autumn Landscape");

	glewInit();
```

```
        initialize();
        glutKeyboardFunc(keyboardP);
        glutKeyboardUpFunc(keyboardR);
        glutDisplayFunc(render);
        glutTimerFunc(0, updateBoids, 0); // start animation
        glutMainLoop();

        glutCloseFunc(cleanup);

        return 0;
}
```

shader.vert

```
#version 330 core
layout(location = 0) in vec4 inPosition;
layout(location = 1) in vec4 inColor; // color
layout(location = 2) in int inObjectType; // object type

uniform mat4 myMatrix;
out vec4 fragColor;
flat out int objectType;

void main()
{
    gl_Position = myMatrix * inPosition;
    fragColor = inColor;
    objectType = int(inObjectType);
}
```

shader.frag

```
#version 330 core
in vec4 fragColor;
flat in int objectType;

out vec4 outColor;

void main()
{
    if (objectType == 1) {
        outColor = vec4(0.6, 0.0, 0.0, 1.0); // red
    } else if (objectType == 0) {
        outColor = vec4(0.0, 0.0, 0.0, 1.0); // green
    } else if(objectType == 2){
        outColor = vec4(0.0, 0.5, 0.0, 1.0); // black (default)
    }
}
```