CREATIVE PROGRAMMING AND COMPUTING

Lab 8:

Evolutionary Systems

Luca Comanducci

luca.comanducci@polimi.it

CONTENTS

ECOSYSTEM SIMULATION

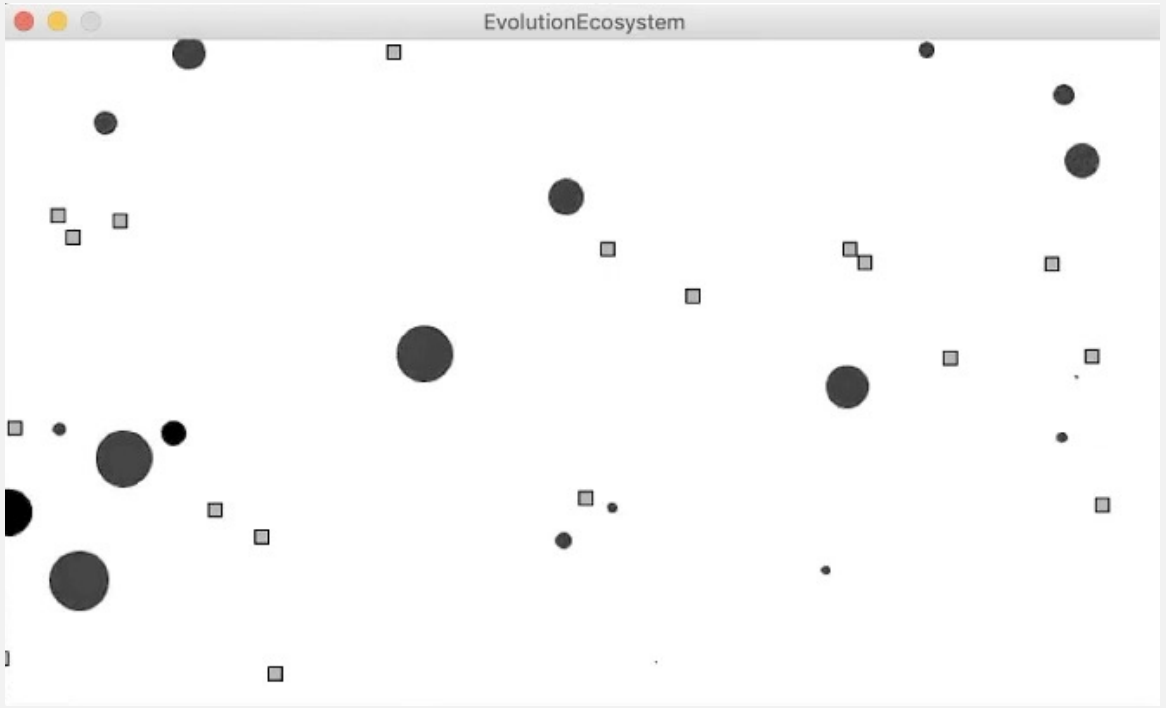
- You will find the code in the folder:

  "*Ex_1_EcosystemSimulation*"

ECOSYSTEM SIMULATION

- In real world we have ‘’survival of the survivors’’
- We might use a genetic algorithm to develop a simulation of a natural system

- Simple Scenario
  - Bloops moves about the screen according to perlin noise
  - Bloops reproduce asexually
  - Their lifespan depends on the food that they eat



  - Each time a bloop ‘’eats’’ it sends a message to SuperCollider via OSC and plays a note depending on its genes

ECOSYSTEM SIMULATION

- **Bloop Class**
  - Circle that moves according to Perlin noise
  - Attributes:
    - Radius
    - Maximum speed
    - (.. Some details are missing)

```
class Bloop {
  PVector location;                                   A location

  float r;                                            Variables for size and speed
  float maxspeed;
  float xoff, yoff;                                   Some variables for Perlin noise calculations

  void update() {
    float vx = map(noise(xoff),0,1,-maxspeed,maxspeed);
    float vy = map(noise(yoff),0,1,-maxspeed,maxspeed);
    PVector velocity = new PVector(vx,vy);            A little Perlin noise algorithm to calculate a velocity
    xoff += 0.01;
    yoff += 0.01;

    location.add(velocity);                           The bloop moves.
  }

  void display() {                                    A bloop is a circle.
    ellipse(location.x, location.y, r, r);
  }
}
```

ECOSYSTEM SIMULATION

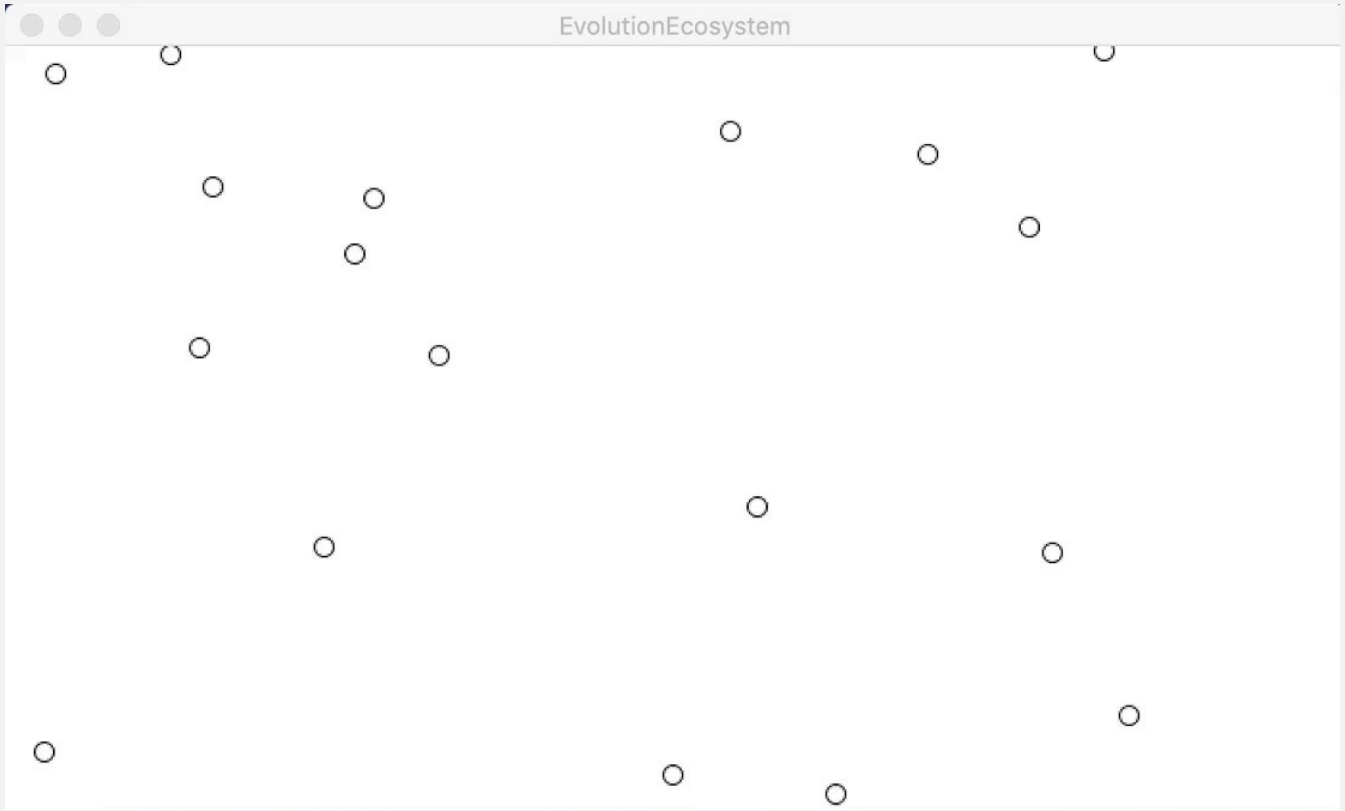
- **World Class**
  - Manages all the elements of the Bloop's world
  - Bloops population stored as ArrayList

(array of objects, but with an ArrayList, items can be easily added and removed and resized dynamically)

```
class World {

  ArrayList<Bloop> bloops;                    A list of bloops

  World(int num) {
    bloops = new ArrayList<Bloop>();

    for (int i = 0; i < num; i++) {
      bloops.add(new Bloop());                Making an initial population of bloops
    }
  }
```

ECOSYSTEM SIMULATION

- Right now what we have is a very simple particle system



ECOSYSTEM SIMULATION

- **LifeCycle PART I**
  - We need to add the features:
    - Bloops die → replacement for a fitness function
    - Bloops are born
  - Notice that a health variable exists in the Bloop class

```
class Bloop {
  float health;
```

  - Bloops lose some health in each frame of animation,
    - modify the update() function in order to reflect this behavior
    - Modify the display() function using stroke() and fill()

      in order to color the bloop depending on its health
  - If health drops to 0→ bloop dies

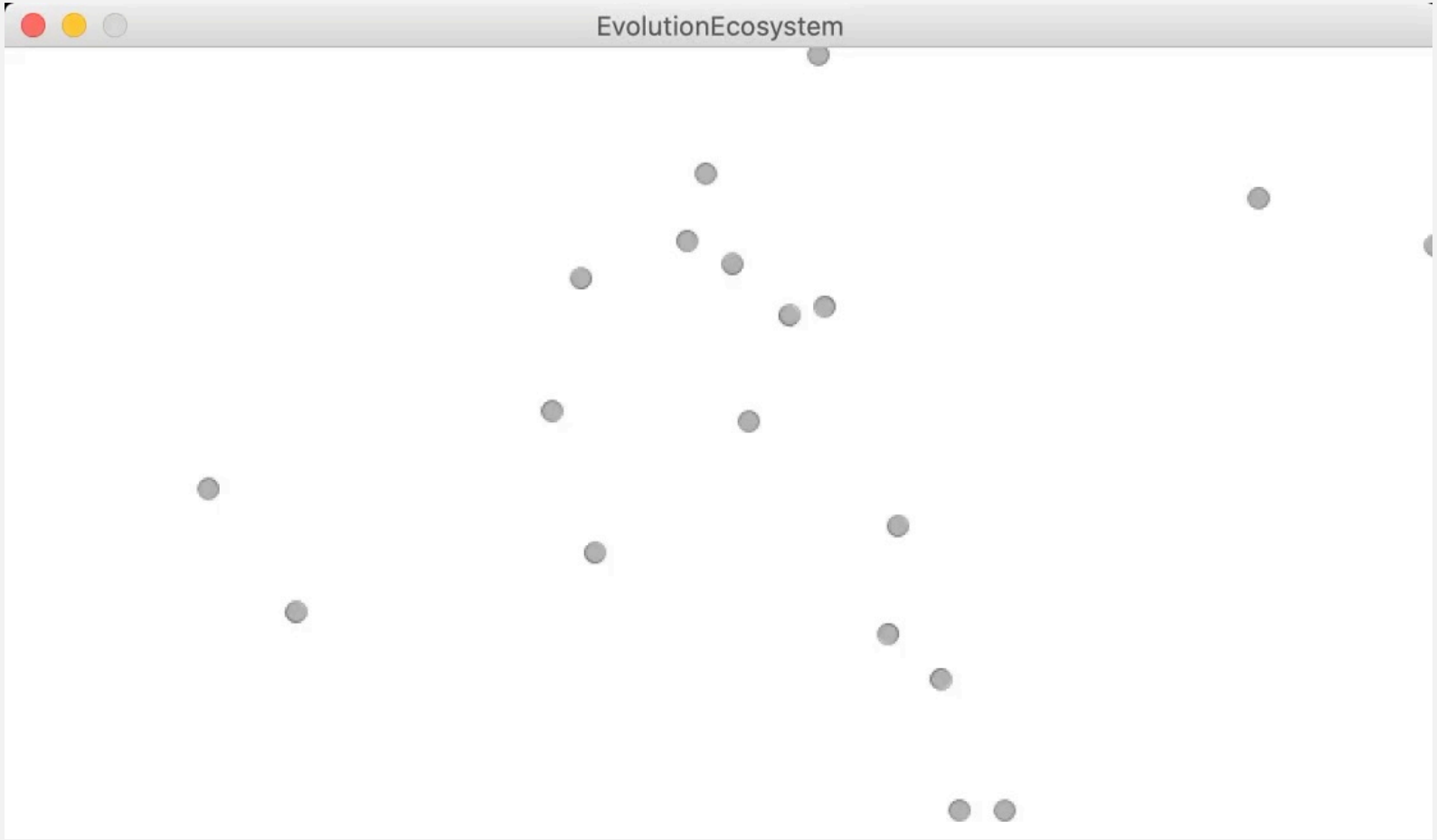
```
boolean dead() {
  if (health < 0.0) {
    return true;
  }
  else {
    return false;
  }
}
```

Bloop.pde

```
void update() {
    // Simple movement based on perlin noise
    float vx = map(noise(xoff),0,1,-maxspeed,maxspeed);
    float vy = map(noise(yoff),0,1,-maxspeed,maxspeed);
    PVector velocity = new PVector(vx,vy);
    xoff += 0.01;
    yoff += 0.01;

    position.add(velocity);

    // FILL THE CODE -> decrease health
    //…… //FILL THE CODE
  }

// ...
// ...

// Method to display
 void display() {
    ellipseMode(CENTER);
    // ... //FILL THE CODE
    // ... //FILL THE CODE
    ellipse(position.x, position.y, r, r);
  }
```

ECOSYSTEM SIMULATION

- **LifeCycle PART I**
  - You should be able to get this kind of behavior



ECOSYSTEM SIMULATION

- **LifeCycle – Part II**
  - Right now all bloops live exactly the same amount of time and die together
  - To simulate different lifespans, we introduce a very simple idea, ***Food***
    - If the bloop eats food, it lives longer
    - If the bloop does not eat, it eventually dies
  - Food can be defined as ar ArrayList of Pvectors, then we can test the following:

    - In the World class, uncomment in the void run() function the food.run() line.
      - Now you if you re-run you should be able to see food appearing in the environment

ECOSYSTEM SIMULATION

- **LifeCycle – Part II**
  - Having food is not enough
  - In order to simulate a real ecosystem we need the bloops to eat it

Bloop.pde

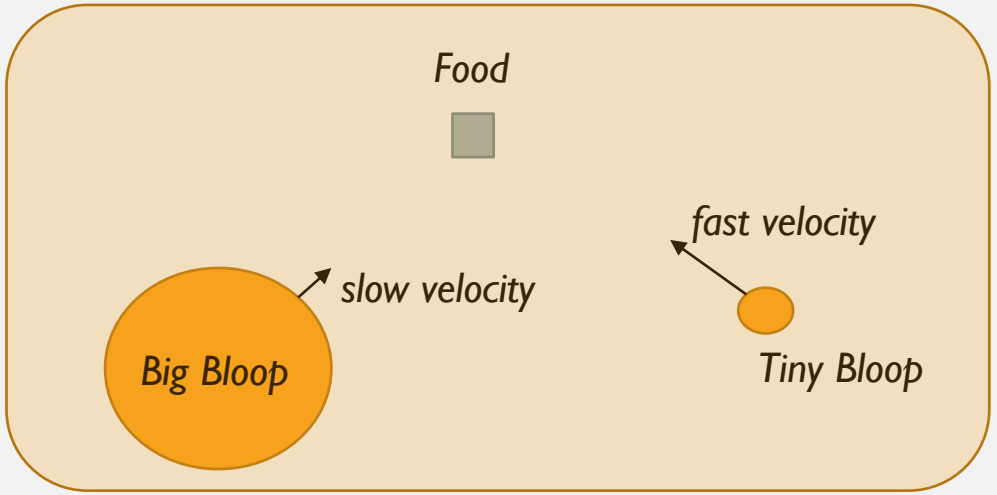
```
void eat() {
    ArrayList<PVector> food = f.getFood();
    // Are we touching any food objects?
    for (int i = food.size()-1; i >= 0; i--) {
      PVector foodposition = food.get(i);
      float d = PVector.dist(position, foodposition);
      // If we are, juice up our strength!
      if (d < r/2) {
           // FILL THE CODE: increase health by 100

           // FILL THE CODE: remove the food element from the world
           // HINT: you can remove elements from an ArrayList using
           // .remove(idx)

        // ... There is some other code here, for now do not mind
        //       about it

      }
    }
  }
```

World.pde

```
void run() {
    // Deal with food
    food.run();

    // Cycle through the ArrayList backwards b/c we
are deleting
    for (int i = bloops.size()-1; i >= 0; i--) {
      // All bloops run and eat
      Bloop b = bloops.get(i);
      b.run();

      /* UNCOMMENT */
      /* b.eat(food);
      // If it's dead, kill it and make food
      if (// FILL THE CODE: check bloop is dead) {
        // FILL THE CODE: remove bloop
        // FILL THE CODE: add food in the same
           position where the bloop died
      }      */
/...
  }
```

ECOSYSTEM SIMULATION

- **Genotype and Fenotype**
  - Ability of Bloop in finding food depends on
    - *Size*
    - *Speed*
  - These two variables are inversely related
    - ***Genotype*** is a scalar




```
class DNA {

  float[] genes;

  DNA() {


    genes = new float[1];

    for (int i = 0; i < genes.length; i++) {
      genes[i] = random(0,1);
    }
  }
```

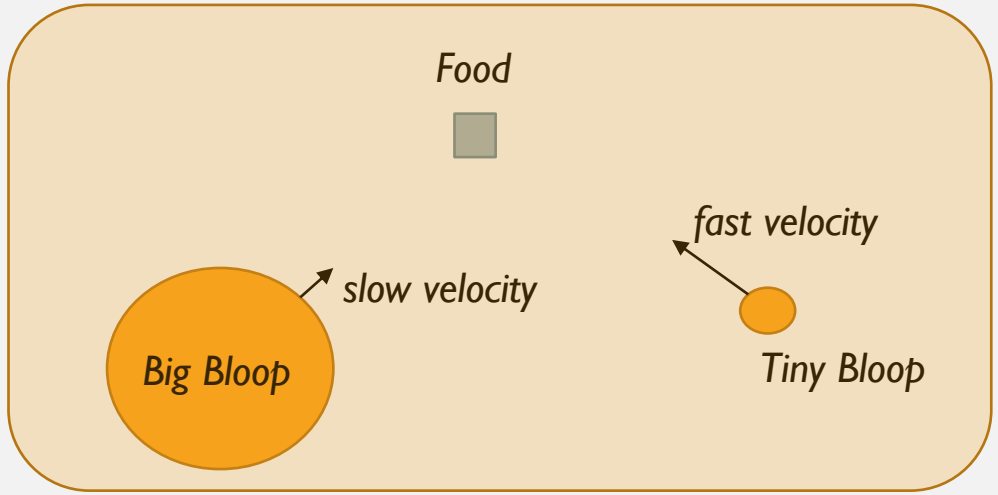
ECOSYSTEM SIMULATION

- **Genotype and Fenotype**
  - ***Fenotype***
    - is the bloop itself, whose size and speed are assigned by adding an instance of a DNA object to the bloop class
    - Modify the bloop constructor
    - Use the dna.genes[0] and map them to maxspeed and r values

Map maxspeed between 15 and 0

Map r between 0 and 50

HINT: use
map(value, start1, stop1, start2, stop2)

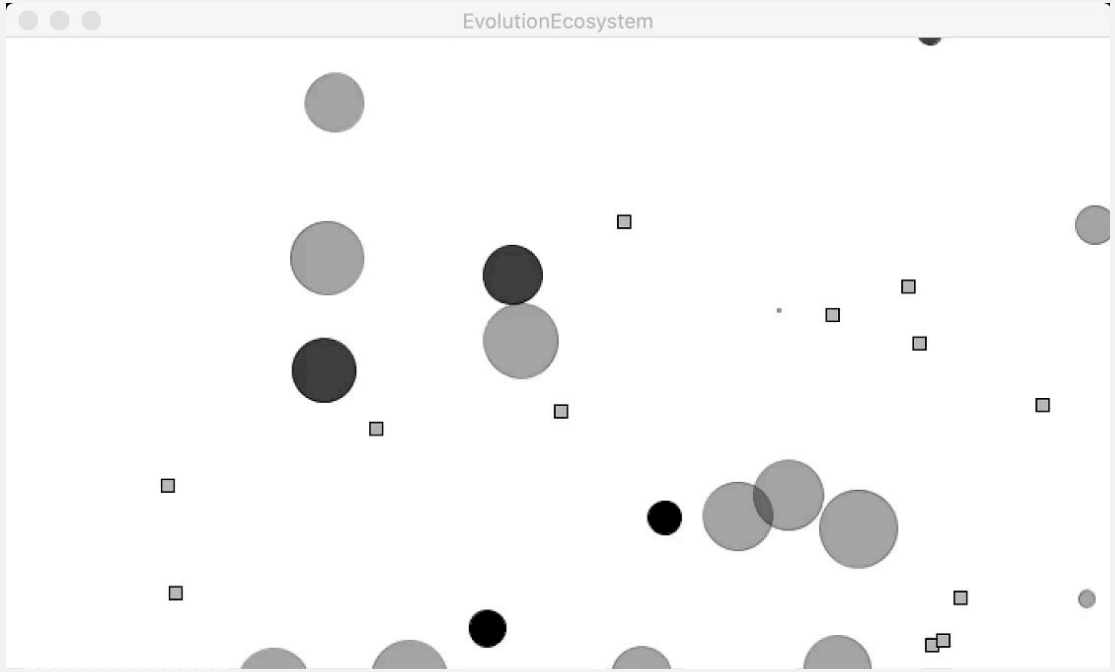


Bloop.pde

```
// Create a "bloop" creature
  Bloop(PVector l /*, … */ ){ FILL THE CODE ADD DNA OBJECT
    position = l.get();
    health = 100; //FILL THE CODE
    xoff = random(1000);
    yoff = random(1000);
    // dna = dna_; UNCOMMENT
    // Gene 0 determines maxspeed and r
    // The bigger the bloop, the slower it is
    maxspeed = 5; // FILL THE CODE
    r = 10; // FILL THE CODE

  }
```

ECOSYSTEM SIMULATION

- **Genotype and Fenotype**
  - Finally modify the World constructor in order to take into account the DNA in the bloop generation



World.pde

```
  // Constructor
  World(int num) {
    // Start with initial food and creatures
    food = new Food(num);
    bloops = new ArrayList<Bloop>();                //
Initialize the arraylist
    for (int i = 0; i < num; i++) {
      PVector l = new PVector(random(width),random(height));
      // FILL THE CODE → instantiate DNA object
      bloops.add(...); // FILL THE CODE! Take into account DNA
    }
  }
```

- Now you can see bloops of different size going around and eating

ECOSYSTEM SIMULATION

- **Selection and Reproduction**
  - Simple method: ”*Asexual reproduction*”
    - A bloop does not require a partner
    - It can make a clone of itself at any moment:

      At any given moment, a bloop has a $x\%$ chance of reproducing
    - The longer the bloop lives the more likely it will make children
    - Reproduction is simply implemented by ”copying” the parent DNA

```
class DNA {

  DNA copy() {

    float[] newgenes = new float[genes.length];

    arraycopy(genes,newgenes);
    return new DNA(newgenes);
  }
}
```

ECOSYSTEM SIMULATION

- **Selection and Reproduction**
  - Fiill the reproduce() function in Bloop.pde
  - Reproduction happens with a probability

    hint: use random, do not use a too high probability

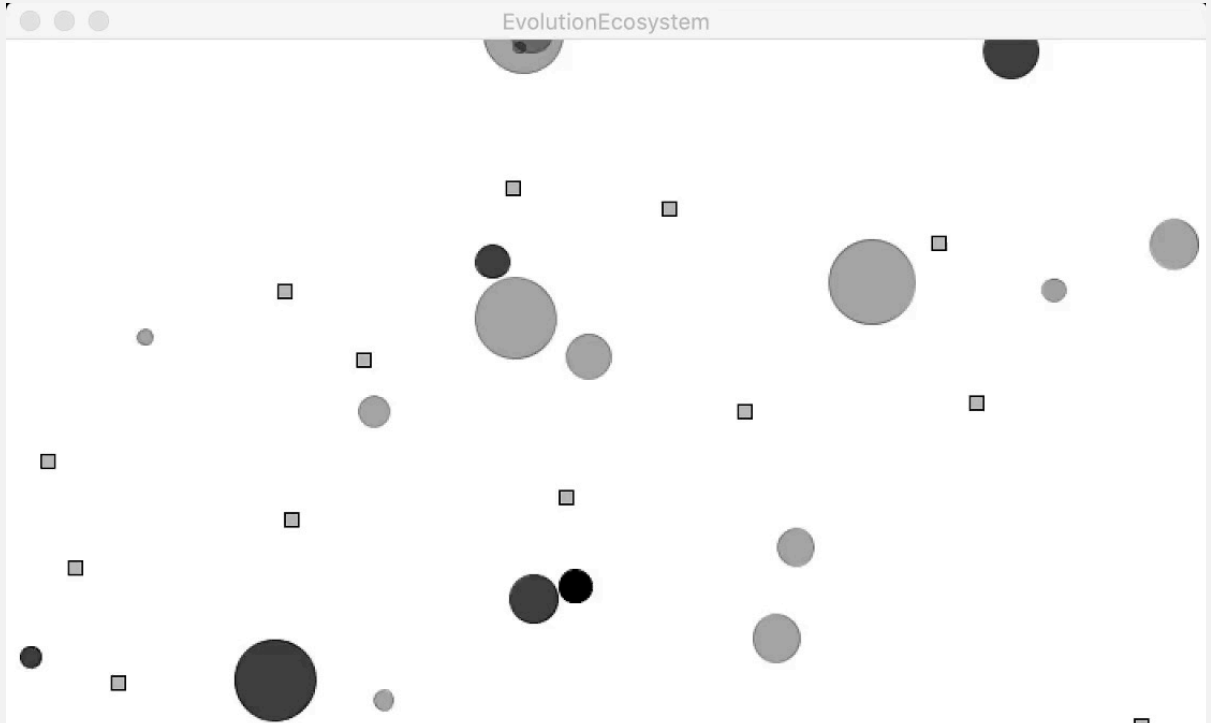
    e.g. p < $5 * 10 - 4$
  - Copy child DNA from parent
  - Mutate the child DNA with a higher probability than reproduction (e.g. 0.1)

Bloop.pde

```
Bloop reproduce() {
   // asexual reproduction
   if (*/FILL THE CODE*/) {
     // Child is exact copy of single parent
     //DNA childDNA = // FILL THE CODE

     // Child DNA can mutate
     // FILL THE CODE

     return new Bloop(position, childDNA);
   }
   else {
     return null;
   }
 }
```

ECOSYSTEM SIMULATION

- **Selection and Reproduction**
- In the World.pde function run() uncomment the lines and complete them
- And now you should finally be seeing bloops eating and reproducing



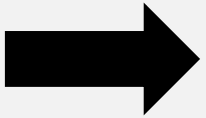
World.pde

```
void run() {
  // Deal with food
  food.run();

      ..


   // Perhaps this bloop would like to make a baby?
   //Bloop child = // FILL THE CODE → perform reproduction
   // FILL THE CODE --> if reproduction is successful add bloop
   // to the ecosystem

  }
}
```

ECOSYSTEM SIMULATION

- **Sonification**
  - We will send OSC messages from Processing to Supercollider each time a bloop eats some food
  - We map the genes to control *note pitch, reverb mix* and *room*

```
//…
OscMessage msg = new
OscMessage("/synth_control");

// map idx of the scale
        msg.add(/*FILL THE CODE */);

        // map reverb mix
        msg.add(/*FILL THE CODE */);
        // map reverb room
        msg.add(/*FILL THE CODE */);

        oscP5.send(msg, location);



print(int(map(dna.genes[0],0,1,0,7)));
//…
```




```
//…
x = OSCFunc( { | msg, time, addr, port |
    var pyIdx, pyRoom,pyMix, scale, offset, freq;
     // Parse message
     pyIdx= msg[1].asFloat; // Idx of note on the scale
     pyMix= msg[2].asFloat; // Reverb mix
     pyRoom= msg[2].asFloat; // Room size (reverb)

     offset = 60; // midi note offset
     scale = Scale.hijaz; // selected scale
     freq = (scale[pyIdx]+ offset).midicps;

     // Play the synth!!!
     Synth(\SimpleNote,[freq:freq,mix:pyMix,room:pyRoom]);

     // Print some info
     ("Message received from Processing").postln
     ( "Note idx is " + pyIdx ).postln;
     ( "Reverb Mix is " + pyMix).postln;
     ( "Room Size is " + pyRoom).postln;
}, '/synth_control' );
//…
```

ECOSYSTEM SIMULATION

- **Sonification**
  - Uncomment the following lines of code in EvolutionEcosystem.pde

Ex_I_EvolutionEcosystem.pde

```
 //import oscP5.*;
//import netP5.*;

//OscP5 oscP5;

//int port = 57120;
//NetAddress location;


World world;

void setup() {
  size(700, 400);
  // World starts with 20 creatures
  // and 20 pieces of food
  //location = new NetAddress("127.0.0.1",port);
  //oscP5 = new OscP5(this,5500);
```

ECOSYSTEM SIMULATION

- **Sonification**
  - Fill the bloop.pde eat function sending messages to SC
  - Map dna.genes[0] to elements from 0 to 7 in order to represent notes on a scale

    HINT: use map, it must be an integer value!
  - Use dna.genes[0] to control reverb mix
  - Use dna.genes[0] to control reverb room
  - If you run supercollider you should be able to hear the generated sound!

Bloop.pde

```
void eat(Food f)
OscMessage msg = new OscMessage("/synth_control");

// map idx of the scale
        msg.add(/*FILL THE CODE */);

        // map reverb mix
        msg.add(/*FILL THE CODE */);
        // map reverb room
        msg.add(/*FILL THE CODE */);

        oscP5.send(msg, location);


        print(int(map(dna.genes[0],0,1,0,7)));
```

ECOSYSTEM SIMULATION

- **Bonus: generate bloops by clicking on the canvas**
  - Add the following lines of code at the end of Ex_I_EvolutionEcosystem.pde

Ex_I_EvolutionEcosystem.pde

```
  void mousePressed() {
  world.born(mouseX,mouseY);
}

void mouseDragged() {
  world.born(mouseX,mouseY);
}
```

  - And the following function to World.pde

World.pde

```
 // FILL THE CODE
  void born(float x, float y) {
    PVector l = new PVector(x,y);
    DNA dna = new DNA();
    bloops.add(new Bloop(l,dna));
  }
```

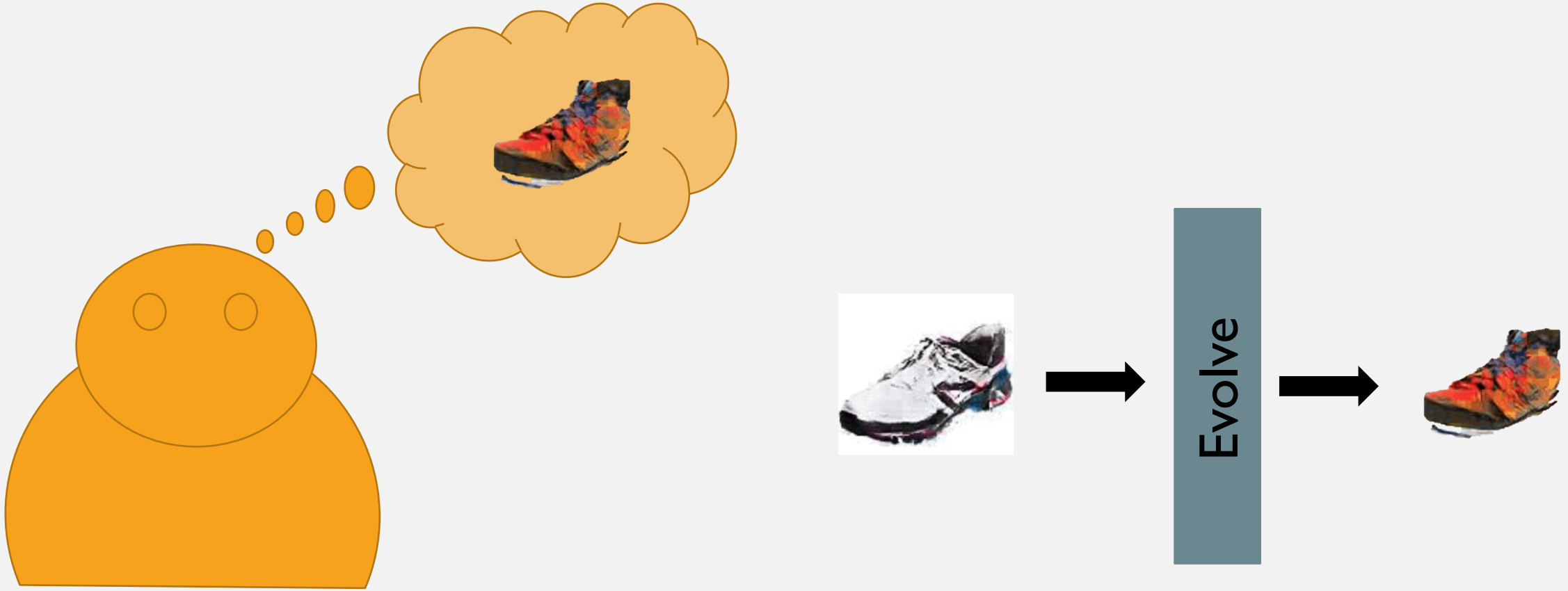
DEEP INTERACTIVE EVOLUTION

- You will find the code in the file:

  "Ex_II_Deep_interactive_evolution.*ipynb*"

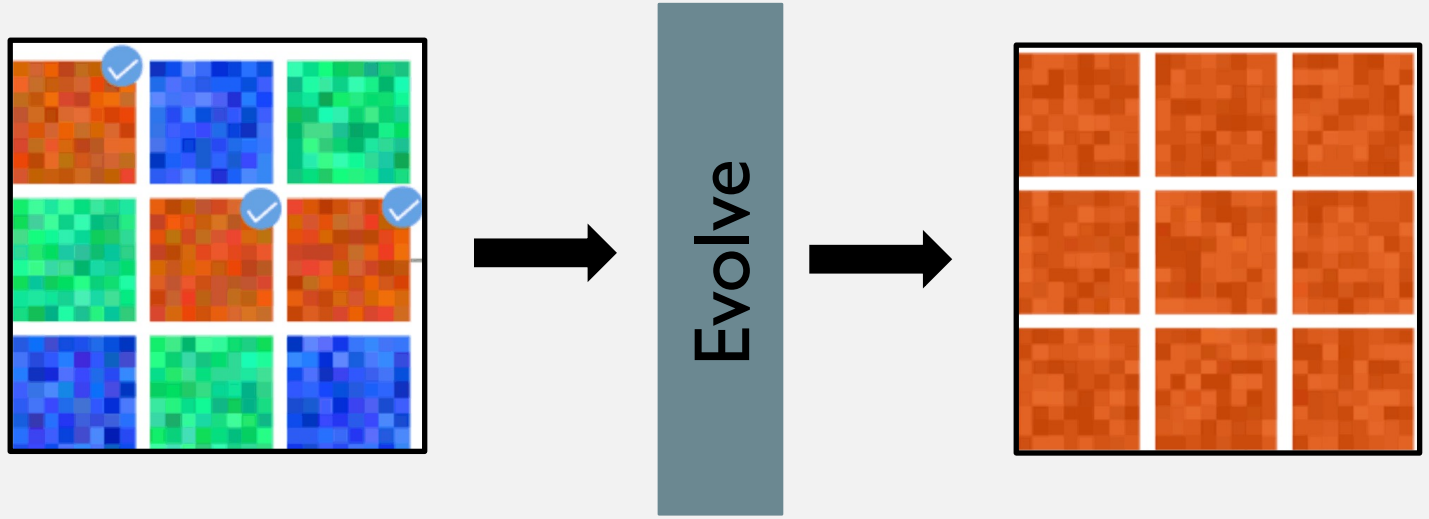
DEEP INTERACTIVE EVOLUTION

- **Combine Deep Generative models and Evolutionary Computation**
  - How can a model take user preferences into account during the generation process?



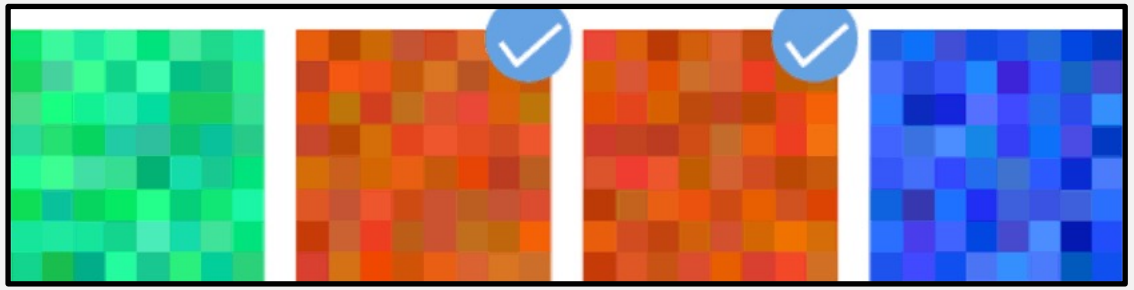
DEEP INTERACTIVE EVOLUTION

- **Genetic Algorithms**
  - The system that we want does the following:
    1. *Start with a batch of random samples*
    2. *Ask the user to select the preferred ones*
    3. *Generate samples with features similar to those of the selected samples*
    4. *Repeat 2-3 until a sample has all desired attributes*



DEEP INTERACTIVE EVOLUTION

- **Genetic Algorithms**
  - We are performing an optimization process
  - *Fitness function is not differentiable, since it is user-defined!*



$$\rightarrow \quad [0, 1, 1, 0]^T$$

  - Selected choices can be represented as a vector of 1's and 0's

  - No back-propagation is possible −> we resort to Genetic Algorithms

DEEP INTERACTIVE EVOLUTION

- **Interactive Evolutionary Computation(IEC)**
  - genetic optimization in response to user inputs
  - Genetic algorithms, combine features of the best samples (+ mutation) to produce better ones

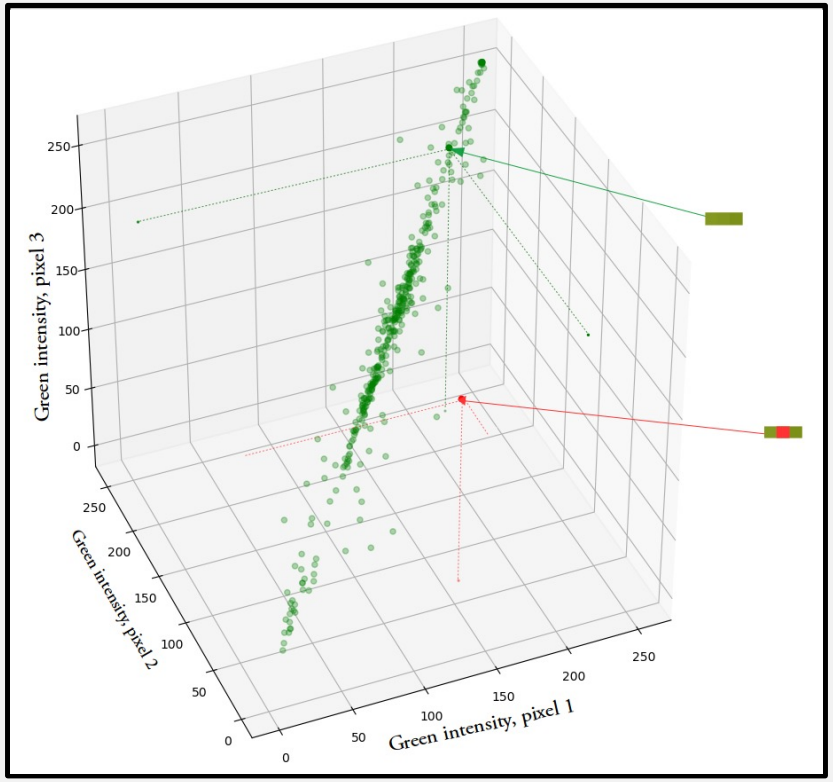
  - Hard to define what is a *feature*
    - If each single pixel is a *feature*:

DEEP INTERACTIVE EVOLUTION

- Genetic algorithm require features to be as much independent as possible
- For most data types (e.g. images, sound, text) this is not true
- E.G. pixels:

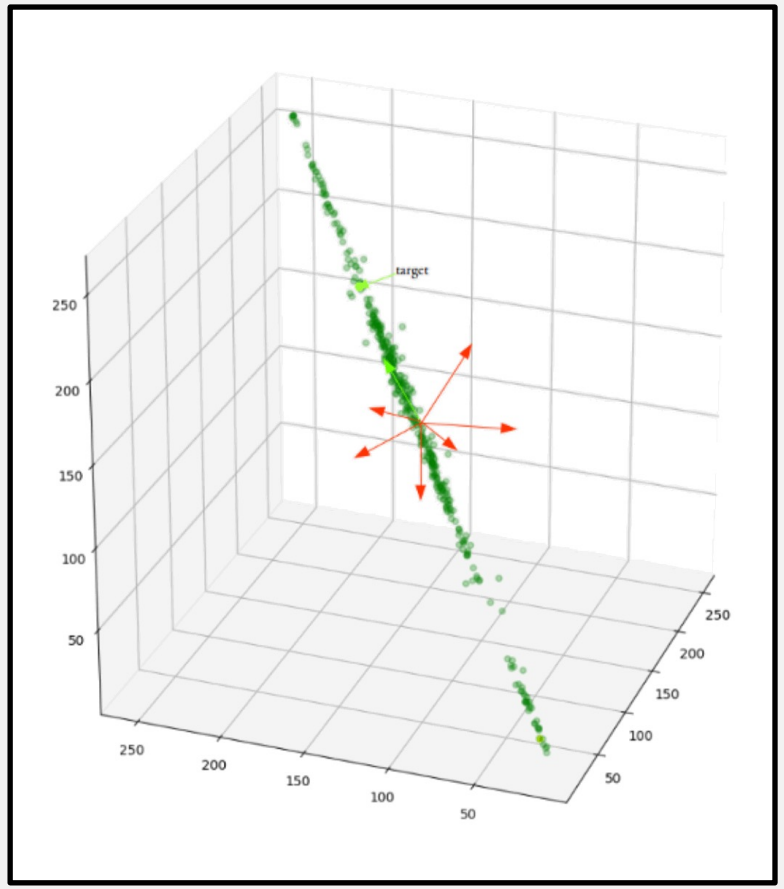
Likely

Unlikely



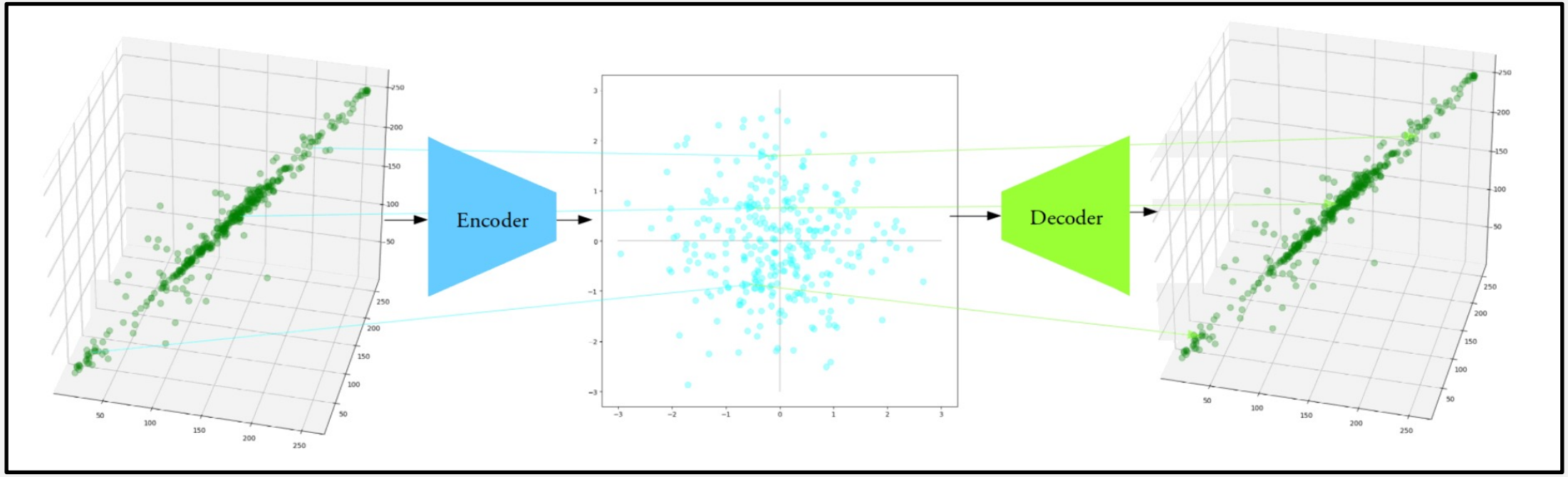
DEEP INTERACTIVE EVOLUTION

- Most Data lie on a small subspace of the total space used to represent it → manifold
- Only this data represents information that would be considered realistic
- Genetic algorithms randomly explore the search space
- Human judgement-based approach is not going to converge in a reasonable amount of time



DEEP INTERACTIVE EVOLUTION

- **Idea**
  - *Optimize the latent vectors of generative models instead of the Raw data*

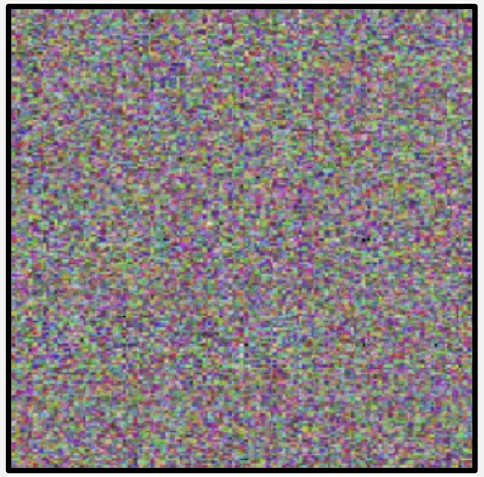


  - Merge IEC with Deep Generative Models
  - General Solution

DEEP INTERACTIVE EVOLUTION

- **Idea**
  - Latent vectors are typically sampled from a gaussian distribution
  - Components are independent of each other
  - Decoder learns to map certain regions of latent space to realistic saples

Sampling from latent space



Sampling from input space

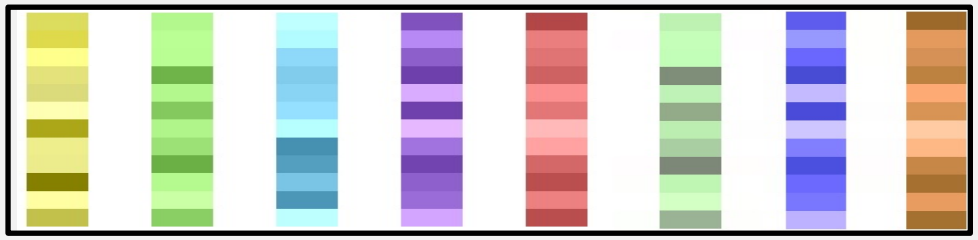
- *One can freely recombine features and add random noise to them, while still generating realistic data*

DEEP INTERACTIVE EVOLUTION

- ***Algorithm***

  A. Sample

  - Sample n random vectors from distribution

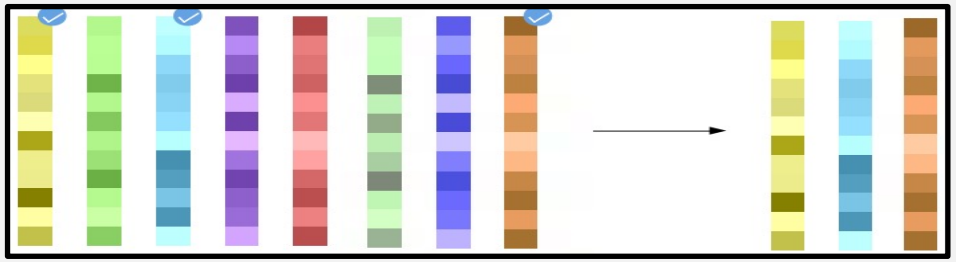


  B. Generate

  - Repeat the following:

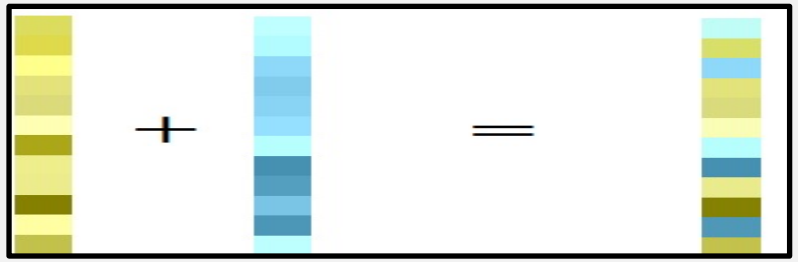
    i. Selection

    - user selects best samples



    ii. Crossover

    - Crossover between selected latent vectors



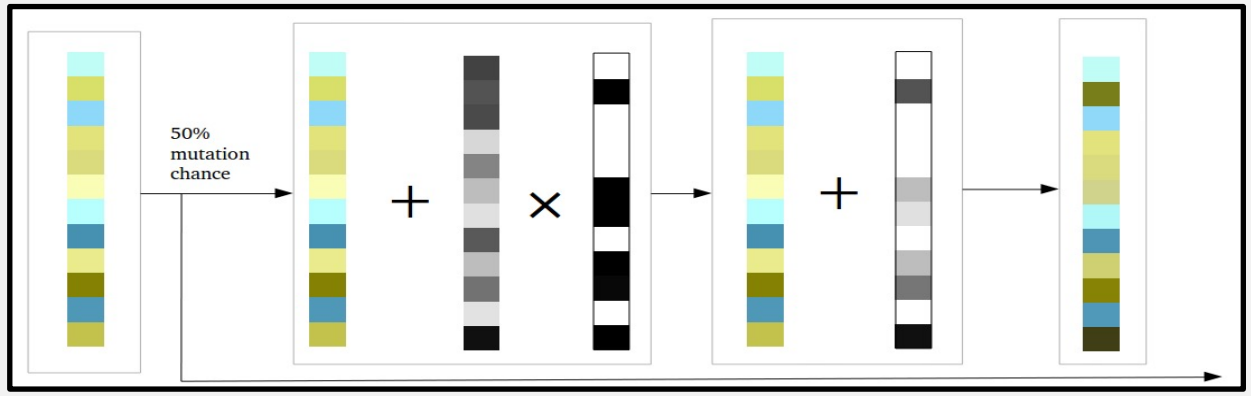
DEEP INTERACTIVE EVOLUTION

- ***Algorithm***

  B. Generate

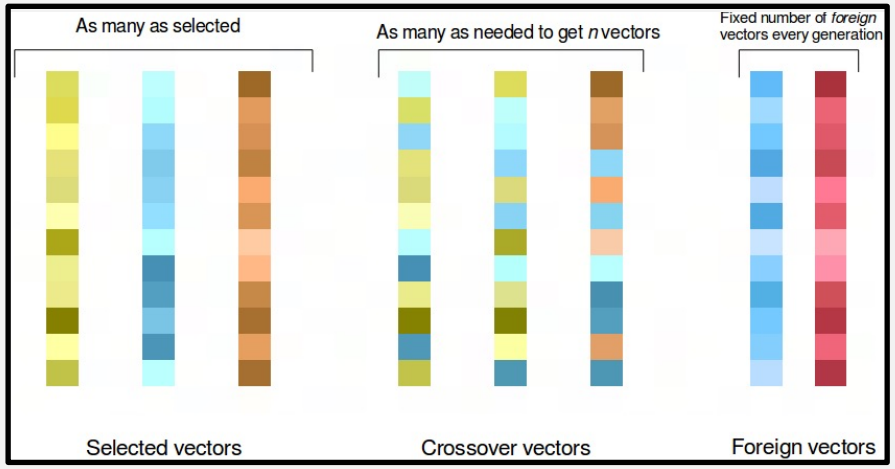
  iii. Mutation

  - Mutate all latent vectors



iv. Foreign Vectors

- Randomly generated vectors



DEEP INTERACTIVE EVOLUTION

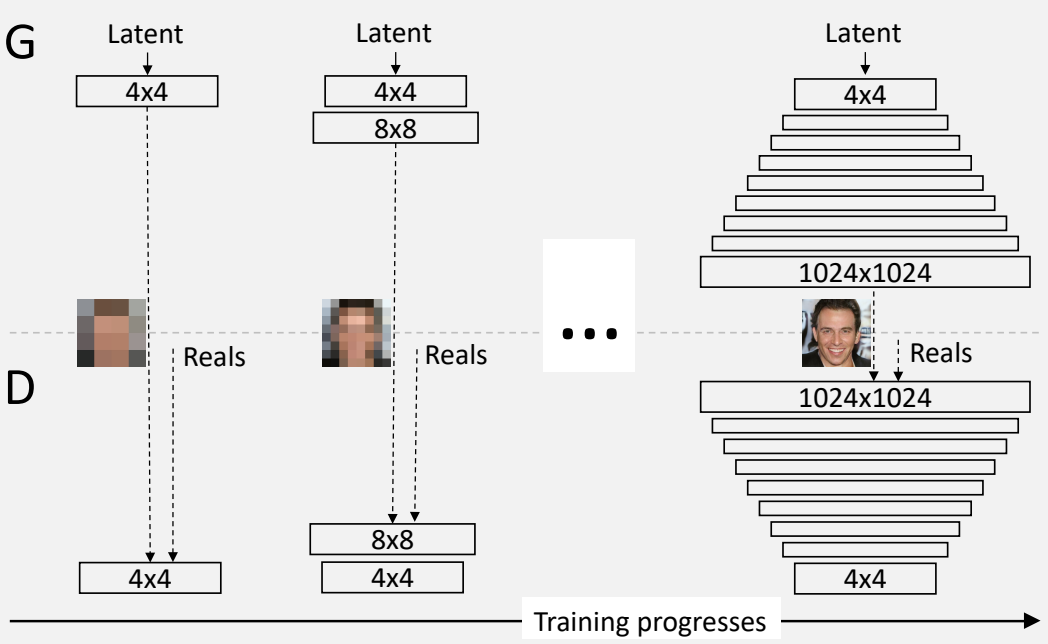
- ***Algorithm***
  - *Here is the algorithm explanation from the paper (you will need it when completing the exercise)*

Algorithm 2 Deep Interactive Evolution.
Defaults $m, n \leftarrow 20, \mu \leftarrow 0, \sigma^2 \leftarrow 1, p \leftarrow .5, foreign \leftarrow 2$

$G_\theta \leftarrow trainGAN(data)$ ▷ we use WGAN-GP [8]
$interface \leftarrow$ Interface()
$Z \leftarrow$ m by n matrix where $z_{i,j} \sim \mathcal{N}(\mu, \sigma^2)$ ▷ population size m with n latent variables each
repeat
    $images \leftarrow G_\theta(Z)$
    interface.display($images$)
    wait until interface.buttonNextPressed() is True
    $indices \leftarrow$ interface.getSelectedImages()
    $selection \leftarrow Z_{indicies}$
    $s^2 \leftarrow$ interface.getMutationParameter()
    $\Delta \leftarrow m - length(selection)$
    $x \leftarrow \max(0, \Delta - foreign)$
    $cross \leftarrow x$ by n matrix where $cross_i \sim mutate(uniform(selection), s^2)$
    $x \leftarrow \max(foreign, \Delta)$
    $new \leftarrow x$ by n matrix where $new_{i,j} \sim \mathcal{N}(\mu, \sigma^2)$
    $selection \leftarrow$ apply $mutate$ with s^2 to each $selection_i$
    $Z \leftarrow selection + cross + new$
until user stops
function UNIFORM(population)
    $a \leftarrow$ random individual from population
    $b \leftarrow$ random individual from population
    $mask \leftarrow$ vector of length n where $mask_i \sim Bernoulli(.5)$
    return $mask \cdot a + (1 - mask) \cdot b$
function MUTATE(individual, std)
    $mutation \leftarrow Bernoulli(p)$
    $noise \leftarrow$ vector of length n where$noise_i \sim \mathcal{N}(\mu, std)$
    return $individual + mutation \cdot noise$

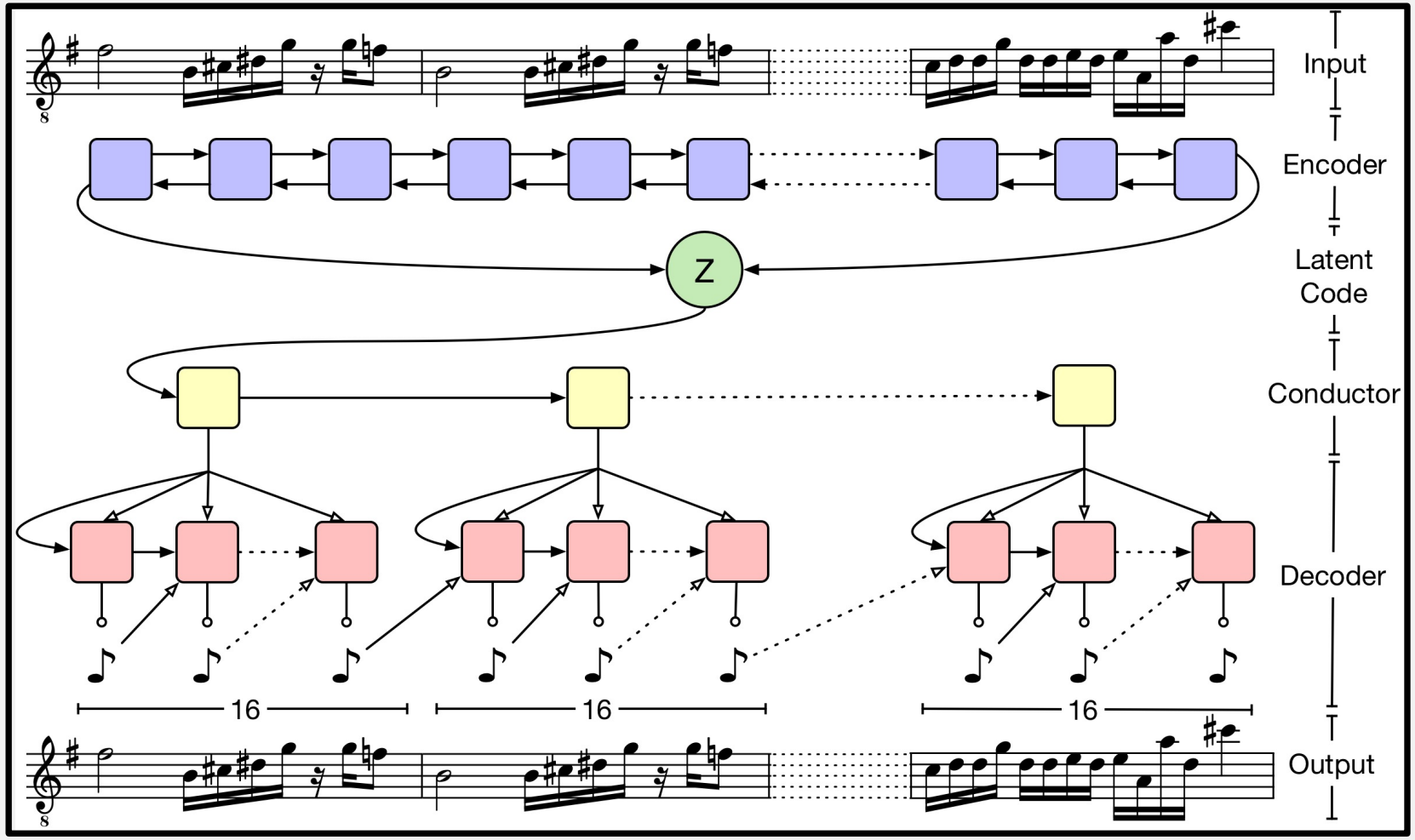
DEEP INTERACTIVE EVOLUTION

- **Progressive GANs**
  - Gan architecture where discriminator and genereator are progressively grown during training
  - Trained on the CelebA dataset



DEEP INTERACTIVE EVOLUTION

- **MusicVAE**
  - Hierarchical autoencoder
  - **Encoder**
    - Bidirectional RNN
  - **Decoder**
    - Autoregressive RNN
- Latent code passed to a conductor RNN, outputs a new embedding for each bar of the output
- Note RNN generates. Each bar independently



REFERENCES

- **Evolutionary Ecosystem**
  - Nature of Code Book Chapter on Genetic Algorithms

- **Deep Interactive Evolution**
  - Original Paper
  - Blog Post
  - **MusicVAE**
    - Original Paper
    - Blog Post
    - Sound Synthesis
  - **Progressive GANs**
    - Original paper
    - Tensorflow colab

- **Genetic Algorithms with Supercollider**
  - Sound Synthesis