KING'S College LONDON

University of London

6CCS3PRJ Final Year Project

2-Dimensional and 3-Dimensional Evolutionary Art Using Genetic Algorithms

Final Project Report

Author: Patrick Whyte

Supervisor: Rita Borgo

Student ID: 1749039

April 23, 2020

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Patrick Whyte

April 23, 2020

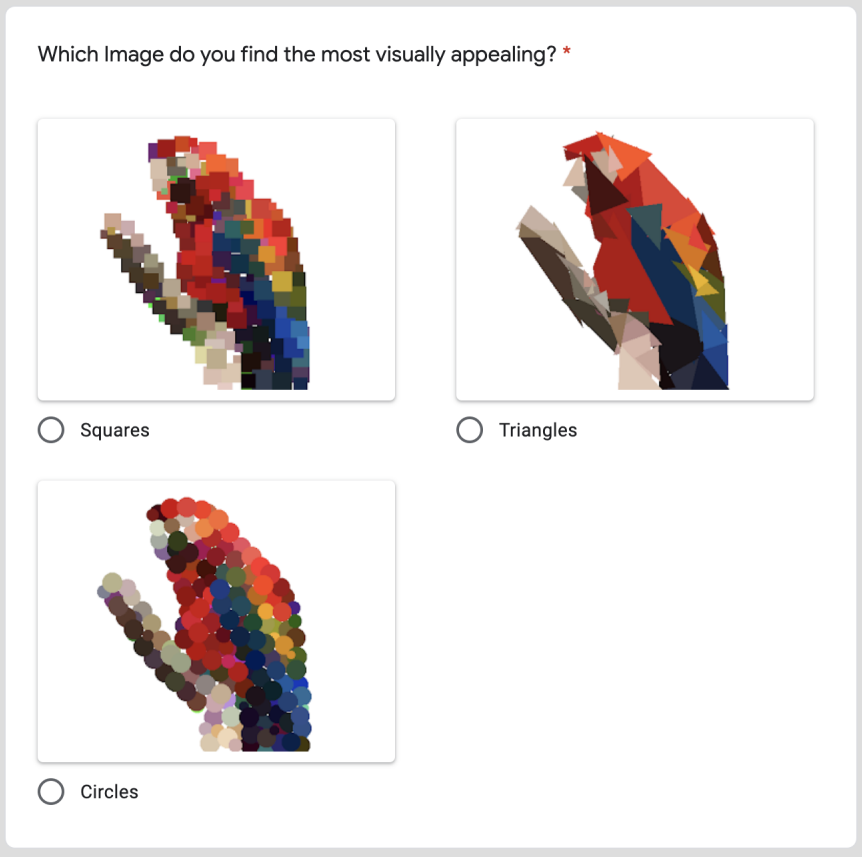
Contents

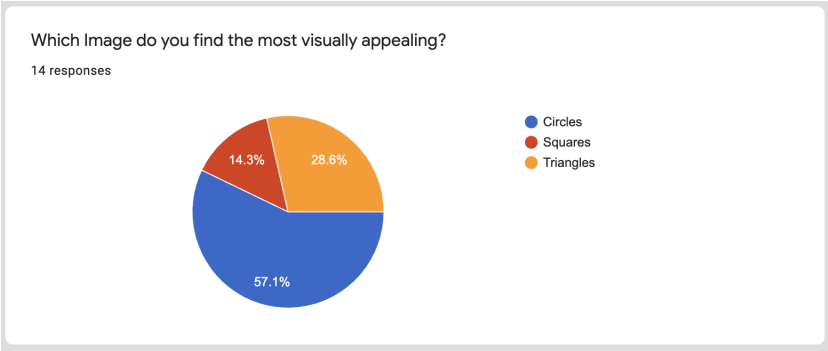
Appendix A

Extra Information

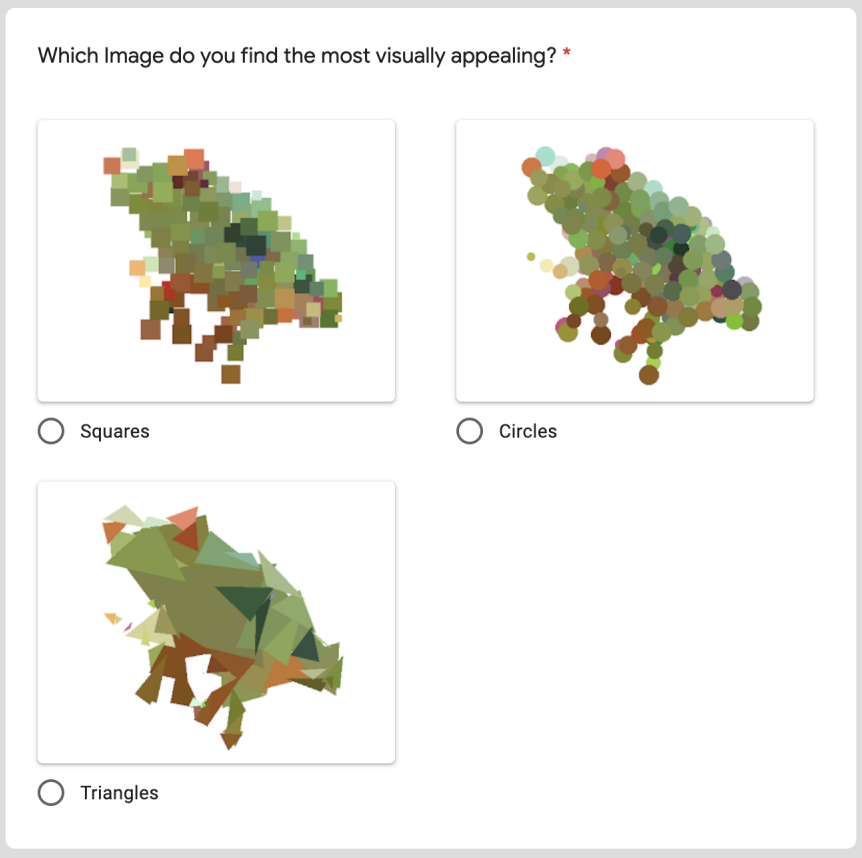
A.1 Survey Questions and Answers

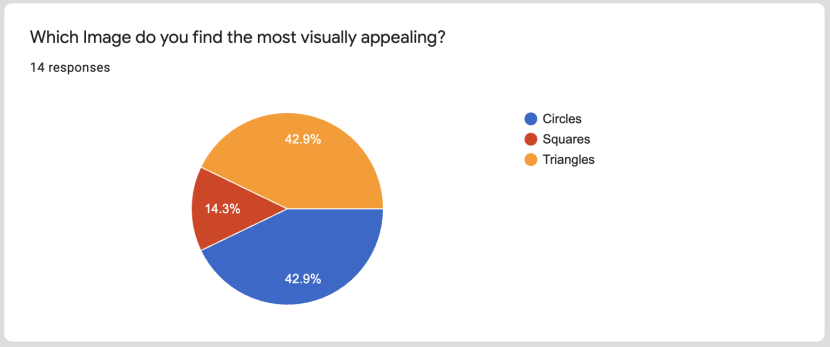
A.1.1 Question 1



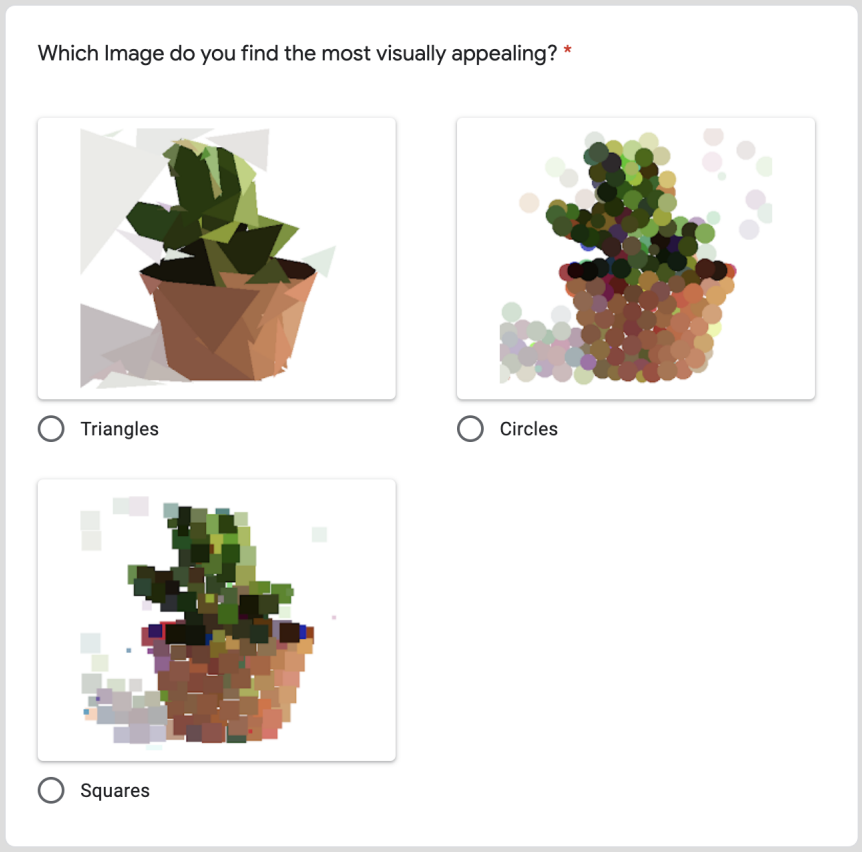


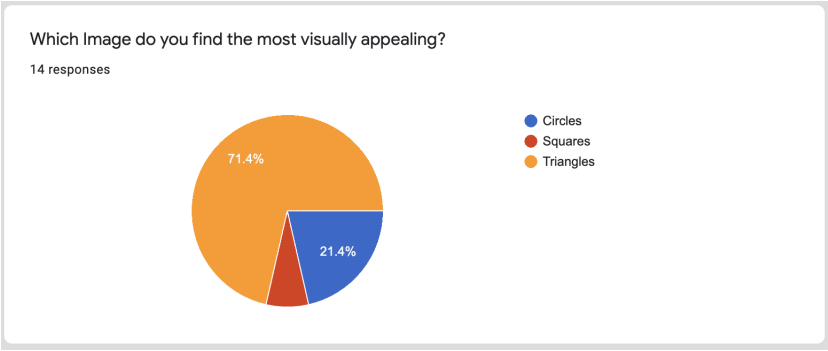
A.1.2 Question 2



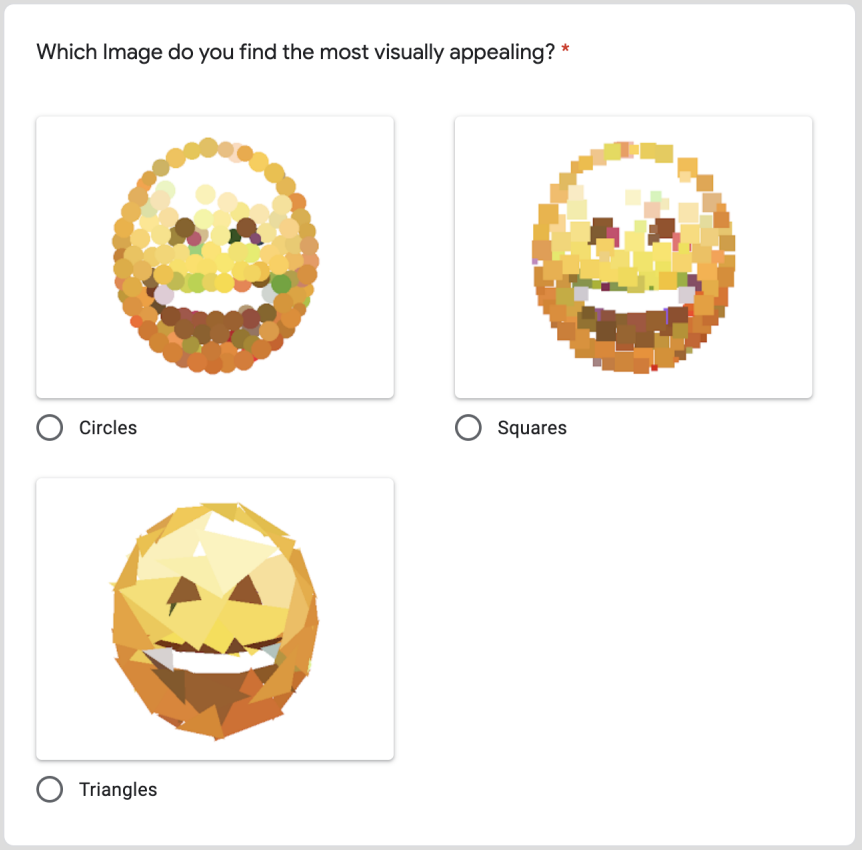


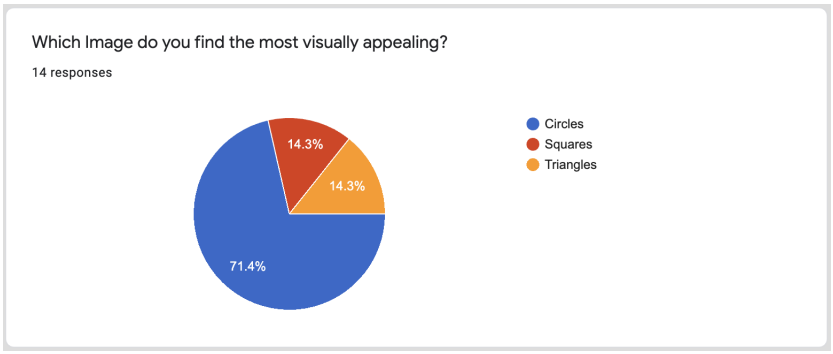
A.1.3 Question 3



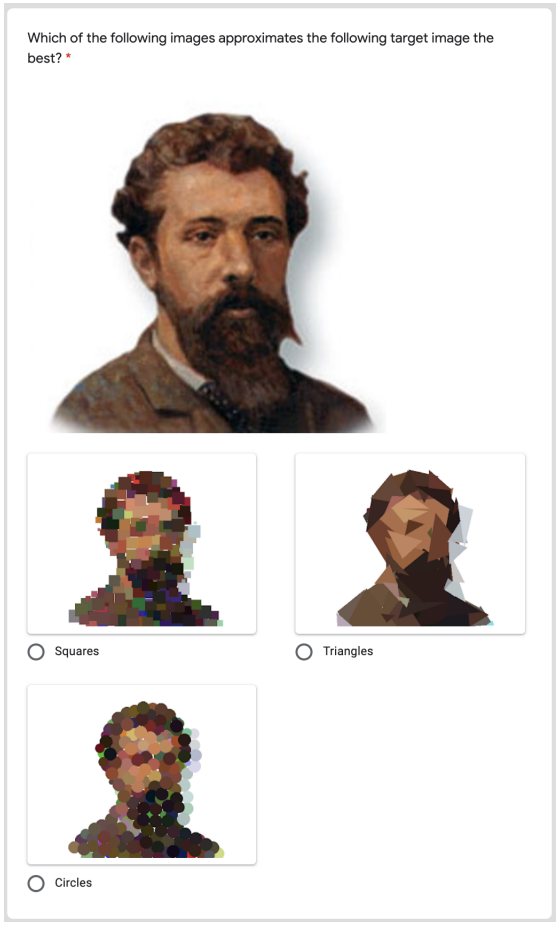


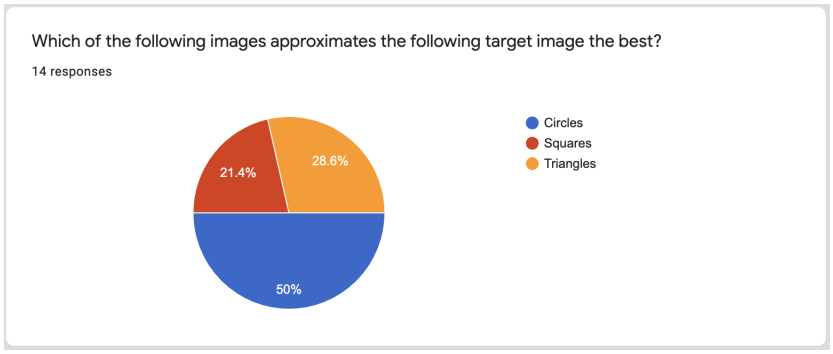
A.1.4 Question 4



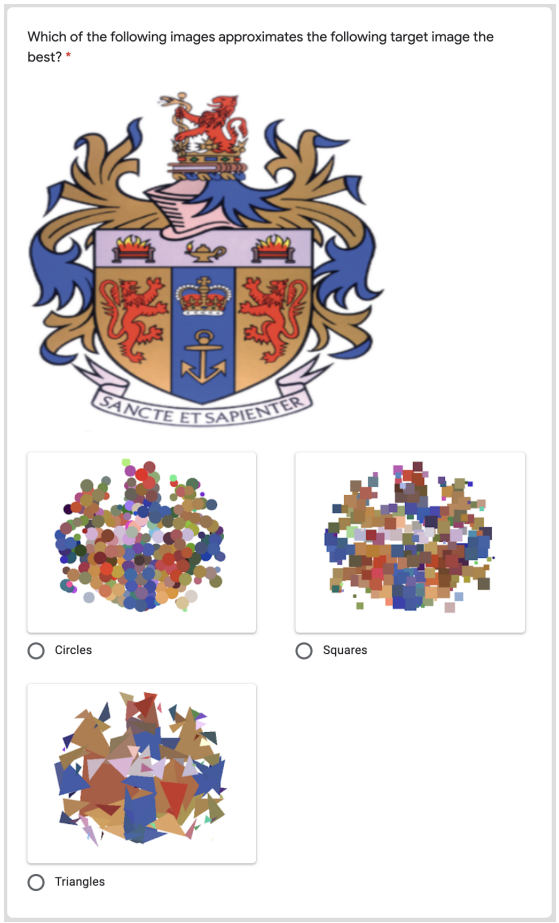


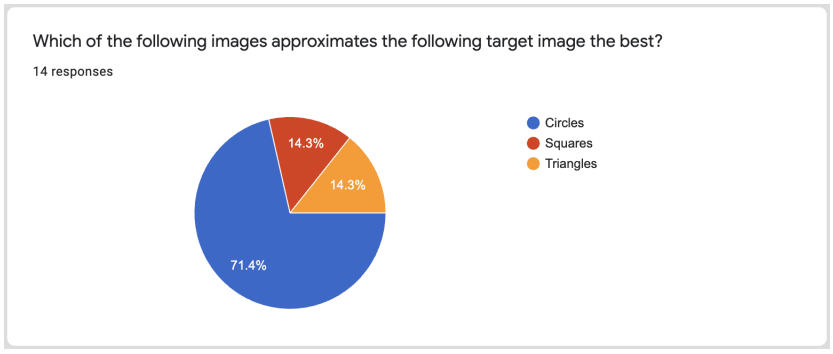
A.1.5 Question 5





A.1.6 Question 6





A.1.7 Question 7

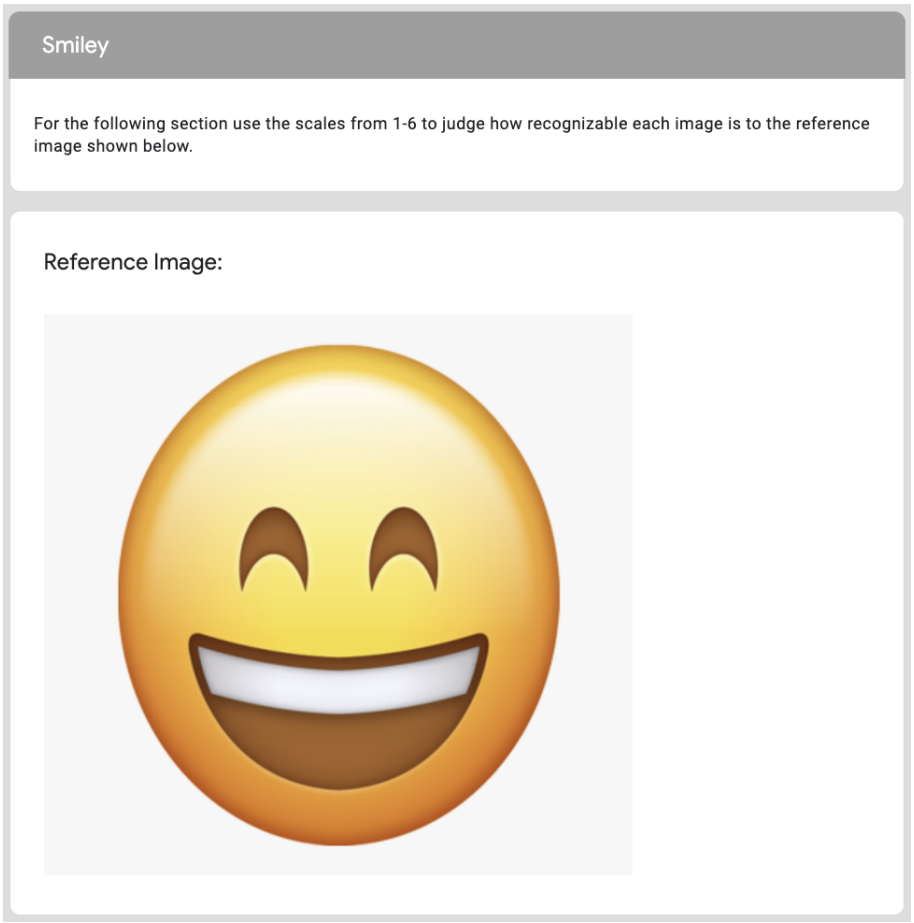
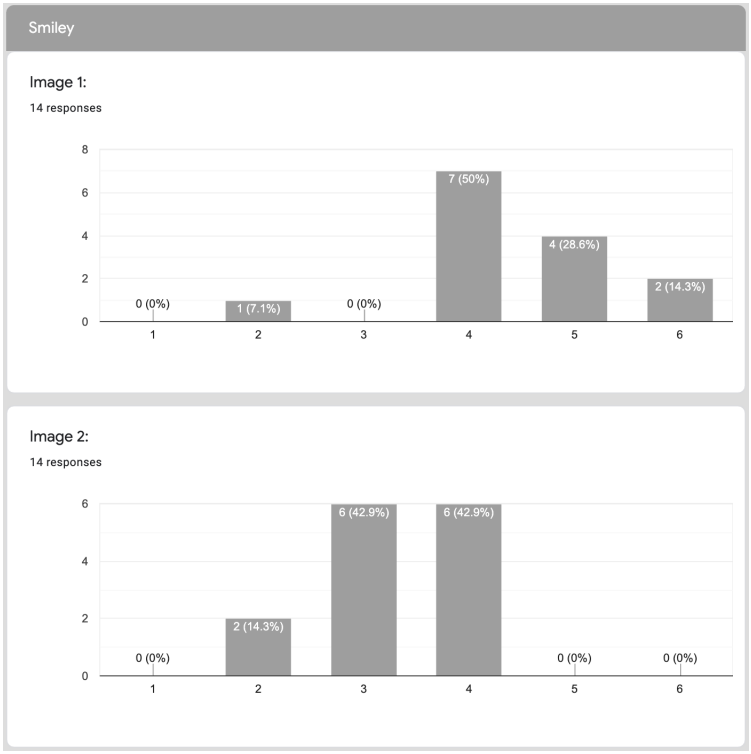



Image 1: *
1 2 3 4 5 6
Unidentifiable
Completely Recognizable
Image 2: *
1 2 3 4 5 6
Unidentifiable
Completely Recognizable


Smiley
Image 1:
14 responses
8
6
4
2
0
0 (0%)
1 (7.1%)
0 (0%)
7 (50%)
4 (28.6%)
2 (14.3%)
1 2 3 4 5 6
Image 2:
14 responses
6
4
2
0
0 (0%)
2 (14.3%)
6 (42.9%)
6 (42.9%)
0 (0%)
0 (0%)
1 2 3 4 5 6

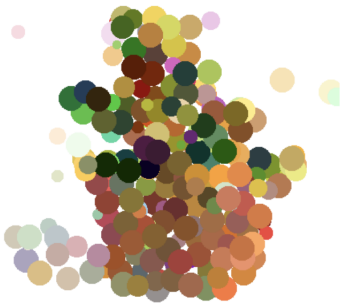
A.1.8 Question 8

Cactus

For the following section use the scales from 1-6 to judge how recognizable each image is to the reference image shown below.

Reference Image:

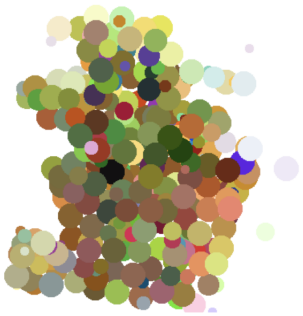
Image 1: *



1 2 3 4 5 6

Unidentifiable ○ ○ ○ ○ ○ ○ Completely Recognizable

Image 2: *



1 2 3 4 5 6

Unidentifiable ○ ○ ○ ○ ○ ○ Completely Recognizable

Cactus

Image 1:

14 responses

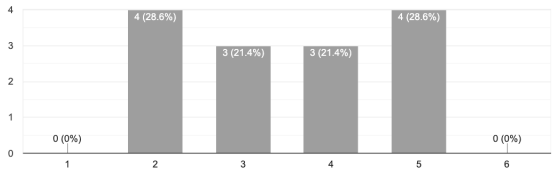
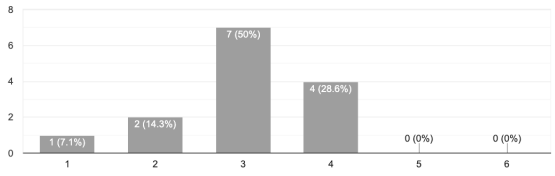


Image 2:

14 responses



A.1.9 Question 9

Bird

For the following section use the scales from 1-6 to judge how recognizable each image is to the reference image shown below.

Reference Image:

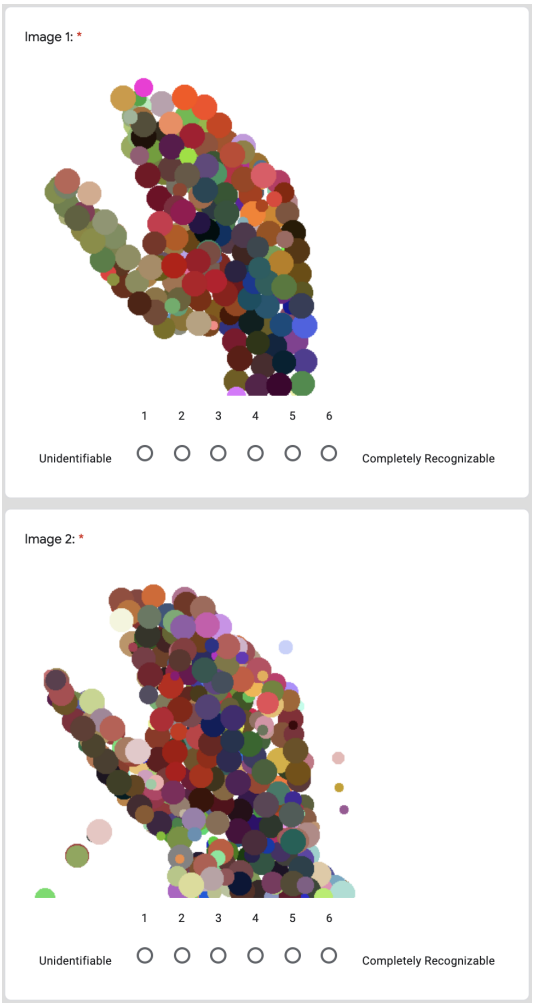
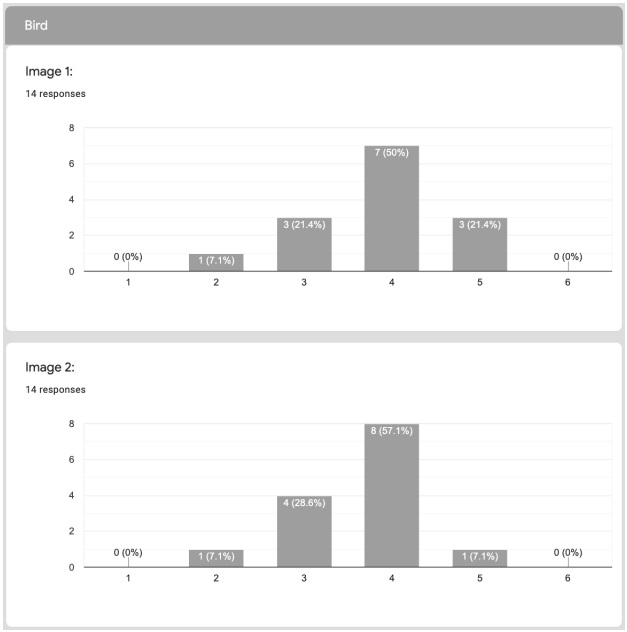


Image 1: *
1 2 3 4 5 6
Unidentifiable
Completely Recognizable
Image 2: *
1 2 3 4 5 6
Unidentifiable
Completely Recognizable


Bird
Image 1:
14 responses
8
6
4
2
0
0 (0%)
1 (7.1%)
3 (21.4%)
7 (50%)
3 (21.4%)
0 (0%)
1 2 3 4 5 6
Image 2:
14 responses
8
6
4
2
0
0 (0%)
1 (7.1%)
4 (28.6%)
8 (57.1%)
1 (7.1%)
0 (0%)
1 2 3 4 5 6

A.1.10 Question 10

Frog

For the following section use the scales from 1-6 to judge how recognizable each image is to the reference image shown below.

Reference Image:

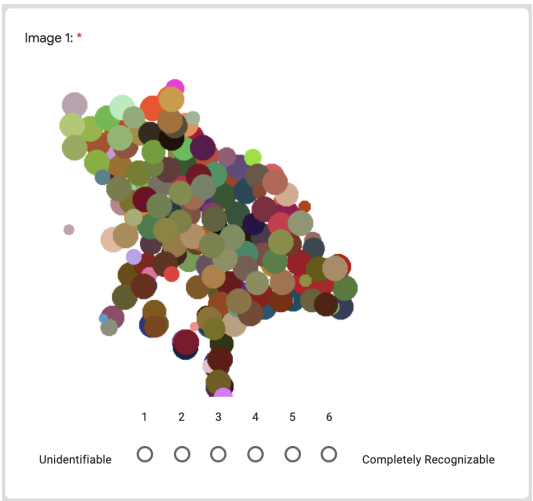


Image 1: *
1 2 3 4 5 6
Unidentifiable
Completely Recognizable

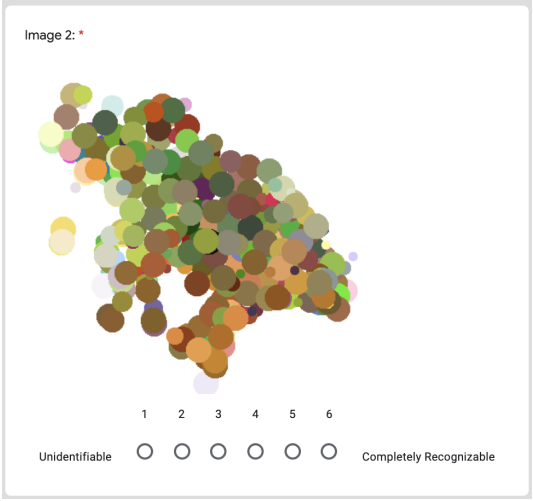
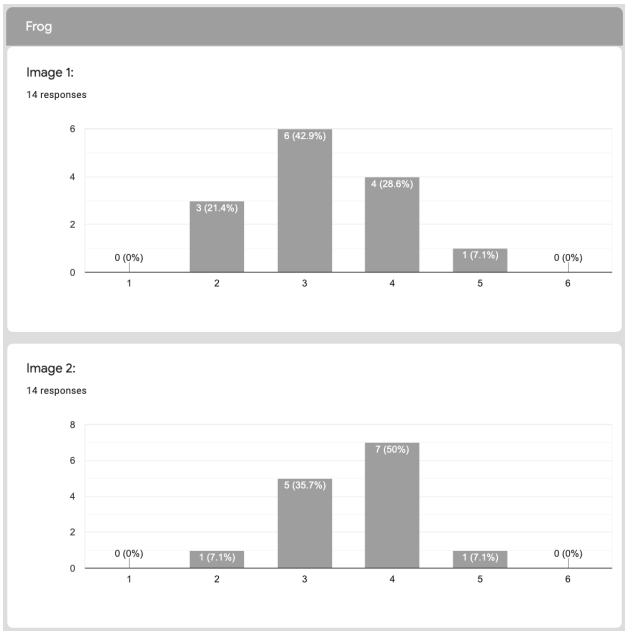


Image 2: *
1 2 3 4 5 6
Unidentifiable
Completely Recognizable


Frog
Image 1:
14 responses
6
4
2
0
0 (0%)
3 (21.4%)
6 (42.9%)
4 (28.6%)
1 (7.1%)
0 (0%)
1 2 3 4 5 6
Image 2:
14 responses
8
6
4
2
0
0 (0%)
1 (7.1%)
5 (35.7%)
7 (50%)
1 (7.1%)
0 (0%)
1 2 3 4 5 6

A.1.11 Question 11

Face

For the following section use the scales from 1-6 to judge how recognizable each image is to the reference image shown below.

Reference Image:

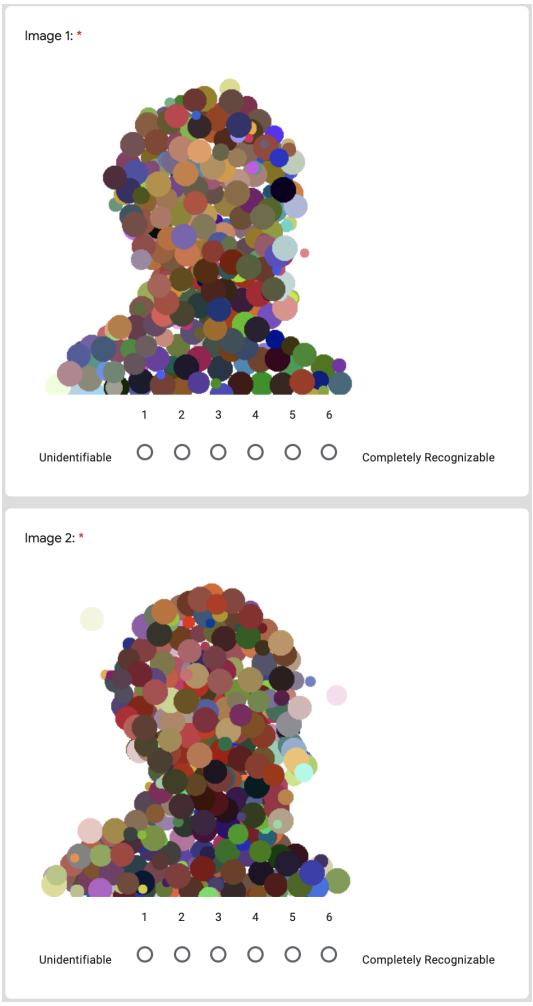
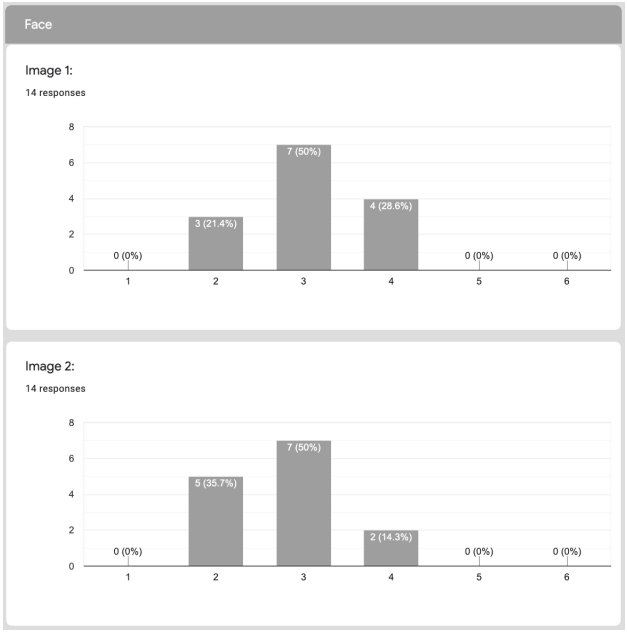


Image 1: *
1 2 3 4 5 6
Unidentifiable
Completely Recognizable
Image 2: *
1 2 3 4 5 6
Unidentifiable
Completely Recognizable


Face
Image 1:
14 responses
8
6
4
2
0
0 (0%)
3 (21.4%)
7 (50%)
4 (28.6%)
0 (0%)
0 (0%)
1 2 3 4 5 6
Image 2:
14 responses
8
6
4
2
0
0 (0%)
5 (35.7%)
7 (50%)
2 (14.3%)
0 (0%)
0 (0%)
1 2 3 4 5 6

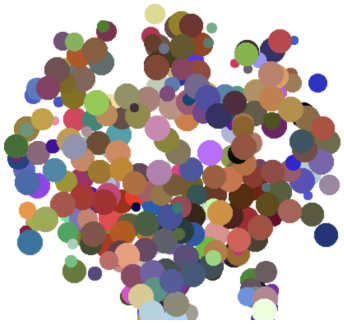
A.1.12 Question 12

King's

For the following section use the scales from 1-6 to judge how recognizable each image is to the reference image shown below.

Reference Image:

Image 1: *



1 2 3 4 5 6

Unidentifiable ○ ○ ○ ○ ○ ○ Completely Recognizable

Image 2: *

1 2 3 4 5 6

Unidentifiable ○ ○ ○ ○ ○ ○ Completely Recognizable

King's

Image 1:

14 responses

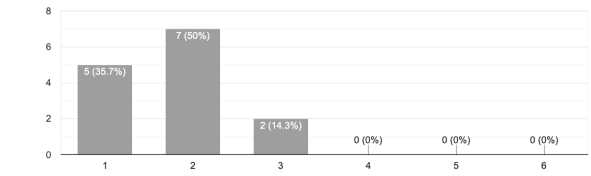
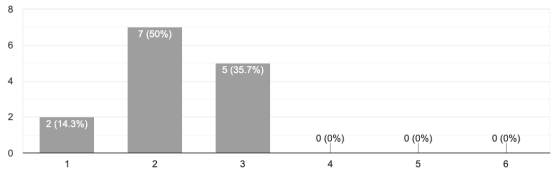


Image 2:

14 responses



Appendix B

User Guide

B.1 Instructions

Libraries Required:

- PIL (Python Image Library)
- Vpython
- numpy
- multiprocessing
- random
- os
- datetime
- copy

Python Version Required: Python 3.8.1 (or higher)

Use pip3 to install the required libraries.

IMPORTANT: The "results" folder generated during the execution of the program will overwrite any previous results still stored in the project folder. To save the results, rename the folder or move it to a different directory.

B.1.1 How to Create Digital Art Using a Single Reference Image

Step 1: Navigate to the project folder

Step 2: Select one of the included reference images provided in the project file that you would like to use and rename it to "reference1.png". If you would like to use your own reference image, add your desired reference image to the project folder and rename it to "reference1.png". *The reference image you use must be of type PNG.*

Step 3: At this point the python file is ready to be executed using the default Control Variables. If you would like to use the default Control Variable skip to Step 7.

Step 4: To change the default Control Variables open the python file **"pyArt-1.py"**.

Step 6: Change the values under the Control Variables to your desired settings and save your changes. Figure B.1 is a screenshot of the Control Variables at their default settings.

```
# ------------------------------------------------------------------------------------------
# Control Variables
# ------------------------------------------------------------------------------------------

# Genetic Variables
POP_SIZE = 50
MUTATION_CHANCE = 0.05
ADD_GENE_CHANCE = 0.7
REM_GENE_CHANCE = 0.8
INITIAL_GENES = 50

# How often to output images
GENERATIONS_PER_IMAGE = 100

# How many generations to run for
# Set to -1 if you do not want to stop automatically
GENERATIONS_MAX = 4000

# The type of shape to use for the Genes
# Options: [Circle, Square, Triangle]
GENE_TYPE = "Circle"
```

Figure B.1: The Control Variables for pyArt-1.py

Step 7: Using your terminal, navigate to the project folder.

Step 8: Run the command "*python3 pyArt-1.py*" in your terminal.

Step 9: The terminal will display the current generation and the current accuracy to the reference image. To view the results, open the newly created "results" folder in the project folder. The images will be saved as *[Current Generation].png*. The log file *log.txt* in the "results" folder contains comma separated data of the best Organism each generation.

Step 10: The program will stop executing once it has reached the maximum generations. To stop the program before press *Control + C* on your keyboard in the terminal window.

B.1.2 How to Create Digital Anamorphic Sculptures Using Two Reference Images

Step 1: Navigate to the project folder

Step 2: Select two of the included reference images provided in the project file that you would like to use and rename them to "reference1.png" and "reference2.png". If you would like to use your own reference images, add your desired reference image to the project folder and rename them to "reference1.png" and "reference2.png". *The reference images you use must be of type PNG and be equally sized.*

Step 3: At this point the python file is ready to be executed using the default Control Variables. If you would like to use the default Control Variable skip to Step 7.

Step 4: To change the default Control Variables open the python file **"pyArt-2.py"**.

Step 6: Change the values under the Control Variables to your desired settings and save your changes. Figure B.2 is a screenshot of the Control Variables at their default settings.

```
# ------------------------------------------------------------------------------------------
# Control Variables
# ------------------------------------------------------------------------------------------

# Genetic Variables
POP_SIZE = 10
MUTATION_CHANCE = 0.1
ADD_GENE_CHANCE = 0.8
REM_GENE_CHANCE = 0.7
INITIAL_GENES = 50

# Toggle for variable mutation rate
VARIABLE_MUT = True

# How often the variable mutation rate decreases by 0.01
VARIABLE_MUT_RATE = 500

# Toggle to set limit to the size the genes can grow
BOUND_GENE_SIZE = True

# How often to output images
GENERATIONS_PER_IMAGE = 100

# How often to 3D render the organism
GENERATIONS_PER_RENDER = 10

# How many generations to run for
# Set to -1 if you do not want to stop automatically
GENERATIONS_MAX = 15000
```

Figure B.2: The Control Variables for pyArt-2.py and pyArt-3.py

Step 7: Using your terminal, navigate to the project folder.

Step 8: Run the command "*python3 pyArt-2.py*" in your terminal.

Step 9: A browser window will pop up and display the 3D Scene.

Step 10: You may need to slightly zoom out the camera in order to view the whole digital sculpture. The instructions next to the 3D scene explain how to manipulate the camera.

Step 11: The terminal will also display the current generation and the current accuracies to the reference images. To view the saved results of the digital anamorphic sculptures seen from each of the perspectives, open the newly created "results" folder in the project folder. The images will be saved as *P[x]_[Current Generation].png* where P[x] indicates which perspective

the image is viewing the sculpture from. The log file *log.txt* in the "results" folder contains comma separated data of the best Organism each generation.

Step 10: The program will pause once it has reached the maximum generations. To unpause use the "run/pause" button provided. To stop the program completely, simply close the browser window or press *Control + C* on your keyboard in the terminal window.

B.1.3 How to Create Digital Anamorphic Sculptures Using Three Reference Images

Step 1: Navigate to the project folder

Step 2: Select three of the included reference images provided in the project file that you would like to use and rename them to "reference1.png", "reference2.png" and "reference3.png". If you would like to use your own reference images, add your desired reference image to the project folder and rename them to "reference1.png", "reference2.png" and "reference3.png". *The reference images you use must be of type PNG and be equally sized.*

Step 3: At this point the python file is ready to be executed using the default Control Variables. If you would like to use the default Control Variable skip to Step 7.

Step 4: To change the default Control Variables open the python file **"pyArt-3.py"**.

Step 6: Change the values under the Control Variables to your desired settings and save your changes. Figure B.2 is a screenshot of the Control Variables at their default settings.

Step 7: Using your terminal, navigate to the project folder.

Step 8: Run the command "*python3 pyArt-3.py*" in your terminal.

Step 9: A browser window will pop up and display the 3D Scene.

Step 10: You may need to slightly zoom out the camera in order to view the whole digital sculpture. The instructions next to the 3D scene explain how to manipulate the camera.

Step 11: The terminal will also display the current generation and the current accuracies to the reference images. To view the saved results of the digital anamorphic sculptures seen from each of the perspectives, open the newly created "results" folder in the project folder. The images will be saved as *P[x]_[Current Generation].png* where P[x] indicates which perspective the image is viewing the sculpture from. The log file *log.txt* in the "results" folder contains comma separated data of the best Organism each generation.

Step 10: The program will pause once it has reached the maximum generations. To unpause use the "run/pause" button provided. To stop the program completely, simply close the browser window or press *Control + C* on your keyboard in the terminal window.

Appendix C

Source Code

C.1 pyArt-1.py

This file is used to create digital art using a single reference image.

```
#
    ------------------------------------------------------------------------

# Imports
#
    ------------------------------------------------------------------------

import os
import sys
import random
from copy import deepcopy
import multiprocessing
from datetime import datetime

import numpy
from PIL import Image, ImageDraw, ImageChops

#
```

```
# ------------------------------------------------------------------------

# Control Variables
#
# ------------------------------------------------------------------------


# Genetic Variables
POP_SIZE = 50
MUTATION_CHANCE = 0.05
ADD_GENE_CHANCE = 0.7
REM_GENE_CHANCE = 0.8
INITIAL_GENES = 50

# How often to output images
GENERATIONS_PER_IMAGE = 100

# How many generations to run for
# Set to -1 if you do not want to stop automatically
GENERATIONS_MAX = 4000

# The type of shape to use for the Genes
# Options: [Circle, Square, Triangle]
GENE_TYPE = "Circle"

#
# ------------------------------------------------------------------------

# Reference Image
#
# ------------------------------------------------------------------------
```

```
try:
    globalTarget = Image.open("reference1.png")
    if globalTarget.mode == "RGBA":
        globalTarget = globalTarget.convert('RGB')
except IOError:
    print("The files reference1.png must be located in the same directory as pyArt-1.py.")
    exit()


#
    ------------------------------------------------------------------------


# Point and Color Classes
#
    ------------------------------------------------------------------------


class Point:
    """
        A 2D point. Has an X and Y coordinate.
    """

    def __init__(self, x, y):
        self.x = x
        self.y = y


class Color:
    """
        A color. Has an R, G and B color value.
    """

    def __init__(self, r, g, b):
        self.r = r
```

```
        self.g = g
        self.b = b


#
    ------------------------------------------------------------------------

# Gene Class and SubClasses
#
    ------------------------------------------------------------------------

class Gene:
    """
        A Gene is the part of the Organism that can be mutated.
    """

    def __init__(self, size):
        # Used to know the maximum position values
        self.size = size

        # The color of the Gene in RGB
        self.color = Color(random.randint(0, 255), random.randint(0, 255), random.randint
            (0, 255))

        # The depth of the gene determines when it is drawn on the canvas relative to the
            other Genes
        # Genes with a larger depth are drawn first so that the subsequent Genes can be
            drawn in front of them
        # We reuse the size of the image in the X dimension to bound the "imaginary" Z
            dimension
        self.depth = random.randint(0, size[0])

    # abstract method
```

```
    def mutate(self):
        pass

    def mutateColor(self):
        r = min(max(0, int(round(random.gauss(self.color.r, 10)))), 255)
        g = min(max(0, int(round(random.gauss(self.color.g, 10)))), 255)
        b = min(max(0, int(round(random.gauss(self.color.b, 10)))), 255)

        self.color = Color(r, g, b)

    def mutateDepth(self):
        self.depth = min(max(0, int(round(random.gauss(self.depth, 3)))), self.size[0])


class Circle(Gene):
    """
        A circle Gene. This Gene has a size, diameter, color, and position on the canvas.
    """

    def __init__(self, size):
        super().__init__(size)

        self.diameter = random.randint(5, 15)
        self.pos = Point(random.randint(0, size[0]), random.randint(0, size[1]) )
        self.params = ["diameter", "color", "pos"]

    def mutate(self):
        # Decide which variable of the Gene to apply the mutation
        mutation_type = random.choice(self.params)

        # Mutate the variables
        if mutation_type == "diameter":
            self.diameter = min(max(5, int(round(random.gauss(self.diameter, 2)))), 15)
```

```
        elif mutation_type == "color":
            self.mutateColor()

        elif mutation_type == "pos":
            x = min(max(0, int(round(random.gauss(self.pos.x, 3)))), self.size[0])
            y = min(max(0, int(round(random.gauss(self.pos.y, 3)))), self.size[1])
            self.pos = Point(x, y)

            # Depth is a sudo z coordinate for the 2-dimensional Gene so mutating the
            # position should also changes the depth
            self.mutateDepth()


class Square(Gene):
    """
        A square Gene. This Gene has a size, color, diagonal length and the position of it'
            s corners on the canvas.
    """

    def __init__(self, size):

        super().__init__(size)

        # The positions of the top left corner of the square on the canvas
        self.pos1 = Point(random.randint(0, size[0]), random.randint(0, size[1]))

        # The diagonal length of the square
        self.diagonal = random.randint(5, 30)

        # The positions of the bottom right corner of the square on the canvas
        self.pos2 = self.calcPos2()
        self.params = ["pos", "color", "diagonal"]
```

```
    # For a square the second position is relative to the first position it's diagonal
        length
    def calcPos2(self):
        x = self.pos1.x + self.diagonal
        y = self.pos1.y + self.diagonal
        return Point(min(x, self.size[0]), min(y, self.size[1]))

    def mutate(self):
        # Decide which variable of the Gene to apply the mutation
        mutation_type = random.choice(self.params)

        # Mutate the variable
        if mutation_type == "pos":
            x = min(max(0, int(round(random.gauss(self.pos1.x, 3)))), self.size[0])
            y = min(max(0, int(round(random.gauss(self.pos1.y, 3)))), self.size[1])

            self.pos1 = Point(x, y)
            self.pos2 = self.calcPos2()

            # Depth is a sudo z coordinate for the 2-dimensional Gene so mutating the
                position also changes the depth
            self.mutateDepth()

        elif mutation_type == "color":
            self.mutateColor()

        elif mutation_type == "diagonal":
            self.diagonal = min(max(5, int(round(random.gauss(self.diagonal, 2)))), 30)


class Triangle(Gene):
    """
```

```
    A triangle Gene. This Gene has a color and 3 positions on the canvas which together
        form a triangle.
"""

def __init__(self, size):

    super().__init__(size)

    self.pos1 = Point(random.randint(0, size[0]), random.randint(0, size[1]))
    # Positions 2 and 3 are initialized relative to position 1 to ensure the triangles
        do not spawn to large
    self.pos2 = self.calcInitialPos()
    self.pos3 = self.calcInitialPos()

    self.params = ["pos", "color"]

def calcInitialPos(self):
    x = min(max(0, int(round(random.gauss(self.pos1.x, 10)))), self.size[0])
    y = min(max(0, int(round(random.gauss(self.pos1.y, 10)))), self.size[1])
    return Point(x, y)

def mutate(self):
    # Decide which variable of the Gene to apply the mutation
    mutation_type = random.choice(self.params)

    # Mutate the variable
    if mutation_type == "pos":
        x1 = min(max(0, int(round(random.gauss(self.pos1.x, 3)))), self.size[0])
        y1 = min(max(0, int(round(random.gauss(self.pos1.y, 3)))), self.size[1])
        self.pos1 = Point(x1, y1)

        x2 = min(max(0, int(round(random.gauss(self.pos2.x, 3)))), self.size[0])
        y2 = min(max(0, int(round(random.gauss(self.pos2.y, 3)))), self.size[1])
```

```
            self.pos2 = Point(x2, y2)

            x3 = min(max(0, int(round(random.gauss(self.pos3.x, 3)))), self.size[0])
            y3 = min(max(0, int(round(random.gauss(self.pos3.y, 3)))), self.size[1])
            self.pos3 = Point(x3, y3)

            # Depth is a sudo z coordinate for the 2-dimensional Gene so mutating the
                position also changes the depth
            self.mutateDepth()

        elif mutation_type == "color":
            self.mutateColor()


#
    ------------------------------------------------------------------------

# Organism Class
#
    ------------------------------------------------------------------------

class Organism:
    """
        The Organism consists of a collection of Genes that work together in order to
            produce the image.
    """

    def __init__(self, size, num):
        self.size = size
        self.score = 0

        # Create random Genes up to the number given
        if GENE_TYPE == "Circle":
```

```
            self.genes = [Circle(size) for _ in range(num)]
        elif GENE_TYPE == "Square":
            self.genes = [Square(size) for _ in range(num)]
        elif GENE_TYPE == "Triangle":
            self.genes = [Triangle(size) for _ in range(num)]
        else:
            print("The GENE_TYPE is not a valid type.")
            sys.exit()

    def mutate(self):
        # Each Gene has a random chance of mutating
        # This is statistically equivalent and faster than looping through all the Genes
        mutSampleSize = max(0, int(round(random.gauss(int(len(self.genes) *
            MUTATION_CHANCE), 1))))
        for g in random.sample(self.genes, mutSampleSize):
            g.mutate()

        # An Organism also has a chance to add or remove a random Gene
        if ADD_GENE_CHANCE > random.random():
            if GENE_TYPE == "Circle":
                self.genes.append(Circle(self.size))
            elif GENE_TYPE == "Square":
                self.genes.append(Square(self.size))
            elif GENE_TYPE == "Triangle":
                self.genes.append(Triangle(self.size))
        if len(self.genes) > 0 and REM_GENE_CHANCE > random.random():
            self.genes.remove(random.choice(self.genes))

    def drawImage(self):
        """
            Using PIL, uses the Genes to draw the Organism as a 2-dimensional image.
        """
        image = Image.new("RGB", self.size, (255, 255, 255))
```

```
canvas = ImageDraw.Draw(image)

# Sort the Genes by their Z coordinate positions so that they can be drawn in the
    correct order
# Genes in the 'back' of the image get drawn first so that the Genes in 'front' get
    drawn on top of them
sortedGenes = sorted(self.genes, key=lambda x: x.depth, reverse=True)

if GENE_TYPE == "Circle":
    for g in sortedGenes:
        color = (g.color.r, g.color.g, g.color.b)
        canvas.ellipse([g.pos.x - g.diameter, g.pos.y - g.diameter, g.pos.x + g.
            diameter, g.pos.y + g.diameter],
                       outline=color, fill =color)

elif GENE_TYPE == "Square":
    for g in self.genes:
        color = (g.color.r, g.color.g, g.color.b)
        canvas.rectangle([(g.pos1.x, g.pos1.y), (g.pos2.x, g.pos2.y)], outline=
            color, fill =color)

elif GENE_TYPE == "Triangle":
    for g in self.genes:
        color = (g.color.r, g.color.g, g.color.b)
        canvas.polygon([(g.pos1.x, g.pos1.y),
                        (g.pos2.x, g.pos2.y),
                        (g.pos3.x, g.pos3.y)],
                       outline=color, fill =color)
else:
    print("The GENE_TYPE is not a valid type.")
    sys.exit()

return image
```

```
#
    ------------------------------------------------------------------------

# Main Functions
#
    ------------------------------------------------------------------------

def run(cores):
    """
        Contains the loop that creates and tests new generations.
    """

    # Create results directory in current directory
    if not os.path.exists("results"):
        os.mkdir("results")

    # Create log file
    f = open(os.path.join("results", "log.txt"), 'a')
    f.write("Generation,Accuracy,Genes\n")

    # Start the timer to calculate the run time of the algorithm
    startTime = datetime.now()

    # Create the first parent Organism (with random Genes)
    generation = 1
    parent = Organism(globalTarget.size, INITIAL_GENES)

    # Save an image of the initial Organism to the results directory
    parent.drawImage().save(os.path.join("results", "{}.png".format(generation)))

    # Calculate the score of the Organism
```

```
parent.score = calcScore(parent.drawImage(), globalTarget)

# Calculate initial accuracy
accuracy = calcAccuracy(parent.score)

# Setup the multiprocessing pool
p = multiprocessing.Pool(cores)

# Infinite loop (until the process is interrupted)
while True:
    # Print the current score and write it to the log file
    print("Generation {} - R1: {}%".format(generation, accuracy))
    f.write("{},{},{}\n".format(generation, accuracy, len(parent.genes)))

    # Save an image of the current best Organism to the results directory
    if generation % GENERATIONS_PER_IMAGE == 0:
        parent.drawImage().save(os.path.join("results", "{}.png".format(generation)))

    # Stop program once we are on the max generation
    if generation == GENERATIONS_MAX:
        sys.exit()

    # Increment the generation count
    generation += 1

    # Create the new population and add parent to the collection
    # in case all children mutations result in worse scores
    population = [parent]

    # Generate the children in the new population by applying mutations to the parent
        )rganism
    try:
        newChildren = generatePopulation(parent, POP_SIZE - 1, p)
```

```
            except KeyboardInterrupt:
                # Print the final gene count and total run time
                print("Final Gene Count: {}".format(len(parent.genes)))
                print("Total Runtime: {}\n".format(datetime.now() - startTime))
                p.close()
                return

            # Add the new children to the population
            population.extend(newChildren)

            # Sort the population of Organisms by their fitness.
            sortedPopulation = sorted(population, key=lambda x: fitness(x))

            # Get the Organism that minimises the fitness value the most and
            # assign it to be the parent for the next generation
            parent = sortedPopulation[0]

            # Update the accuracy
            accuracy = calcAccuracy(parent.score)


def fitness(o):
    """
        This functions calculates the fitness of the individual Organisms in the
            population.
        The fitness is the score for the reference image added to the
        number of Genes the Organisms has multiplied by a weight. The number of Genes is
            included
        in the score to encourage the Organisms to use the fewest amount of Genes possible
            .
        The best Organism is the one that is able to minimise this value the most.
    """
    return o.score + len(o.genes) * 2
```

```
def calcAccuracy(score):
    """
        This functions takes the score values of an Organism and calculates how accurate
        the image produced by the Organism is to the reference images as a percentage.
        The accuracy is calculated by finding the inverse of the percentage difference.
    """

    # Calculate the total number of color values in the target image by calculating the
        number of pixels and
    # multiplying by 3 for the three RGB color values.
    imageDimensions = globalTarget.size[0] * globalTarget.size[1] * 3

    return 100 - ((score / 255.0 * 100) / imageDimensions)


def calcScore(im1, im2):
    """
        This function is used to determine how close the image produced by the Organism
            is to the reference image.
        It uses ImageChops and numpy to quickly compute the sum of the differences
        in the pixels between the two images.
    """
    difImg = ImageChops.difference(im1, im2)
    diff = numpy.sum(difImg)
    return diff


def createChild(o):
    """
        Takes a parent Organism and mutates it's Genes to create a child Organism. The
            child Organism is then
```

```
        tested on the reference image to determine its score.
    """
    try:
        child = deepcopy(o)
        child.mutate()
        img = child.drawImage()
        child.score = calcScore(img, globalTarget)
        return child
    except KeyboardInterrupt as e:
        pass


def generatePopulation(o, number, p):
    """
    Uses the multiprocessing module to create a new population of Organisms.
    """
    population = p.map(createChild, [o] * int(number))
    return population


#
    ------------------------------------------------------------------------


# Main Function
#
    ------------------------------------------------------------------------


if __name__ == "__main__":

    # Set number of available cores
    cores = max(1, multiprocessing.cpu_count() - 1)

    # run the Genetic Algorithm
```

```
    run(cores)
```

C.2 pyArt-2.py

This file is used to create digital anamorphic sculptures using 2 reference images simultaneously.

```
#
    ------------------------------------------------------------------------

# Imports
#
    ------------------------------------------------------------------------

from vpython import *

from datetime import datetime
import os
import random
from copy import deepcopy
import multiprocessing

import numpy
from PIL import Image, ImageDraw, ImageChops

#
    ------------------------------------------------------------------------

# Control Variables
#
    ------------------------------------------------------------------------


# Genetic Variables
POP_SIZE = 10
```

```
MUTATION_CHANCE = 0.1
ADD_GENE_CHANCE = 0.8
REM_GENE_CHANCE = 0.7
INITIAL_GENES = 50

# Toggle for variable mutation rate
VARIABLE_MUT = True

# How often the variable mutation rate decreases by 0.01
VARIABLE_MUT_RATE = 500

# Toggle to set limit to the size the genes can grow
BOUND_GENE_SIZE = True

# How often to output images
GENERATIONS_PER_IMAGE = 100

# How often to 3D render the organism
GENERATIONS_PER_RENDER = 10

# How many generations to run for
# Set to -1 if you do not want to stop automatically
GENERATIONS_MAX = 15000

# Allows us to run and pause the execution of the program
runProgram = True

#
    ------------------------------------------------------------

# Reference Images
#
    ------------------------------------------------------------
```

```
try:
    # Get the first reference image
    globalTarget1 = Image.open("reference1.png")
    if globalTarget1.mode == "RGBA":
        globalTarget1 = globalTarget1.convert('RGB')
    # Get the second reference image
    globalTarget2 = Image.open("reference2.png")
    if globalTarget2.mode == "RGBA":
        globalTarget2 = globalTarget2.convert('RGB')
except IOError:
    print("Files reference1.png and reference2.png must be located in the same directory as
        pyArt-2.py.")
    exit()


#
    ----------------------------------------------------------------------

# Point and Color Classes
#
    ----------------------------------------------------------------------

class Point:
    """
        A 3D point. Has an X, Y and Z coordinate.
    """

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

```
class Color:
    """
        A color. Has an R, G and B color value.
    """

    def __init__(self, r, g, b):
        self.r = r
        self.g = g
        self.b = b


#
    ------------------------------------------------------------------------

# Genetic Classes
#
    ------------------------------------------------------------------------

class Gene:
    """
        A Gene is the part of the Organism that can be mutated.
        This Gene has a size, diameter, color, and position in the 3D sculpture.
    """

    def __init__(self, size):
        # Used to know the maximum position values
        self.size = size

        self.diameter = random.randint(5, 15)
        # Reusing the size of the image in the X dimension to bound the "imaginary" Z
            dimension
```

```
            self.pos = Point(random.randint(0, size[0]), random.randint(0, size[1]), random.
                randint(0, size[0]))
            self.color = Color(random.randint(0, 255), random.randint(0, 255), random.randint
                (0, 255))
            self.params = ["diameter", "pos", "color"]

        def mutate(self):
            # Decide which variable will be mutated
            mutation_type = random.choice(self.params)

            # Mutate the variable
            if mutation_type == "diameter":
                if BOUND_GENE_SIZE:
                    self.diameter = min(max(5, int(round(random.gauss(self.diameter, 2)))),
                        15)
                else:
                    self.diameter = max(5, int(round(random.gauss(self.diameter, 2))))

            elif mutation_type == "pos":
                x = min(max(0, int(round(random.gauss(self.pos.x, 5)))), self.size[0])
                y = min(max(0, int(round(random.gauss(self.pos.y, 5)))), self.size[1])
                z = min(max(0, int(round(random.gauss(self.pos.z, 5)))), self.size[0])
                self.pos = Point(x, y, z)

            elif mutation_type == "color":
                r = min(max(0, int(round(random.gauss(self.color.r, 20)))), 255)
                g = min(max(0, int(round(random.gauss(self.color.g, 20)))), 255)
                b = min(max(0, int(round(random.gauss(self.color.b, 20)))), 255)
                self.color = Color(r, g, b)


    class Organism:
        """
```

```
        The Organism consists of a collection of Genes that work together in order to
            produce the images and 3D sculpture.
    """

    def __init__(self, size, num):
        self.size = size
        # Create random Genes up to the number given
        self.genes = [Gene(size) for _ in range(num)]
        # The score is first initialized to (0, 0) and is then calculated later
        self.score = (0, 0)

    def mutate(self):
        # Each Gene has a random chance of mutating
        # This is statistically equivalent and faster than looping through all the Genes
        mutSampleSize = max(0, int(round(random.gauss(int(len(self.genes) *
            MUTATION_CHANCE), 1))))
        for g in random.sample(self.genes, mutSampleSize):
            g.mutate()

        # The organism also has a chance to add or remove a Gene when it mutates
        if ADD_GENE_CHANCE > random.random():
            self.genes.append(Gene(self.size))
        if len(self.genes) > 0 and REM_GENE_CHANCE > random.random():
            self.genes.remove(random.choice(self.genes))

    def drawImage(self, perspective):
        """
            Using PIL, the Genes draw the Organism as a 2-dimensional image from a
                specified perspective.
        """
        image = Image.new("RGB", self.size, (255, 255, 255))
        canvas = ImageDraw.Draw(image)
```

```
        if perspective == 1:
            # Sort the Genes by their Z coordinate positions so that they can be drawn in
                the correct order
            sortedGenes = sorted(self.genes, key=lambda x: x.pos.z, reverse=True)

            for g in sortedGenes:
                color = (g.color.r, g.color.g, g.color.b)
                canvas.ellipse ([g.pos.x - g.diameter, g.pos.y - g.diameter, g.pos.x + g.
                    diameter, g.pos.y + g.diameter],
                                outline=color, fill =color)
        else:
            # Sort the Genes by their X coordinate positions so that they can be drawn in
                the correct order
            sortedGenes = sorted(self.genes, key=lambda x: x.pos.x, reverse=True)

            for g in sortedGenes:
                color = (g.color.r, g.color.g, g.color.b)
                canvas.ellipse (
                    [(self.size[0] - g.pos.z) - g.diameter, g.pos.y - g.diameter, (self.
                        size[0] - g.pos.z) + g.diameter,
                     g.pos.y + g.diameter],
                    outline=color, fill =color)

        return image


#
    ------------------------------------------------------------------------

# Main Functions
#
    ------------------------------------------------------------------------
```

```
def run(cores):
    """
        Contains the loop that creates and tests new generations.
    """
    global runProgram

    # Create results directory in current directory
    if not os.path.exists("results"):
        os.mkdir("results")

    # Create log file
    logFile = open(os.path.join("results", "log.txt"), 'a')
    logFile.write("Generation,Acc1,Acc2,Genes\n")

    # Start the timer to calculate the run time of the algorithm
    startTime = datetime.now()

    # Create the parent Organism (with random Genes)
    # Can use size of either target image as they must be the same
    generation = 1
    parent = Organism(globalTarget1.size, INITIAL_GENES)

    # Save an image of the initial Organism
    parent.drawImage(1).save(os.path.join("results", "P1_{}.png".format(generation)))
    parent.drawImage(2).save(os.path.join("results", "P2_{}.png".format(generation)))

    # Calculate the score of the Organism from the different perspectives compared to the
        two target images
    parent.score = (calcScore(parent.drawImage(1), globalTarget1), calcScore(parent.
        drawImage(2), globalTarget2))

    # Calculate initial accuracy
    accuracy = calcAccuracy(parent.score)
```

```
# Create the counters for the display
generationCounter = wtext(pos=display.title_anchor, text="\n<b>Generation:</b>
    {}\n".format(generation))
refAccuracy1 = wtext(pos=display.title_anchor, text="<b>Accuracy to reference1:</b
    > {}%\n".format(accuracy[0]))
refAccuracy2 = wtext(pos=display.title_anchor, text="<b>Accuracy to reference2:</b
    > {}%\n".format(accuracy[1]))
geneCounter = wtext(pos=display.title_anchor, text="<b>Number of Genes:</b> {}\n
    ".format(len(parent.genes)))

# Create the curves for the Graph
accuracyCurve1 = gcurve(color=color.blue, width=4, label='Ref. 1')
accuracyCurve2 = gcurve(color=color.red, width=4, label='Ref. 2')

# Render the initial Organism in 3D
renderOrganism(parent)

# Setup the multiprocessing pool
p = multiprocessing.Pool(cores)

# Infinite loop (until the process is interrupted)
while True:
    # Pause and run the GA
    if runProgram:
        # Print the current score and write generation info to the log file
        print("Generation {} - R1: {}%, R2: {}%".format(generation, round(accuracy
            [0], 3), round(accuracy[1], 3)))
        logFile.write("{},{},{},{}\n".format(generation, accuracy[0], accuracy[1], len(
            parent.genes)))

        # Save an image of the current best Organism to the results directory
        if generation % GENERATIONS_PER_IMAGE == 0:
```

```
        parent.drawImage(1).save(os.path.join("results", "P1_{}.png".format(
            generation)))
        parent.drawImage(2).save(os.path.join("results", "P2_{}.png".format(
            generation)))

    # Update the accuracy
    accuracy = calcAccuracy(parent.score)

    # Update the display counters
    generationCounter.text = "\n<b>Generation:</b> {}\n".format(generation)
    refAccuracy1.text = "<b>Accuracy to reference1:</b> {}%\n".format(accuracy
        [0])
    refAccuracy2.text = "<b>Accuracy to reference2:</b> {}%\n".format(accuracy
        [1])
    geneCounter.text = "<b>Number of Genes:</b> {}\n".format(len(parent.genes)
        )

    # Render the current best Organism in 3D
    if generation % GENERATIONS_PER_RENDER == 0:
        renderOrganism(parent)
        # Update the graph
        accuracyCurve1.plot(generation, accuracy[0])
        accuracyCurve2.plot(generation, accuracy[1])

    # Pause program once we are on the max generation
    if generation == GENERATIONS_MAX:
        logFile.write("Total Runtime: {}\n".format(datetime.now() - startTime))
        runProgram = False

    # Decrease mutation rate if using variable mutation
    if VARIABLE_MUT and generation % VARIABLE_MUT_RATE == 0:
        global MUTATION_CHANCE
        if MUTATION_CHANCE > 0.01:
```

```
                MUTATION_CHANCE = MUTATION_CHANCE - 0.01

                # Increment the generation count
                generation += 1

                # Create the new population and add parent to the collection
                # in case all children mutations result in worse scores
                population = [parent]

                # Generate the children in the new population by applying mutations to the
                    parent organism
                try:
                    newChildren = generatePopulation(parent, POP_SIZE - 1, p)
                except KeyboardInterrupt:
                    # Print the final Gene count and total run time
                    print("Final Gene Count: {}".format(len(parent.genes)))
                    print("Total Runtime: {}\n".format(datetime.now() - startTime))
                    p.close()
                    return

                # Add the new children to the population
                population.extend(newChildren)

                # Sort the population of Organisms by their fitness.
                sortedPopulation = sorted(population, key=lambda x: fitness(x))

                # Get the Organism that minimises the fitness value the most and
                # assign it to be the parent for the next generation
                parent = sortedPopulation[0]


def fitness(o):
    """
```

```
        This functions calculates the fitness of the individual Organisms in the
            population.
        The fitness is the sum of the scores for each of the reference images added to the
        number of Genes the Organisms has multiplied by a weight. The number of Genes is
            included
        in the score to encourage the Organisms to use the fewest amount of Genes possible
            .
        The best Organism is the one that is able to minimise this value the most.
    """
    return o.score[0] + o.score[1] + len(o.genes) * 2


def calcAccuracy(score):
    """
        This functions takes the score values of an Organism and calculates how accurate
        the image produced by the Organism is to the reference images as a percentage.
        The accuracy is calculated by finding the inverse of the percentage difference.
    """

    # Calculate the total number of color values in the target images by calculating the
        number of pixels and
    # multiplying by 3 for the three RGB color values. Only need size of one, images must
        be same sized
    imageDimensions = globalTarget1.size[0] * globalTarget1.size[1] * 3

    # Since the score is a calculation of the sum of the differences between the two
        images
    # we are easily able to calculate the % difference and then use the inverse of this for
         the accuracy
    return (100 - ((score[0] / 255.0 * 100) / imageDimensions)), (100 - ((score[1] / 255.0
        * 100) / imageDimensions))
```

```
def calcScore(im1, im2):
    """
        This function is used to determine how close the image produced by the Organism
            is to the reference image.
        It uses ImageChops and numpy to quickly compute the sum of the differences
        in the pixels between the two images.
    """
    difImg = ImageChops.difference(im1, im2)
    diff = numpy.sum(difImg)
    return diff



def createChild(o):
    """
        Takes a parent Organism and mutates it's Genes to create a child Organism. The
            child Organism is then
        tested on both reference images and given a score.
    """
    try:
        child = deepcopy(o)
        child.mutate()
        img1 = child.drawImage(1)
        score1 = calcScore(img1, globalTarget1)
        img2 = child.drawImage(2)
        score2 = calcScore(img2, globalTarget2)
        child.score = (score1, score2)
        return child
    except KeyboardInterrupt as e:
        pass



def generatePopulation(o, number, p):
    """
```

```
        Uses the multiprocessing module to create a new population of Organisms in parallel
            .
    """
    population = p.map(createChild, [o] * int(number))
    return population


#
    ----------------------------------------------------------------------

# Vpython helper methods
#
    ----------------------------------------------------------------------


# Toggle the execution of the main program
def runButton(b):
    global runProgram
    if b.text == 'Pause':
        runProgram = False
        b.text = 'Run'
    else:
        runProgram = True
        b.text = 'Pause'


# Deletes all the objects in the scene
def clearScene():
    for obj in display.objects:
        obj.visible = False
        del obj
```

```
# Takes an Organism and renders it in Vpython
def renderOrganism(organism):
    # Clear the scene from any previously rendered Organisms
    clearScene()

    # Find the midpoint of the 3D image so that we can subtract it from each
    # point position so that the rendered image is centered around (0,0,0).
    xTranslate = organism.size[0] // 2
    yTranslate = organism.size[1] // 2
    # Reusing xTranslate for zTranslate
    zTranslate = xTranslate

    for g in organism.genes:
        # Reformat Gene attributes to be compatible with Vpython
        p = vec((g.pos.x - xTranslate), (0 - (g.pos.y - yTranslate)), (0 - (g.pos.z -
            zTranslate)))
        # Doubling the radius because the size of the balls was too small before
        r = g.diameter
        c = vec((g.color.r / 255), (g.color.g / 255), (g.color.b / 255))

        # Create sphere object from Gene
        sphere(pos=p, radius=r, color=c)


# Takes a screen shot of the 3D scene and saves it as a .png in your downloads folder
def takeScreenShot():
    display.capture("PyArt2")


#
    ------------------------------------------------------------------------


# Main Function
```

```
#
    ------------------------------------------------------------------------

if __name__ == "__main__":
    # Set number of available cores
    cores = max(1, multiprocessing.cpu_count() - 1)

    # Create the 3D scene
    display = canvas()
    display.width = display.height = 600
    display.background = color.white
    display.align = 'left'
    display.range = 1

    # Setting the FOV to be small to produce a Orthographic projection
    display.fov = 0.01

    # lighting
    display.lights = []  # gets rid of default lighting (shadows and distant white light)
    display.ambient = color.gray(1)  # adds bright ambient light

    # Add title
    display.title = '<b>PyArt:</b> Using a Genetic Algorithm to create digital art from
        multiple reference \n' \
                    'images in the style of Pointillism rendered in 3D.'

    # buttons
    button(text='Pause', bind=runButton)
    button(text='Screen Shot', bind=takeScreenShot)

    # Add a Graph to visualize the improvements
    graph(title='Rate of Improvements', xtitle='Generations', ytitle='Accuracy', align='
        right', fast=False)
```

```
    # Add instructions
    display.append_to_caption("""
    <b>Instructions:</b>
    To rotate "camera", drag with right button or Ctrl-drag.
    To zoom, drag with middle button or Alt/Option depressed, or use scroll wheel.
      On a two-button mouse, middle is left + right.
    To pan left/right and up/down, Shift-drag.
    Touch screen: pinch/extend to zoom, swipe or two-finger rotate.""")

    # run the genetic algorithm
    run(cores)
```

C.3 pyArt-3.py

This file is used to create digital anamorphic sculptures using 3 reference images simultaneously.

```
#
    ------------------------------------------------------------

# Imports
#
    ------------------------------------------------------------

from vpython import *

from datetime import datetime
import os
import random
from copy import deepcopy
import multiprocessing

import numpy
from PIL import Image, ImageDraw, ImageChops
```

```
#
    ------------------------------------------------------------------------

# Control Variables
#
    ------------------------------------------------------------------------


# Genetic Variables
POP_SIZE = 10
MUTATION_CHANCE = 0.1
ADD_GENE_CHANCE = 0.8
REM_GENE_CHANCE = 0.7
INITIAL_GENES = 50

# Toggle for variable mutation rate
VARIABLE_MUT = True

# How often the variable mutation rate decreases by 0.01
VARIABLE_MUT_RATE = 500

# Toggle to set limit to the size the Genes can grow
BOUND_GENE_SIZE = True

# How often to output images
GENERATIONS_PER_IMAGE = 100

# How often to 3D render the organism
GENERATIONS_PER_RENDER = 10

# How many generations to run for
# Set to -1 if you do not want to stop automatically
```

```
GENERATIONS_MAX = 15000

# Allows us to run and pause the execution of the program
runProgram = True

#
    ------------------------------------------------------------------------

# Reference Images
#
    ------------------------------------------------------------------------



try:
    # Get the first reference image
    globalTarget1 = Image.open("reference1.png")
    if globalTarget1.mode == "RGBA":
        globalTarget1 = globalTarget1.convert('RGB')
    # Get the second reference image
    globalTarget2 = Image.open("reference2.png")
    if globalTarget2.mode == "RGBA":
        globalTarget2 = globalTarget2.convert('RGB')
    # Get the third reference image
    globalTarget3 = Image.open("reference3.png")
    if globalTarget3.mode == "RGBA":
        globalTarget3 = globalTarget3.convert('RGB')
except IOError:
    print("Files reference1.png, reference2.png and reference3.png must be located in the
        same directory as pyArt-3.py.")
    exit()


#
```

```
    ------------------------------------------------------------------------

# Point and Color Classes
#
    ------------------------------------------------------------------------

class Point:
    """
        A 3D point. Has an X, Y and Z coordinate.
    """

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z


class Color:
    """
        A color. Has an R, G and B color value.
    """

    def __init__(self, r, g, b):
        self.r = r
        self.g = g
        self.b = b


#
    ------------------------------------------------------------------------

# Genetic Classes
#
```

```
# ------------------------------------------------------------------------------

class Gene:
    """
        A Gene is the part of the Organism that can be mutated.
        This Gene has a size, diameter, color, and position in the 3D sculpture.
    """

    def __init__(self, size):
        # Used to know the maximum position values
        self.size = size

        self.diameter = random.randint(5, 15)
        # Reusing the size of the image in the X dimension to bound the "imaginary" Z
            dimension
        self.pos = Point(random.randint(0, size[0]), random.randint(0, size[1]), random.
            randint(0, size[0]))
        self.color = Color(random.randint(0, 255), random.randint(0, 255), random.randint
            (0, 255))
        self.params = ["diameter", "pos", "color"]

    def mutate(self):
        # Decide which variable will be mutated
        mutation_type = random.choice(self.params)

        # Mutate the variable
        if mutation_type == "diameter":
            if BOUND_GENE_SIZE:
                self.diameter = min(max(5, int(round(random.gauss(self.diameter, 2)))),
                    15)
            else:
                self.diameter = max(5, int(round(random.gauss(self.diameter, 2))))
```

```
        elif mutation_type == "pos":
            x = min(max(0, int(round(random.gauss(self.pos.x, 5)))), self.size[0])
            y = min(max(0, int(round(random.gauss(self.pos.y, 5)))), self.size[1])
            z = min(max(0, int(round(random.gauss(self.pos.z, 5)))), self.size[0])
            self.pos = Point(x, y, z)

        elif mutation_type == "color":
            r = min(max(0, int(round(random.gauss(self.color.r, 20)))), 255)
            g = min(max(0, int(round(random.gauss(self.color.g, 20)))), 255)
            b = min(max(0, int(round(random.gauss(self.color.b, 20)))), 255)
            self.color = Color(r, g, b)


class Organism:
    """
        The Organism consists of a collection of Genes that work together in order to
            produce the images and 3D sculpture.
    """

    def __init__(self, size, num):
        self.size = size
        # Create random Genes up to the number given
        self.genes = [Gene(size) for _ in range(num)]
        # The score is first initialized to (0, 0, 0) and is then calculated later
        self.score = (0, 0, 0)

    def mutate(self):
        # Each Gene has a random chance of mutating
        # This is statistically equivalent and faster than looping through all the Genes
        mutSampleSize = max(0, int(round(random.gauss(int(len(self.genes) *
            MUTATION_CHANCE), 1))))
        for g in random.sample(self.genes, mutSampleSize):
            g.mutate()
```

```
        # The organism also has a chance to add or remove a Gene when it mutates
        if ADD_GENE_CHANCE > random.random():
            self.genes.append(Gene(self.size))
        if len(self.genes) > 0 and REM_GENE_CHANCE > random.random():
            self.genes.remove(random.choice(self.genes))


    def drawImage(self, perspective):
        """
            Using PIL, the Genes draw the Organism as a 2-dimensional image from a
                specified perspective.
        """
        image = Image.new("RGB", self.size, (255, 255, 255))
        canvas = ImageDraw.Draw(image)

        if perspective == 1:
            # Sort the Genes by their Z coordinate positions so that they can be drawn in
                the correct order
            sortedGenes = sorted(self.genes, key=lambda x: x.pos.z, reverse=True)

            for g in sortedGenes:
                color = (g.color.r, g.color.g, g.color.b)
                canvas.ellipse([g.pos.x - g.diameter, g.pos.y - g.diameter, g.pos.x + g.
                    diameter, g.pos.y + g.diameter],
                                outline=color, fill=color)
        elif perspective == 2:
            # Sort the Genes by their X coordinate positions so that they can be drawn in
                the correct order
            sortedGenes = sorted(self.genes, key=lambda x: x.pos.x, reverse=True)

            for g in sortedGenes:
                color = (g.color.r, g.color.g, g.color.b)
                canvas.ellipse(
```

```
                    [( self.size[0] - g.pos.z) - g.diameter, g.pos.y - g.diameter, (self.
                        size[0] - g.pos.z) + g.diameter,
                     g.pos.y + g.diameter],
                    outline=color, fill=color)
        else:
            # Sort the Genes by their Y coordinate positions so that they can be drawn in
                the correct order
            sortedGenes = sorted(self.genes, key=lambda x: x.pos.y, reverse=True)

            for g in sortedGenes:
                color = (g.color.r, g.color.g, g.color.b)
                canvas.ellipse([g.pos.x - g.diameter, g.pos.z - g.diameter, g.pos.x + g.
                    diameter, g.pos.z + g.diameter],
                                outline=color, fill=color)

        return image


#
    ------------------------------------------------------------------------

# Main Functions
#
    ------------------------------------------------------------------------

def run(cores):
    """
        Contains the loop that creates and tests new generations.
    """
    global runProgram

    # Create results directory in current directory
    if not os.path.exists("results"):
```

```
    os.mkdir("results")

# Create log file
logFile = open(os.path.join("results", "log.txt"), 'a')
logFile.write("Generation,Acc1,Acc2,Acc3,Genes\n")

# Start the timer to calculate the run time of the algorithm
startTime = datetime.now()

# Create the parent Organism (with random Genes)
# Can use size of either target image as they must be the same
generation = 1
parent = Organism(globalTarget1.size, INITIAL_GENES)

# Save an image of the initial Organism
parent.drawImage(1).save(os.path.join("results", "P1_{}.png".format(generation)))
parent.drawImage(2).save(os.path.join("results", "P2_{}.png".format(generation)))
parent.drawImage(3).save(os.path.join("results", "P3_{}.png".format(generation)))

# Calculate the score of the Organism from the different perspectives compared to the
    three target images
parent.score = (calcScore(parent.drawImage(1), globalTarget1),
                calcScore(parent.drawImage(2), globalTarget2),
                calcScore(parent.drawImage(3), globalTarget3))

# Calculate initial accuracy
accuracy = calcAccuracy(parent.score)

# Create the counters for the display
generationCounter = wtext(pos=display.title_anchor, text="\n<b>Generation:</b>
    {}\n".format(generation))
refAccuracy1 = wtext(pos=display.title_anchor, text="<b>Accuracy to reference1:</b
    > {}%\n".format(accuracy[0]))
```

```
refAccuracy2 = wtext(pos=display.title_anchor, text="<b>Accuracy to reference2:</b
    > {}%\n".format(accuracy[1]))
refAccuracy3 = wtext(pos=display.title_anchor, text="<b>Accuracy to reference3:</b
    > {}%\n".format(accuracy[2]))
geneCounter = wtext(pos=display.title_anchor, text="<b>Number of Genes:</b> {}\n
    ".format(len(parent.genes)))

# Create the curves for the Graph
accuracyCurve1 = gcurve(color=color.blue, width=4, label='Ref. 1')
accuracyCurve2 = gcurve(color=color.red, width=4, label='Ref. 2')
accuracyCurve3 = gcurve(color=color.green, width=4, label='Ref. 3')

# Render the initial Organism in 3D
renderOrganism(parent)

# Setup the multiprocessing pool
p = multiprocessing.Pool(cores)

# Infinite loop (until the process is interrupted)
while True:
    # Pause and run the GA
    if runProgram:
        # Print the current score and write generation info to the log file
        print("Generation {} - R1: {}%, R2: {}%, R3: {}%".format(generation, round(
            accuracy[0], 3), round(accuracy[1], 3), round(accuracy[2], 3)))
        logFile.write("{},{},{},{},{}\n".format(generation, accuracy[0], accuracy[1],
            accuracy[2], len(parent.genes)))

        # Save an image of the current best Organism to the results directory
        if generation % GENERATIONS_PER_IMAGE == 0:
            parent.drawImage(1).save(os.path.join("results", "P1_{}.png".format(
                generation)))
            parent.drawImage(2).save(os.path.join("results", "P2_{}.png".format(
```

```
            generation)))
        parent.drawImage(3).save(os.path.join("results", "P3_{}.png".format(
            generation)))

    # Update the accuracy
    accuracy = calcAccuracy(parent.score)

    # Update the display counters
    generationCounter.text = "\n<b>Generation:</b> {}\n".format(generation)
    refAccuracy1.text = "<b>Accuracy to reference1:</b> {}%\n".format(accuracy
        [0])
    refAccuracy2.text = "<b>Accuracy to reference2:</b> {}%\n".format(accuracy
        [1])
    refAccuracy3.text = "<b>Accuracy to reference3:</b> {}%\n".format(accuracy
        [2])
    geneCounter.text = "<b>Number of Genes:</b> {}\n".format(len(parent.genes)
        )

    # Render the current best Organism in 3D
    if generation % GENERATIONS_PER_RENDER == 0:
        renderOrganism(parent)
        # Update the graph
        accuracyCurve1.plot(generation, accuracy[0])
        accuracyCurve2.plot(generation, accuracy[1])
        accuracyCurve3.plot(generation, accuracy[2])

    # Pause program once we are on the max generation
    if generation == GENERATIONS_MAX:
        logFile.write("Total Runtime: {}\n".format(datetime.now() - startTime))
        runProgram = False

    # Decrease mutation rate if using variable mutation
    if VARIABLE_MUT and generation % VARIABLE_MUT_RATE == 0:
```

```
        global MUTATION_CHANCE
        if MUTATION_CHANCE > 0.01:
            MUTATION_CHANCE = MUTATION_CHANCE - 0.01

    # Increment the generation count
    generation += 1

    # Create the new population and add parent to the collection
    # in case all children mutations result in worse scores
    population = [parent]

    # Generate the children in the new population by applying mutations to the
        parent organism
    try:
        newChildren = generatePopulation(parent, POP_SIZE - 1, p)
    except KeyboardInterrupt:
        # Print the final gene count and total run time
        print("Final Gene Count: {}".format(len(parent.genes)))
        print("Total Runtime: {}\n".format(datetime.now() - startTime))
        p.close()
        return

    # Add the new children to the population
    population.extend(newChildren)

    # Sort the population of Organisms by their fitness.
    sortedPopulation = sorted(population, key=lambda x: fitness(x))

    # Get the Organism that minimises the fitness value the most and
    # assign it to be the parent for the next generation
    parent = sortedPopulation[0]
```

```
def fitness(o):
    """
        This functions calculates the fitness of the individual Organisms in the
            population.
        The fitness is the sum of the scores for each of the reference images added to the
        number of Genes the Organisms has multiplied by a weight. The number of Genes is
            included
        in the score to encourage the Organisms to use the fewest amount of Genes possible
            .
        The best Organism is the one that is able to minimise this value the most.
    """
    return o.score[0] + o.score[1] + o.score[2] + len(o.genes) * 2


def calcAccuracy(score):
    """
        This functions takes the score values of an Organism and calculates how accurate
        the image produced by the Organism is to the reference images as a percentage.
        The accuracy is calculated by finding the inverse of the percentage difference.
    """

    # Calculate the total number of color values in the target images by calculating the
        number of pixels and
    # multiplying by 3 for the three RGB color values. Only need size of one, images must
        be same sized
    imageDimensions = globalTarget1.size[0] * globalTarget1.size[1] * 3

    # Since the score is a calculation of the sum of the differences between the three
        images
    # we are easily able to calculate the % difference and then use the inverse of this for
         the accuracy
    return (100 - ((score[0] / 255.0 * 100) / imageDimensions)), \
           (100 - ((score[1] / 255.0 * 100) / imageDimensions)), \
```

```
            (100 - ((score[2] / 255.0 * 100) / imageDimensions))


def calcScore(im1, im2):
    """
        This function is used to determine how close the image produced by the Organism
            is to the reference image.
        It uses ImageChops and numpy to quickly compute the sum of the differences
        in the pixels between the two images.
    """
    difImg = ImageChops.difference(im1, im2)
    diff = numpy.sum(difImg)
    return diff


def createChild(o):
    """
        Takes a parent Organism and mutates it's Genes to create a child Organism. The
            child Organism is then
        tested on all of the reference images and given a score.
    """
    try:
        child = deepcopy(o)
        child.mutate()
        img1 = child.drawImage(1)
        score1 = calcScore(img1, globalTarget1)
        img2 = child.drawImage(2)
        score2 = calcScore(img2, globalTarget2)
        img3 = child.drawImage(3)
        score3 = calcScore(img3, globalTarget3)
        child.score = (score1, score2, score3)
        return child
    except KeyboardInterrupt as e:
```

```
        pass


def generatePopulation(o, number, p):
    """
        Uses the multiprocessing module to create a new population of Organisms in parallel
            .
    """
    population = p.map(createChild, [o] * int(number))
    return population


#
    ------------------------------------------------------------------------


# Vpython helper methods
#
    ------------------------------------------------------------------------


# Toggle the execution of the main program
def runButton(b):
    global runProgram
    if b.text == 'Pause':
        runProgram = False
        b.text = 'Run'
    else:
        runProgram = True
        b.text = 'Pause'


# Deletes all the objects in the scene
def clearScene():
```

```
    for obj in display.objects:
        obj.visible = False
        del obj


# Takes an Organism and renders it in Vpython
def renderOrganism(organism):
    # Clear the scene from any previously rendered Organisms
    clearScene()

    # Find the midpoint of the 3D image so that we can subtract it from each
    # point position so that the rendered image is centered around (0,0,0).
    xTranslate = organism.size[0] // 2
    yTranslate = organism.size[1] // 2
    # Reusing xTranslate for zTranslate
    zTranslate = xTranslate

    for g in organism.genes:
        # Reformat Gene attributes to be compatible with Vpython
        p = vec((g.pos.x - xTranslate), (0 - (g.pos.y - yTranslate)), (0 - (g.pos.z -
            zTranslate)))
        # Doubling the radius because the size of the balls was too small before
        r = g.diameter
        c = vec((g.color.r / 255), (g.color.g / 255), (g.color.b / 255))

        # Create sphere object from Gene
        sphere(pos=p, radius=r, color=c)


# Takes a screen shot of the 3D scene and saves it as a .png in your downloads folder
def takeScreenShot():
    display.capture("PyArt3")
```

```
#
    ------------------------------------------------------------------------

# Main Function
#
    ------------------------------------------------------------------------

if __name__ == "__main__":
    # Set number of available cores
    cores = max(1, multiprocessing.cpu_count() - 1)

    # Create the 3D scene
    display = canvas()
    display.width = display.height = 600
    display.background = color.white
    display.align = 'left'
    display.range = 1

    # Setting the FOV to be small to produce a Orthographic projection
    display.fov = 0.01

    # lighting
    display.lights = []  # gets rid of default lighting (shadows and distant white light)
    display.ambient = color.gray(1)  # adds bright ambient light

    # Add title
    display.title = '<b>PyArt:</b> Using a Genetic Algorithm to create digital art from
        multiple reference \n' \
                    'images in the style of Pointillism rendered in 3D.'

    # buttons
    button(text='Pause', bind=runButton)
```

```
button(text='Screen Shot', bind=takeScreenShot)

# Add a Graph to visualize the improvements
graph( title='Rate of Improvements', xtitle='Generations',  ytitle ='Accuracy', align='
    right',  fast=False)

# Add instructions
display.append_to_caption("""
<b>Instructions:</b>
To rotate "camera", drag with right  button or Ctrl-drag.
To zoom, drag with middle button or Alt/Option depressed, or use scroll wheel.
  On a two-button mouse, middle is left + right.
To pan left/right  and up/down, Shift-drag.
Touch screen: pinch/extend to zoom, swipe or two-finger rotate.""")

# run the genetic algorithm
run(cores)
```