Evolved Art with Transparent, Overlapping and Geometric Shapes

Joachim Berg, Nils Gustav Andreas Berggren, Sivert Allergodt Borgeteien,
Christian Ruben Alexander Jahren, and Arqam Sajid

OsloMet – Oslo Metropolitan University

Dated: November 22, 2018

Contents

List of Figures

List of Tables

1 Preface

This project is an introduction to artificial intelligence and its abilities to recreate images using evolutionary algorithms and transparent, overlapping circles, polygons, and lines. When we first began this project in the month of September in 2018, none of us in the group had any real experience with using AI to iteratively optimise a given output (or a given task). Stefano Nichele, our supervisor, presented us a report titled *Evolving Images Using transparent Overlapping Polygons* (Dong, Li, Tarimo, Xing, 2013), and we were inspired to develop, optimise and extend a program such as the one in that paper, with some tweaks and improvements. We were also inspired to research the technology of AI to understand its artistic capabilities as well as understand how similar techniques could be used in other domains.

Our motivation behind this project is to get a deeper understanding of the field of AI and evolutionary algorithms. The former is relevant to our studies and is an increasingly dominating industry in the world of programming. We believe having a proper understanding of the technical aspects of AI and its practical uses in academics and in the workplace, is as important as ever in order to be relevant and to follow the development of technology.

We would like to thank Stefano Nichele for supervising us throughout the project and his professional insight on AI and evolutionary algorithms.

2 Aims

Our aims in this project are to explore different techniques in evolutionary algorithm, and figure out what the benefits and drawbacks of these techniques are for generating and recreating art. We intend to develop a fully functional Python based program which will render images based on a target image and a set of given parameters. We expect to have some tangible data that can be used to determine what the most optimal parameters are. We also expect to develop some examples of how our program approximates images, that can be shown in this report.

3 Introduction

3.1 Background theory

In nature, all species of all organisms develop through natural selection by passing its traits to the next generation, but with small variations, to increase the next generation's ability to survive, compete, and reproduce.[1] This is called evolution by natural selection and can be seen as an algorithm to search for an increasingly better or fitter solution. The algorithm adapts to the environment by making small changes, or mutations, to its previous solution, and by repeating this process the algorithm will iteratively find an equal or a better solution each time. The DNA sequences are passed from a parent to a child, and this compressed representation of DNA sequences is called a genotype. A genotype is a complete heritable genetic identity used to pass genetic information from one generation to the next generation. It can be seen as the recipe for the phenotype which is the actual visual representation of the organism. By applying this to evolution, one can say that the organism passes on its genotype with small variations to the next generation. While it is the genotype that is passed through the generations, it is the phenotype that is evaluated and subjected to fitness.

The phenotype's ability to compete, survive, and reproduce is determined by its compatibility with the surrounding environment. If the child of this phenotype has traits that are less compatible with the environment than its parent, the child's ability to compete and survive may not be enough for it to reproduce. The initial phenotype has to produce children with an equal or better genotype to make sure the newer generations can continue to reproduce.

3.2 Theoretical approach

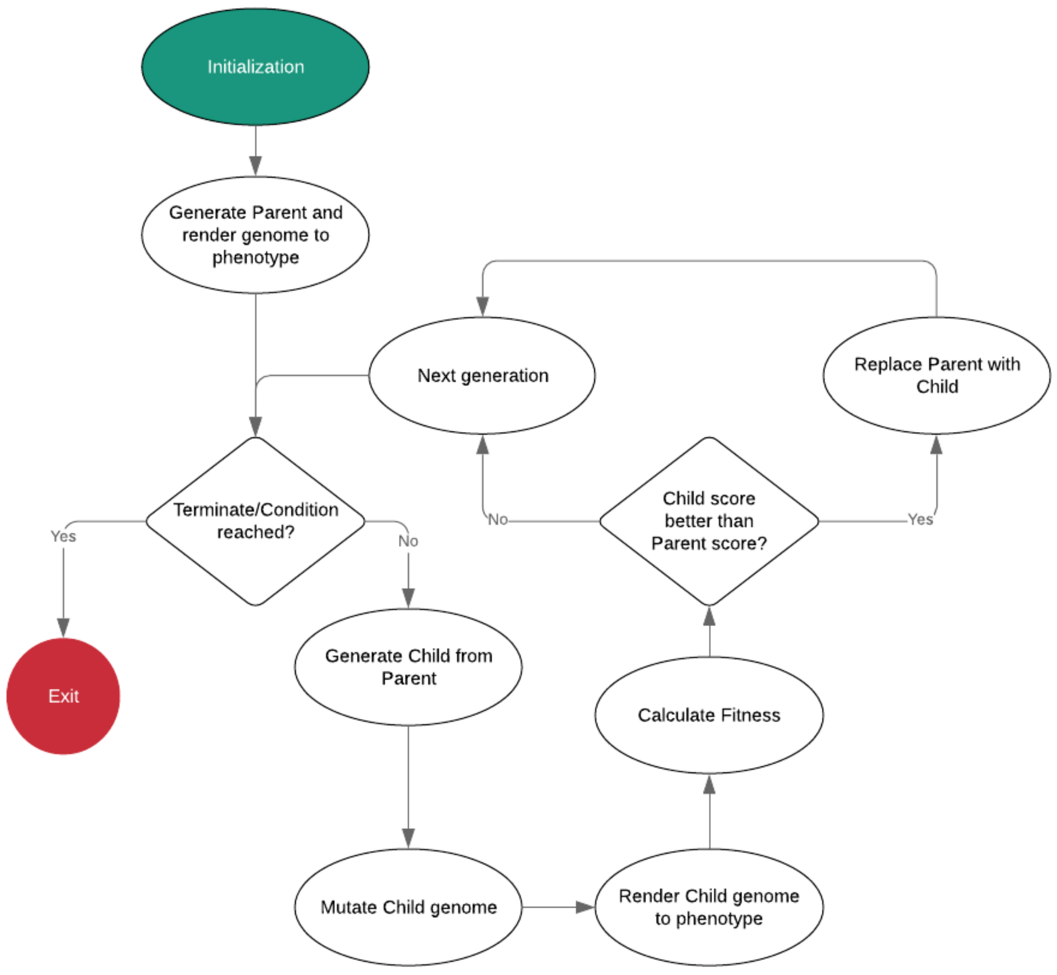
Using this concept of evolutionary algorithm, we can develop a program which uses the same logic as in natural selection, to recreate a target image using arbitrary geometrical shapes such as transparent overlapping circles, polygons, and lines. By this we mean circles, polygons, and lines of varying size, colour, transparency, and placement. In each iteration, or generation, of this process, a collection of shapes are created, and their structures are represented as instance objects[2] with changeable parameters. These objects are the genes in the genotype. We will refer to this collection of genes as an image genome.

To distinguish a bad solution from a better solution, a fitness function is needed. If the new solution has a better fitness score than the previous one, the new solution will replace the old one and be used for the next generation. Using this technique the program will find an increasingly better solution, and when the fitness score is good enough or a termination criteria is met, the program will stop.

We will describe every step of our algorithm below (see Figure 3.1 on page 7 for a visual diagram): 1. The program receives a target image and parameters set by the user as an input, 2. It generates the initial genome(s) based on the parameters and renders the phenotype. This is the parent(s) for the first generation, 3. If the termination criteria is met (e.g number of generations), the program is terminated,

[1]Dorin, A. (2014), *Biological Bits: A brief guide to the ideas and artefacts of computational artificial life*

[2]In object-oriented programming, an object is an instance of a class. All genes are instantiated from their respective classes, but the information stored within the objects vary within their limits.



Initialization
Generate Parent and render genome to phenotype
Next generation
Replace Parent with Child
Terminate/Condition reached?
Yes
No
Child score better than Parent score?
No
Yes
Exit
Generate Child from Parent
Calculate Fitness
Mutate Child genome
Render Child genome to phenotype

Figure 3.1: The step-by-step flow of the algorithm

the program stops. Otherwise the loop begins. 4. A child genome is created based on the parent genome and is mutated based on parameters set in the program, 5. It renders the child genome into a phenotype, 6. The fitness score of the child genomes phenotype are calculated, 7. If the child phenotype's fitness score is better than the parent phenotype's fitness score, the parent is replaced by the child. Otherwise, the parent is kept while the child is discarded.

4 Methodology

4.1 Image genome structure

All image genomes are implemented in the program as objects consisting of genes for each shape. Each gene is made up by parameters that define the size of the canvas, the shape's colour, the transparency/alpha of that shape, and its coordinates on the canvas, and depending on the shape, number of vertices, the radius length and the thickness. A shape's initial gene structure is made up by these parameters: The width and height of the target image, an array for the colours with values from 0-255, a transparency/alpha value between 0-1, and the coordinates as x (value from 0 to width of image) and y (value from 0 to height of image). Additional parameters such as the number of vertices, the length of the radius, and the thickness of the line are added to the end of the gene when it is generated. See Table 4.1 on page 8 for examples for each of the shapes. Their phenotypes are represented visually in Figure 4.1 on page 8.

Table 4.1: Sample gene values

Type	Width x Height	Colour	Alpha	Coordinates	Vertices/Radius/Thickness
Polygon	200 x 200	(65, 6, 197)	0.64	[[22, 36], [110, 172], [72, 0]]	3
Circle	200 x 200	(243, 159, 253)	0.77	(59, 182)	97
Line	200 x 200	(35, 89, 71)	0.12	(51, 130), (162, 60)	6

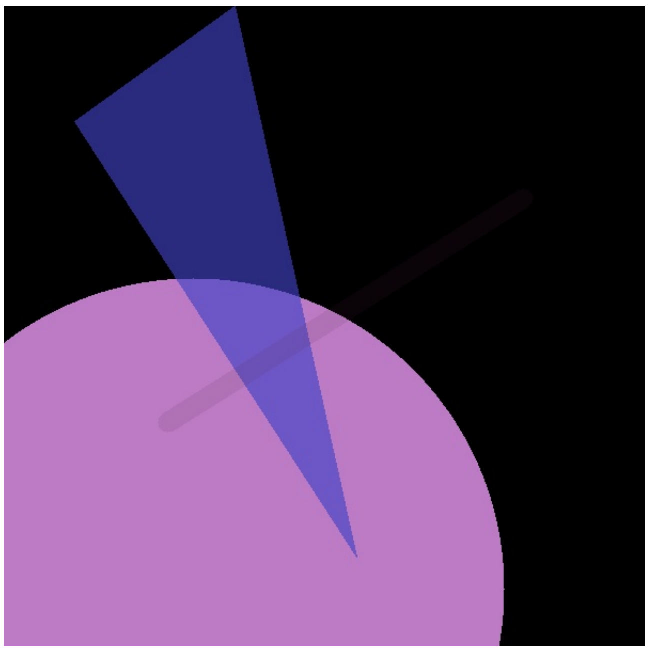


Figure 4.1: Visual representation of the genes in table 4.1

4.2 Mutation operations

Any gene in the genome can be mutated based on a set of given parameters in the program. A modification of a specific or random parameter of the gene is done by retrieving the values of the relevant parameters and modifying them within the limitations given. In our implementation we have included three mutation operations, soft mutation, medium mutation, and a hybrid mutation. The soft mutation updates parameters within a limit, whilst the medium operation replaces existing parameters with new values. The hybrid mutation combines the two former mutations by first doing two soft mutation and then one medium mutation (ratio 2:1). We have also added two mutation factors, probability mutation and chunk mutation. The probability mutation is based on the parameter mutation_probability, set in the program with a value from 0 to 1. This gives all genes in a genome a probability of mutating decided by the parameter's value. When using the chunk mutation, a number of genes in the genome are always mutated. The number is based on the same parameter, mutation_probability, but is multiplied with the number of genes in the genome. Example with the probability mutation: If the parameter is set to 0.5, all genes in the genome will have a 50% probability of mutation. Example with the chunk mutation: If a genome has 100 genes and the variable is set to 0.5, then 50 mutations will take place. This means that there is a probability for the same gene being mutated twice (or more).[3]

The following is a description of the mutation selection process: 1. A mutation operation (soft or medium) is selected, 2. A mutation factor (probability or chunk) is selected. See Figure 4.2 for a visual representation of this process.

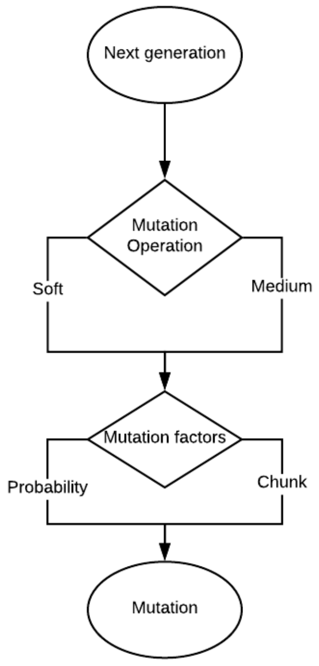


Figure 4.2: Diagram of the mutation operations and the mutation factors

[3]The minimum number of genes to be mutated is 1.

4.3 From genotype to phenotype

As seen in Figure 3.1 on page 7, new solutions are rendered when their mutation is completed in the previous step. To render the genotype's phenotype, a black canvas is created, and then each gene in the genotype is rendered onto the canvas one by one. In our implementation we have used a Python library called OpenCV, as it is capable of rendering shapes with correct alpha values. It also provides simple utility for working with red, green, and blue channels of a 24-bit image. As every gene represents one shape, the circles, polygons, and lines are drawn, filled, and rendered in their respective order, in compliance with the genome structure. The resulting phenotype is evaluated by the fitness function.

4.4 Fitness function

The point of the fitness function is to measure how close the generated image is to the target image, and to distinguish bad solutions from good ones. Our implementation of the fitness function is done by summarising the pixel by pixel difference of the images and that way determine a score. This score is used to determine whether to replace the parent image genome with the child image genome, or discard it. If the score of the child is lower than the parent image genome, the parent is replaced.

To improve the readability of the fitness score, we convert the absolute score to a relative fitness in percent. The percentage score is calculated by dividing the actual fitness score by the theoretical worst fitness score: The maximum difference (255) in each channel of the image (r, g and b) multiplied by the dimensions of the image in pixels (255 x 3 x width x height). Furthermore, the ratio is converted to a percent and then flipped (100.0% minus the calculated percentage) to reflect approximation towards 100% instead of 0%. The difference decreases as the approximation improves. This way the fitness score is comparable between images of different dimensions and easier to put into context.

The fitness score does not necessarily determine the best rendered image for human eyes. (e.g image with worse score can be more recognisable than another image with better score.) Certain defined features of an image can be more important for recognition by humans.

5 The program

The program we developed was initially based on the MATLAB code from the paper *Evolving Images Using transparent Overlapping Polygons* (Dong, et al. 2013). As the project progressed and other parameters were added, we developed a Python (version 3.7) based program. Only one member of the group had earlier experience with programming in Python, so the rest of the group had to spend some hours in the first two weeks of this project learning the language. We experienced the transition of going from MATLAB code to Python as fairly easy, and it gave us the opportunity to really implement anything we wanted to in our program. The group was already very familiar with Java and its object-oriented programming. This helped us understand the Python syntax.

5.1 Programming environment

The programming was done in PyCharm, an IDE developed by Jet Brains. We used GitHub as the repository. To keep track of the progress, we found Trello, a service providing us overview of the project, the implementation done so far, and the to do list we had.

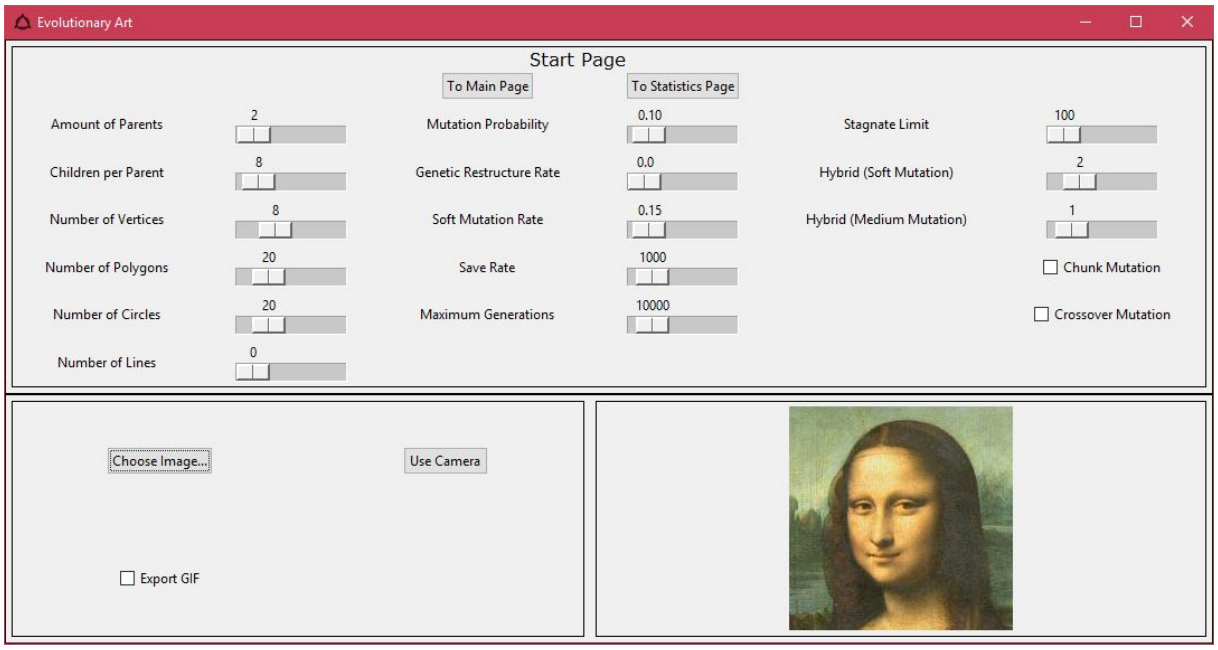
The program and the data generated from the experiments are available on GitHub:
https://github.com/joacber/Evolved-art-with-transparent-overlapping-and-geometric-shapes

5.2 Graphical User Interface (GUI)

A GUI was implemented in order to make the program more accessible and interactive to us and potential users (see Figure 5.1 on page 12). We chose TkInter for this task, as it is the de-facto standard Graphical User Interface package in Python. A big portion of our project hours was spent working on the program itself, and the GUI was a central part of said program. From the beginning of the project, we wanted to be able to tie together every feature that we would implement.

TkInter provides tools for implementing graphical program windows through the use of frames, which can be switched between with some clever tinkering. The package also comes with a decent amount of widgets for reassigning variables and calling functions through the GUI. This allows other to reproduce the configuration of the parameters that affect the algorithm, making the program much more user friendly than it would have been through terminal commands.

MatPlotLib is a plotting library, which produces 2D graphics in different environments. It is mostly used for plotting graphs and enabling exploration of data. It can also be integrated in the GUI through TkInter. We found that by performing this integration, we could switch seamlessly between running our evolutionary algorithm and plotting the fitness data. This was mostly used in the earlier stages of development, to understand the impact of our design choices in the code of the algorithm. When the time came to analyse results of our tests, the plotting functionality in our GUI was not capable enough, and we didn't have the resources to implement it within a reasonable time-frame. Therefore, while the plotting is carried out using MatPlotLib, it is not done through integration within our software. The GUI does however, in its current state, give the user a straightforward way to explore the program and reproduce the experiments presented in this report.



Amount of Parents	2
Children per Parent	8
Number of Vertices	8
Number of Polygons	20
Number of Circles	20
Number of Lines	0
Mutation Probability	0.10
Genetic Restructure Rate	0.0
Soft Mutation Rate	0.15
Save Rate	1000
Maximum Generations	10000
Stagnate Limit	100
Hybrid (Soft Mutation)	2
Hybrid (Medium Mutation)	1

☐ Chunk Mutation

☐ Crossover Mutation

☐ Export GIF

Figure 5.1: Screenshot of the Start Page in the GUI

5.3 Graphic Interchange Format (GIF)

An option to export the evolving images as a GIF file was implemented later in the development. This implementation saves the parent images to an array. When the algorithm is terminated, the array of images will be encoded into a GIF. By only using the parent images, the fitness score will always be increasingly better. In reality, this will have a more noticeable effect when running many generations, because it will display fast progression the beginning, and slower at the end.

5.4 Webcam

We also chose to implement webcam functionality, which uses the computer's webcam to capture a target image. At the start page of the program, the user has the option to either import an existing image file or capture a new image. The main reason we wanted to implement this functionality was for the purpose of exhibiting our work at a teacher conference at our university. The idea was to set up a demonstration booth, and let passers-by and onlookers have a go at it. Having a webcam handy would allow them to use their own face as the target image, and they would have received the computed image by email, or a physical copy of it.

6 Experiments and testing

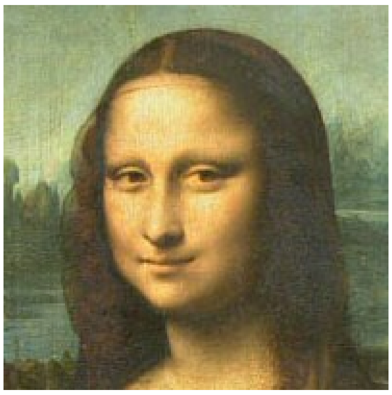


Figure 6.1: Target image used in experiments

Table 6.1: Initial Test Parameter Values

Parameter Name	Parameter Value
Number of Parents	1
Children per Parent	1
Genes Total	20
Polygons	20
Circles	0
Lines	0
Vertices	3
Mutation Probability	0.1
Genetic Restructure Rate	0
Soft Mutation Rate	0.1
Hybrid (Soft Mutation)	0
Hybrid (Medium Mutation)	0
Chunk Mutation	False
Crossover Mutation	False
Save Rate	1,000
Maximum Generations	10,000

Because the number of variables were so numerous and the time frame was short between the completion of the development and the deadline for delivering the project, the testing was conducted with one variable at a time. By doing it like this, it was easier to find an optimal value for the variable for the remaining tests. Each test was conducted either 5, 15, or 20 times, depending on the progression, to ensure the results are reliable and accurate. To make the tests comparable the same target image was used as in the paper by *Evolving Images Using transparent Overlapping Polygons* (Dong, et al. 2013). This image was also chosen because it is easily recognisable. See Figure 6.1 on page 13.

To accurately compare the results, the best result from each generation was logged to a file for each run, as well as the input parameters saved to another file. That way it is possible to track the progress for each generation, and determine how fast or slow the improvements are for the specific parameters that were tested. The initial test parameters can be seen in Figure 6.1 on page 13. These were modified during the experimenting and testing to reflect the optimisation results we got.

The following paragraphs are descriptions of Table 6.1 on page 13: The *Number of Parents* parameter reflects how many parents the algorithm has to work with. The best parent has its phenotype shown in the GUI, but every parent's phenotype is saved in the output folder. The *Children per Parent* parameter is the number of children each parent will produce. The minimum value of both is 1, so the program has a population to evolve and compare, and the maximum is set to 100 to limit the processing and the total time it takes. As previously explained, the genome of a genotype consists of a number of genes. The next parameter you see here is *Genes Total*, which is made up of the values of the three parameters below it. The initial number in these tests is 20 genes per genotype. As the experimenting progresses, the number of *Polygons*, *Circles*, and *Lines* will change. The number of *Vertices* for the polygons is an editable parameter, but the radius of the circle and the thickness of the lines are set to random and are not editable by the user (and therefore not present in the table).

The *Mutation Probability* parameter defines the rate of probability and chunk mutation factors. The *Genetic Restructure Rate* parameter is based on the previous parameter, *Mutation Probability.* If the current number of generations during the process is below one tenth of the maximum number of generations, each gene in the genotype will have 0.1 probability to mutate. The *Soft Mutation Rate* parameter decides the level of change that any gene's parameter can be subjected to. *Hybrid (Medium Mutation)* determines for how many generations in a row the algorithm should run with medium mutation. If it is 0 it only runs soft. *Hybrid (Soft Mutation)* determines how many generations in a row the algorithm should run with soft mutation. If it is 0 and *Hybrid (Medium Mutation)* is not 0 it only runs medium mutation.

The *Save Rate* parameter defines how often the image is saved (e.g. every 1,000th generation). The *Maximum Generations* is set to 10,000. That means reaching generation number 10,000 is a termination condition, and thus the algorithm will terminate. *Chunk Mutation* decides whether to use chunk mutation as a factor in the mutation operations, instead of probability, which is the standard. The parameter is true or false. The functions of these two mutation factors are explained in section 4.2 Mutation Operations. *Crossover Mutation* determines if 2 parents should cross-mutate when producing a child. The parameter is true or false. The child is made up of the main parent's coordinates and shape relevant parameters, and the second parent's colour and alpha values.

6.1 Number of vertices

The initial test was to determine the most optimal number of vertices to use in further testing of polygons. All tests were done 15 times with the same values. Using a genome consisting of 20 polygons, we were able to determine the number of vertices with the most positive impact on the fitness score over 10,000 generations. The best average score was achieved by genomes consisting of polygons with 8 vertices, followed by 10 and 15 (see Table 6.2 on page 15). The top graph in Figure 6.2 on page 15 displays the average results of 15 tests with the number of vertices increasing from 3 to 20. The differences are apparently small. The bottom graph in the same figure displays the standard deviation (STD). If the fitness score has a high variation during the process, the STD value will also be high. Here we can see that the STD value is relatively dynamic in the first 5,000 generations, but flattens out later. This makes sense because in the early stages the image is being created from scratch, and later on the image is mostly being fine-tuned. Figure 6.3 on page 15 displays all the results from the 15 runs with 8 vertices.

Table 6.2: Top results from the vertices tests

Rank	Number of Vertices	Average Score	Average Relative Score
1	8	2,885,838	90.57%
2	10	3,014,961	90.15%
3	15	3,032,196	90.09%

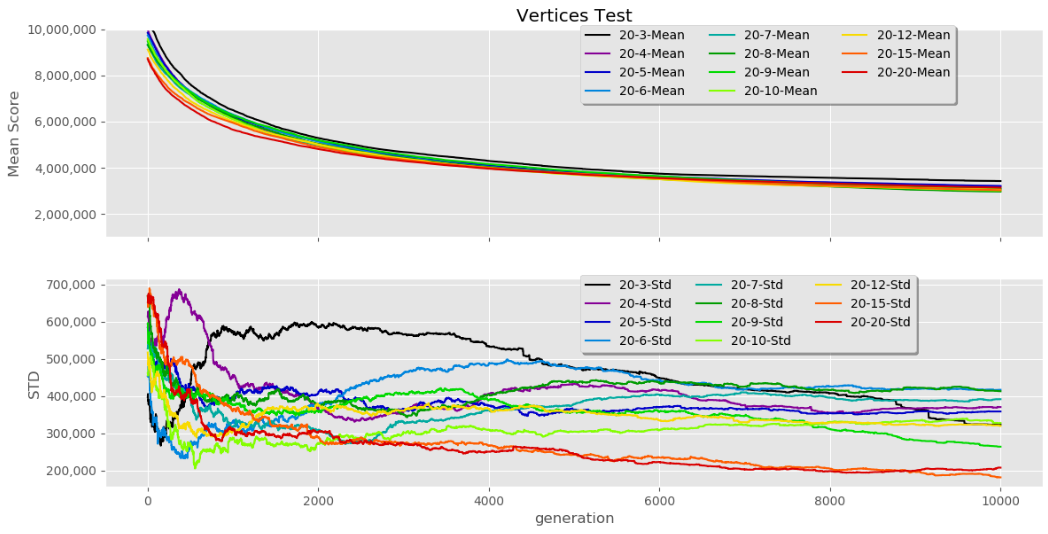


Figure 6.2: Results from the vertices test with 20 polygons and a number of vertices

Top graph: The average result from all the vertices tests

Bottom graph: The standard deviation from the same tests

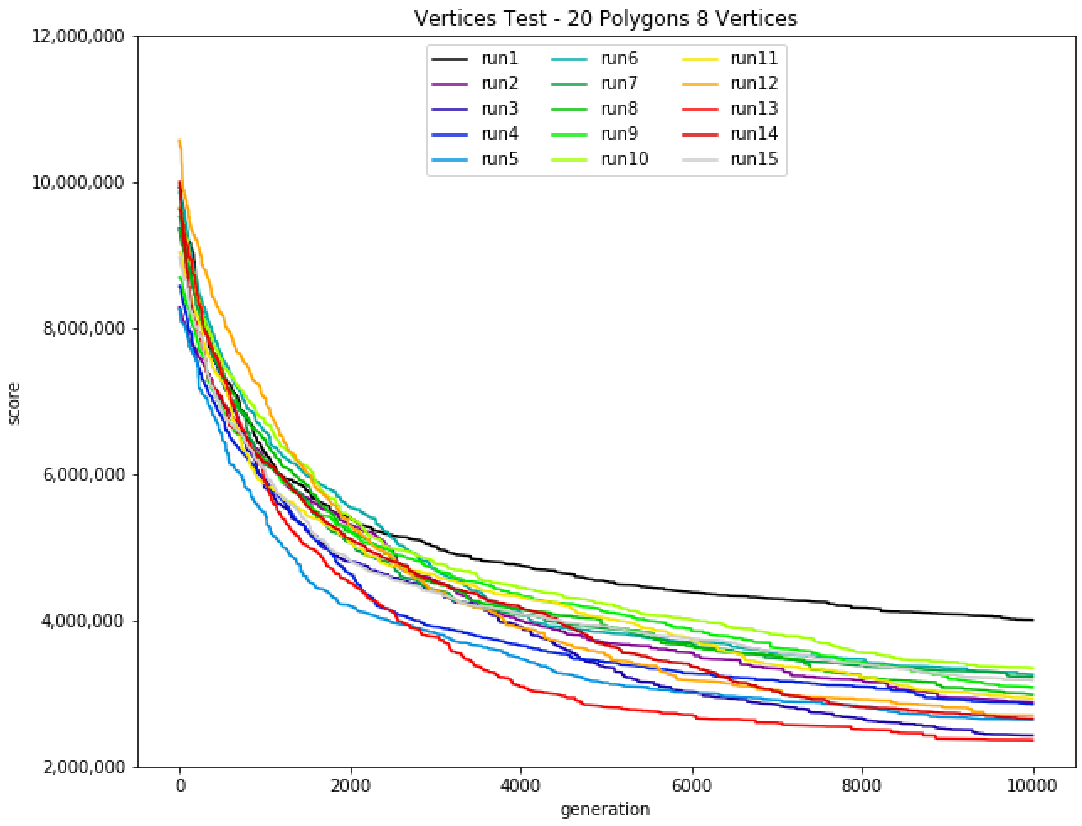
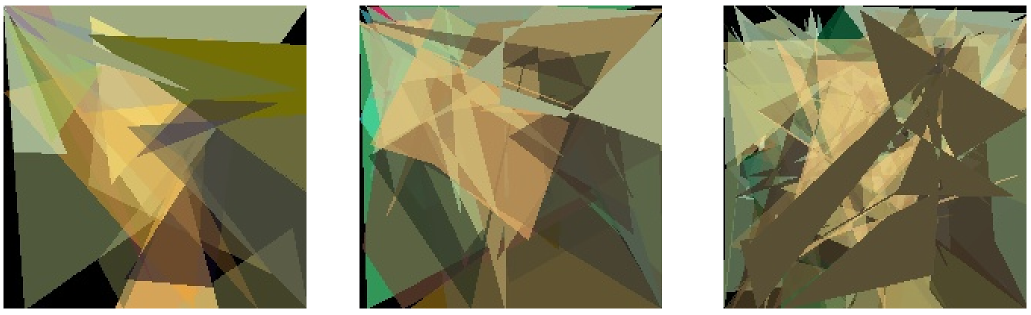


Figure 6.3: Results from 15 runs with 20 polygons with 8 vertices

Legend shows number of polygons - number of vertices



(a) 3 vertices (b) 8 vertices (c) 20 vertices

Figure 6.4: Samples from the vertices tests (10,000 generations)

6.2 Polygons

Using genomes consisting only of polygons with 8 vertices, we found that the fitness score increases almost linearly with an increase in the genome complexity (more polygons). The genome size was increased by 5 after 15 consecutive runs with the same parameters. We saw that the bigger leaps in fitness happened between 5 and 15 polygons (see top graph in Figure 6.5 on page 17) - hitting 3,072,442 (89.96%) average score at 15 contrary to 4,391,165 (85.65%) at 5. 20 polygons is the milestone hitting 2,996,858 (90.21%) average score, and 25 polygons subsequently hitting 2,751,771 (91.01%) average score with an increase of 0.80 percentage points. Increases were minimal with more complex genomes and disproportionate increases in process time up to 40 polygons, with fitness score even dropping between 40 (Avg. Score: 2,494,330) and 50 (Avg. Score: 2,507,859) polygons, thus making 25 polygons the apparent winner for 10,000 generations with an approximation of above 90% and a relatively big increase (0.80 percent points) from 20 polygons. In the bottom graph of Figure 6.5 on page 17, we can see that the STD value flattens out after 4,000 generations.

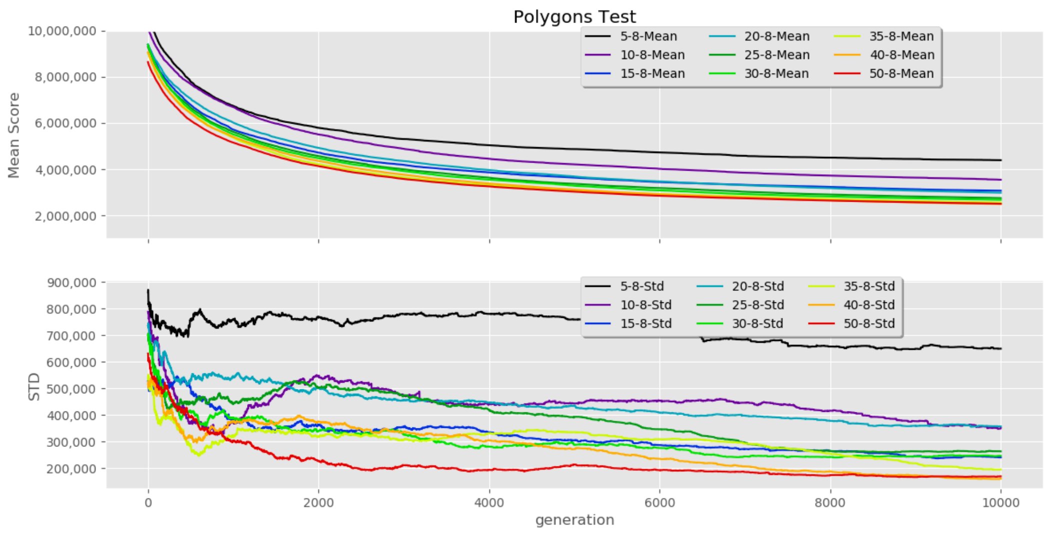
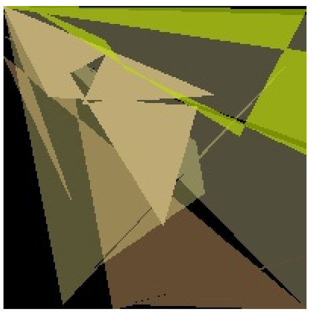
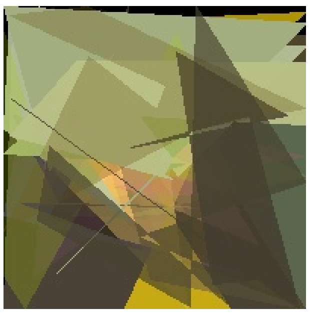


Figure 6.5: Results from the polygons tests with 8 vertices and an increasing number of polygons Legend shows number of polygons - number of vertices


(a) 5 polygons


(b) 25 polygons

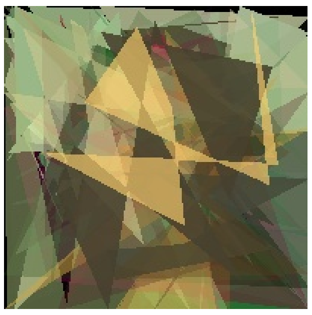

(c) 50 polygons

Figure 6.6: Samples from the polygon tests (10,000 generations)

6.3 Circles

Using genomes consisting only of circles, testing showed that, similarly to polygons, the fitness score increases steadily when adding more circles to the genome (see Figure 6.7 on page 18). An all circles genome gets an average approximation of over 90% at 15 circles. (Whereas polygons were right below that at 15.) The increases in fitness score when increasing genome complexity are more volatile with circles than polygons. We only tested up to 40 circles per genome and do not know if average fitness score decreases between 40 and 50 circles as with polygons. Using 40 circles was however only slightly better than using 35 (92.20% over 92.15% (0.05 percent point increase)), whereas using 30 circles was a great deal better than using 25 (91.75% over 91.17% (0.58 percent point increase)). There are no clear winners, but genomes consisting of 15, 20 and 30 circles stand out among the rest.

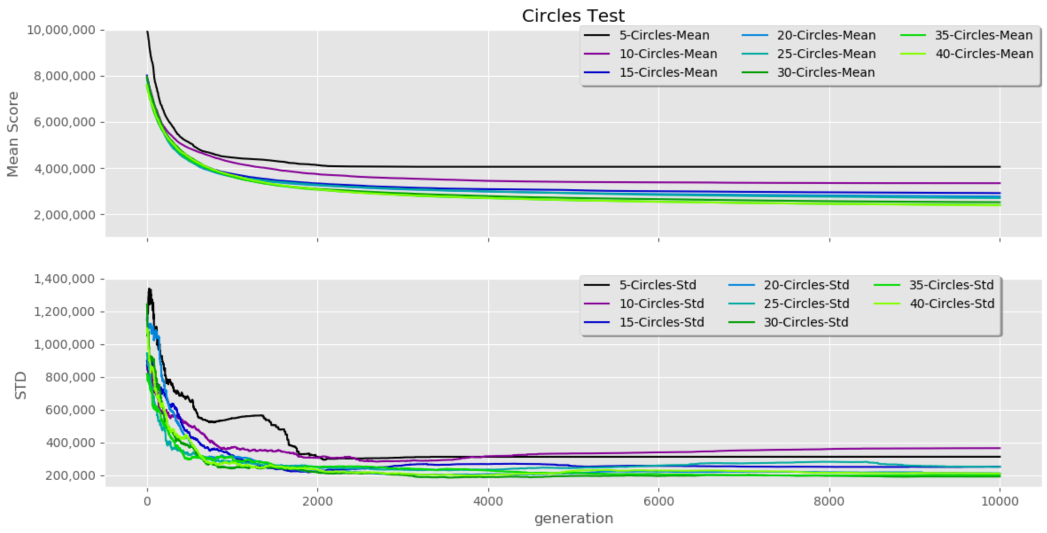
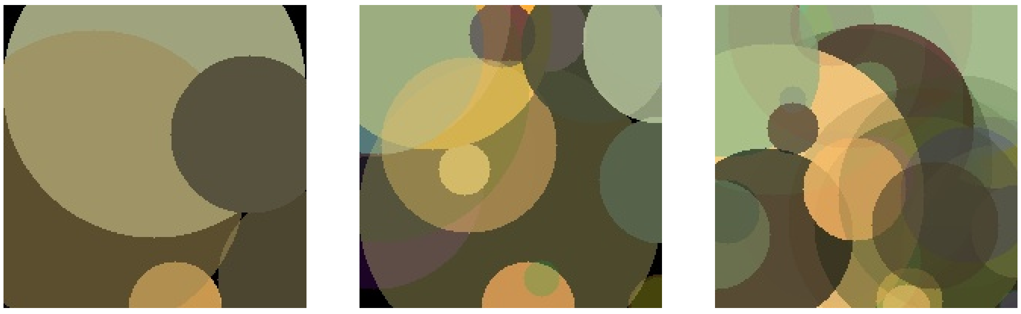


Figure 6.7: Result from the circles tests from 5 to 40 circles

Top graph: Collection of the circles tests.

Bottom graph: The STD flattens out already after 2,000 generations

Legend shows number of circles



(a) 5 circles (b) 20 circles (c) 40 circles

Figure 6.8: Samples from the circles tests (10,000 generations)

6.4 Lines

Testing with genomes consisting only of lines was shown to be an ineffective option. Starting at an average approximation of 65.50% using 5 lines in the genome, ending at an average approximation of

83.86% using 40 lines per genome. While using lines is worse at getting desired fitness results, it may still provide a certain aesthetic and artistic value for human eyes. And may prove useful in approximating certain pictures containing elements with straight lines.

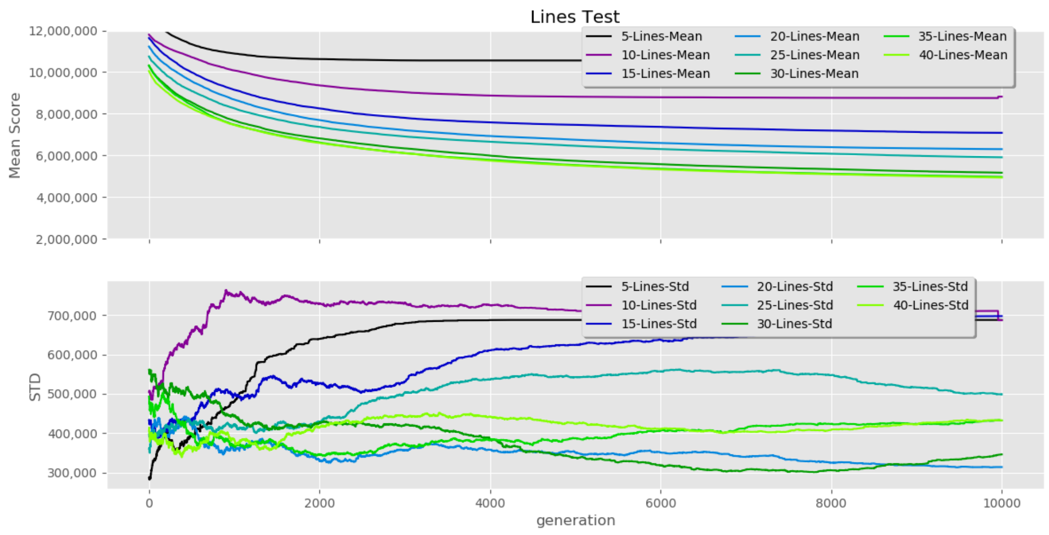
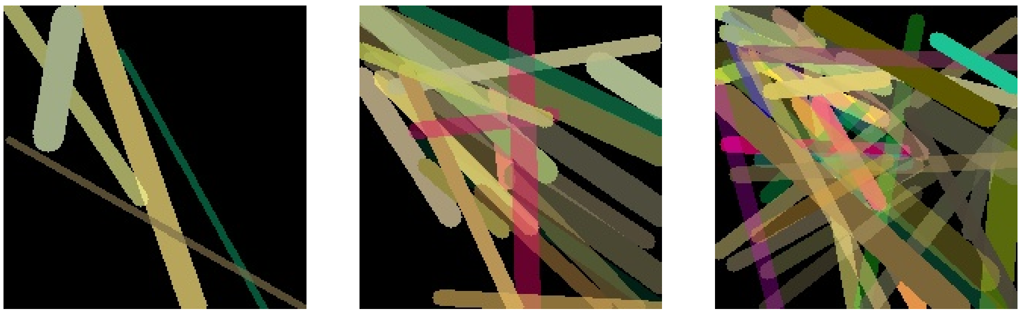


Figure 6.9: Result from the line tests

Legend shows number of lines



(a) 5 lines (b) 20 lines (c) 40 lines

Figure 6.10: Samples from the lines tests (10,000 generations)

6.5 Combinations of polygons, circles, and lines

To determine if a combination of different genes had any advantage over single gene types, we tested 6 different compositions with 20 genes in total. The tests showed that the combination of circles and polygons had the best results. The best composition of genes was a 1:1 ratio of polygons and circles, and the second best a 3:1 ratio of polygons to circles. Based on the experiments, lines make very little impact on the fitness results over the course of 10,000 generations, but as mentioned in section 6.4, they may provide functionality for certain images and artistic purposes.

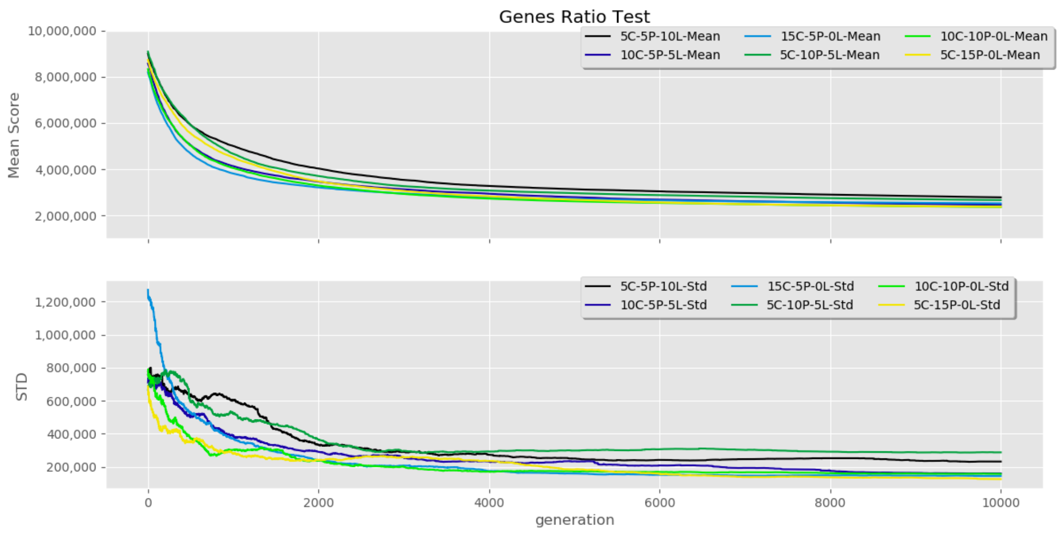
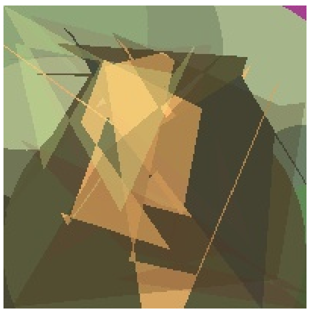
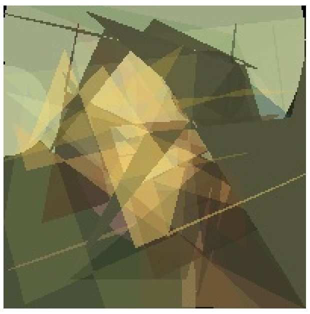


Figure 6.11: Average results from the tests with combinations of polygons, circles, and lines Legend shows number of circles - number of polygons - number of lines


(a) 10 Circles 10 Poly


(b) 5 Circles 15 Polygons

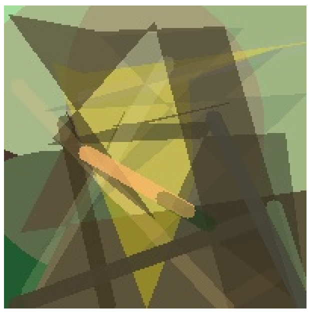

(c) 5 Circles 5 Polygons 10 Lines

Figure 6.12: Samples from the gene combination tests (10,000 generations)

6.6 Mutation Probability

To find the best mutation probability we chose to only do tests on the smallest population size of 1, because of the limited time-frame we had. Since the project we were inspired by mostly used low probability, we also focused mostly on the lowest values (see Figure 6.13 on page 21), but also did some testing on high probability (e.g. 0.7 and 1.0). The result showed that a mutation probability higher than 50% overall gave poor results over 10,000 generations.

This prompted us to limit the future testing to 10% and 30%. A mutation probability of 10% entails that every single gene has a 10:1 chance of mutating. One genome consisting of 40 genes will *on average* have 4 of its genes mutated every generation. If the probability is 30%, then the average will be 12 genes per generation.

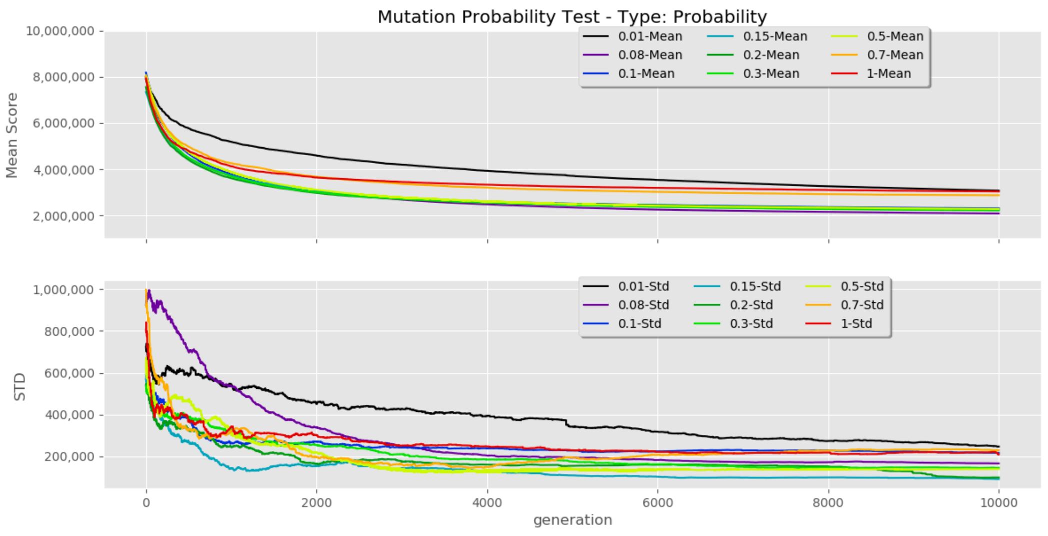


Figure 6.13: Average scores with standard deviations from mutation probability tests

6.7 Soft Mutation Rate

To determine the optimal values for soft mutation rate the tests were conducted with 25 polygons and 8 vertices. The results showed that the program performed better with the lower values than with higher values. The optimal values was shown to be between 0.1 and 0.2 (see Table 6.3 on page on page 21), where 0.15 had the best overall result.

Table 6.3: Optimal Soft Mutation Rate Values

Rank	Soft Mutation Rate	Average Score	Average Relative Score
1	0.15	2,706,987	91.15%
2	0.1	2,761,374	90.98%
3	0.2	2,810,075	90.82%

6.8 Hybrid Mutation Rate

The optimal value for the *Hybrid Mutation* parameter appear to be using a ratio of 2 soft mutations to 1 medium mutation (see Table 6.4 on page 22). The most significant jump in performance seems to be when a combination of soft and medium mutation is used. Only using medium mutation had the worst results by a significant margin (avg. score: 2,923,076), and only using soft mutation also performed worse than most other combinations (avg. score: 2,687,659).

Table 6.4: Optimal Hybrid Mutation Rate Values

Rank	Nr of Soft	Nr of Medium	Average Score	Average Relative Score
1	2	1	2,295,092	92.50%
2	4	1	2,355,187	92.30%
3	1	2	2,363,117	92.28%

6.9 Compound Mutation (Combination)

In these tests, we wanted to put together some of best parameter settings from previous tests. Using genomes consisting of 40 genes at a 1:1 ratio, 20 polygons with 8 vertices and 20 circles (omitting the ineffective lines), we performed tests with a variation of the best mutation settings. In order to get the best possible result we chose the two best values from the mutation probability tests, soft mutation tests and hybrid mutation ratio test and used them as the parameters.

The lower mutation probability of 0.1 performed better in general than the higher of 0.3, and the tests with a hybrid mutation ratio of 2:1 performed better than those with a 4:1 ratio (see Table 6.5 on page 22). The differences between the tests with the mutation rate set to 0.1 and 0.15 bore very similar results, but those with 0.1 were slightly better. A compound mutation setting of 0.1 probability, 0.1 soft mutation rate and a 2:1 hybrid mutation ratio was the best with a resulting average approximation of 93.34% (Score: 2037951) over 10,000 generations.

Table 6.5: Optimal Compound Mutation Values

Rank	Mut.Prob.	Soft Mut. Rate	Hyb. Mut. Ratio (Soft/Medium)	Avg. Score	Avg. Rel. Score
1	0.1	0.1	2/1	2,037,951	93.34%
2	0.1	0.15	2/1	2,056,403	93.28%
3	0.1	0.1	4/1	2,062,254	93.26%

6.10 Parent/Child distribution

In order for each set of experiments to be comparable, the population multiplied by number of generations had to be similar for each set of tests. So when the number of generations to decreased, the population size had to increase proportionately. If the population size was 5 the number of generations had to be 10,000, and if the population size was 10 the number of generations had to decrease to 5,000. The results from these tests showed that increasing the population in general resulted in a poorer fitness score per computed solution. Tests where the number of children per parent were greater than the number of parents yielded better results than tests where the opposite criteria was met.

6.11 Probability vs Chunk

In order to figure out the difference between Probability and Chunk(see section 3.1) The test parameters were modified to a high population size(18) as well as a higher number of generations(100,000). Since these tests were so resource intensive and time consuming they were only done 5 times for each set of parameters.This Somewhat lowers the accuracy from that of the previous experiments, but still is a good indication of how Probability and Chunk affects the results. The based on the results we saw that there were not much difference in the average final score results between the two. But Probability had a greater variance, compared to Chunk(see Figure 6.14 on page 23).

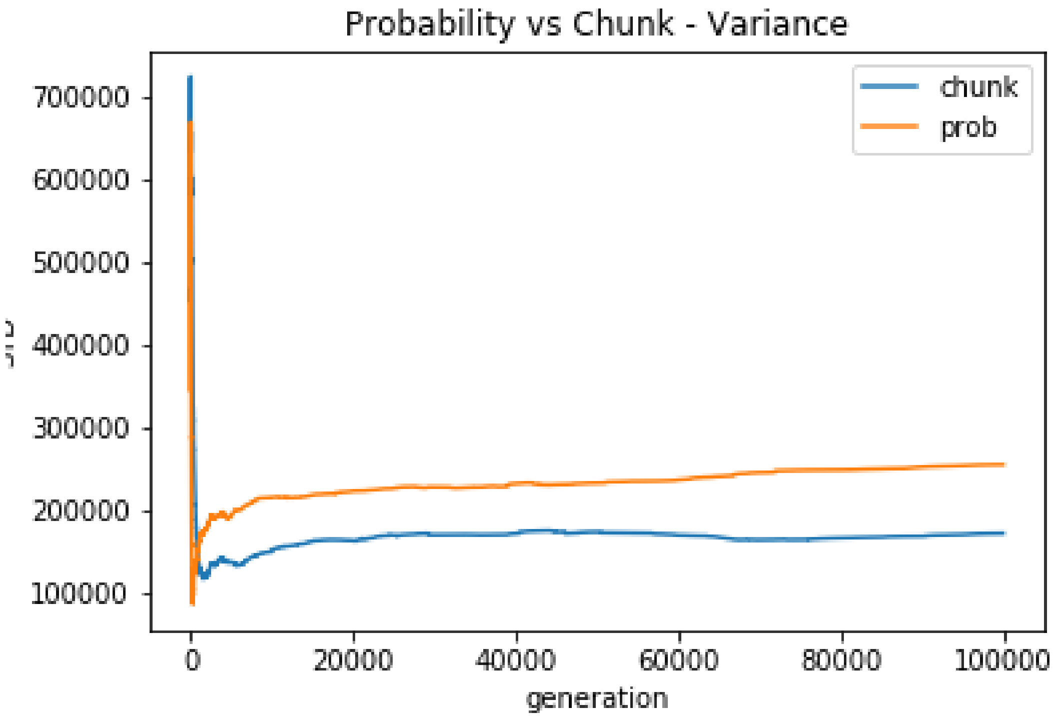
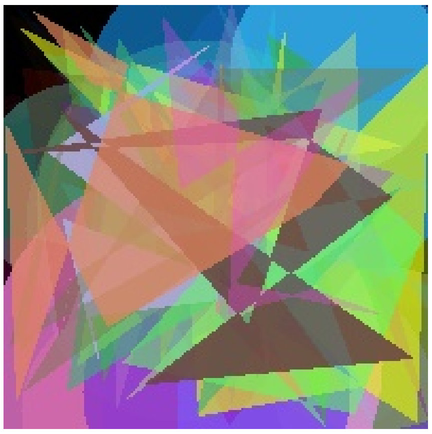
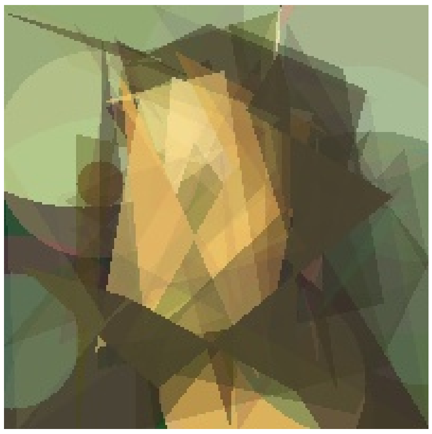
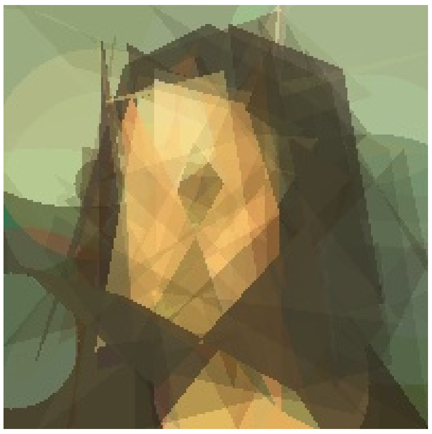


Figure 6.14: Probability vs Chunk Variance

Plot shows the standard deviation over each generation


(a) Generation 0, Fitness 7,555,657


(b) Generation 2,000, Fitness 1,735,902


(c) Generation 50,000, Fitness 1,456,640

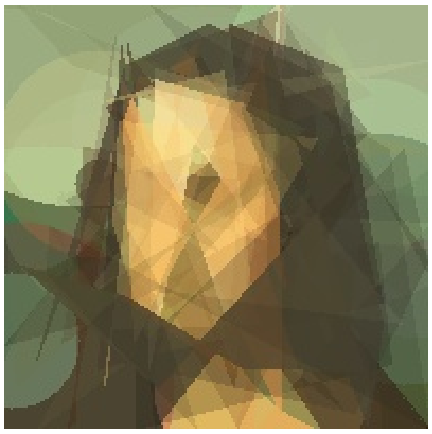

(d) Generation 100,000, Fitness 1,405,469

Figure 6.15: Samples from tests with mutation type probability over 100,000 generations

6.12 Conclusion

Table 6.6: Best Parameter Values

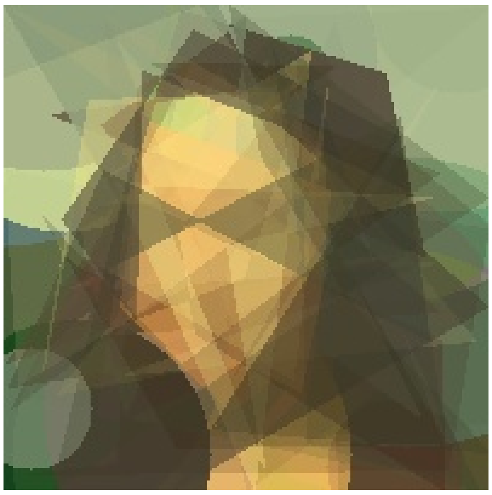
Parameter Name	Parameter Value
Number of Parents	1
Children per Parent	5
Genes Total	50
Polygons	25
Circles	25
Lines	0
Vertices	8
Mutation Probability	0.1
Genetic Restructure Rate	0
Soft Mutation Rate	0.15
Hybrid (Soft Mutation)	2
Hybrid (Medium Mutation)	1
Chunk Mutation	False
Crossover Mutation	False
Save Rate	1,000
Maximum Generations	10,000

After thorough testing, we have found a collection of *best values* for every parameter. The criteria for being the best value, is having contributed to the best mean fitness score in the test batch. Where the best values disproportionately increase the computation time, we had to consider milestones in mean fitness score and the rate of improvement between the values. Some parameter values that have been omitted[4] might have potential to give aesthetically pleasing or interesting results. However, we needed to prioritise measurable, numeric results in order to be able to make any conclusions at all. Some of the parameters we tested didn't have the impact we initially thought they would, like *Lines* and *Crossover Mutation*, and have been omitted from the collection. *Genetic Restructure Rate* has not been tested, and is also omitted. *Chunk Mutation* did not differ much from the standard Probability Mutation. Polygons and circles were both relatively good at around 25 genes, and are therefore both represented at 25 in order to maintain a 1:1 ratio, which was the best distribution of genes in the tests. *Mutation Probability*, *Soft Mutation Rate* and *Medium (Soft & Medium Mutation* were all straight-forward top picks based on

[4]Set to 0 or False

mean fitness score.

Our work shows that it is possible to generate artistic images through evolutionary algorithms, and could be used for artistic purposes. We've certainly been amazed and puzzled by the images our algorithm has produced.


(a) Batman

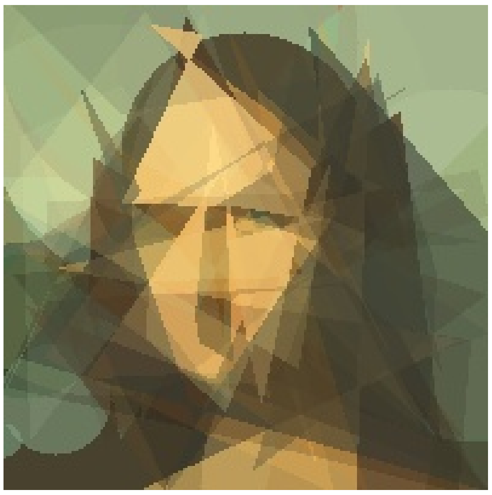

(b) One-Eye

Figure 6.16: Interesting Results

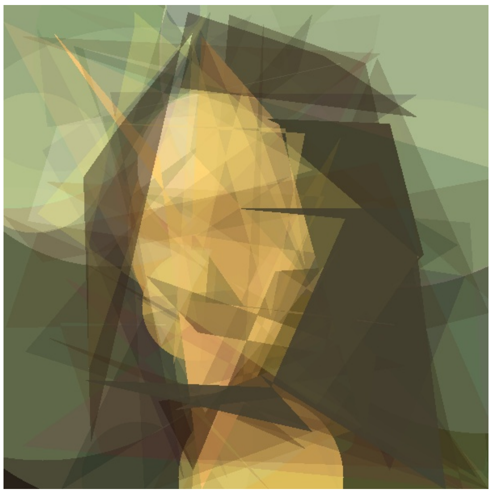


Figure 6.17: Phenotype of Mona Lisa with the most optimal parameter values (Score: 1,599,405 (94.77%))

7 Future work

Evolutionary art is a comprehensive field of research and there are several aspects that could be improved upon this project.

The program today uses Microsoft's own optimisation system for the algorithm. This made the program have some optimisation limitations on larger images. A solution to this is to split up the image in 4 different segments, and run the algorithm on every segments separately. We gave each segment their own thread and a 10,000 generation limit. When the limit is reached, the algorithm merges the top two segments together and the bottom two segments together. The algorithm runs again after the merging. The settings were the same except the generation limit, which were set to 1/2 of the starting limit. This is done to erase the sharp edges and not to continue the evolving. When the limit is reached on both the top and bottom segments, the algorithm combines them and run it one last time to erase the sharp edges. The generation limit is now set to 1/4 of the the starting limit. The segments and the result is shown in Figure 7.1, 7.2, and 7.3 on page 28.

In this project we have used direct encoding, where the genotype directly represents the phenotype, as opposed to the paper titled *"Evolved art via control of cellular automata"* (Ashlock, D., and Tsang, J., 2009), which uses indirect encoding. By this we mean it indirectly represents the genotype as a phenotype by having the genotype act like a seed. The phenotype is generated through an evolving of the genotype. Reasons for doing this can be modularity and flexibility when generating the phenotype from the genotype. We believe this implementation would make this project even more interesting, but because of the time-frame we had, we were not able to do it.

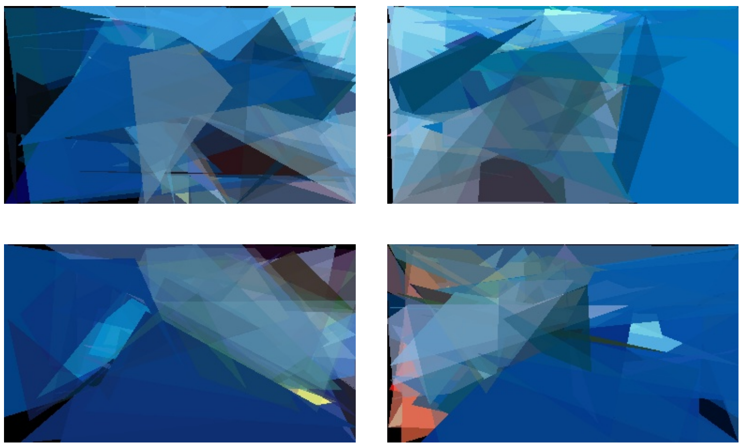


Figure 7.1: Split with 4 segments, 10,000 generations

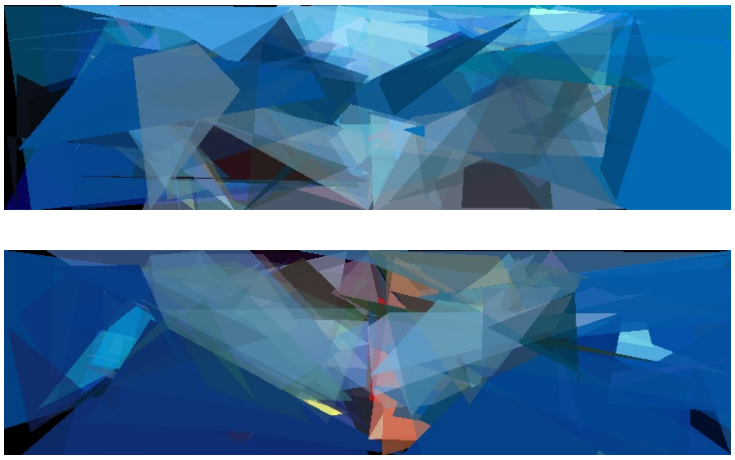


Figure 7.2: Split with 2 segments, 5,000 generations after combination

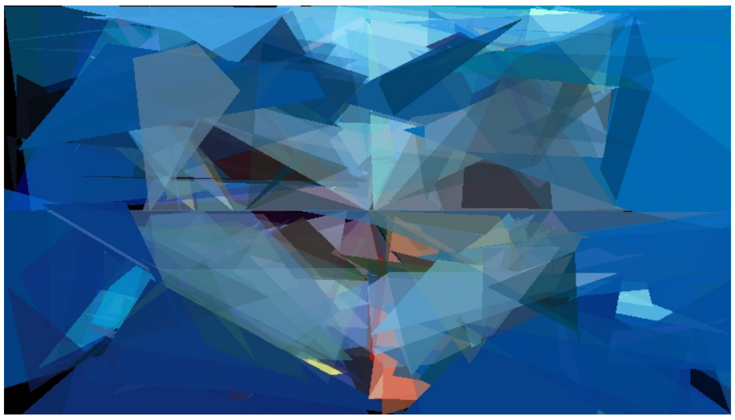


Figure 7.3: Final segments, 2,500 generations after last combination

8 REFERENCES

1. Ashlock, D., and Tsang, J. (2009) *Evolved art via control of cellular automata.* Evolutionary Computation. CEC'09. IEEE Congress on. IEEE,.

2. Dong, L., Li, C. (2013), Tarimo, W., Xing, C. *Evolving Images Using transparent Overlapping Polygons.*

3. Dorin, A. (2014), *Biological Bits: A brief guide to the ideas and artefacts of computational artificial life* (pp. 10-17, pp. 73-76, pp. 121-141), Animaland.

4. J. Romero and P. Machado. (2008), *The Art of Artificial Evolution, A Handbook on Evolutionary Art and Music.* Springer, New York.

5. Personal Genetics Education Project, *What is genotype? What is phenotype?*, Retrieved on 2018-11-20 from https://pged.org/what-is-genotype-what-is-phenotype/

Appendices

1. Appendices folder in GitHub: https://github.com/joacber/Evolved-art-with-transparent-overlapping-and-geometric-shapes/tree/master/Appendices
2. Interesting results: https://github.com/joacber/Evolved-art-with-transparent-overlapping-and-geometric-shapes/tree/master/Appendices/InterestingResults
3. Program details: https://github.com/joacber/Evolved-art-with-transparent-overlapping-and-geometric-shapes/blob/master/Appendices/Appendix%20-%20Program%20Details.docx
4. Screenshot of GitHub: https://github.com/joacber/Evolved-art-with-transparent-overlapping-and-geometric-shapes/blob/master/Appendices/Evo_Art_GitHub_Projects.JPG
5. Screensot of GUI Start Page: https://github.com/joacber/Evolved-art-with-transparent-overlapping-and-geometric-shapes/blob/master/Appendices/GUI01.JPG
6. Screenshot of GUI Main Page: https://github.com/joacber/Evolved-art-with-transparent-overlapping-and-geometric-shapes/blob/master/Appendices/GUI02.JPG
7. Screenshot of GUI Statistics Page: https://github.com/joacber/Evolved-art-with-transparent-overlapping-and-geometric-shapes/blob/master/Appendices/GUI03.JPG
8. Screenshot of Trello: https://github.com/joacber/Evolved-art-with-transparent-overlapping-and-geometric-shapes/blob/master/Appendices/Evo_Art_Trello.JPG