Проектна задача на тема:

„СИМУЛИРАЊЕ ЕВОЛУЦИЈА НА ОРГАНИЗМИ“

Ученик: Вељко Аџиќ

2023

Содржина

1. ВОВЕД

Цел на ова проектна задача е да се направи програма со која лесно и едноставно можеме да претставиме како една популација поединци под влијание на притисоците од околината доведува до адаптација и специјализација на целата популација.

Општиот концепт на кој начин ќе можеме да симулираме еволуција е така што на почетокот на симулацијата ќе креираме одреден број организми со случајни вредности и ќе има одреден број храна низ околината. Како организмите понатаму живеат така повеќе трошат енергија, и ако опстанат доволно за да се размножат, односно да креираат нов организам што има слични карактеристики, нивниот генетски код опстанува, но ако не успеат да опстанат доволно за да се размножат, нивниот генетски код се отстранува од популацијата. Ова симулација на едноставен модел на еволуција ни дава слика како популација организми може да се приспособи на притисоците од околината.

За ова симулација имаме два објекта: објект за храна (Food) и објект за организам (Creature). Food објектот има променлива за позиција и променлива за енергетска вредност која дава кога ќе се изеде, и има една метода со која ќе се исцрта на прозорецот. Creature објектот е покомплексен, така што ги има променливите за: позиција, големина, брзина, видливост, енергија, позиција; и има методи за: пронаоѓање на најдобрата храна, движење кон најдобрата храна, размножување, исцртување на прозорецот и други методи кои ќе бидат објаснети понатаму. Исто така во „генетскиот код“ се зачувуваат три карактеристики: брзина на движење, големина на организмот и големина на видливоста.

Ќе користам Processing за да ја реализирам ова задача. Processing е библиотека за Java која содржи корисни функции за полесно креирање прозорец и исцртување елементи во него и функции и класи за полесно извршување операции и за зачувување на податоци. Processing е исто така наменет за графички апликации, што значи дека е оптимизиран за вакви задачи.

2. ГЛАВНАТА ПРОГРАМА

Главната програма во Processing се состои од две функции: setup и draw. Функцијата setup се извршува еднаш на почетокот кога се стартува програмата и обично се користи за декларирање и доделување на вредности на променливи. А функцијата draw се извршува повеќе пати во секунда сè додека не се исклучи програмата (стандардно е поставено да се извршува 60 пати во секунда, но ова може да се смени по потреба) и тука обично се извршуваат пресметки и исцртување.

```
ArrayList<Food> food = new ArrayList<Food>();
ArrayList<Creature> creatures = new ArrayList<Creature>();

void setup(){
  size(1000, 1000);

  //pregen food
  for(int i = 0; i < 50; i++)
    food.add( new Food() );
  //~

  //pregen creatures
  for(int i =0 ; i < 30; i++)
    creatures.add( new Creature() );
  //~
}
```

На почетокот имам декларирано две глобални листи, **food** и **creatures**, и ми служат за зачувување на сите објекти организми и храна кои постојат. Овие се низи објекти кои имаат променлива големина, односно нема ограничување на тоа колку членови можам да додадам. Во **setup()** делот ја имам функцијата **size()** со која ја дефинирам големината на прозорецот, односно големината на околината. Тука сум одбрал околината да ми е широка 1.000 пиксели и висока 1.000 пиксели. Потоа имам два циклуса за повторување. Првиот во листата **food** додава 50 нови

објекти за храна кои имаат случајна позиција и енергетска вредност. А вториот циклус генерира 30 случајни организми во соодветната листа.

```
18 void draw(){
19   background(220);
20 
21   //periodically add food if it's less than 100
22   if(frameCount % 10 == 0 && food.size() < 200)
23     food.add( new Food() );
24   //~
25 
26   //display the food
27   for(int i = food.size() - 1; i >= 0; i--)
28     food.get(i).show();
29   //~
30 
```

Во **draw()** делот прво ја имам функцијата **background()** и со неа ја одредувам бојата на позадината. Со еден аргумент (како во програмата) се поставува ахроматска боја, односно црна боја има вредност 0 и бела има вредност 255, но со три аргумента може да се постават хроматски бои. Јас имам избрано ахроматска боја со вредност 220, која е малку потемнета бела боја. Потоа има if структура за избор и со неа на секои 10 рамки се додава нова произволен објект на храна ако има помалку од 200 објекти за храна во околината. Променливата **frameCount** е системска променлива која брои колку пати се извршила методата **draw()**. Потоа со структурата за повторување се исцртуваат сите објекти за храна од листата **food**. Со бројачот броиме одназад бидејќи во ретки случаи ако се отстрани објект во листата додека одиме низ неа може да настанат грешки или прерипувања.

```
31   //removing creatures and optimisation
32   if(creatures.size() == 0){
33     noLoop();
34     println("There are no more creatures");
35   }else{
```

И на крај има структура за избор. Во случај ако сите организми изумрат, односно ако листата **creatures** е празна ќе настане прекин на повтарање на функцијата **draw()** со функцијата **noLoop()** за да не се извршуваат непотребни операции и во конзолата се испечатува порака дека симулацијата завршила.

```
35    }else{
36      //~kill dead creatures
37      for(int i = creatures.size() - 1; i >= 0; i--){
38        if(!creatures.get(i).alive)
39        creatures.get(i).die();
40      }
41      //~~
42
43      //~eat meals
44      for(int i = creatures.size() - 1; i >= 0; i--)
45        creatures.get(i).eat();
46      //~~
47
48      //~update and draw the creatures
49      for(int i = creatures.size() - 1; i >= 0; i--){
50        creatures.get(i).update();
51        creatures.get(i).show();
52      }
53      //~
54    }
55    //~
56
57  }
```

Но во случај ако условот не е исполнет, односно има организми кои се сè уште живи, имаме три циклуса за повторување. Првиот циклус оди низ листата **creatures** и врши проверка дали секој организам е мртов, односно дали логичката променлива **alive** има вредност false. Ако условот важи, за тој организам се извршува методата **die()**, која ќе биде објаснета подолу.

Во следниот циклус за повторување пак одиме низ листата **creatures** и за секој организам ја извршуваме методата **eat()**. Ова метода ја отстранува сета храна која е доволно блиска да биде изедена, но исто така и се отстрануваат сите

организми кои се помали од организмот на кој му се извршува методата. Затоа циклусот минува низ листата одназад поради тоа што ако во случај се отстрани организам од листата да не настанат грешки или прерипувања.

И последниот циклус за повторување исто така оди низ листата **creatures** и со него се извршуваат методите **update()** и **show()**. Со повикување на методата **update()** се извршуваат повеќе пресметки кои се клучни за целата симулација и однесувањата на организмите. А со методата **show()** се исцртува организмот на прозорецот. Овие методи ќе бидат објаснети во последователен дел.

3. КЛАСА Food

```
float minE = -50,  //minimum energy of food
      maxE = 50;   //maximum energy of food

class Food {
 PVector pos;  //position
 float energy; //energy from food
```

Објект за храна треба да има позиција и енергетска вредност која се добива кога ќе биде изедено. Енергетската вредност **energy** паѓа во интервал од -50 до 50, каде што се ограничена со глобалните променливите **minE** и **maxE**, а променливата за позиција **pos** е PVector, односно е вектор променлива која чува x и y координати.

```
//Constructor for inputed pos and energy
Food(float _x, float _y, float _e){
  pos = new PVector(_x, _y);

  //capping energy
  energy = constrain(_e, minE, maxE);
}
//~

//Overloaded constructor for random pos and energy
Food(){
 pos = new PVector(random(width), random(height));
 energy = random(-30, maxE);
}
//~
```

Потоа во линија 9 имаме конструктор кој прима три аргументи, првите два за x и y позиција и последниот за енергетската вредност. Во променливата **pos** креираме вектор со внесените x и y вредности. **PVector** е класа дел од Processing која служи за полесно вршење пресметки со вектори, но и често се користи за зачувување координати во простор. Внесената енергетска вредност ја зачувуваме во соодветната променлива, но пред тоа ја ограничуваме со Processing функцијата **constrain()**.

Во линија 18 го преклопуваме (overload) конструкторот без аргументи. Во случај ако сакаме да добиеме објект храна со случајна позиција и енергетска вредност при дефинирање нема да внесеме аргументи во конструкторот. Во **pos** исто така креираме вектор, но користејќи ја функцијата **random()** и системските променливи **width** и **height** добиваме случајна позиција некаде во прозорецот. Исто така доделуваме случајни вредности на променливата **energy** со помош на функцијата **random()**. При случајно креирање на храна енергетската вредност паѓа во опсегот од -30 до 50.

Функцијата **random()** е дел од Processing која прима еден или два аргумента. Во случај ако е внесен еден аргумент тогаш функцијата враќа случаен природен број помеѓу 0 и внесениот број. Но во случај кога се внесени два аргумента функцијата враќа природен број помеѓу двата внесени.

```
24  //display the food
25  void show(){
26    push();
27    //calculate color based on energy
28    color col = lerpColor( color(150, 220, 0),
29                           color(200, 10, 0),
30                           map(energy, minE, maxE , 0, 1)
31                         );
32
33    noStroke();
34    fill(col);
35    circle(pos.x, pos.y, 5);
36    pop();
37  }
38  //~
39 }
```

Исто така ни треба и метода за исцртување **show()**. Со неа визуелно ќе ја прикажеме позицијата и енергетската вредност на објектот храна. Во телото на ова метода на почеток и на крај ги има функциите **push()** и **pop()**. Поради тоа што стилот е глобално зачуван и која било измена во еден дел од програмата може да доведе до несакани промени во други делови, секогаш ги обиколувам сегментите од програмата со **push()** и **pop()** во кој се врши некоја промена на стилот. Со функцијата **push()** се зачувува стилот со меморија и кога се повикува **pop()** стилот е враќа на претходно.

Потоа дефинираме променлива **col** со вид **color**, односно променлива за боја. Со неа ја претставуваме енергетската вредност на храната. На ова променлива и се доделува вратената вредност од функцијата **lerpColor()** во која имаме ставено зелена боја во првиот аргумент, црвена во вториот, и во третиот аргумент функција за пресликување **map()** која враќа вредност помеѓу 0 и 1. **lerpColor()** враќа боја помеѓу двете внесени бои во зависност од последниот аргумент. Ако замислиме градација помеѓу зелена и црвена боја, и ако 0 ни ја означува зелена и 1 ни ја означува црвената, тогаш со број помеѓу 0 и 1 ни означува боја која е делумно зелена и делумно црвена. Таа меѓу вредност ја добиваме од функцијата **map()**.

Функцијата **map()** е дел од Processing и служи за пресликување на некоја вредност од еден опсег во друг. Ова функција прима пет аргумента, каде првиот аргумент е број од опсег А, па вториот и третиот се почетна и крајна вредност на опсег А, и четвртиот и петтиот се почетната и крајната вредност на опсегот Б. Во овој случај ние треба да ја пресликаме енергетската вредност, која е во опсег од -50 до 50, во опсег од 0 до 1.

Функцијата **noStroke()** ни го менува стилот на контурната линија, односно да не се исцртува контурна линија. Функцијата **fill()** ни менува со која боја се избојува внатрешноста, и во овој случај ја имам поставено претходно пресметана бојата **col**. Со функцијата **circle()** се исцртува круг со неговиот центар со истите координати како и позицијата на храната и со радиус од 5 пиксели. Со овој дел од

кодот храната ја претставуваме со круг со боја која е пропорцијална на енергетската вредност.

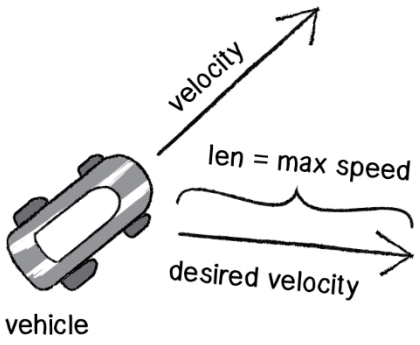
4. КЛАСА Creature

Клучен концепт со кој организмите се движат и се насочуваат кон некоја посакана локација е принципот на автономни агенти (возила).

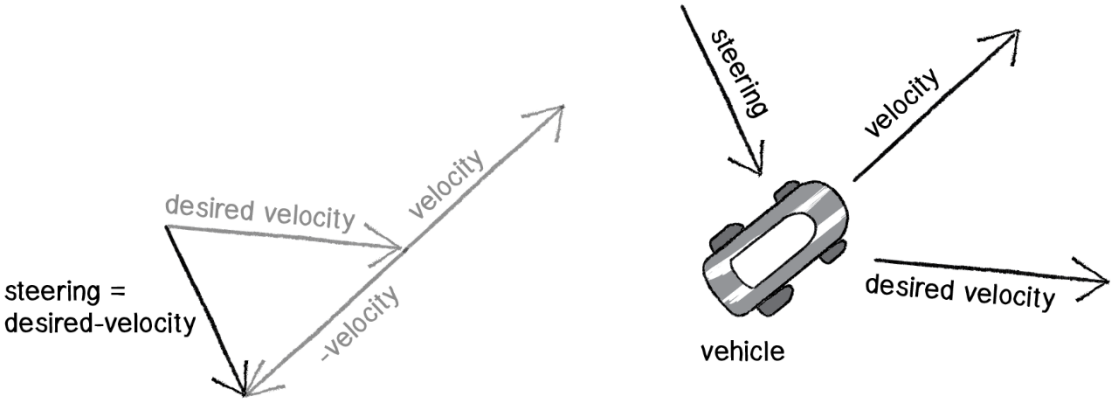
Кога симулираме физика на некое тело, за тоа тело треба да ја имаме зачувано неговата позиција P, брзина V и забрзување A. Секој временски такт можеме да имаме некоја сила F која влијае на телото, која влијае на неговото забрзување, а забрзувањето влијае на брзината која влијае на неговата позиција.

$$F \rightarrow A \rightarrow V \rightarrow P$$

Во разлика, автономен агент не само што има позиција P, брзина V и забрзување A, туку има и посакана позиција до која сака да се достигне P_D и посакана брзина V_D со која ќе достигне до P_D.



Автономниот агент за да ја достигне посаканата брзина V_D ќе треба секој временски такт да ја пресмета грешката како разлика од моменталната брзина V и посаканата брзина V_D. Таа грешка е со која ќе се насочува агентот кон посаканата позиција, односно ќе ни биде силата F. Со тоа секој временски такт агентот ќе си го менува својот правец на движење за да ја достигне посаканата позиција.



Со ова основа можеме да креираме организми кои имаат некаква логика или „свест“ за нивната околина и како според нивната околина можат да се приспособат и да ги направат најдобрите одлуки.

```
float breedingChance = 0.005;   //Chance of repruduction in %
float steeringStrenght = 0.05;  //Strength of applied steering in %
float eatableSize = 0.95;        //% of how smaller creature can be canibalised

//Minimum and maximum ranges
float[] capSight = {25, 150};
float[] capSize =  {10, 150};
float[] capSpeed = {1 , 5 };
//~

//maximum energy possible
float maxEnergy = ( capSight[1] + pow(capSize[1], 3) * pow(capSpeed[1], 2) ) / 2;
```

За класата Creature прво имаме дефинирано неколку глобални променливи кои ќе ни служат понатаму во пресметките, но и за контрола на симулацијата.

Променливата **breedingChance** е шансата за организмот да се размножи секој временски такт. Променливата **steeringStrenght** е колку од насочувачката сила ќе влијае а правецот на организмот, а **eatableSize** е колку помал треба друг организам да биде за да може да биде изеден.

Потоа ги имаме низите **capSight**, **capSize** и **capSpeed** кои се користат за ограничување на големината, брзината на организмот и колку далеку може да гледа организмот. Првиот член, односно член со индекс 0 е минималната вредност, а вториот член, со индекс 1, е максималната вредност.

На крај ја имаме променливата **maxEnergy** која е најголемата енергија можна некој организам да има. Како енергија на организам се обработува ќе биде објаснето понатаму.

```
14 class Creature {
15   //~~~~~~~~~~~~~~~~~~~~~~ Setup code ~~~~~~~~~~~~
16
17   boolean alive;
18   float sight, size, speed, energy, fullEnergy;
19   float[] DNA = {0.0, 0.0, 0.0};
20   int breedingCooldown;
21
22   PVector pos, vel, acc, wonder;
```

Во класата Creature прво ги имаме декларирано следниве променливи: логичка променлива **alive** која чува дали организмот е жив или не; променливата **sight** за тоа колку може организмот да гледа; променливата **size** за големината на организмот; **speed** за брзината на организмот; **energy** за колку енергија има организмот; и променливата **fullEnergy** која чува колку е капацитетот на енергијата, односно максималната енергија која може да ја има некој организам.

Потоа ја имаме низата **DNA** која има 3 члена, со индекс 0 е големината, со индекс 1 е колку може да види и со индекс 2 е брзината на организмот. Ова е „генетскиот код“ на организмот кој ќе се користи за размножување. Исто така ја имаме декларирано променливата **breedingCooldown** која се користи за размножување.

Ги имаме и вектор променливите: **pos** за позиција; **vel** за брзина на организмот; **acc** е забрзувањето на организмот; и **wonder** е случајна точката во прозорецот која ќе биде објаснета понатаму.

```
Creature(float _x, float _y, float _size, float _sight, float _speed){
  alive = true;
  breedingCooldown = 0;

  size = constrain(_size, capSize[0], capSize[1]);
  sight = size + constrain(_sight, capSight[0], capSight[1]);
  speed = constrain(_speed, capSpeed[0], capSpeed[1]);

  energy = ( sight + pow(size, 3) * pow(speed, 2) ) / 2;
  fullEnergy = energy;

  DNA[0] = size;
  DNA[1] = sight;
  DNA[2] = speed;


  pos = new PVector(_x, _y);
  vel = PVector.random2D();
  acc = new PVector(0, 0);

  wonder = new PVector( random(width), random(height) );
}
```

Потоа имаме дефинирано конструктор со пет аргумента: позиција x, позиција y, големина, колку гледа, и брзината на организмот.

Во конструкторот променливата **alive** се поставува како true и на променливата **breedingCooldown** и е доделено вредност 0. Потоа променливите **size**, **sight** и **speed** и им доделува вредност од внесените аргументи, но пред тоа ограничени со функцијата **constrain()**. Само за променливата **sight** е збир од големината и внесената вредност поради тоа што организмот не гледа од неговиот центар, односно не гледа во него, туку од каде завршува телото на организмот до одредено растојание.

Следно ги имаме променливите **energy** и **fullEnergy** кои на почетокот имаат иста вредност. Енергијата ја пресметуваме со формулата:

$$E = \frac{h + b^3 + s^2}{2}$$

каде колку гледа организмот е h, големината е b и брзината е s. Така ја составив ова формула бидејќи сите тие влијаат некако на енергијата. Големината има најголем ефект бидејќи колку поголемо е некое тело, толку повеќе може да држи енергија; Исто така и големината на погледот и брзината имаат некакво влијание на енергетскиот капацитет, но со помал степен.

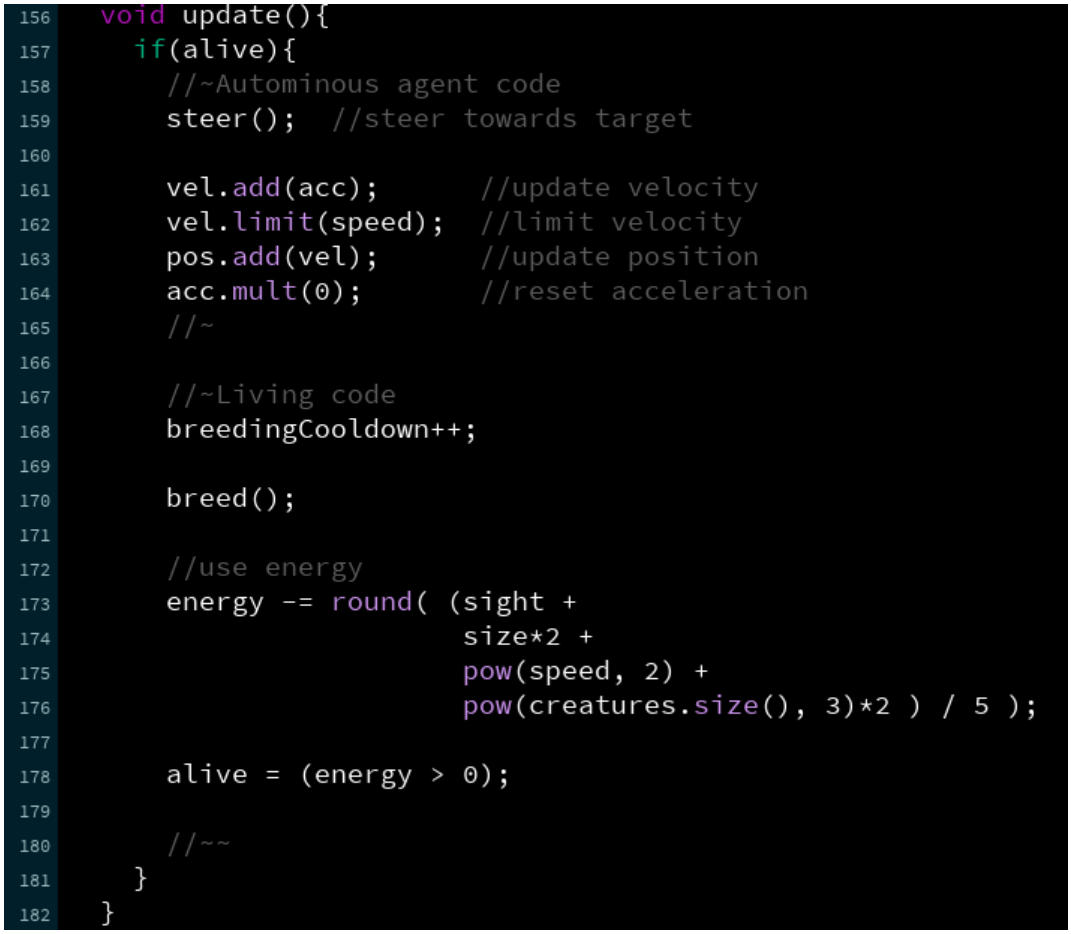
Следно што имаме е како вредностите за генетскиот код се зачувуваат во низата **DNA**. Овие се трите претходни карактеристики и се оние ќе се пренесува за следнава генерација.

Потоа следат променливите **pos**, **vel** и **acc**. Во променливата **pos** се креира нов PVector објект со внесените x и y координати за почетната позиција и во променливата **acc** се декларира нов PVector објект со координати (0,0), односно без забрзување. Во променливата **vel** од класата PVector ја користиме функцијата **random2D()** која враќа случаен два димензионален вектор.

На крај на **wonder** и доделуваме нов PVector објект со случајни координати некаде во прозорецот.

```
50  Creature(){
51    alive = true;
52    breedingCooldown = 0;
53
54    size = random(capSize[0], capSize[1]);
55    sight = size + random(capSight[0], capSight[1]);
56    speed = random(capSpeed[0], capSpeed[1]);
57
58    energy = ( sight + pow(size, 3) * pow(speed, 2) ) / 2;
59    fullEnergy = energy;
60
61    DNA[0] = size;
62    DNA[1] = sight;
63    DNA[2] = speed;
64
65
66    pos = new PVector( random(width), random(height) );
67    vel = PVector.random2D();
68    acc = new PVector(0, 0);
69
70    wonder = new PVector( random(width), random(height) );
71  }
```

Потоа го преклопуваме конструкторот без аргументи. Овде е исто како конструкторот со аргументите, но само што тука со функцијата **random()** при креирање Creature објект им доделуваме случајни вредности на променливите **size**, **sight**, **speed** и **pos**.



```
156   void update(){
157     if(alive){
158       //~Autominous agent code
159       steer();  //steer towards target
160 
161       vel.add(acc);      //update velocity
162       vel.limit(speed);  //limit velocity
163       pos.add(vel);      //update position
164       acc.mult(0);       //reset acceleration
165       //~
166 
167       //~Living code
168       breedingCooldown++;
169 
170       breed();
171 
172       //use energy
173       energy -= round( (sight +
174                         size*2 +
175                         pow(speed, 2) +
176                         pow(creatures.size(), 3)*2 ) / 5 );
177 
178       alive = (energy > 0);
179 
180       //~~
181     }
182   }
```

Методата **update()** ги содржи голем дел од пресметките што се извршуваат за организмот. Во главната програма за да не го испишувам сето ова во for циклус за организмите, само ќе ја повикам ова метода. Исто така е и по прегледно бидејќи сите овие променливи и методи се на едно место.

Прво во ова метода имаме проверка дали организмот е жив со if структурата. Во случај ако не е жив да се прескокне непотребно извршување на пресметки. Ако е жив прво што се повикува е методата **steer()** која поставува некоја вредност на променливата **acc**, но ќе биде објаснета понатаму. Потоа во линија 161 забрзувањето се додава на брзината **vel** и во линија 162 брзината со функцијата **limit()** се ограничува до брзината на организмот. Потоа брзината влијае на позицијата, и така брзината се додава на позицијата **pos**. И потоа се ресетира забрзувањето за следниот временски такт на 0 со функцијата **mult()**, односно **acc** се множи со 0.

Следно променливата **breedingCooldown** се инкрементира секој временски такт, односно брои колку временски такта поминале од последното размножување. Под тоа се повикува методата **breed()** која секој временски такт се обидува да се размножи организмот, ако условите се задоволени. Детално ќе биде објаснета методата понатаму.

Потоа ја имаме пресметката за искористена енергија. Секој временски такт организмот извршува некоја работа и со тоа користува некоја количина енергија. Ја Пресметуваме новата енергетска состојба E_{novo} со:

$$E_{novo} = E - \frac{h + 2b + s^2 + 2p^3}{5}$$

каде E е старата енергетска вредност, h е колку може да види, b е големината на организмот, s е брзината на организмот и p е големината на популацијата.

На крај се доделува нова вредност на променливата **alive**. Тука наместо да вршам проверка дали условот важи и според тоа да ја сменам вредноста во false или да ја чувам како што е, туку секој пат ја пресметувам вредноста. Ова го правам поефикасно и со помалку линии код.

```
145   void steer(){
146    PVector desire = desire();
147    desire.setMag(speed);
148
149    PVector steer = PVector.sub(desire, vel);
150    steer.mult(steeringStrenght);
151    acc.add(steer);
152   }
```

Со методата **steer()** го насочуваме организмот кон посаканата цел. Прво дефинираме вектор **desire** кој е насочен кон некоја посакана цел и му се доделува вратената вредност од методата **desire()**. Потоа со функцијата **setMag()** му се поставува величината да биде брзината на организмот. Потоа дефинираме нов вектор **steer** кој е деклариран како PVector од разликата помеѓу посаканата насока и моменталната насока на организмот. Со **mult()** се одредува колку ќе влијае

насочувањето. И на крај насочувачкиот вектор **steer** се додава на забрзувањето **acc**.

```
77  PVector desire(){
78    PVector target = null;
79    float ratio = -100000;
80
81    //~loop thru food and see which is the best target
82    for(int i = food.size() - 1; i >= 0; i-- ){
83      if(food.get(i).pos.dist(pos) <= sight){
84        if(ratio < food.get(i).energy / food.get(i).pos.dist(pos)){
85          ratio = floor(food.get(i).energy / food.get(i).pos.dist(pos));
86          target = food.get(i).pos;
87        }
88      }
89    }
```

Следната метода што ја има класата Creature е **desire()**. Во неа прво имаме декларирано променлива **target** и **ratio** со соодветни почетни вредности -100.000 и null. Во **target** ја чуваме позицијата на најдобрата точка на интерес, а во променливата **ratio** го зачувуваме највисокиот однос помеѓу енергетска вредност на храната и растојанието помеѓу организмот и таа храна. Со тоа ја одбираме најдобрата храна наместо најблиската.

Следно имаме for циклус со кој минуваме низ листата food. За секој член во листата прво проверуваме дали храната лежи внатре во видот на организмот, и ако тој услов е исполнет проверуваме дали досега зачуваниот однос е помал од односот на енергетската вредност на храната и оддалеченоста на храната од организмот. Ако тој услов е исполнет новиот однос го мемориираме во **ratio** и позицијата на тој член ја мемориираме во **target**.

```
93    for(int i = creatures.size() - 1; i >= 0; i-- ){
94      if(creatures.get(i).pos.dist(pos) <= sight &&
95         creatures.get(i) != this &&
96         creatures.get(i).size < size * eatableSize){
97
98        if(ratio < creatures.get(i).energy / creatures.get(i).pos.dist(pos)){
99          ratio = floor(creatures.get(i).energy / creatures.get(i).pos.dist(pos));
100         target = creatures.get(i).pos;
101       }
102     }
103   }
```

Исто така организмите можат да изедат други организми ако се со одреден процент поголеми од оние што ги јадат, па затоа имаме for циклус со кој минуваме низ листата creatures и за секој член вршиме проверка дали е во видот на организмот, проверуваме да не се проверува сам себе си и дали организмот е доволно поголем од членот што го проверуваме. Ако сите тие услови важат пак проверуваме дали новиот однос е поголем од стариот и, во случај ако важи условот, го зачувуваме новиот однос во **ratio** и ја зачувуваме позицијата на членот во **target**.

```
107     if(wonder.dist(pos) < 10){
108       wonder.x = random(width);
109       wonder.y = random(height);
110     }
111     //~~
112
113     target = (target == null) ? wonder : target;
114
115     return PVector.sub(target, pos);
116   }
```

Следно имаме if структура со кој проверуваме ако растојанието помеѓу позицијата на организмот и неговата точка за шетање **wonder** е помала од 10 пиксели, и во ако тој услов важи поставуваме нови координати на неа.

Во линија 113 имаме троен оператор кој доделува вредност на **target**. Ако во случај двете претходни прелистувања не најдат ниеден член од двете листи кој ги исполнува сите соодветни услови, организмот ќе треба да оди кон неговата точка за шетање. Во тројниот оператор проверуваме дали **target** има вредност null и ако има неговата вредност ќе биде поставена како **wonder**, но ако не важи условот си останува иста.

На крај методата враќа PVector објект што е разлика меѓу **target** и **pos**, односно враќа вектор кој покажува кон некоја точка на интерес.

```
120   void eat(){
121     //~loop thru food and see which is the best target
122     for(int i = food.size() - 1; i >= 0; i-- ){
123       if(food.get(i).pos.dist(pos) <= size/4){
124           energy += food.get(i).energy;
125           food.remove(i);
126       }
127     }
```

Со методата **eat()** организмот секој објект храна и секој организам кој е доволно блиску го „јаде“, односно ја додава енергијата од тој објект на својата и го отстранува од соодветната листа. Во ова метода прво што имаме е for циклус кој прелистува низ листата food и за секој член се проверува дали е доволно блиску за да биде изеден. Ако условот важи на својата енергија си ја додава енергетската вредност на членот и потоа тој член го отстранува од листата.

```
131     for(int i = creatures.size() - 1; i >= 0; i--){
132       if(creatures.get(i) != this &&
133          creatures.get(i).pos.dist(pos) <= size/4 &&
134          creatures.get(i).size < size * eatableSize){
135           energy += creatures.get(i).energy;
136           creatures.remove(i);
137       }
138     }
139     //~~
140   }
```

И потоа имаме for циклус кој минува низ листата creatures. За секој член се проверува да не е самиот себе организмот што ја извршува ова метода, дали е доволно блиску да биде изеде и дали членот е доволно помал за да биде изеде. Ако сите услови се исполнети енергијата на членот му се додава на организмот и потоа тој член се отстранува од листата.

```
185  void breed(){
186    float[] newDNA = {0.0, 0.0, 0.0};
187    if(breedingCooldown >= 10 * frameRate &&
188       energy >= 9000 &&
189       random(1) < breedingChance){
190        newDNA[0] = constrain(floor(DNA[0] + DNA[0] * random(-0.05, 0.05)), capSize[0], capSize[1]);
191        newDNA[1] = constrain(floor(DNA[1] + DNA[1] * random(-0.05, 0.05)), capSight[0], capSight[1]);
192        newDNA[2] = constrain(floor(DNA[2] + DNA[2] * random(-0.05, 0.05)), capSpeed[0], capSpeed[1]);
193
194        energy -= 10000;
195
196        breedingCooldown = 0;
197
198        creatures.add( new Creature(pos.x + random(-size, size), pos.y  + random(-size, size),
199                                    newDNA[0], newDNA[1], newDNA[2])
200                     );
201    }
202  }
```

Методата **breed()** служи за размножување, односно за креирање на нов организам со слични „генетски код“. Во неа прво што имаме е дефинирање на низа **newDNA** во која ќе се пресметаат и зачуваат карактеристиките на новиот организам.

Следно имаме if структура која проверува дали поминало доволно време од претходното размножување, дали организмот има доволно енергија за да се размножи и дали со функцијата **random()** во тој временски такт успешно е размножувањето. Ако сите овие услови се исполнети се копира низата **DNA** во **newDNA** со случајни промени од ±5%.

Потоа се одзема 10.000 од енергијата бидејќи во реалноста за размножување потребно е голема инвестиција на енергија и ресурси. Исто така **breedingCooldown** се праќа пак на 0.

И на крај во листата creatures се додава нов организам со новото пресметани гени и со позиција близу до позицијата на организмот родител, но со некоја случајна варијација.

```
206  void die(){
207    int nof = floor(size/5);
208
209    for(int i = 0; i < nof; i++){
210      if(i <= nof * 0.7){  //70% will be good food
211        food.add( new Food( pos.x + random(-size/2, size/2),
212                            pos.y + random(-size/2, size/2),
213                            map(fullEnergy, 0, maxEnergy, 20, 50) ) );
214      } else {             //30% will be bad
215        food.add( new Food( pos.x + random(-size/2, size/2),
216                            pos.y + random(-size/2, size/2),
217                            map(fullEnergy, 0, maxEnergy, -30, -1) ) );
218      }
219    }
220
221    creatures.remove(this);
222  }
```

Со методата **die()** организмот се отстранува од листата creatures и во негово место според неговата големина се поставуваат објекти храна. Во неа прво имаме дефинирано целобројна променлива **nof** и на неа и се доделува една петина од големината на организмот. Ова променлива ни означува колку објекти храна треба да бидат ставени на местото на организмот кога умре.

Следно имаме for циклус кој се пофтара онолку пати колку што е вредноста на **nof**. Во него имаме if else структура во кој проверуваме ако бројачот **i** надминал 70% од вкупниот број храна, односно со ова структура осигуруваме 70% од храната што ќе се постави ќе биде со висока енергетска вредност, а рестото ќе бидат со ниска енергетска вредност. Во двата случаја ако условот е или не е исполнет во листата food се додава нов храна објект со случајна позиција околу позицијата на организмот и енергетска вредност што зависи од максималната енергија на организмот **fullEnergy**.

На крај се отстранува изумрениот организам од листата creature со функцијата **remove()**.

```
228   void show(){
229     push();
230     //~calculate colour based on energy
231     color fillCol = lerpColor( color(220, 10, 0, 150),
232                                color(220, 220, 0, 150),
233                                map( constrain(energy, 0, fullEnergy),
234                                     0, fullEnergy, 0, 1 )
235                              );
236
237     color strokeCol = lerpColor( color(220, 10, 0, 200),
238                                  color(230, 220, 0, 250),
239                                  map( constrain(energy, 0, fullEnergy),
240                                       0, fullEnergy, 0, 1 )
241                                );
242     //~~
243
244     stroke(strokeCol);           //set stroke colour
245     strokeWeight(3);             //set stroke size
246     fill(fillCol);               //set fill colour
247     circle(pos.x, pos.y, size);  //draw the creature
248
249     pop();
250   }
251   //~
252 }
```

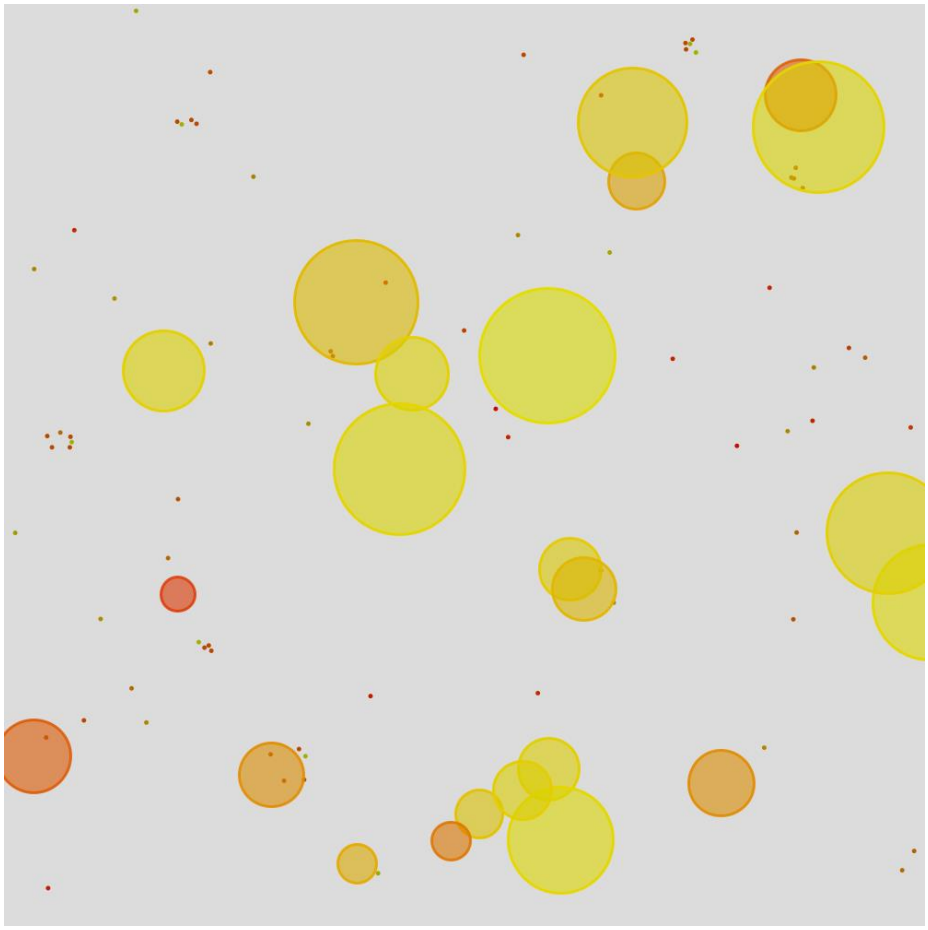
Методата **show()** служи за да се исцрта организмот на прозорецот. Телото на методата е обиколена со функциите **push()** и **pop()** за да не дојде до несакани промени на стилот што можат да влијаат во други делови на кодот.

Потоа има декларирано променливи **fillCol** и **strokeCol** од вид color и двете имаат доделена вредност со користење на функцијата **lerpColor()**. Секое исцртување бојата на контурната линија и на внатрешноста зависат од **energy**, односно со ниска енергија бојата на организмот ќе е црвена, а со висока енергија ќе биде обоена жолта боја.

Следно со **stroke()** се поставува бојата на контурната линија и со **strokeWeight()** се поставува дебелината. Со функцијата **fill()** се поставува бојата на внатрешноста. И на крај со **circle()** се исцртува круг со центар координати од позицијата **pos** и радиус големината на организмот **size**.

5. Заклучок

Се комплетно, резултатот што се добива со ова програма изгледа како како на сликата подолу. Ова програма ни дава слика на тоа како може да се доведе до некои клучни карактеристики кои се возвишени низ популацијата што им овозможува најдобри шанси за да преживеат организмите и да создадат потомство. Но исто така со менување на вредностите во програмата може да се врши и истражување за тоа какви адаптации ќе се појават во разни услови.



6. Користена литература

https://processing.org/

https://natureofcode.com/book/chapter-6-autonomous-agents/