Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Laboratory session #07: Particle Systems

Student: Vargas Romero Erick Efrain

Professor: Rosas Trigueros Jorge Luis

Practice completion date: November 8, 2021

Report delivery date: November 04, 2021

1

Theoretical framework

1.1 Particle Systems

In 1982, William T. Reeves, a researcher at Lucasfilm Ltd., was working on the film Star Trek II: The Wrath of Khan. Much of the movie revolves around the Genesis Device, a torpedo that when shot at a barren, lifeless planet has the ability to reorganize matter and create a habitable world for colonization. During the sequence, a wall of fire ripples over the planet while it is being "terraformed." The term particle system, an incredibly common and useful technique in computer graphics, was coined in the creation of this particular effect.

"A particle system is a collection of many many minute particles that together represent a fuzzy object. Over a period of time, particles are generated into a system, move and change from within the system, and die from the system."

—William Reeves

Since the early 1980s, particle systems have been used in countless video games, animations, digital art pieces, and installations to model various irregular types of natural phenomena, such as fire, smoke, waterfalls, fog, grass, bubbles, and so on.

1.2 Basic Model of Particle Systems

A particle system is a collection of many particles that together represent a fuzzy object. Over a period of time, particles are generated into a system, move and change from within the system, and die from the system.

To compute each frame in a motion sequence, the following sequence of steps is performed:

1. New particles are generated into the system.
2. Each new particle is assigned its individual attributes.

3. Any particles that have existed within the system past ther prescribed lifetime are extinguished.
4. The remaining particles are moved and transformed according to their dynamic attributes.
5. An image of the living particles is rendered in a frame buffer.

The particle system can be programmed to execute any set of instructions at each step. Because it is procedural, this approach can incorporate any computational model that describes the appearance of dynamics of the object. For example, the motions and transformations of particles could be tied to the solution of a system of partial different equations or particle attributes could be assigned on basis of statistical mechanics. We can therefore, take advantage of models with have been developed in other scientific or engineering disciplines [1].

2 Material and equipment

The necessary material for this practice is:

- A computer with the latest *Python* stable version installed
- A text editor

Or is possible to use the google site `https://colab.research.google.com/` that allows us to use a virtual machine with an *Ubuntu* operative system with *Python* installed.

3

Practice development

3.1 Particle Systems

To develop this practice we used the Google platform called *Colab* as this platform uses a virtual machine with linux (specifically Ubuntu) we can install some packets and of course we can verify if we have already *Python* installed. To check it we must use the command:

```
python --version
```

If we run this command in our linux terminal using *Colab* we can see the next as the result:

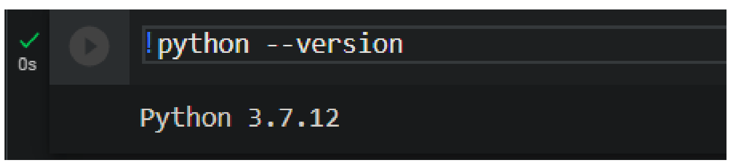

Figure 3.1: Verifying python version

3.2 Implement a particle system in 2D with nearest neighbour interactions in a grid

To solve this practice we based in the code provided by the professor. The mexican wave we already known that works in one dimension and works with the interaction between adjacent particles. So to solve this first task we have something very similar but instead of one dimension we have two dimensions. In code to make easier to understand the behaviour of the system we built a class called *Particle* this class contains a *constructor* that receives the initial position of the particle, radius (can be also the force of the particle) and a boolean state called isGrowing that indicates that the $i-th$

particle is changing or can be affected by their neighbours. This class also contains a method called *draw*, basically, this function draws a particle in the space.

```python
class Particle:
    #Constructor
    def __init__(self, x, y):
        self.radius = 0.0
        self.x = x
        self.y = y
        self.isGrowing = True

    def draw(self, img):
        color = (255, 255, 255)
        cv2.circle(img, (int(self.x), int(self.y)), int(self.radius), color, -1)
```

Other interesting function in this problem is the called *nextIteration*. This function makes possible to change every particle in the space, we iterate particle by particle and for each particle we get the "force" of the neighbors and we add that "force" in the current particle but of course the force is decreased in every iteration.

```python
def nextIteration(particles):
    global maxRadius, activeParticles, decayValue, increaseConstant

    neighbors = [[1,0], [0,1], [-1,0], [0,-1]]
    xParticles = len(particles)
    yParticles = len(particles[0])
    activeParticles = 0

    for x in range(0, xParticles):
        for y in range(0, yParticles):
            if particles[x][y].isGrowing:
                for xOffset, yOffset in neighbors:
                    #We add to the current particle the "force" applied by the neighbours.
                    if(0 <= x + xOffset < xParticles) and (0 <= y  + yOffset < yParticles):
                        particles[x][y].radius += increaseConstant * particles[x + xOffset][y + yOffset].radius

            #We reach the maximum radius of the current particle we don't need to use it again
            if particles[x][y].isGrowing and particles[x][y].radius >= maxRadius:
                particles[x][y].isGrowing = False

            #If the current particle is not growing we set the particle to the initial state (talking about "force")
            if not particles[x][y].isGrowing:
                if particles[x][y].radius <= 1.0:
                    particles[x][y].radius = 0

            #We count the number of particles that are in use
            if particles[x][y].radius > 1.0:
                activeParticles += 1

            #Decay
            particles[x][y].radius *= decayValue
```

The entire code to solve this task is here to get a better understanding about the functionality:

```python
import cv2
import numpy as np
from IPython import display as display
import ipywidgets as ipw
import PIL
import random
import time
from io import BytesIO

class Particle:
    #Constructor
    def __init__(self, x, y):
        self.radius = 0.0
        self.x = x
        self.y = y
        self.isGrowing = True

    def draw(self, img):
        color = (255, 255, 255)
        cv2.circle(img, (int(self.x), int(self.y)), int(self.radius), color, -1)

#Creates all the possible particles in the space
def generateParticles(x0, y0, maxX, maxY):

    global spaceBetweenParticles

    xValues = []
    yValues = []
    particles = []

    xi = x0
    while xi > spaceBetweenParticles:
        xValues.append(xi)
        xi -= spaceBetweenParticles
    xi = x0
    while xi < maxX:
        xValues.append(xi)
        xi += spaceBetweenParticles
    yi = y0
    while yi > spaceBetweenParticles:
        yValues.append(yi)
        yi -= spaceBetweenParticles
    yi = y0
    while yi < maxY:
        yValues.append(yi)
        yi += spaceBetweenParticles

    xValues.sort()
    yValues.sort()

    xSize = len(xValues)
    ySize = len(yValues)

    for x in range(0, xSize):
        currParticles = []
        for y in range(0, ySize):
            currParticles.append(Particle(xValues[x], yValues[y]))
        particles.append(currParticles)
```

```

    return particles

def chooseRandomParticle(particles):
    xRandomPosition = random.randrange(1, len(particles) - 1)
    yRandomPosition = random.randrange(1, len(particles[0]) - 1)
    particles[xRandomPosition][yRandomPosition].radius = 1

def nextIteration(particles):
    global maxRadius, activeParticles, decayValue, increaseConstant

    neighbors = [[1,0], [0,1], [-1,0], [0,-1]]
    xParticles = len(particles)
    yParticles = len(particles[0])
    activeParticles = 0

    for x in range(0, xParticles):
        for y in range(0, yParticles):
            if particles[x][y].isGrowing:
                for xOffset, yOffset in neighbors:
                    #We add to the current particle the "force" aplyied by the neighbours.
                    if(0 <= x + xOffset < xParticles) and (0 <= y  + yOffset < yParticles):
                        particles[x][y].radius += increaseConstant * particles[x + xOffset][y + yOffset].radius

            #We reach the maximum radius of the current particle we don't need to use it again
            if particles[x][y].isGrowing and particles[x][y].radius >= maxRadius:
                particles[x][y].isGrowing = False

            #If the current particle is not growing we set the particle to the initial state (talking about "force")
            if not particles[x][y].isGrowing:
                if particles[x][y].radius <= 1.0:
                    particles[x][y].radius = 0

            #We count the number of particles that are in use
            if particles[x][y].radius > 1.0:
                activeParticles += 1

            #Decay
            particles[x][y].radius *= decayValue

def drawParticles(particles, img):

    xParticles = len(particles)
    yParticles = len(particles[0])

    for x in range(0, xParticles):
        for y in range(0, yParticles):
            particles[x][y].draw(img)

#Or rows
height = 500
#Or cols
width = 600
```

```
#Max particle radius
maxRadius = 10
#Max particle radius
spaceBetweenParticles = maxRadius * 2 + 1
activeParticles = 1
decayValue = 0.9
increaseConstant = 0.15

x0 = width // 2
y0 = (height) // 2

maxIterations = 200

wIm = ipw.Image()
display.display(wIm)

#We create our grid
image = np.zeros((height + spaceBetweenParticles, width + spaceBetweenParticles, 3),
     dtype = "uint8")
#Creating all the particles
#We add an offset in height because without that we can't see the leyend (Iteration:
     n)
particles = generateParticles(x0, y0, width, height - (30 + maxRadius))

xParticles = len(particles)
yParticles = len(particles[0])

drawParticles(particles, image)

for iteration in range(0, 200):
    image[:] = (0, 0, 0)

    if activeParticles == 0:
        for x in range(0, xParticles):
            for y in range(0, yParticles):
                particles[x][y].isGrowing = True

        chooseRandomParticle(particles)

    nextIteration(particles)
    drawParticles(particles, image)
    #Show the current iteration in the image
    cv2.putText(image, "Iteration: " + str(iteration + 1), (spaceBetweenParticles,
    height - 10), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))
    #Show the current iteration in the image
    cv2.putText(image, "Particles: " + str(activeParticles), (250 +
    spaceBetweenParticles, height - 10), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255,
    255))

    pilIm = PIL.Image.fromarray(image, mode = "RGB")

    with BytesIO() as fOut:
        pilIm.save(fOut, format = "png")
        byPng = fOut.getvalue()

    # set the png bytes as the image value;
    # this updates the image in the browser.
    wIm.value = byPng
```

```
    time.sleep(0.15)
```

3.3 Implement a Particle System in 2D with collisions (A box with a small window)

To solve this task we used the code provider by the teacher, This task was a little bit more complicated than the previous one because of the physics implyied in the behaviour of each particle. Talking about the physics of this task, we used the first and second Newton's law, and as we are using a 2D particle system is mandatory to use vectorial calculus to modelate all the physical phenomena

Something really important about this task is to make all the particles move to the right, because in the right side of the box, we created a small exit in order to see the behaviour of the particles. So to solve this we must increase the force in the y component of the vector and decrease this force in each iteration this step is done after calculate the force applied by each particle around the current particle.

```
for count in range(MaxIterations):
    img[:] = (0, 0, 0)
    for i in range(NumParticles):
        for j in range(NumParticles):
            if i != j:
                Fij = particles[i].calculateForce(particles[j])
                particles[i].force += Fij
                #As we already known we must guide all the particles to the right
                result = exitAtractor - particles[i].position
                magnitudeResult = np.linalg.norm(result)
                if magnitudeResult <= 0:
                    magnitudeResult = 1
                #Calculating the force of the vector
                resultForce = result / magnitudeResult
                #DECAY
                resultForce[0] *= 0.001
                resultForce[1] *= 0.009
                particles[i].force += resultForce

    for particle in particles:
        particle.updateSpeed()
        particle.updatePosition()
        particle.draw(x0, y0, img)
        particle.force[:] = 0
```

The rest of the code is almost the same, here we add the entire code in order to check carefully how this task works.

```
import cv2
import numpy as np
from IPython import display as display
import ipywidgets as ipw
import PIL
from io import BytesIO
import math
import random
import time
```

```

class Particle:
    MaxV = np.sqrt(2)

    def __init__(self, x, y, speedX, speedY, radius, WallParticle=False):
        self.position = np.array([float(x), float(y)])
        self.speed = np.array([float(speedX), float(speedY)])
        self.WallParticle = WallParticle
        self.force = np.array([0.0, 0.0])
        self.radius = radius

    #By definition an unit vector it's magnitude must be 1 otherwise it isn't a unit vector
    def normalizeVector(self, x):
        norm = np.linalg.norm(x)
        if norm == 0.0:
            return x * np.inf
        return x / norm

    def calculateForce(self, r2):
        if self.WallParticle == True:
            return np.array([0.0, 0.0])
        result = self.position - r2.position
        r12magnitude = np.linalg.norm(result)
        if r12magnitude <= r2.radius + self.radius:
            #We get the unitary vector and then we split by the magnitude of R where R is a new vector from r1 to r2
            return self.normalizeVector(result) / (r12magnitude ** 2) * wallParticlesSize * 20
        return np.array([0.0, 0.0])

    def updatePosition(self):
        if self.WallParticle == True:
            return
        self.position += self.speed
        return

    def updateSpeed(self):
        if self.WallParticle == True:
            return
        self.speed += self.force
        vmag = np.linalg.norm(self.speed)

        if vmag > self.MaxV:
            self.speed = self.normalizeVector(self.speed) * self.MaxV
        return

    def draw(self, x0, y0, img):
        if self.WallParticle == True:
            color = (0, 255, 255)
        else:
            color = (255, 255, 255)
        cv2.circle(img, (int(x0 + self.position[0]), int(y0 - self.position[1])), self.radius, color, -1)
        return


particles = []
```

```python
def lineOfWallParticles(x1, y1, x2, y2, N):
    global particles
    x = np.linspace(x1, x2, N)
    y = np.linspace(y1, y2, N)
    for i in range(N):
        particles.append(Particle(x[i], y[i], 0, 0, wallParticlesSize, WallParticle=
    True))


wIm = ipw.Image()
display.display(wIm)

maxX = 500
maxY = 500
x0 = int(maxX / 2)
y0 = int(maxY / 2)
nParticles = 70
particleSize = 7
wallParticlesSize = 15

img = np.zeros((500, 500, 3), dtype="uint8")

#Upper wall
lineOfWallParticles(-150, 100, 150, 100, 20)
#Bottom wall
lineOfWallParticles(-150, -100, 150, -100, 20)
#Left wall
lineOfWallParticles(-150, 100, -150, -100, 20)
#Right wall
lineOfWallParticles(150, -100, 150, -30, 10)
lineOfWallParticles(150, 100, 150, 30, 10)
#Exit atractor
exitAtractor = np.array([300.0, 0.0])

for i in range(0, nParticles):
    x = random.randrange(-150 + 2 * wallParticlesSize, -100 + 2 * wallParticlesSize)
    y = random.randrange(-100 + 2 * wallParticlesSize, 100 - 2 * wallParticlesSize)
    speedX = random.random()
    speedY = random.random()
    newParticle = Particle(x, y, speedX, speedY, particleSize)
    particles.append(newParticle)


MaxIterations = 500
NumParticles = len(particles)

for count in range(MaxIterations):
    img[:] = (0, 0, 0)
    for i in range(NumParticles):
        for j in range(NumParticles):
            if i != j:
                Fij = particles[i].calculateForce(particles[j])
                particles[i].force += Fij
                #As we already known we must guide all the particles to the right
                result = exitAtractor - particles[i].position
                magnitudeResult = np.linalg.norm(result)
                if magnitudeResult <= 0:
                    magnitudeResult = 1
```

```
                #Calculating the force of the vector
                resultForce = result / magnitudeResult
                #DECAY
                resultForce[0] *= 0.001
                resultForce[1] *= 0.009
                particles[i].force += resultForce

    for particle in particles:
        particle.updateSpeed()
        particle.updatePosition()
        particle.draw(x0, y0, img)
        particle.force[:] = 0

    cv2.putText(img, "Iteration: " + str(count + 1), (20, 40), cv2.
    FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))

    pilIm = PIL.Image.fromarray(img, mode="RGB")
    with BytesIO() as fOut:
        pilIm.save(fOut, format="png")
        byPng = fOut.getvalue()

    # set the png bytes as the image value;
    # this updates the image in the browser.
    wIm.value = byPng
```

3.4 Implement a Particle System in 2D with collisions (A box with a small window and an obstacle)

To this final task we used the same code than in the previous task, but with an extra particle that is static and works as an obstacle to the rest of particles that can move. As you can see below we only add an extra line with a particle.

```
#Obstacle
particles.append(Particle(100, 0, 0, 0, wallParticlesSize, WallParticle=True))
```

Here is the entire code to a better understanding

```
import cv2
import numpy as np
from IPython import display as display
import ipywidgets as ipw
import PIL
from io import BytesIO
import math
import random
import time

class Particle:
    MaxV = np.sqrt(2)

    def __init__(self, x, y, speedX, speedY, radius, WallParticle=False):
        self.position = np.array([float(x), float(y)])
        self.speed = np.array([float(speedX), float(speedY)])
        self.WallParticle = WallParticle
        self.force = np.array([0.0, 0.0])
```

```
        self.radius = radius

    def normalizeVector(self, x):
        norm = np.linalg.norm(x)
        if norm == 0:
            return x * np.inf
        return x / norm

    def calculateForce(self, r2):
        if self.WallParticle == True:
            return np.array([0.0, 0.0])
        result = self.position - r2.position
        r12magnitude = np.linalg.norm(result)
        if r12magnitude <= r2.radius + self.radius:
            #We get the unitary vector and then we split by the magnitude of R where
 R is a new vector from r1 to r2
            return self.normalizeVector(result) / (r12magnitude ** 2) *
    wallParticlesSize * 20
        return np.array([0.0, 0.0])

    def updatePosition(self):
        if self.WallParticle == True:
            return
        self.position += self.speed
        return

    def updateSpeed(self):
        if self.WallParticle == True:
            return
        self.speed += self.force

        vmag = np.linalg.norm(self.speed)

        if vmag > self.MaxV:
            self.speed = self.normalizeVector(self.speed) * self.MaxV
        return

    def draw(self, x0, y0, img):
        if self.WallParticle == True:
            color = (0, 255, 255)
        else:
            color = (255, 255, 255)

        cv2.circle(img, (int(x0 + self.position[0]), int(y0 - self.position[1])),
    self.radius, color, -1)
        return


particles = []
def lineOfWallParticles(x1, y1, x2, y2, N):
    global particles
    x = np.linspace(x1, x2, N)
    y = np.linspace(y1, y2, N)
    for i in range(N):
        particles.append(Particle(x[i], y[i], 0, 0, wallParticlesSize, WallParticle=
    True))


```

```
wIm = ipw.Image()
display.display(wIm)

maxX = 500
maxY = 500
x0 = int(maxX / 2)
y0 = int(maxY / 2)
nParticles = 50
particleSize = 7
wallParticlesSize = 15

img = np.zeros((500, 500, 3), dtype="uint8")

#Upper wall
lineOfWallParticles(-150, 100, 150, 100, 20)
#Bottom wall
lineOfWallParticles(-150, -100, 150, -100, 20)
#Left wall
lineOfWallParticles(-150, 100, -150, -100, 20)
#Right wall
lineOfWallParticles(150, -100, 150, -30, 10)
lineOfWallParticles(150, 100, 150, 30, 10)
#Exit atractor
exitAtractor = np.array([300.0, 0.0])
#Obstacle
particles.append(Particle(100, 0, 0, 0, wallParticlesSize, WallParticle=True))

for i in range(0, nParticles):
    x = random.randrange(-150 + 2 * wallParticlesSize, -100 + 2 * wallParticlesSize)
    y = random.randrange(-100 + 2 * wallParticlesSize, 100 - 2 * wallParticlesSize)
    speedX = random.random()
    speedY = random.random()
    newParticle = Particle(x, y, speedX, speedY, particleSize)
    particles.append(newParticle)


MaxIterations = 500
NumParticles = len(particles)

for count in range(MaxIterations):
    img[:] = (0, 0, 0)
    for i in range(NumParticles):
        for j in range(NumParticles):
            if i != j:
                Fij = particles[i].calculateForce(particles[j])
                particles[i].force += Fij
                #As we already known we must guide all the particles to the right
                result = exitAtractor - particles[i].position
                magnitudeResult = np.linalg.norm(result)
                if magnitudeResult <= 0:
                    magnitudeResult = 1
                #Calculating the force of the vector
                resultForce = result / magnitudeResult
                resultForce[0] *= 0.001
                resultForce[1] *= 0.009
                particles[i].force += resultForce

    for particle in particles:
```

```
        particle.updateSpeed()
        particle.updatePosition()
        particle.draw(x0, y0, img)
        particle.force[:] = 0

    cv2.putText(img, "Iteration: " + str(count + 1), (20, 40), cv2.
    FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))

    pilIm = PIL.Image.fromarray(img, mode="RGB")
    with BytesIO() as fOut:
        pilIm.save(fOut, format="png")
        byPng = fOut.getvalue()

    # set the png bytes as the image value;
    # this updates the image in the browser.
    wIm.value = byPng
```

4

Screens, graphs and diagrams

4.1 Implement a particle system in 2D with nearest neighbour interactions in a grid

As you can see in the img. 4.1 when we run the simulation randomly a particle is chosen and from that particle the neighbors are affected.

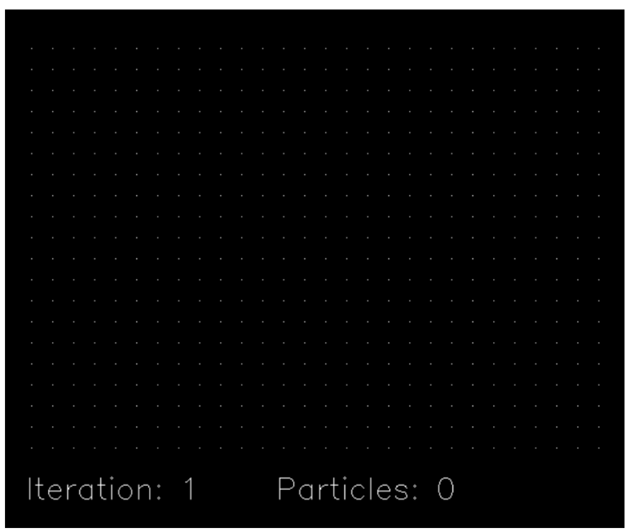

Figure 4.1: 2D Particle system with neighbour interactions, iteration 1

In the img 4.2 as you can see the program choose randomly a particle and now this particle affected more particles around of it and that particles affected other ones and so on, as you can see it the pattern grow.

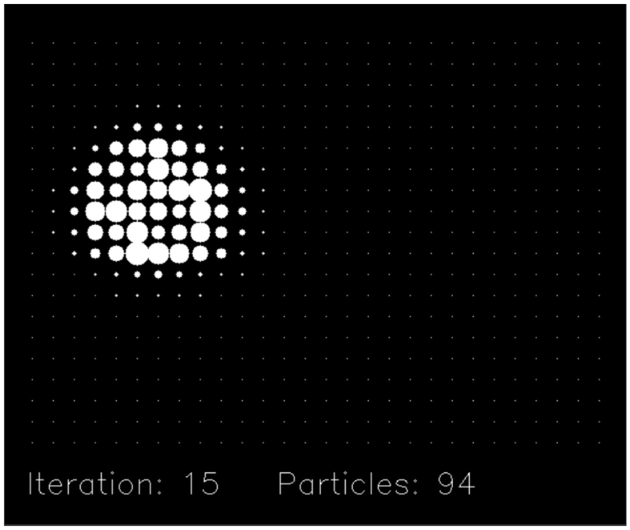

Figure 4.2: 2D Particle system with neighbour interactions, iteration 15

Now in img. 4.3 the pattern grow more but the first particles affected now are decaying.

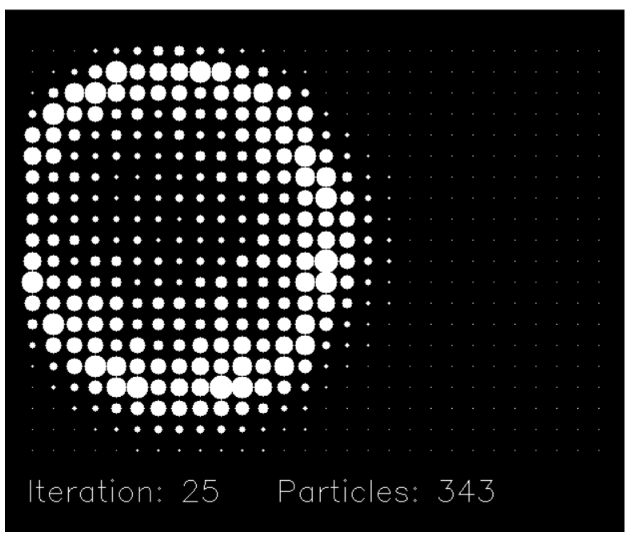

Figure 4.3: 2D Particle system with neighbour interactions, iteration 25

In img. 4.4 and img. 4.5 we have the same behaviour than in the previous image. It looks like a wave.

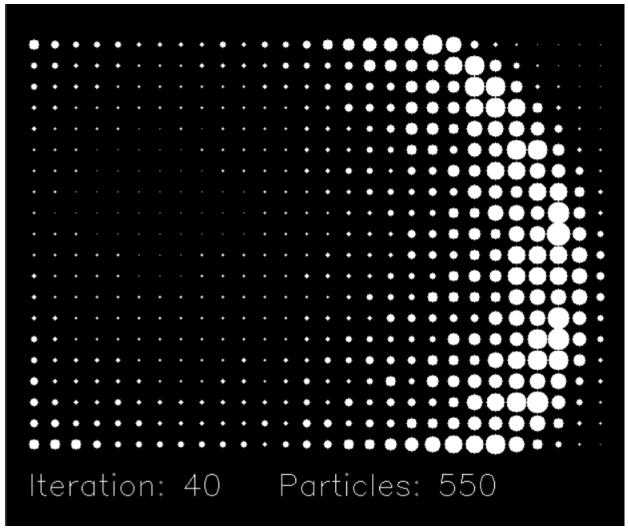

Figure 4.4: 2D Particle system with neighbour interactions, iteration 40

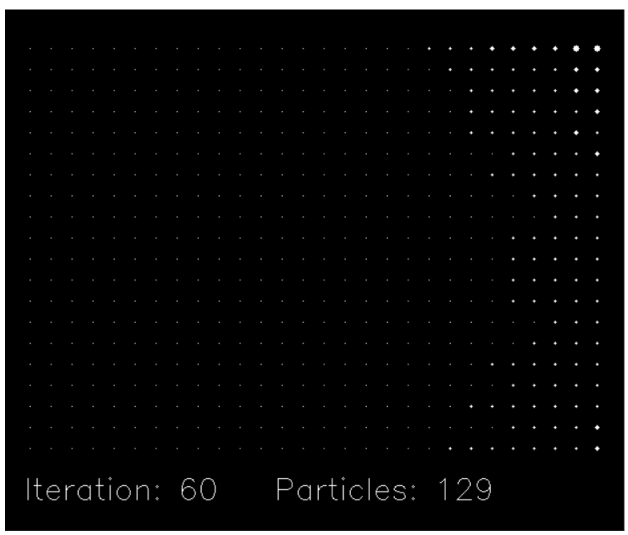

Figure 4.5: 2D Particle system with neighbour interactions, iteration 60

4.2 Implement a Particle System in 2D with collisions (A box with a small window)

For the second task we generate in random positions 70 particles, all of them are located at the left side of the box img. 4.6.

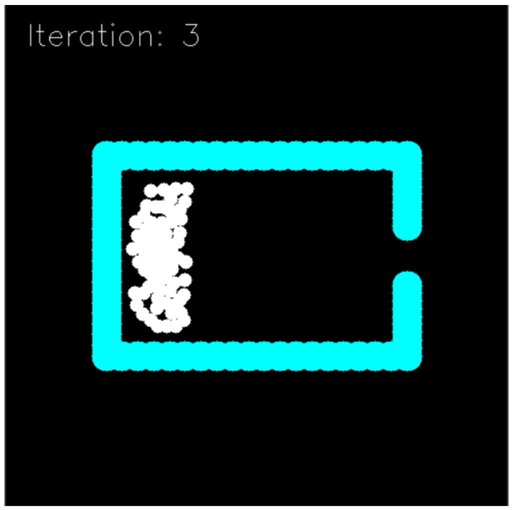

Figure 4.6: 2D Particle collisions (A box with small window), iteration 3

As you can see en each iteration looks like every particle moves to the right looking for the exit img. 4.7.

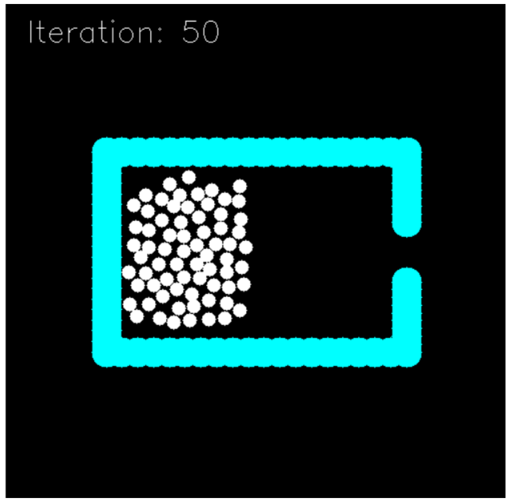

Figure 4.7: 2D Particle collisions (A box with small window), iteration 50

And this behaviour keeps going, but now looks like the particles are moving specifically to the exit, looks like they are moving in the y axis to the middle of the box 4.8.

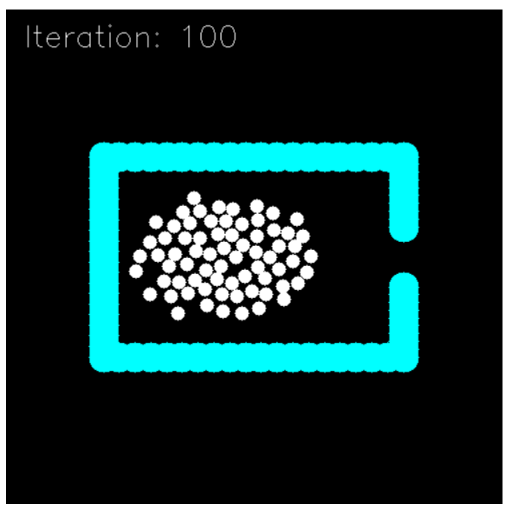

Figure 4.8: 2D Particle collisions (A box with small window), iteration 100

In the img. 4.9 we have the same behaviour but in the img. 4.10 as you can see some particles found the exit and the rest of them start following that particle.

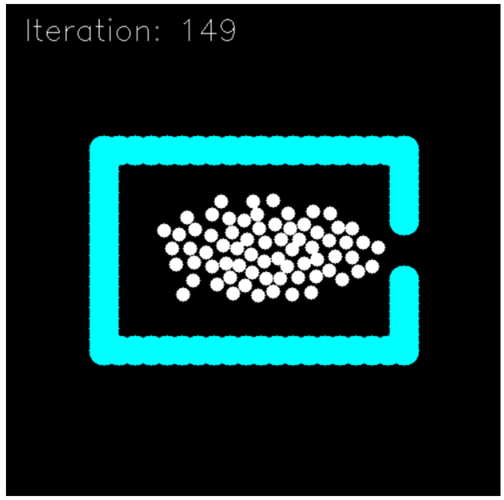

Figure 4.9: 2D Particle collisions (A box with small window), iteration 149

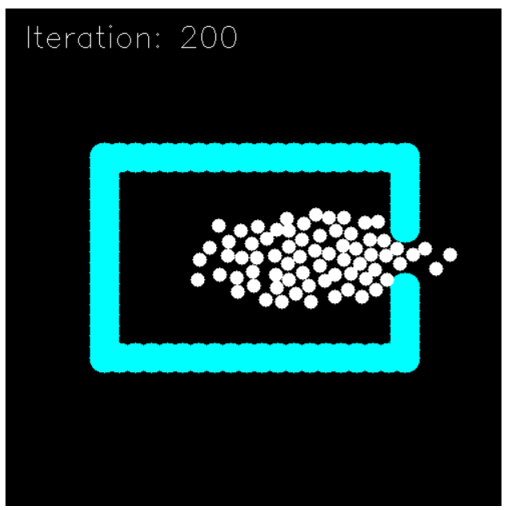

Figure 4.10: 2D Particle collisions (A box with small window), iteration 200

4.3 Implement a Particle System in 2D with collisions (A box with a small window and an obstacle)

We have a very similar behaviour than in the previous task, the particles moves from left to right img. 4.11

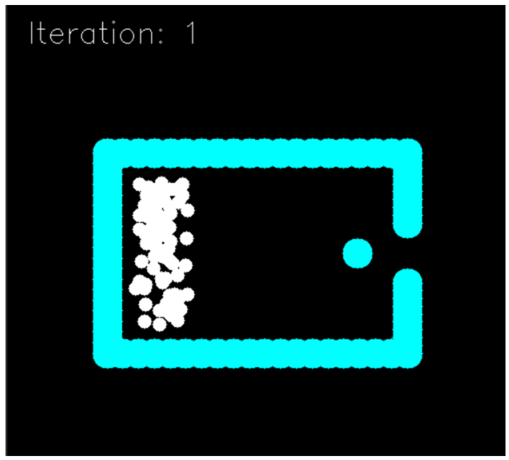

Figure 4.11: 2D Particle collisions (A box with small window and obstacle), iteration 1

And again looks like all the particles starts moving in the y axis too img. 4.12.

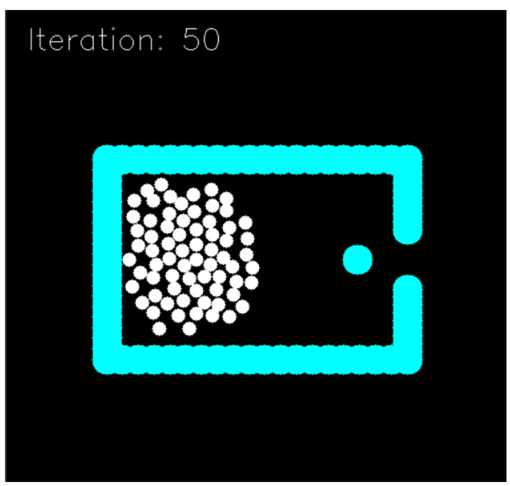

Figure 4.12: 2D Particle collisions (A box with small window and obstacle), iteration 50

But now the particles found an obstacle they must surround that obstacle in order to move to the exit 4.13.

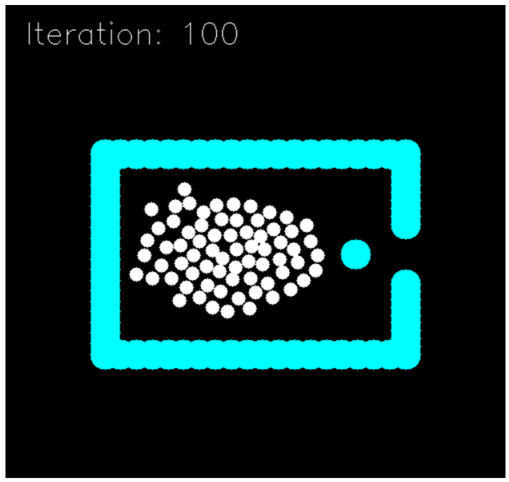

Figure 4.13: 2D Particle collisions (A box with small window and obstacle), iteration 100

And as we described previouslly, the particles surronded that obstacle and continue moving to the exit 4.14.

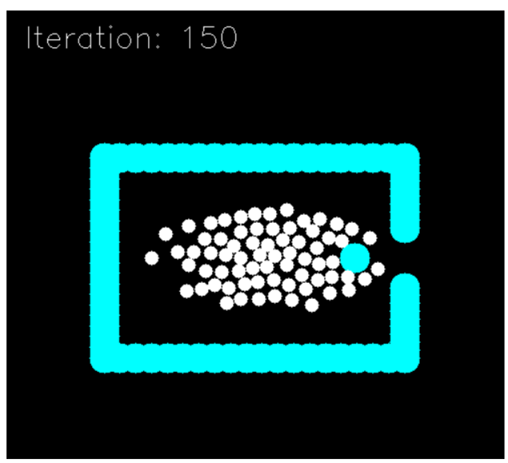

Figure 4.14: 2D Particle collisions (A box with small window and obstacle), iteration 150

Looks like the particles goes to the exit in pairs and the exit of that particles is ordered 4.15.

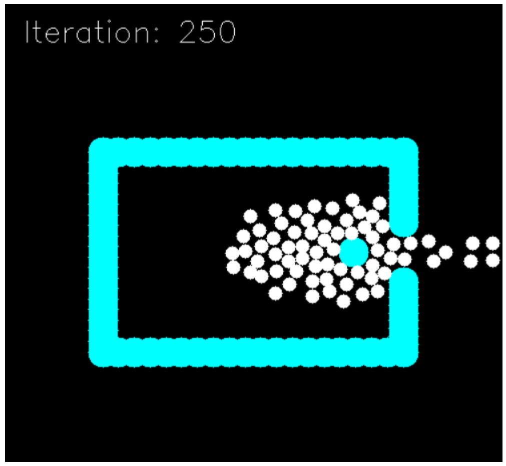

Figure 4.15: 2D Particle collisions (A box with small window and obstacle), iteration 250

5 Conclusions

This practice has been very interesting at least for me, because we used some physic concepts to simulate some behaviours that we have in the real world. The first task was related to interaction, here every particle interacts with their neighbors (at least the up, down, left, right neighbour) and we get something very similar to a wave for example when we throw a rock in water. The other two tasks used the Newton's first and second law to describe the behaviour of each particle, but as we are using a 2-dimensional system we used vectors to modelate the behaviour of position, force and speed.

As a commentary I believe that provide us a template about how something works to make the practices is good but to make easier to understand the code I suggest to use more descriptive variable names or at least add a comment in the code with a brief description about what is the objective of a variable.

Bibliography

[1] William T. Reeves *Particle systems - A technique for modeling a Class of Fuzzy Objects.* Accessed on November 7th, 2021. Available online: `https://www.lri.fr/~mbl/ENS/IG2/devoir2/files/docs/fuzzyParticles.pdf`