MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

Cuzmin Simion Faf-221

Report

Laboratory work n.5

of Computer Graphics

Checked by: [i]

Элементы оглавления не найдены.

Olga Grosu, *university assistant*

DISA, FCIM, UTM

Chișinău – 2023

Task A

- Do the sketch using the function:

  Use the concept of steering forces to drive the behavior of the creatures in your ecosystem. Some possibilities:
  Create “schools” or “flocks” of creatures.
  Use a seeking behavior for creatures to search for food (for chasing moving prey, consider “pursuit”).
  Use a flow field for the ecosystem environment. For example, how does your system behave if the creatures live in a flowing river?
  Build a creature with countless steering behaviors (as many as you can reasonably add). Think about ways to vary the weights of these behaviors so that you can dial those behaviors up and down, mixing and matching on the fly. How are creatures’ initial weights set? What rules drive how the weights change over time?
  Complex systems can be nested. Can you design a single creature out of a flock of boids? And can you then make a flock of those creatures?
  Complex systems can have memory (and be adaptive). Can the history of your ecosystem affect the behavior in its current state? (This could be the driving force behind how the creatures adjust their steering force weights.)

- The program code with relevant comments:

```
import processing.core.*;
import java.util.ArrayList;
import java.util.Iterator;

void setup() {
  size(800, 600);
  creatures = new ArrayList<Creature>();
  foods = new ArrayList<Food>();

  // Create 3 creatures
  for (int i = 0; i < 5; i++) {
    creatures.add(new Creature(random(width),
random(height), 30, 0.02, 50));
  }

  // Create 100 food items
  for (int i = 0; i < 100; i++) {
    foods.add(new Food(random(width),
random(height), 10));
  }

  // Create poison food
  for (int i = 0; i < 10; i++) {
    foods.add(new PoisonFood(random(width),
random(height), 10));
  }
```

```
}

void draw() {
  background(224, 158, 118);

  // Draw the attraction circle in the center
  drawAttractionCircle(width / 2, height / 2, 200);

  // Iterate over creatures using Iterator
  Iterator<Creature> creatureIterator =
creatures.iterator();
  while (creatureIterator.hasNext()) {
    Creature creature = creatureIterator.next();

    // Check if the creature is outside the
attraction circle (reached the borders)
    if
(isOutsideAttractionCircle(creature.center.x,
creature.center.y, creature.radius * 2.0)) {
      creature.seek(foods);
      creature.update();
      creature.display();

      if (creatures.size() > 1) {
        creature.checkCollisions(creatures, foods);
```

```
      }
    } else {
      // Check the size during collision with the
attraction circle
      if
(creature.collidesWithAttractionCircle(width / 2,
height / 2, 200) && creature.radius < 30) {
        creatureIterator.remove(); // Remove the
creature when it reaches the borders and is smaller
than 2.0 times the initial one
      }
    }
  }

  // Iterate over foods using Iterator
  Iterator<Food> foodIterator = foods.iterator();
  while (foodIterator.hasNext()) {
    Food food = foodIterator.next();
    food.display();
  }

  // Close the program if only one creature is left
  if (creatures.size() == 1) {
    exit();
  }
}
```

```
// Function to draw the attraction circle
void drawAttractionCircle(float x, float y, float
radius) {
  noFill();
  stroke(255, 0, 0); // Red color for the circle
  ellipse(x, y, radius * 2, radius * 2);
}

// Function to check if a point is inside or on the
attraction circle
boolean isOutsideAttractionCircle(float x, float y,
float circleRadius) {
  float distance = dist(x, y, width / 2, height /
2);
  return distance >= circleRadius;
}



ArrayList<Creature> creatures;
ArrayList<Food> foods;

class Food {
  PVector position;
```

```
  float radius;

  Food(float x, float y, float radius) {
    // Ensure the initial position is at least 100
pixels away from all borders
    float adjustedX = max(radius + 100, min(x,
width - radius - 100));
    float adjustedY = max(radius + 100, min(y,
height - radius - 100));
    position = new PVector(adjustedX, adjustedY);
    this.radius = radius;
  }

  void display() {
    fill(0, 255, 0);
    ellipse(position.x, position.y, radius * 2,
radius * 2);
  }
}

class PoisonFood extends Food {

  PoisonFood(float x, float y, float radius) {
    super(x, y, radius);
  }
```

```
  @Override
  void display() {
    fill(139, 0, 0); // RGB color for poison food
    triangle(
      position.x, position.y - radius,
      position.x - radius, position.y + radius,
      position.x + radius, position.y + radius
    );
  }
}


class Creature {
  PVector center;
  float radius;
  float angle;
  float angleVelocity;
  float speed;
  PVector acceleration;
  PVector velocity;
  PVector direction;
  ArrayList<Wings> wingsList;
  float lastMillis;
```

```
  Creature(float x, float y, float radius, float
angleVelocity, int numWings) {
    center = new PVector(x, y);
    this.radius = radius;
    this.angle = 0;
    this.angleVelocity = angleVelocity;
    this.speed = 20;
    this.acceleration = new PVector(0.1, 0.1);
    this.velocity = new PVector(this.speed, 0);
    this.direction = new PVector(0, 0);
    wingsList = new ArrayList<Wings>();
    lastMillis = millis();

    for (int i = 0; i < numWings; i++) {
      wingsList.add(new Wings());
    }
  }

  boolean collidesWithAttractionCircle(float
attractionX, float attractionY, float
attractionRadius) {
    float distance = dist(center.x, center.y,
attractionX, attractionY);
    return distance <= (attractionRadius + radius)
/ 2;
  }
```

```
  void seek(ArrayList<Food> foods) {
    if (foods.size() > 0) {
      Food target = findNearestFood(foods);
      PVector desired =
PVector.sub(target.position, center);
      desired.setMag(speed);

      // Calculate the steering force
      PVector steer = PVector.sub(desired,
velocity);
      steer.limit(0.1); // Limit the steering force
to avoid abrupt changes

      // Apply steering force to update the
direction
      this.direction.add(steer).normalize();

      // Set the new velocity based on the updated
direction

this.velocity.set(PVector.mult(this.direction,
this.speed));
    }
  }
```

```
  void applyForce(PVector force) {
    acceleration.add(force);
  }

  void update() {
    float deltaTime = (float) (millis() -
lastMillis) / 1000.0;
    lastMillis = millis();

    // Update speed based on acceleration
    this.speed += this.acceleration.mag() *
deltaTime;

    // Update velocity based on acceleration

this.velocity.add(PVector.mult(this.acceleration,
deltaTime));

    // Update position based on velocity
    this.center.add(PVector.mult(this.velocity,
deltaTime));

    // Oscillate the angle of the creature's wings
based on speed
    this.angle += this.speed * this.angleVelocity;
```

```
    // Update wings position
    for (Wings wings : wingsList) {
      wings.update(this.center, this.angle);
    }

    // Check boundaries and reverse direction if
needed
    if (this.center.x > width - this.radius) {
      this.center.x = width - this.radius;
      this.velocity.x *= -1; // Reverse x-direction
    } else if (this.center.x < this.radius) {
      this.center.x = this.radius;
      this.velocity.x *= -1; // Reverse x-direction
    }

    if (this.center.y > height - this.radius) {
      this.center.y = height - this.radius;
      this.velocity.y *= -1; // Reverse y-direction
    } else if (this.center.y < this.radius) {
      this.center.y = this.radius;
      this.velocity.y *= -1; // Reverse y-direction
    }

    // Reset acceleration
```

```
    this.acceleration.mult(0);
  }

  void display() {
    pushMatrix();
    translate(this.center.x, this.center.y);

    ellipse(0, 0, this.radius * 2, this.radius *
2);

    fill(216, 240, 232);
    ellipse(-10, -10, 5, 5);
    ellipse(10, -10, 5, 5);

    arc(0, 5, 30, 20, 0, PI);
    fill(133, 122, 106);

    for (Wings wings : wingsList) {
      wings.display(this.radius);
    }

    popMatrix();
  }

void checkCollisions(ArrayList<Creature> creatures,
ArrayList<Food> foods) {
```

```
  Iterator<Creature> creatureIterator =
creatures.iterator();
  while (creatureIterator.hasNext()) {
    Creature otherCreature =
creatureIterator.next();
    if (this != otherCreature &&
collidesWith(otherCreature)) {
      if (this.radius > otherCreature.radius) {
        this.grow(1.3);
        creatureIterator.remove(); // Use iterator
to remove the smaller creature
        if (this.radius > MAX_CREATURE_RADIUS) {
          handleTooBigError();
        }
      } else if (this.radius <
otherCreature.radius) {
        otherCreature.grow(1.3);
        creatureIterator.remove(); // Use iterator
to remove the smaller creature
        if (otherCreature.radius >
MAX_CREATURE_RADIUS) {
          handleTooBigError();
        }
        break;
      } else {
```

```
        // Equal-sized creatures, randomly choose
one to survive
        if (random(1) < 0.5) {
          this.grow(1.3);
          creatureIterator.remove();
          if (this.radius > MAX_CREATURE_RADIUS) {
            handleTooBigError();
          }
        } else {
          otherCreature.grow(1.3);
          creatureIterator.remove();
          if (otherCreature.radius >
MAX_CREATURE_RADIUS) {
            handleTooBigError();
          }
          break;
        }
      }
    }
  }

  Iterator<Food> foodIterator = foods.iterator();
  while (foodIterator.hasNext()) {
    Food food = foodIterator.next();
    if (collidesWith(food)) {
      if (food instanceof PoisonFood) {
```

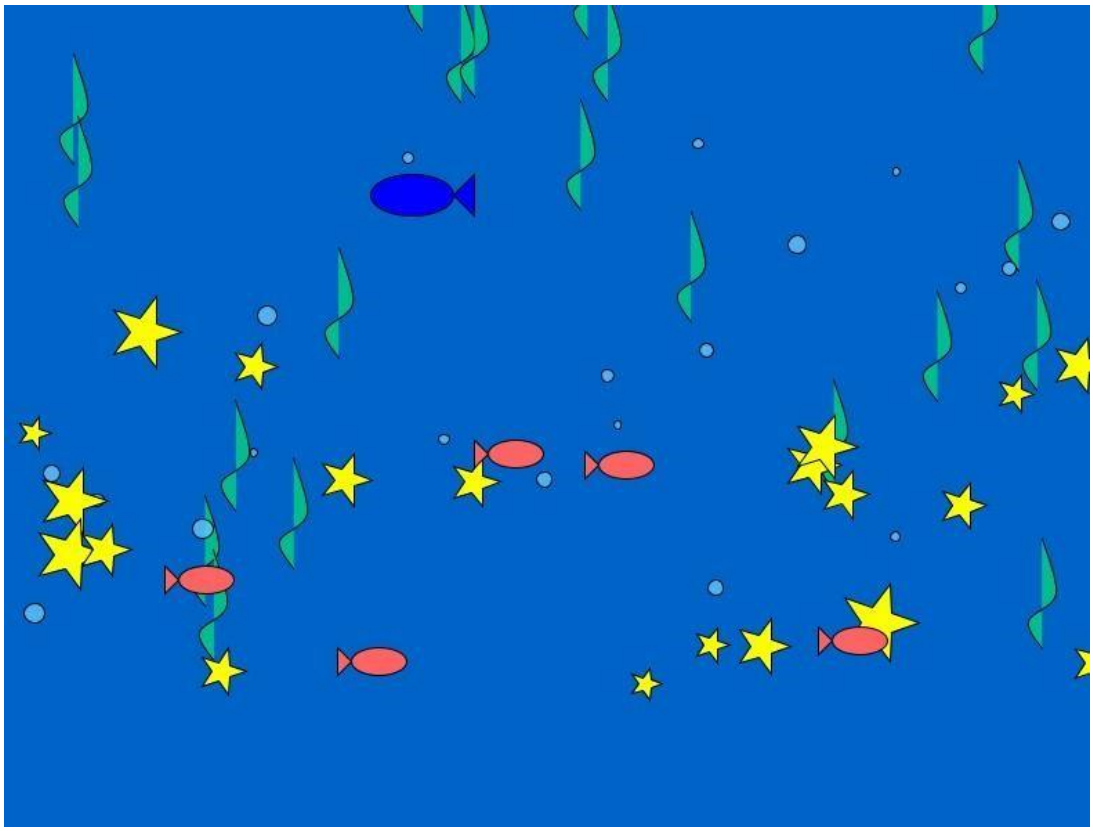
```
        this.grow(0.5); // Poisonous food, reduce
growth
      } else {
        this.grow(1.1); // Regular food
      }
      foodIterator.remove(); // Use iterator to
remove the food
      if (this.radius > MAX_CREATURE_RADIUS) {
        handleTooBigError();
      }
      break;
    }
  }
}


  void handleTooBigError() {
    println("Error: Creature is too big! Exiting
program.");
    exit();
  }

  static final float MAX_CREATURE_RADIUS = 200; //
Adjust the maximum allowed creature radius as
needed
```

```
  void grow(float scaleFactor) {
    this.radius *= scaleFactor;
    for (Wings wings : wingsList) {
      wings.changeSize(scaleFactor);
    }
  }

  boolean collidesWith(Creature otherCreature) {
    float distance = dist(center.x, center.y,
otherCreature.center.x, otherCreature.center.y);
    return distance < (radius +
otherCreature.radius) / 2;
  }

  boolean collidesWith(Food food) {
    float distance = dist(center.x, center.y,
food.position.x, food.position.y);
    return distance < (radius + food.radius) / 2;
  }

  Food findNearestFood(ArrayList<Food> foods) {
    if (foods.size() > 0) {
      Food nearest = foods.get(0);
```

```
      float record = dist(nearest.position.x,
nearest.position.y, center.x, center.y);
      for (Food food : foods) {
        float d = dist(food.position.x,
food.position.y, center.x, center.y);
        if (d < record) {
          record = d;
          nearest = food;
        }
      }
      return nearest;
    } else {
      return null;
    }
  }
}

class Wings {
  float wingLength;
  float oscillationAmplitude;
  float wingAngleOffset;

  Wings() {
    wingLength = 30;
    oscillationAmplitude = 20;
```

```
    wingAngleOffset = random(TWO_PI);
  }

  void update(PVector center, float angle) {
    // Wings update logic, if needed
  }

  void display(float creatureRadius) {
    float wingOscillation = sin(frameCount * 0.05 +
wingAngleOffset) * oscillationAmplitude;

    line(creatureRadius, 0, creatureRadius +
wingLength, wingOscillation);
    line(-creatureRadius, 0, -creatureRadius -
wingLength, wingOscillation);
  }

  void changeSize(float scaleFactor) {
    this.wingLength *= scaleFactor;
    this.oscillationAmplitude *= scaleFactor;
  }
}
```

- Screen printing of program execution:



- Student's conclusions and reflections:

This code sets up a simulation environment in which creatures interact with food and poison. The canvas has a size of 800x600 pixels, and three creatures are initialized with random positions, a radius of 30, and 50 wings each. Additionally, 100 regular food items and 10 poison food items are randomly placed.

The draw function manages the simulation. It draws an attraction circle at the center with a radius of 200 pixels. Creatures outside the circle seek food, update their positions, and check for collisions. If a creature is inside the circle and has a radius smaller than 30, it is removed.

The program exits when only one creature remains. The Creature class includes methods for growth, collision detection with other creatures and food, and seeking the nearest food. The Wings class controls the display and size change of wings. The code effectively models a dynamic ecosystem with evolving creatures and environmental challenges.

---

i