Non-photorealistic Rendering with Cartesian Genetic Programming using Graphic Processing Units

Illya Bakurov

Submitted in partial fulfilment
of the requirements for the degree of

Master of Science

Department of Computer Science
Brock University
St. Catharines, Ontario

Abstract

Non-photorealistic rendering (NPR) is concerned with the algorithm generation of images having unrealistic characteristics, for example, oil paintings or watercolour. Using genetic programming to evolve aesthetically pleasing NPR images is a relatively new approach in the art field, and in the majority of cases it takes a lot of time to generate results. With use of Cartesian genetic programming (CGP) and graphic processing units (GPUs), we can improve the performance of NPR image evolution. Evolutionary NPR can render images with interesting, and often unexpected, graphic effects. CGP provides a means to eliminate large, inefficient rendering expressions, while GPU acceleration parallelizes the calculations, which minimizes the time needed to get results. By using these tools, we can speed up the image generation process. Experiments revealed that CGP expressions are more concise, and search is more exploratory, than in tree-based approaches. Implementation of the system with GPUs showed significant speed-up.

Acknowledgements

Writing thesis is challenging, demanding and nerve wracking, but with the dash of a help it becomes pleasant, exciting and challenging (Yes, it does not stop being challenging no matter what). And even though, there have been many people around, who were supportive of me through this time, I would like to say special thanks to some of them.

I am grateful to my supervisor Prof. Brian Ross for all his support, amazing attention to details, constant availability and understanding. He believed in me, even when I thought I was not able to deliver. It was a wonderful experience and a sense of nostalgia will be always present in my mind.

Special thanks to Michael Gircy for sum-of-ranks implementation, which saved valuable time. Thanks to authors of ECJ, CGP, and jCuda frameworks, which allowed me to stick with the Java language (for the most part) for implementation of this thesis.

I would love to express unbelievable gratitude to my wife, Yaroslava Bakurova. You have been next to me all these years, with all advices, intelligent decisions, good eye for finding bugs, and hot tea and chocolates to boost the energy and keep us going forward. It has been a high wave for us, but we successfully rode it with excitement, and I sincerely believe that more of amazing adventures await.

I would like to thank my family, everyone of you, who believed in me, and pushed me forward. I would like to say special thanks to my dad, Andrey, my mom, Nataly, and brother, Artem. You gave me inspiration, raised my curiosity, and challenged me during this path. I always felt your love and support, even though you have been far away.

I hope that my thesis will inspire readers for even bigger experiments, and will lead to more exciting results.

> "Do not be afraid to tackle big challenges, always try to break them down into smaller pieces, and that's how you may succeed against them."

or in other words

"Big challenge is always small inside."

Illya Bakurov

Contents

1 Introduction **1**

1.1 Goals and Motivations . . . 1

1.2 Thesis Structure . . . 2

2 Background **3**

2.1 Evolutionary computation . . . 3

2.1.1 Genetic Programming . . . 4

2.1.2 Cartesian Genetic Programming . . . 4

2.1.3 Multi-objective Evaluation . . . 9

2.2 Evolutionary Art . . . 10

2.2.1 Non-Photorealistic Rendering . . . 12

2.2.2 NPR with CGP . . . 13

2.2.3 Automation of Image Evaluation . . . 15

2.2.3.1 Aesthetic model . . . 15

2.2.3.2 Colour matching . . . 16

2.2.3.3 Luminance . . . 16

2.3 GPUs . . . 17

2.3.1 GPU basics . . . 18

2.3.2 CUDA basics . . . 18

2.3.3 GP using GPUs . . . 22

2.3.4 CGP using GPUs . . . 23

3 CGP implementation of NPR **24**

3.1 Architecture . . . 24

3.2 Paint stroke application . . . 25

3.3 Parameters . . . 28

3.4 Language . . . 29

3.5 Fitness Evaluation . . . 30

4 Experiments **32**

4.1 CGP Results 32

4.2 $\mu + \lambda$ compared to population-based search 40

4.3 Conclusions 42

5 GPU acceleration of CGP NPR **46**

5.1 Use of GPU 46

5.1.1 Collecting data and sending it to GPU 47

5.1.2 GPU execution 49

5.1.3 Rendering the RGB data 49

5.2 Discussion 50

6 Conclusion **53**

6.1 Future work 53

6.1.1 Further GPU acceleration 53

6.1.2 Fitness evaluation and solution image rendering 55

6.1.3 Advanced NPR ideas 55

Bibliography **61**

List of Tables

2.1 Objectives and there corresponding ranks for 5 individuals. Low scores are preferred. SR is the final value for the normalized sum of ranks formula. max is the maximum value in specific rank. 10

3.1 Typical values which were used in setup for CGP run. 28
3.2 Inputs (terminals). 29
3.3 Functions. 31

List of Figures

2.1 General form of CGP program. It is a grid of nodes. The grid has n_c columns and n_r rows. The column of inputs n_i is on the most left side and the column of outputs O_i is on the most right side of a figure. Each node can take different amount of inputs based on the arity of the node a [35]. 5

2.2 The decoding procedure for a CGP individual (genotype). (a) Output O_1 is connected to the output node 7; move to node 7. (b) Node 7 is connected to the nodes 2 and 3; move to nodes 2 and 3. (c) Nodes 2 and 3 are both connected to the program inputs 0 and 1, thus the output O_1 is decoded. Decoding output O_2. (d) Output O_2 is connected to the output node 9; move to node 9. (e) Node 9 is connected to the input 0 and the node 5; move to the node. (f) Node 5 is connected to the inputs of the program 1 and 0, thus output O_2 is decoded. Decoding output O_3. Steps (g) and (h) describe decoding of the output O_3. After all outputs are decoded, the genotype is fully decoded [35]. 7

2.3 An example of a point mutation operator. (a) is before point mutation is applied to the CGP genotype, (b) is after point mutation is applied [35]. 8

2.4 The original image of horse (a) and the target image for CHISTQ (b). The result of the best individual (c) of a run. 17

2.5 Kepler architecture of modern NVIDIA GPUs [43]. 19

2.6 SMX of GPU with Kepler architecture [43]. 20

2.7 Structure of grids, blocks, and threads in GPU when using CUDA [45]. 21

3.1 Examples of brushstroke bitmaps. 26

3.2 Two images generated by CGP program. (a) is a small canvas to which we applied fitness, and (b) is a bigger canvas done at the end of the run. See original image in Fig. 2.4(a) 28

4.1 These are the original input images that have been used in experiments. 32
4.2 These are some of the interesting results we received from our experiments with CGP. 33
4.3 Examples from a single run. (a) Individual from generation 3386 has very high LUM objective match. (b) Individual from generation 3387 starts to satisfy a different objective. (c) Individual from generation 3786 has better colour match (CHISTQ), more visible brushstrokes (DFN), and a legible shape of cherries (LUM), thus satisfying more than one objective compared to (a) and (b). 34
4.4 Mean, SD, DFN, CHISTQ and LUM scores in 1 run with 800 generations and averaged per generation. 35
4.5 Mean, SD, DFN, CHISTQ and LUM scores in 7 runs with 800 generations each and averaged per every 10 generations. 36
4.6 The graph with scores for all 5 objectives, averaged on 7 runs with 800 generations, then averaged over a moving window of 10 generations, and normalized so they all appear in one scale from 0 to 1. 37
4.7 (a) The best individual out of all generations, but not the most aesthetically pleasing. (b) Not the fittest individual, but arguably a preferable result in which the shape of a cherry is clearly visible. 37
4.8 (a)-(f) are 6 images which show the best individuals selected by the system out of 5000 generations from 6 different runs. (g)-(l) are 6 hand picked images by the user from the same 6 runs. (g) is from generation 1753, (h) - 1369, (i) - 3322, (j) - 1200, (k) - 3884, (l) - 40. 38
4.9 Results from the run with images with a non-white background. . . . 39
4.10 (a) is the first individual (285) in the run which has a shape of the cherries visible. The graph (e) shows how fitness improved once the shape became visible. (b) is the later individual (293) in which the shape of the object is not visible and the graph (e) shows a spike in the fitness. Same goes with (c) (362) and (d) (433), where the former one has the visible shape again, and the later one does not have cherries on it. The correspondent drops and spikes are shown in the graph (e). . 39
4.11 These images are the last 16 individuals from population based run. . 41
4.12 These images are the last 16 individuals from $\mu + \lambda$ run. 41
4.13 (a) & (b) are end of a run solutions from the population-based search strategy. (c) & (d) are from $\mu + \lambda$ $(1 + 4)$ strategy. 42

4.14 Mean, SD, DFN, CHISTQ and LUM scores in 1 run with 60 generations, averaged per generation with population based strategy. 43
4.15 Cherries. 44
4.16 Blue Jay (CHISTQ target of a rainbow image is used; see Fig. 2.4(b)). 45

5.1 Conversions of genotype into phenotype and genotype into reverse Polish notation. Note how terminal and function indices are increased by 1 in (3). Also, note an extra 0 (exit symbol) at the end. (4) is reverse Polish notation with numbers replaced by terminals and functions, and without exit symbol. 48
5.2 Estimated speed comparison: GPU versus CPU. 51

Chapter 1

Introduction

Evolutionary art [32, 2] is a field in evolutionary computation which uses Darwinian evolution to automatically generate objects of art. The search for a solution in evolutionary art is not constrained, but rather, finds unexpected effects on the rendered images that are evolved. It can be used for generating textures, non-photorealistic rendering (NPR) effects, and 3D models [6]. It is also used in music, for generating musical effects and tunes.

NPR is an area of computer graphics which renders images so that they do not look like photographs. Broadly speaking, many paintings or texture renderings are forms of NPR. A difficulty, however, is to render images that are aesthetically or artistically interesting. This is hard due to the subjectivity of the assessment of aestheticity of images. It is either time-consuming, if performed interactively by a user, or not always of sufficient quality if performed automatically by the system.

1.1 Goals and Motivations

Cartesian Genetic Programming (CGP) [35] is a graph-based version of genetic programming (GP) [26] which simplifies program representation and reduces program bloat problem. Graphic Processing Units (GPUs) [28, 42] are highly efficient hardware for computational tasks, which allow the parallelization of computations into multiple threads. If we combine all of these ideas, we can get a simplification and speed-up NPR processing [54, 16], potentially leading to real-time renderings. This provides us with opportunity for an improved, faster NPR system.

In this thesis, we develop an NPR implementation with CGP, and make its computations take less time with the use of GPUs. We borrow some existing approaches [46, 36, 17] to achieve our results, in order to automate the evolutionary process. Our system uses multi-objective fitness evaluation and sum of ranks scoring. When combined with CGP, it leads to an exploratory process, which is extremely beneficial for artistic applications [49, 14, 38].

We are using CGP to implement NPR, because it solves an important issue tree-based GP for NPR faced - bloat [5]. The graph is of fixed size and is much smaller than typical GP trees [5]. This prevents CGP from generating long, ineffective expressions. Because the CGP graph is evaluated backwards, starting from output nodes, it keeps expressions concise. CGP does not spend any time evaluating those parts of graphs which are not connected to output nodes.

To achieve NPR effects, we apply brush strokes to different areas of an image, which involves calculating pixel values. This is where the efficiency of GPUs helps, as they boost the speed of calculations, and results are generated faster. With GPUs, we parallelize the evaluation of the graphs by calculating all pixels' values inside of one brushstroke at the same time. However, although possible, we do not use GPUs for fitness calculations. When considering GPUs, it is always worth evaluating whether GPUs will benefit a particular application. For example, we need to transfer data from main memory to the GPU's memory, and that can involve an overhead. With smaller data sets, GPUs may be inefficient.

1.2 Thesis Structure

The thesis is organized as follows. Chapter 2 introduces definitions and background work which this thesis is based on, and discusses evolutionary art and NPR, the CGP structure and processing, and GPUs and their use. Chapter 3 discusses how NPR is implemented with CGP. It discusses the architecture of the system, and how brushstrokes are applied to canvases. It also provides pseudo-code for the paint function, description of the CGP language, and details of the parameters that are used for the CGP runs. Chapter 4 provides details on experimental results. It discusses how CGP behaviour is different from that of tree-based systems, how multi-objective fitness influences the search process, and comparisons between different strategies. Example renderings are also given. Chapter 5 investigates how GPUs speed-up the system, and how the system architecture changes when GPUs are used. It also explains how data is passed from the CPU to the GPU and how it is processed. Finally, Chapter 6 summarizes the thesis, and discusses what improvements could be made to the system in the future.

Chapter 2

Background

2.1 Evolutionary computation

Evolutionary computation is a branch of computational intelligence which uses heuristic search algorithms inspired by Darwin's theory of evolution. Holland originally proposed the genetic algorithm in 1975 [23].

The solution to many computational problems can be stated as a task of finding a string of symbols that describes a solution. Consider the travelling salesman problem (TSP), which is a well-known problem, which in most cases does not have a tractable solution. The problem is in finding the shortest path between all the points C_i. The path starts and ends in the same point. If we label every point in the following way; C_1, C_2, C_3,..., C_n, then some permutation of this string will be our solution.

The evolutionary algorithm would approach this problem in multiple steps as follows.

1. Randomly generate an initial population, meaning a number of strings (chromosomes) out of labels (might not use all of them) by random permutations.

2. Evaluate each individual in the population and assign appropriate fitness scores to the individuals, so we could compare these individuals to each other later. Fitness in this particular problem will be the distance covered by the route between points. Thus, we are trying to minimize the distance.

3. Based on the fitness scores, the program selects chromosomes to create the next generation. These selected chromosomes are called 'parents'

4. Individuals produced by 'parents' are called 'children'. The children are generated via the reproduction operators such as crossover and mutation. Crossover

generates child from two parts, each of which is inherited from different parent. Mutation generates child by substituting a part of a parent.

5. Repeat steps 2-4 until solution is found or time limit is reached.

The detailed description of how the further populations generated and possible pitfalls and solutions to them are described in [35].

2.1.1 Genetic Programming

Genetic programming (GP) is the automatic evolution of computer programs [26]. It uses a tree-based data structure. Genetic programming is capable of searching the space of possible solutions to the problem and identifying the fittest solution to the problem at hand. Generated solutions by GP are rarely the most compact structures. Usually, they have substructures which do not influence the solution and are left totally unused [26]. This is called bloat. The problem with this phenomenon is that chromosomes become larger and larger without any increase in performance (fitness). Programs have large subexpressions that are redundant and inefficient, moreover, the execution time of bloated programs is longer. In addition, the solutions that are generated are hard to understand.

2.1.2 Cartesian Genetic Programming

CGP uses directed acyclic graphs instead of trees, which solves the problem of bloat, as well as simplifies computations. Each node in a graph represents a function with multiple inputs and outputs. This way the program is more compact, because it allows the reuse the values of previously computed subgraphs. The graphs are described by a two-dimensional grid of nodes. Integer indices represent which nodes are connected to which, depicting inputs and outputs as well as the operations that a node should perform on input data. These indices are called genes, and they make up the genotype of CGP. CGP is decoded backwards, starting from outputs. This means that some nodes may be ignored, because some node's outputs are not connected to any inputs of other nodes, and do not lead to the inputs of the graph. Such nodes are called 'non-coding'. The program we receive after decoding of genotype is called the phenotype [35]. CGP cannot suffer from inefficient chromosome growth, because the genotype of the chromosome is of fixed size. Moreover, when large genotypes are allowed by the program, the phenotypes remain small [35].

The types of functional nodes used in CGP are decided by the user and stored in a look-up table. Each node in the graph represents a function, and this node is encoded by a number of genes. One gene, which is called a function gene, is an address of the node's function in the look-up table. The rest of the genes represent where the node gets its data from. These genes are called connection genes. Nodes' inputs are taken from either nodes in a previous column or from input of a program [35].

Fig. 2.1 shows the general form of a structure of a CGP program. The n_O integers are added to the end of the genotype, respectively to needed amount of program outputs. Any two nodes in one column cannot be connected to each other. There are three parameters in CGP which are chosen by the user: number of columns (n_c), number of rows (n_r) and levels-back (l). The number of columns and rows determine the size of the grid, and thus it determines the number of computational nodes, which CGP graph will generate: $L_n = n_c * n_r$. This means that this is the maximum amount of nodes the graph may has. It is worth mentioning that there is a possibility that CGP might generate fewer nodes than this number specifies. Levels-back determine the number of previous columns the current node can get input from. If $l = 1$, a node can get its input values from a node in the immediate previous column (column with computational nodes or program inputs), and not further than the previous column. If the user wishes to allow connectivity between nodes without such constraint, then $l = n_c$.

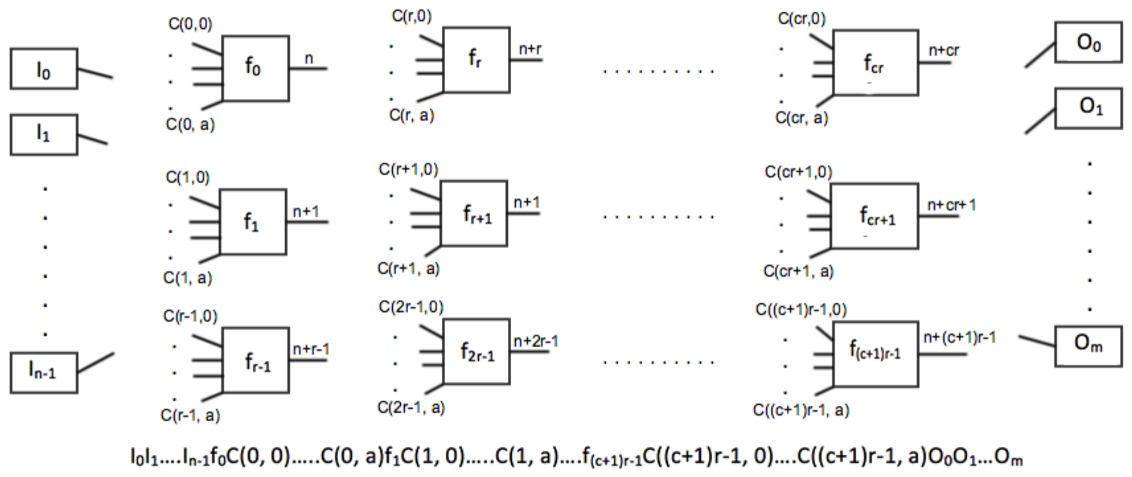


Figure 2.1: General form of CGP program. It is a grid of nodes. The grid has n_c columns and n_r rows. The column of inputs n_i is on the most left side and the column of outputs O_i is on the most right side of a figure. Each node can take different amount of inputs based on the arity of the node a [35].

The decoding of the CGP genotype into a phenotype is what we are interested in the most, because that is what gives CGP an advantage over tree-based GP. The decoding is a recursive process that starts from the outputs, and goes all the way to the inputs. Because of this, if the node's output is not connected to any further node's input or the output of the program, it will not be added to the phenotype, and it won't be considered in the evaluation of the individual at all. Representation of the decoding process can be found in Fig. 2.2

Crossover operators are not normally used in CGP. Originally, one-point crossover operator was used in CGP, but later studies of it show that it affects subgraphs within the genotype and has a detrimental effect on the total performance [34].

Point mutation operator is a mutation operator which is used in CGP. A node on a randomly chosen position of a gene is changed to another valid random node. It should be valid in terms of the arity of inputs and outputs as well as the type of the data it processes. The number of point mutation operations in a single application of the mutation is defined by the user. Usually, a percentage of the total number of genes is taken as a value of this parameter. This parameter is called mutation rate and is described by μ_r. Fig. 2.3 shows how one mutation can dramatically change the output of the individual.

A variation on a simple evolutionary algorithm [35] $1+\lambda$ is used in CGP programs. It is common to set $\lambda = 4$. It has the following procedure. We select the fittest individual, which is going to be parent for the next generation. We generate 4 different mutations of the parent to generate offspring. If an offspring individual has a better or equal fitness to the parent, then offspring is selected. Else the chromosome of the parent remains to be the fittest and we go to the next offspring. It is important to note that in the previous step, we check the fitness of an offspring chromosome to be better or equal. This way, even though, an offspring has the same fitness, it brings variety into the population and increases the diversity of search.

As with any type of GP, to find out a good set of values for the program parameters, one requires some experimentation on the problem being considered. There are some general suggestions that can be given. The user should set the mutation rate based on the length of the genotype. As a general rule, it should not go over 1% of mutation if 100 nodes are used as a maximum value of nodes in genotype. For fast evolution it is advised to find a mutation rate that it would be a slowly growing function of the genotype length. Finding optimal mutation rates is beneficial to the speed of a run of a program, because larger genotypes require fewer fitness evaluations to find a solution than do smaller individuals. In CGP, small sizes of population are usually

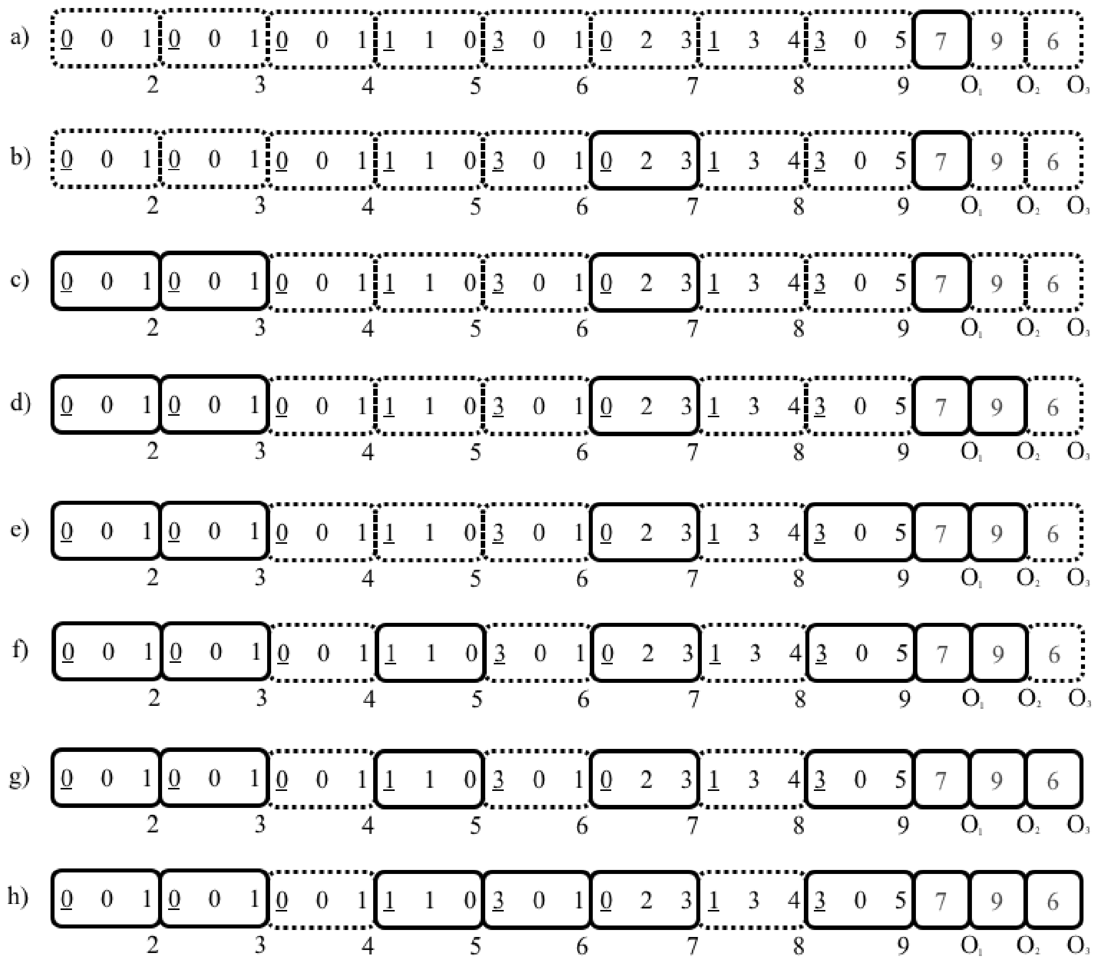
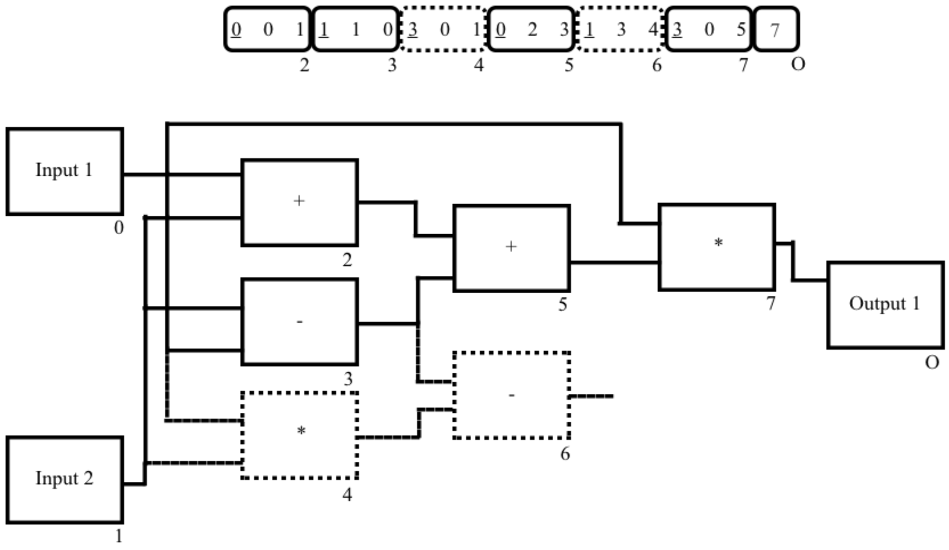


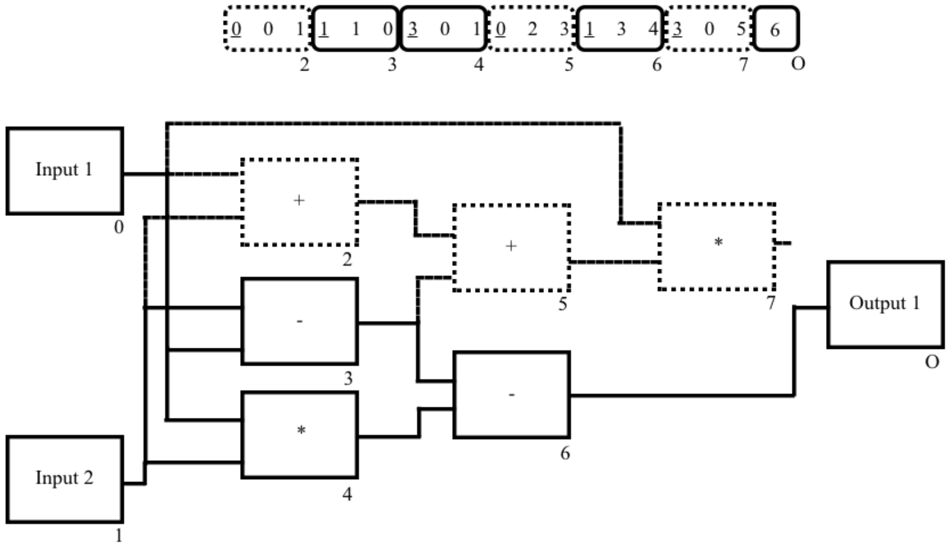
Figure 2.2: The decoding procedure for a CGP individual (genotype). (a) Output O_1 is connected to the output node 7; move to node 7. (b) Node 7 is connected to the nodes 2 and 3; move to nodes 2 and 3. (c) Nodes 2 and 3 are both connected to the program inputs 0 and 1, thus the output O_1 is decoded. Decoding output O_2. (d) Output O_2 is connected to the output node 9; move to node 9. (e) Node 9 is connected to the input 0 and the node 5; move to the node. (f) Node 5 is connected to the inputs of the program 1 and 0, thus output O_2 is decoded. Decoding output O_3. Steps (g) and (h) describe decoding of the output O_3. After all outputs are decoded, the genotype is fully decoded [35].

used, and 5 is typical as a value. Because of this, the user should set a large number of generations to be used, e.g. 2000.

Large genotypes usually lead to more efficient solutions and evolution overall. The question here would be what values should parameters n_r (number of rows,) n_c (number of columns) and l (levels-back) have? If implementing arbitrary directed graphs is not a problem in a CGP program, then the rule of thumb is to use $n_r = 1$



(a)



(b)

Figure 2.3: An example of a point mutation operator. (a) is before point mutation is applied to the CGP genotype, (b) is after point mutation is applied [35].

with $l = n_c$. But, if there are any constraints or limitations, such as in the problem of generating circuits which are going to be implemented on digital devices, then the user should choose $n_c = n_r$. l in this case is arbitrary and should be tested by the user for a specific problem [35].

CGP can represent many computational structures, from digital circuits to mathematical equations. The one of the most interest to us is for evolutionary art. To give an example: there is a simple way to generate a picture with a CGP. We set the (x, y) coordinates of the pixel on a canvas as integer inputs to CGP, and declare three outputs for this pixel, meaning red, green and blue values of it. We should keep the output values in the range in between 0 and 255, because we need a valid colour value for the pixel components. Once the value is computed, the pixel is coloured. CGP continues to process all the pixel coordinates until the whole two-dimensional canvas is rendered.

2.1.3 Multi-objective Evaluation

There are problems which have single criterion, which means there is a single objective to satisfy in the fitness evaluation. There are different fitness formulas which could be used in this case: sum of absolute errors, number of correct examples classified, etc.. However, there are many application problems with several objectives that must be optimized. This type of problem is divided into two subtypes: multi-objective and many-objective problems. Multi-objective problems have 4 or fewer objectives. We can use weighted sum or Pareto rank [15] as fitness formulas to optimize objectives. A weighted sum combines the separate objective scores into a single score. The weighted sum approach depends heavily on the weights which we assign to objectives. This can bias the system to satisfy objectives accordingly to the weights chosen.

Pareto ranking is a popular multi-objective technique. One individual is said to dominate another if it is better in at least one objective, and is tied in the remaining objectives [15]. When Pareto ranking is used, diversity becomes an issue, because of the convergence towards the top rank values. Moreover, Pareto dominance fails with too many objectives, because then almost all individuals are scored equally. Because of this, there is no selective pressure, and it becomes more like a random search.

Many-objective problems are those with more than 4 objectives. One strategy for many-objective optimization is to use normalized sum-of-ranks [7, 12]. Sum-of-ranks ranks each objective separately for all individuals, then all scores are summed [7, 12]. The smaller score is better. However, there can be a broad range of scores for each objective, and this can cause some objectives to dominate the overall sum. This is solved by normalizing the objective values, so that they are all in the range between 0 and 1. Now, after summation of all objectives for each individual, a more balanced score arises.

An example of normalized sum of ranks calculations for 5 individuals is in Table 2.1. At first, we assign ranks (R_1 to R_5) to each individual for each objective (O_1 to O_5) based on how they scored compared to other individuals. To normalize these ranks, we divide each rank by the maximum rank for that objective (row with 'max' values shows the maximum value for the objective). Finally, we sum all ranks for individual and end up with values which we can compare and pick the best individual out of 5 (SR). In our case (Table 2.1) it is the fifth individual.

	Objectives					Ranks					
	O_1	O_2	O_3	O_4	O_5	R_1	R_2	R_3	R_4	R_5	SR
1)	1.542	0.071	1.406	0.559	0.621	3	2	3	2	4	3
2)	1.579	0.039	1.338	0.567	0.589	5	1	2	3	1	2.6
3)	1.550	0.091	1.168	0.567	0.629	4	3	1	3	5	3.5
4)	1.478	0.166	1.445	0.576	0.603	2	4	5	4	3	4
5)	1.476	0.039	1.410	0.563	0.594	1	1	4	1	2	1.9
max:						5	4	5	4	5	

Table 2.1: Objectives and there corresponding ranks for 5 individuals. Low scores are preferred. SR is the final value for the normalized sum of ranks formula. max is the maximum value in specific rank.

2.2 Evolutionary Art

Evolutionary art is a technique which is used to evolve aesthetically pleasing images with evolutionary computation [6]. The biggest difference of evolutionary art in comparison with other search problems is that it is hard to assess generated individuals and assign fitness score to them due to highly subjective understanding of what could be attractive, since this understanding varies from person to person. Instead of trying to optimize or find an exact solution to the problem, evolutionary art is more exploratory in nature. There is not one exact solution to be found, nor are there strict fitness criteria towards what is an artistic image. Diversity of results is one of the main advantages of evolutionary art applications.

Karl Sims was a pioneer who used GP to render 2 dimensional art, and his work affected most of the presently accessible developmental art software applications [51]. William Latham and Stephen Todd used evolution to generate 3D models [55]. Richard Dawkins showed how evolution could be used to generate a variety of 2D images [13]. In early attempts, evolutionary art did not have automatic fitness functions to assess the attractiveness of an image, and instead the user interactively provides the fitness of an image.

Most early evolutionary art programs require a person as a fitness evaluator. The reason for this is that it is hard to define a selection criteria to automate this process, because we are dealing with subjective artistic content. Thus, evolutionary programs require a user to subjectively select individuals and assign them a fitness. Moreover, some tools allow users to tweak mutation rates to control the variety of results produced by an application. However, this approach limits the variety of results and effectiveness of evolution possibilities. Because we have a real person sitting and selecting images one-by-one, we are risking to exhaust the user, or end up with a minor amount of diversity in the results.

Automatic texture evolution overcomes this limitation. It eliminates person from the fitness evaluation process and automates it. Early research using this approach could detect only simple characteristics of an image such as colour, luminosity and shape of objects [1, 25, 56, 48]. When the scores for these parameters are calculated they are compared to the target image, and the fitness scores reflect how close they are to the target image. The aim is to evolve a solution with characteristics as close as possible to those of the target.

An even more advanced problem is automating aesthetic texture evolution. There are a few examples of research that do this [30, 53], and the approach is still making its way into the evolutionary art field. Baluja *et al.* [3] were the first who used an artificial neural network as a fitness evaluation function in their GA program. The network would be trained by the target images, and then it would evaluate a fitness score for new individuals. Unfortunately, such an approach does not let us know how the neural network assessed individuals and what kind of criteria it built. It is also unclear if it learned any principles to assess aesthetics of a generated individual.

Another approach by Machado *et al.* [31] is the NEvAr system. The model assesses aesthetics from two parameters: if an image is visually complex but also easy to perceive by human and further interpret. There are two corresponding mathematical measurements: JPEG compression ratio is used to assess image complexity, and fractal compression ratio is used to assess how hard it is to perceive an image. The results of using this model were extremely satisfactory. A study has been used it on a standardized psychological test called Test of Drawing Appreciation [30].

2.2.1 Non-Photorealistic Rendering

Non-photorealistic rendering (NPR) [54, 16] is a major area of computer graphics which enables new ways and tools to render visual effects. In particular, NPR algorithms are meant to create artistic renderings for aesthetic purposes. NPR is less constrained than photorealistic rendering, because any rendering effect which is not intended to express realism can qualify as an NPR technique. This could refer to oil paintings, watercolour effects, pen and/or pencil drawings, 3D rendered objects, abstract computer graphics, etc.. These all can be separated into two main groups [5]:

- Physical modeling: Modeling of 3D objects, their appearance and physical properties. This could address construction tools, animal models, medicines, chemistry and biology elements, etc. Everything that can be and should be emulated on a computer by representing it as a 3D model is falling under this category. These models are done with the purposes of simulating properties and reactions or educational purposes.
- Artistic modeling: Application of artistic tools, such as paint brush or watercolour or oil to already existing images, or creating and painting from scratch on an empty canvas. These are done mostly for aesthetic purposes.

Because the aim of this thesis is mainly focusing on the second approach to render NPR, we will describe it in more detail. It is further divided into three themes which are commonly seen as a discussion topic in the vast majority of research [5].

- Modeling physical properties: This is almost the same as the first type of NPR effects presented above, but the difference is that we are considering it in the context of paintings. This may refer to properties of paint, pens and pencils, how they move in 3D space or how they are applied to canvas.
- Modeling optical properties: This area discusses how paints change when mixed together, how oil and watercolour could mix and create visual effects. It can refer to many more different effects in terms of how they appear on a canvas and how they change after interfering with each other.

- Simulating application of brushes: This explores the ways in which humans work with natural media by applying brush strokes or pencil or pen to the canvas. This is important because it gives an actual feeling that the image is not computer rendered, rather drawn by human. The closer we can simulate this effect, better the resulting effect on final images.

There has been much work in NPR. A few examples are as follows. Hertzmann [21] generates images with a hand-painted effects using a series of spline brush strokes onto a canvas. Analyzing source images, he also applied different sized and rotated brush strokes which led to more pleasing results.

Colton *et al.* [11] presented the tool called the Painting Fool. The main idea is to analyze a short video with a person on it, and recognize an emotion with the help of machine learning. After the emotion recognized, it is mapped to an artistic style. This style is applied to an image which results in interesting and unexpected NPR effects.

Hertzmann *et al.* [22] presented a technique called image analogies. The idea is to use machine learning to render NPR effects. They use an input image and a generated image with NPR effect on it as a training data. After the tool is trained, it can reconstruct the transformation of generated image on a different target image. They observed a good results with the approach.

Shirashi and Yamaguchi [50] presented an efficient method to automatically render paint strokes on a canvas. The system analyzes an image for colour layouts and details, and then applies brush strokes, rendering an image as if it was painted by a human.

2.2.2 NPR with CGP

One of the benefits of CGP usage in NPR is in providing valuable domain specific knowledge to the program. This could be done in two ways: through the input parameters with pre-calculated values (in case of image processing, it can be the red, green and blue values of a pixel, or pixel coordinates, etc.); or through the function set (for example, high-level image processing filters such as mean, standard deviation, luminosity, etc.)

It should be pointed out that NPR implementations using CGP are not common. There were two articles related to the topic of NPR with CGP [2, 19].

The work of Harding [19] uses OpenCV image processing functions in the CGP language. His aim is to show the efficacy of CGP in multiple image processing domains. One of them is the noise reduction problem. There were two noise reduction problems on which program was tested: 'salt and pepper' and Gaussian noise. Both of them have been solved by CGP with good results compared to previous research on this problem. Gaussian noise is a harder problem to solve, and thus CGP showed a little decrease in performance than with the 'salt and pepper' noise problem.

For robotics vision, Harding [19] used CGP with MCC based fitness evaluation. The problem overall is hard, especially when we consider robots in everyday life. There are different light sources and types of lights. Moreover, objects are not predefined and angles under which robots may examine the same object are infinite. It is a different situation in industrial scene, where companies can keep the same lightning conditions at all times and mark products with data which robots can easily identify. Here, problem images from the robot's cameras are passed to CGP for identification of objects. CGP program is trained with multiple images of the same object from different angles and with different lightning conditions. The results were successful and showed that CGP is applicable to vision problems [19].

Representation of the individual is usually a key factor in evolutionary algorithms. Ashmore and Miller [2] focus on this to achieve artistic evolutionary image generation. The graph has two inputs which represent coordinates of the pixel on the image (x, y) and three outputs (red, green, blue) - the colour of this pixel. Thus, CGP generating the whole image pixel by pixel. In this case, more nodes in the chromosome result in more complex images. But more diversity in results means more noise, so it is important to balance the graph size.

Evolution operators such as crossover and mutation are both used in CGP, but usually we use mutation in CGP than a crossover, because crossover is thought to be more destructive operator for the graph structure. However, the work in [2] shows that crossover is useful: sometimes evolved individuals would have a colour from one parent and shape from another [2]. Another good approach is to use adaptive mutation. It means that the mutation rate will be higher in the early generations and will decrease in later generations.

We already pointed out that the biggest problem of evolutionary art is fitness evaluation. It is time-consuming and hard to automate. But it also might be hard for a human to choose the best individuals in early generations, if the run is randomly seeded. This is due to the great randomness of the first generated results. To overcome this, we may use some kind of predefined fitness functions which target one or another characteristic of a target image [2], and which pre-select images show to the user.

2.2.3 Automation of Image Evaluation

The following subsection reviews ideas used in [5, 46, 36] that are used in this thesis for an automatic image evaluation. In particular, we review Ralph's aesthetic model (DFN), as well as colour and shape matching.

2.2.3.1 Aesthetic model

Fitness evaluation of aesthetics and artistic quality is not obvious to implement. It is due to the fact that aesthetics is very subjective. There is some pioneering research [46, 17] concerning this problem and they both offer potential solutions to it. Also, the problem is discussed in detail in [36].

Research shows that human's response is attracted to changes in colour in an image. However, the change in colour cannot be drastic (noise), nor be static (uniformity). Ralph [46] proposed a measurement called Deviation from Normality (DFN) for colour change in an image. He found that many famous paintings conform to a normal distribution of colour gradient changes, even though this was probably a subconscious task of the artists.

DFN can be used as a fitness criteria in evolutionary art [46]. DFN measures deviation of gradient response from a normal curve distribution [46]. Thus, we are targeting DFN to be equal to zero, which means that it matches perfectly with normal distribution. In addition to DFN, Ralph uses mean and standard deviation (SD) measures as additional parameters for fitness scoring. It was shown that if DFN scores are less than 3.0, then it is often the case that mean = 3.1, and SD = 0.75 for works of art. To implement these scores, we minimize the DFN, and guide the mean and SD towards supplied target values (e.g. 3.1 and 0.75 respectively).

Ralph's DFN approach was used in numerous works such as evolutionary art research in texture generation [9, 47], 3D model generation [8], NPR with GP [4] and image filtering [36]. However, it should be noted that, as with any heuristic search for aesthetically pleasing art, obtaining a perfectly scoring individual does not guarantee an amazingly pleasing result. DFN is helpful in a way that it moderates complexity of an image and keeps it balanced in terms of visual appeal. Finally, one more advantage of using DFN is that images with lower DFN scores often have more visible brushstrokes, and thus makes them more interesting results for NPR applications[36].

2.2.3.2 Colour matching

Another useful measurement is colour matching. The intent is for generated images to use the colours close to a provided target image. This is not a perfect colour matching score, as it does not constrain images to have colours in the same locations as a target image. A generated image is processed to calculate a quantized colour histogram ($histI$). This is repeats for the target image, resulting in a quantized colour histogram ($histT$) for it [52]. The quadratic distance between the two histograms is then calculated [5]:

$$colourDist = \sum_{i,j} |histT_i - histI_i| * sim(i,j) * |histT_j - histI_j|$$

where *sim(i, j)* is the RGB colour distance between the quantized colours in bins i and j. The formula calculates the quadratic distance between two histograms, where each of those histograms is a summation of the colour distance between all colours on an image. Lower scores mean closer images match [4]. This approach to colour fitness evaluation has two advantages:

1. It can be used to keep generated images to be close to the original one in colouring (if original image used as a target image for CHISTQ)
2. It can allow mixing colours from different images, resulting in unexpected results (green sunset, purple buildings, etc.). See example on Fig. 2.4

2.2.3.3 Luminance

Luminance can preserve features of a target image, such as shape, size and positioning of the objects. CHISTQ and DFN discussed above do not do this, and this is why luminance is useful. To calculate LUM, the luminance (greyscale image) for the target image and generated image are computed. Luminance of an image is calculated by the following formula [57]:

$$luminosity = 0.299 * Red + 0.5879 * Green + 0.114 * Blue$$

We then compare the pixel differences between the source and target luminance values pixel-by-pixel. The sum of all the differences becomes the LUM value. When LUM is

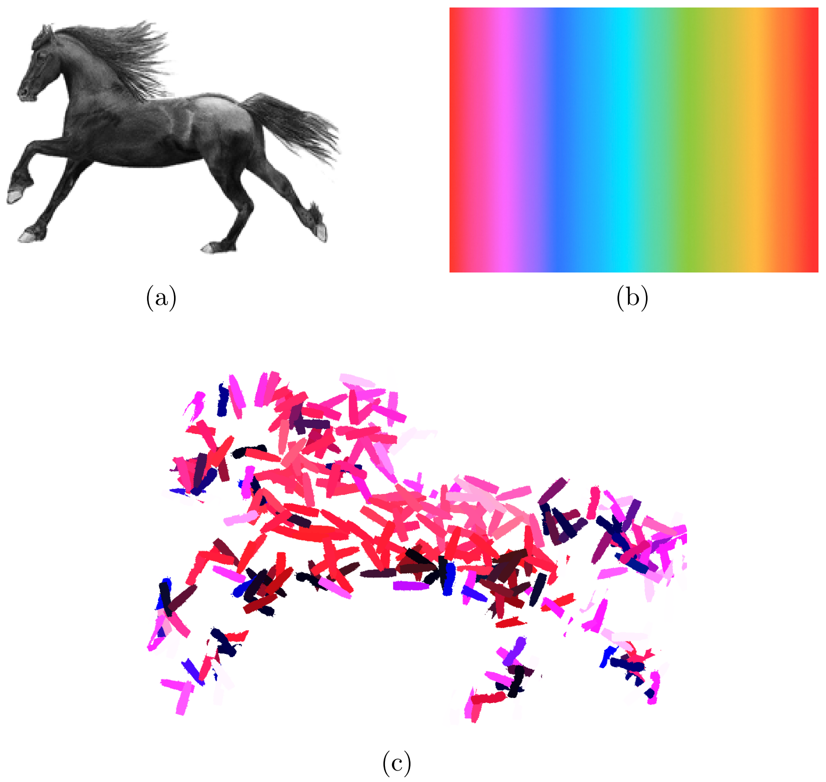


Figure 2.4: The original image of horse (a) and the target image for CHISTQ (b). The result of the best individual (c) of a run.

zero it would mean that we have two identical greyscale images. Of course, we would not want generated images to be the exact copy of the target image, because that would eliminate any NPR effect. But when LUM is used along DFN and CHISTQ, it preserves features of the target image. Otherwise, image details can be lost.

2.3 GPUs

Graphic Processing Units are increasingly popular tools in scientific fields, because of their powerful computational performance. GPUs were created to efficiently calculate colours (red, green, blue) for the pixels on the screen, notably in game applications. Regular video has 60 frames per second, and screens nowadays have 4K (3820x2160 pixels) resolution, which ends up to almost 500 million computations per second, normally calculated in parallel. That is not the maximum performance bar for modern GPUs, because new cards have more and more cores, which leads to more parallelized actions. NVIDIA GPUs [42] can be programmed on almost any computer with the CUDA [39] programming language (which was also created by NVIDIA) to get the most efficiency from the hardware for scientific applications.

It is possible to mount multiple GPUs to one computer, although more than two GPUs per PC are rare. There are multiple reasons for this, such as physical constraints, problems with motherboard connections, and power consumption. However, it is common to see more than two GPUs (usually Tesla - they are the most powerful in the market in 2017) in many backend systems for artificial network applications [28].

We are using the CUDA language to program GPUs [42]. However, there are also AMD's ATI series GPU on the market, which are competitive with NVIDIA's GPUs. Despite this fact, almost all recent GP applications use NVIDIA's graphic cards and CUDA. There are multiple reasons for this. First, there is a practical library and coding environment for NVIDIA's GPUs, which is not present within AMD's eco-sphere. Second, modern versions of CUDA SDKs are more advanced in their implementation and less restrictive; for example, they support recursion and function pointers.

2.3.1 GPU basics

New GPUs are always increasing the amount of cores which fit into the single streaming multiprocessor (SM) and how they operate with memory. We are using NVIDIA GeForce GT 750M [40] with Kepler architecture [41] (features from GK104 and GK110) which has 192 cores in each SM, compared to Fermi architecture [41] which had only 32 cores per SM. Each SM also has shared memory, cache memory, instructions cache, load/store units and other components which may wary from model to model. See Fig. 2.5 and Fig. 2.6 for the full list of components and architecture outline.

2.3.2 CUDA basics

CUDA language is an extension of C language. CUDA programs compile with NVIDIA's CUDA compiler which is called nvcc. It can also compile regular C/C++ code. nvcc works with many command line parameters used by the GNU gcc compiler. We can set conditional switches, or pass custom parameters to the main function of the program. CUDA works with both 32 bit and 64 bit computer systems. But it is always necessary to check what kind of library is installed, to be sure that the current PC supports the CUDA library, and to link the proper library when compiling an executable program [28].

Fig. 2.7 shows the structure of grids, blocks, and threads when using CUDA. We specify how many blocks we need for our program and how many kernel launches

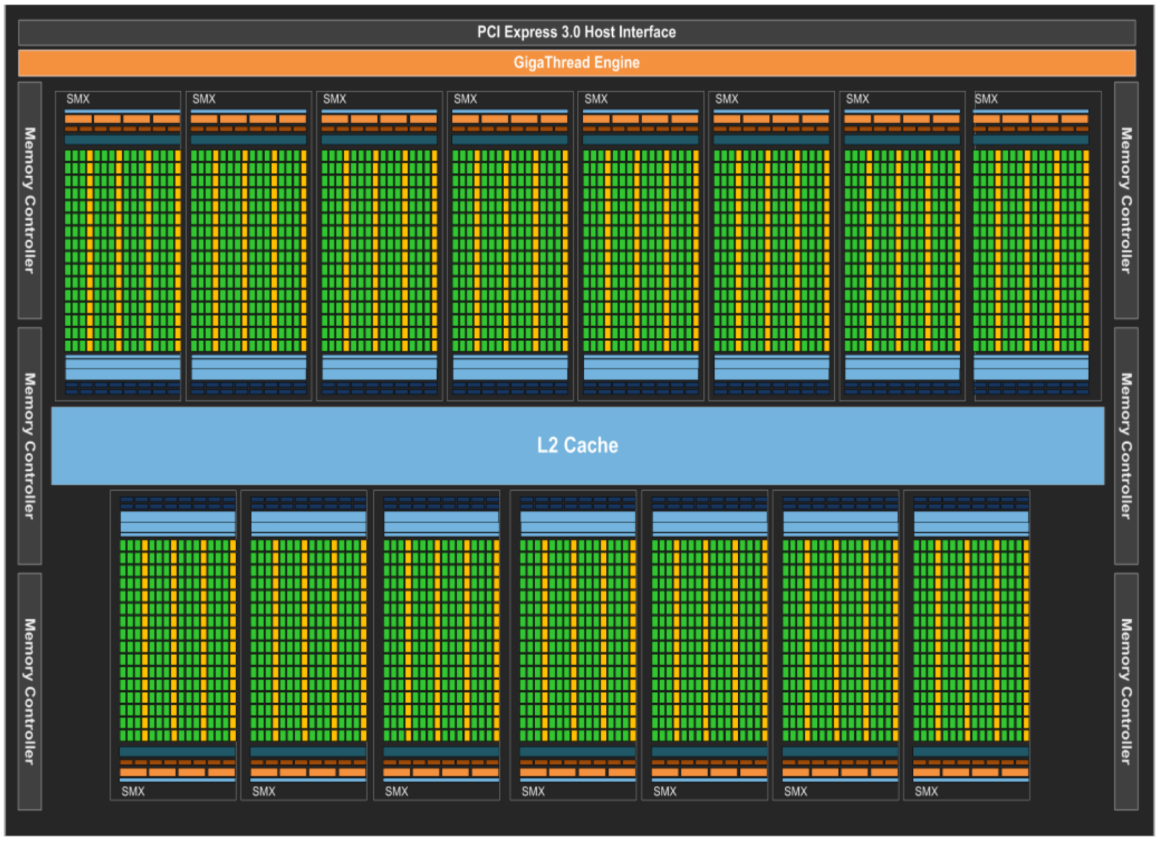


Figure 2.5: Kepler architecture of modern NVIDIA GPUs [43].

we will be doing, and then the GPU sets up the grids based on that. Every block has unique ID ($blockIdx.x$), and every thread has unique ID ($threadIdx.x$) inside the block. We use this data inside the kernel function to determine which data ($dataIndex$) we should process in specific kernel call. We also use the total number of threads in a block ($blockDim.x$) to skip thread IDs, if we are not in the first block. It is done with the following formula: $dataIndex = threadIdx.x + blockIdx.x * blockDim.x$.

We will describe some rules of thumb to consider when one is designing a CUDA application, as well as give an overall introduction into CUDA performance, implementation, profiling and limitations (see [28, 39] for more info). The best practices for CUDA programming are as follows:

- The very first question to think about is, how much of the application is going to be transferred to the GPU? In other words, according to Langdon [28], what part of the application can be run in parallel? Usually, if it is less than 90% of application, then it might not be worthwhile to use GPUs. Even if the part which the user is trying to speed up, could be sped up infinitely, the overall application speed up would not be tremendous in such cases [28].

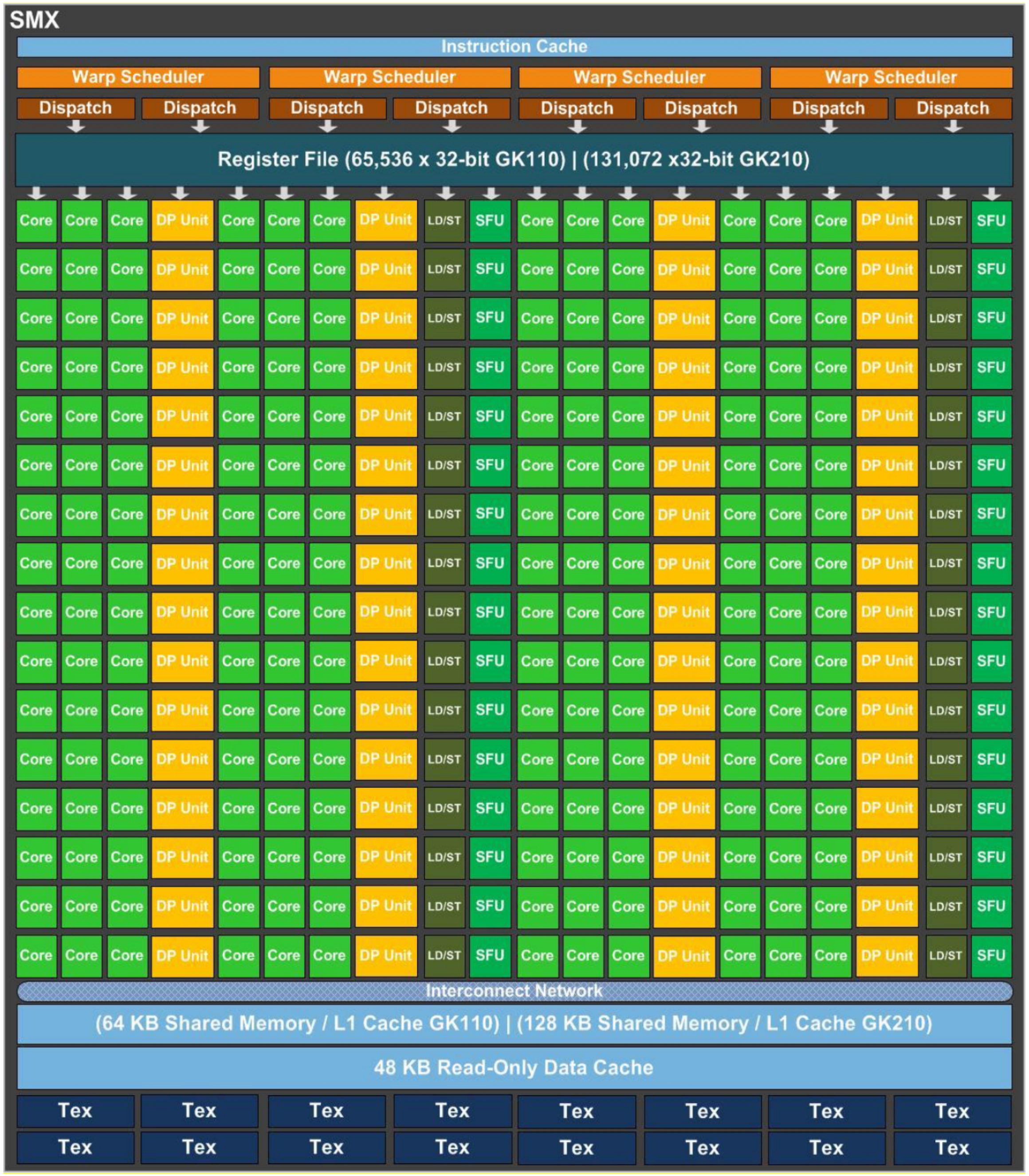

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit GK110) | (131,072 x32-bit GK210)
Core
DP Unit
LD/ST
SFU
Interconnect Network
(64 KB Shared Memory / L1 Cache GK110) | (128 KB Shared Memory / L1 Cache GK210)
48 KB Read-Only Data Cache
Tex

Figure 2.6: SMX of GPU with Kepler architecture [43].

- In GP, and all evolutionary computing, the most resource-intensive step of applications is fitness evaluation. Fortunately, the fitness evaluation of each individual in the population could be run in parallel independently, and that is why this part of an application is perfect for GPU acceleration.

- It is always worth while to try to locate bottlenecks [28]. For example

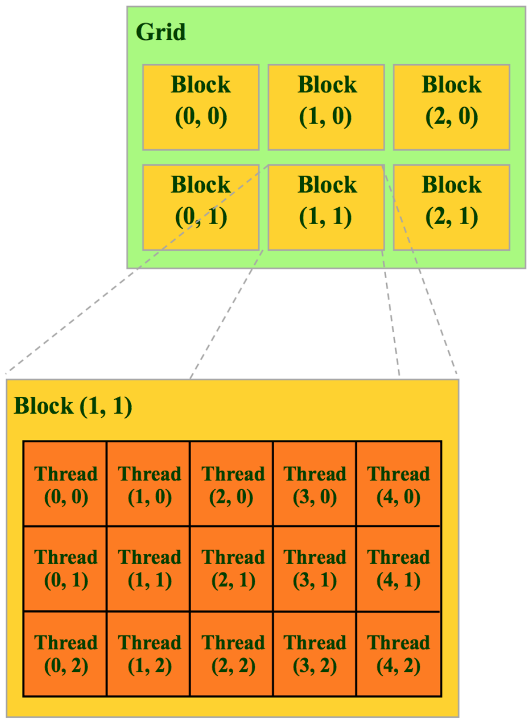


Figure 2.7: Structure of grids, blocks, and threads in GPU when using CUDA [45].

  - Amount of data which is uploading to GPU.
  - Amount of data which is transferring back from GPU to PC.
  - Number of interactions between PC and GPU

  Trying to code kernel as it would not use the local memory is a good approach. We are thinking about first three items in this list in terms of flows between global memory and kernel. With good coding it is relatively easy to get accurate estimates, but with internal transfers it is more dependent on the details of the system: effectiveness of caches, threads overlapping computation with fetching and transferring data.

- Ratio of the volume of PCIe data size to the PCIe's data buffer size will approximately determine number of times the operating system has to make a call to PC code to transfer data. Usually there are a few data instantiations/deinstantiations and transfers before and after each kernel launch. Typically, the system overheads of rescheduling processes and CUDA kernel launches are taking under a millisecond, nevertheless, if an application is designed to perform thousands of PCIe input/output operations or kernel launches per second, it is a better practice to initiate overhead at the beginning [28].

- If the only bottleneck in the system is low computational speed of GPU, then it makes sense to keep writing a program on GPU and simply upgrade to the better GPU when possible. Maybe multiple GPUs are required to achieve appropriate speed up of the heavy computational program [28].
- If the bottleneck is in bandwidth, then it is worth to check which bus is limiting the bandwidth. Maybe less data should be passed through the bottleneck, or we could try to make it wider.

There is a very helpful profiler tool provided by NVIDIA. It can be downloaded from NVIDIA's CUDA web page. The CUDA profiler tool consists from two parts. One part of it monitors the GPU performance right on the GPU. It records the launch of certain operations and logs time of data-transfers between host and GPU, as well as times of the launch and the end of kernel operations. The profiler can provide us with the total number of memory read and write operations. The second part of a profiler runs on a host computer. It monitors the quality of logging of the first part (which runs on GPU), holds the results which GPU profiler part transfers to PC, and presents this information to us. The profiler is a good tool to get a deeper understanding of what is going on with the program, how the GPU is performing and to see where the bottlenecks are.

2.3.3 GP using GPUs

Evolutionary programming overall and genetic programming in particular, is known to be very adoptable to parallelization. This is because the evaluation of every individual is an independent process and can be run simultaneously for multiple individuals. We can parallelize genetic program even with a CPU-threaded parallelism, but there are some drawbacks to it, such as monitoring all the threads and syncing them. Moreover, the number of threads which can be used is much smaller compared to GPUs. To get the most power and speed up from GPUs, we should run thousands of parallel threads. That is due to the wait-times to transfer data to and from GPU. It should be noted that moving data within the GPU can be even more expensive task than computing the data after it arrived from host.

Usually GP uses GPUs to run an evolved individual to evaluate it and assign a fitness score. The selection and all genetic operations such as crossover and mutation, selection of the fittest individual and generating the individuals are made on a host computer. Even though this is not a requirement, since all these operations can be ported to be run on GPUs, it has been shown that it will not provide a big speed up. That is due to the fact that the most computationally heavy part of evolutionary algorithm is the fitness evaluation [27].

We should always be aware of thread divergence. It may happen when a program has if statements or switch cases. In this case the threads may take different branches of the program and they will be locked up until the statement from other thread will be calculated. Whenever the values of all branches are calculated and they are returned to the thread which waits for these values, it will be resumed and other thread will run different calculations, and thus parallelism will continue.

2.3.4 CGP using GPUs

Although, there have been multiple implementations of CGP with GPUs, it is still relatively a new approach to speed up calculations. Mostly it has been used in bioinformatics applications, and because of this, its development has been limited.

Harding and Banzhaf [18] were the first to run CGP programs on GPUs. They used Microsoft's tools based on C# language to compile CGP genotypes into snippets and run them on a GPU.

Chitty [10] used NVIDIA's tools instead of Microsoft's ones [27]. Since 2007, there were CGP implementations written using different programming languages and on different OSs. Regression and Boolean problems were the first ones to be implemented on GPUs and they have shown the acceleration of the calculations, comparing it to the CPU based implementations. The paper used the Microsoft Accelerator framework, which is a .Net library for the GPU [20]. In 2011, Harding and Banzhaf used CUDA.net library to program CGP to use the NVIDIA GPUs [35]. They faced challenges for example, the CUDA compiler did not process long expressions, and functions with too many input variables causes the compiler to fail. Workarounds for these issues were eventually found [35].

Chapter 3

CGP implementation of NPR

Our CGP implementation of NPR is based on Baniasadi's work [5]. She implemented NPR with tree-based GP. We are using CGP. On the one hand, because we changed the structure of individuals (from tree-based to graph-based), we had to adjust multiple functions, terminals, and the way we retrieve results after genotype is evaluated. On the other hand, graph-based GP brought a viable speed-up to the overall performance of the GP runs and kept individuals concise. In this chapter we discuss how our CGP system is constructed, the parameters, NPR language and brushstroke application.

3.1 Architecture

The following are the steps our system executes:

1. Set the values for GP parameters such as mutation rate, number of generations and individuals inside of each generation.

2. Upload source image and colour target image. There 2 sizes of source and target images. One is small for faster processing by GP during training. Another larger image is used for generating a high-resolution image of evolved solutions at the end of the run.

3. We convert each pixel on an image into a data object, which we store in the system. This is used by CGP.

4. We process the target image to calculate static values of image characteristics, such as luminance, and other image filters.

5. We launch GP processing.

6. When results are ready, the system runs the best individual on the large target image.

Now, we will discuss each of the steps in more detail and depth.

3.2 Paint stroke application

The GP process is run in a loop, iterating through the population of individuals over many generations. The initial step in this loop process is initialization of the canvas. This could be done in 2 different ways, depending on the user:

- We copy original image to the canvas.
- We initialize canvas of the same dimensions as the image, and apply one solid RGB colour to it.

In this research, we used the second option, and chose a white colour as a background for it. This eliminates effects like “image” filters, and produces stronger NPR results.

The next step involves choosing which pixels on the canvas we are going to work with. To aid in the proper brushstroke application, we maintain a Boolean value for each pixel of the canvas, which represents whether this pixel has been coloured before or not. Initially, all values are false. Moreover, there are multiple ways our system could go through the canvas and apply brushstrokes to it.

1. Go through the whole canvas processing pixel-by-pixel This approach means that a lot of brushstrokes would cover each other, and this effect may be wanted or may be not, thus should be used with caution. It is also very time consuming.
2. Picking random pixels and checking if it was painted before [37]. If pixel has not been painted before then we process it in an ordinary fashion; otherwise we skip to the next loop iteration.
3. Picking random pixels and checking if it was painted before [37]. If pixel has not been painted before then we process it in an ordinary fashion; otherwise we go through the canvas and look for the next unpainted pixel, and assign it as the one we work with. It should be pointed out that steps 2 and 3 can be limited to finite number of tries or continue until all pixels have been selected. This gives 2 options for steps 2 and 3.

Through trial-and-error, we decided that we should use random selection with finite number of pixels to be coloured, and if we land on pixel which has been selected before, we should look for the next "available" pixel. The number of pixels to be selected is a user option. It is based on the dimensions of brushstrokes in ratio to the dimensions of the canvas, and personal preference to how many brushstrokes are wanted. We also use two different values, for small and large size canvases.

Every pixel which is selected is the centre of the brushstroke, which is going to be applied to the canvas. After it is picked, we calculate values for mean, standard deviation, minimum and maximum of *nxn* squares around pixel at the same (x, y) coordinates on original image. We calculate these values for 5 different squares: 5x5, 7x7, 9x9, 11x11, 13x13. We store them in variables which are to be used in the CGP graph as terminals (inputs). Along with these values, we also pass luminance values for the whole input image, ephemeral, and RGB values for the current pixel on original image as well as on canvas. Three more values which we pass to CGP as inputs are x and y coordinates of pixel, and the opacity on a corresponding brushstroke bitmap. Examples of brushstrokes bitmaps are in Fig. 3.1

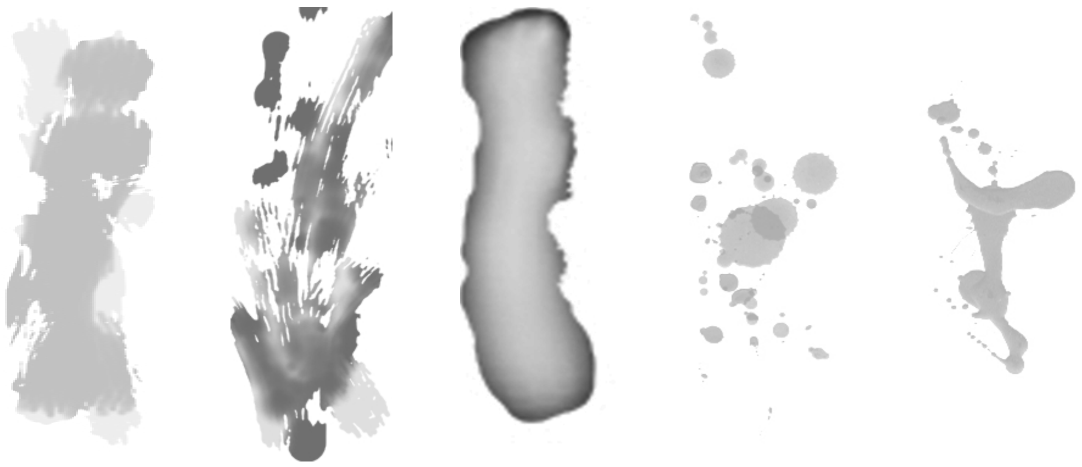


Figure 3.1: Examples of brushstroke bitmaps.

When the CGP individual is first executed, we receive outputs with values. We extract three values: scale, rotation and brushstroke bitmap image number. Even though there are more outputs, we ignore them initially, because they are used in pixel colouring (which we discuss below). We pass these parameters, along with the pixel data (both coordinates and RGB colour of the pixel) and CGP expressions, to a function which applies the brushstroke to the canvas. The paint function has the following declaration:

$$applyBrushStrokeFunction(bitmapNumber, scale, rotation, pixelData, CGPExpressions)$$

where bitmapNumber, scale, and rotation are the first, second and third arguments of CGP output, respectively. pixel variable is the pixel we are currently working with. This pixel is the center of the brushstroke which is going to be applied to the canvas.

The steps undertaken to apply a brush stroke are as follows. First, we determine and select the proper bitmap image of a brushstroke based on the first argument (bitmapNumber). Then we determine the proper positioning of the brushstroke on the canvas, taking into account scale (second argument) and rotation (third argument) values generated by CGP. Once the brushstroke is placed over the canvas, we re-evaluate the CGP graph for every pixel on the brushstroke bitmap. This involves taking the opacity value of the brushstroke pixel and assigning it to CGP input's array. We re-calculate spatial filter values and evaluate the graph again. This time, we read forth, fifth, and sixth output from evaluated outputs. They represent red, green, and blue values for pixel's colour respectively. In such way we receive RGB values for the pixel, and we colour it to those new values. This process repeats every pixel on brushstroke, and for all brushstrokes on a canvas.

Algorithm 1 Brush stroke application function

```
1: procedure APPLYBRUSHSTROKE(INT BITMAPNUMBER, FLOAT SCALE, FLOAT
   ROTATION, PIXELRECORD PIXEL)
2:     bitmap ← retrieve bitmap based on bitmapNumber
3:     scale the bitmap appropriately
4:     rotate the bitmap appropriately
5:     find (x, y) coordinates for all 4 corners of the rectangle of the bitmap
6:     for x = minimumX; x < maximumX; x++ do
7:         for y = minimumY; y < maximumY; y++ do
8:             if bitmap.opacity != 255 then
9:                 data ← collect data about pixel on (x, y) coordinate
10:                execute CGP graph on the data
11:                RGB ← outputs from CGP
12:                apply RGB to the pixel
```

After the canvas is painted according to the individuals' genotype, we evaluate it with multi-objective processing, and assign fitness scores to the individual.

At the end of the GP run, the fittest individual is selected from all generations, and is used to render a larger sized version of the target image. The result with the larger image is always a little different from what is seen in the smaller version for the same individual. However, the similarities in style can be seen as well (e.g. Fig 3.2).

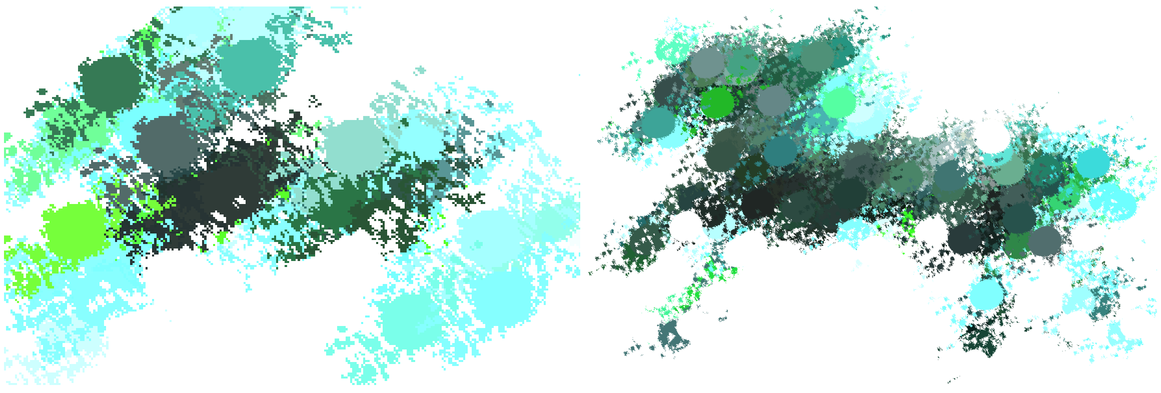


Figure 3.2: Two images generated by CGP program. (a) is a small canvas to which we applied fitness, and (b) is a bigger canvas done at the end of the run. See original image in Fig. 2.4(a)

3.3 Parameters

The ECJ [29] system is used in this research to implement GP. It is extended by the CGP framework [44] to support CGP processing.

Parameter	Value
Generations	5000
Population size	5
Max. number of nodes	100
Mutation probability	.04
$\mu + \lambda$	1+4

Table 3.1: Typical values which were used in setup for CGP run.

Table 3.1 shows typical parameters which were used for runs. The population size is kept small as discussed in Chapter 2, as well as mutation probability, and $\mu + \lambda$. The number of generations is a parameter which we changed a lot in different runs, and it ranged from 500 up to 7000. The number of nodes is the maximum number of nodes which can reside in the CGP genotype.

3.4 Language

We use float for the data type. Values of the data can have any single-precision float value. Table 3.2 and 3.3 show of all terminals and functions used. Overall we use 31 terminals and 22 functions. Terminals are CGP inputs, and functions are functional nodes. We also have 6 outputs. First three outputs are used for rotation, scale and bitmap number, other three outputs are used for RGB value of the pixel on a brushstroke bitmap (see section earlier).

Inputs (terminals)	Description
x, y	Coordinates of the pixel
opacity	Greyscale of the brushstroke's pixel
original RGB, canvas RGB	RGB of the pixel on input image and canvas image respectively
meanKxK	Mean of KxK square centered at the pixel with (x, y). K is one of 5, 7, 9, 11, 13
stdKxK	Standard deviation of KxK square centered at the pixel with (x, y). K is one of 5, 7, 9, 11, 13
minKxK	Minimum of KxK square centered at the pixel with (x, y). K is one of 5, 7, 9, 11, 13
maxKxK	Maximum of KxK square centered at the pixel with (x, y). K is one of 5, 7, 9, 11, 13
luminance	Sum of greyscale values from all pixels of input image
ERC	Ephemeral random float generated once per run. Range is (0, 1)

Table 3.2: Inputs (terminals).

The coordinates of the pixel to paint are (x, y) inputs. Opacity is how dark or bright a pixel is on a brushstroke bitmap. We also pass RGB values of this pixel from both original image and the canvas. We pass a canvas value, because if current pixel has been coloured before, it can have a different value from white. The ephemeral value is randomized between 0.0 and 1.0 at the start of the run, and stays the same till the end.

Finally, mean, standard deviation, minimum and maximum terminals are spatial filters on canvas image. They are calculated before an individual is evaluated. All the values are stored, separately for each dimension and provided as inputs when necessary. When we calculate mean, we convert image to black and white. We do it with the luminance formula provided above.

The function set is divided into two parts. There are simple mathematical and logical functions, such as +, -, sin, cos, min, abs, iflez, etc.. The iflez function means 'if a is less than b then do c else do d'.

There are also functions which work with the pixel's colour information. The last 7 functions in the Table 3.3 work with a pixel's intensity, opacity, brightness and darkness. The brightening, darkening and overlay blending functions demand 2 arguments, meanwhile the burning, dodging, normal blending and difference blending require 3 arguments. All of these functions manipulate their arguments through the number of mathematical formulas and return value in range of 0 and 255.

3.5 Fitness Evaluation

We use the following 5 objectives: deviation from normality (DFN), mean, standard deviation (SD), colour matching (CHISTQ), luminance (LUM). We also utilize normalized sum-of-ranks [7, 12] approach to score individuals. We use multi-objective fitness over the weighted sum technique, because in the case of the latter, weights would introduce bias to one or another objective and that would degrade quality of results and limit the exploratory power of evolution.

Functions	Description
+, -, *, /	Mathematical functions
neg	Negative function. Multiplying value by -1
abs	Absolute function
round	Rounding value
avg, min, max	Calculating average, minimum or maximum value between two values, respectively
sin, cos	Trigonometric functions
exp	Exponential function
log	Logarithmic function with base 10
iflez	Logical function. if a is less than b then do c else do d
Brightening	Brightening the pixel using intensity
Darkening	Darkening the pixel using intensity
Burning	Burning darkens a pixel by blending two layers of pixels
Dodging	Dodging lightens a pixel by blending two layers of pixels
Normal blend	Blending two layers of pixels with normal blend mode
Difference blend	Blending two layers of pixels with difference blend mode
Overlay blend	Blending two layers of pixels with overlay blend mode

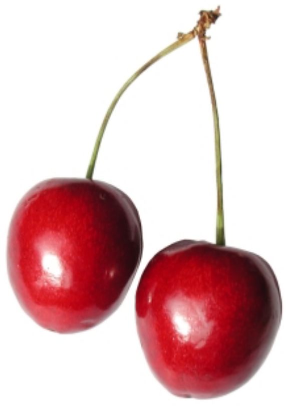
Table 3.3: Functions.

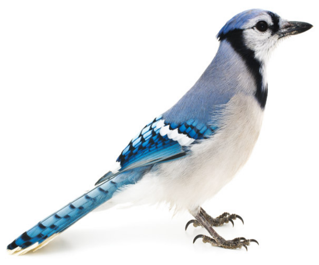
Chapter 4

Experiments

This chapter gives examples of the CGP NPR system in use. Results are presented along the discussion of what lead CGP to produce these images. The chapter examines some of the differences between CGP and tree-based evolution. We also briefly experimented with two different evolution strategies ($\mu + \lambda$ and population-based search), and compared their results. For detailed examination of fitness and NPR, see [5]. Note that we are not focused on rigorous statistical analyses in these experiments.

4.1 CGP Results


(a)


(b)

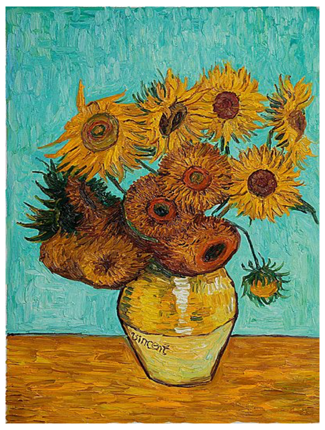

(c)

Figure 4.1: These are the original input images that have been used in experiments.

Implementation of NPR with CGP showed us that the execution of NPR effects with graphs instead of trees leads to different results. For detailed results and study of NPR with tree-based GP refer to [5]; it compares runs with different parameters,

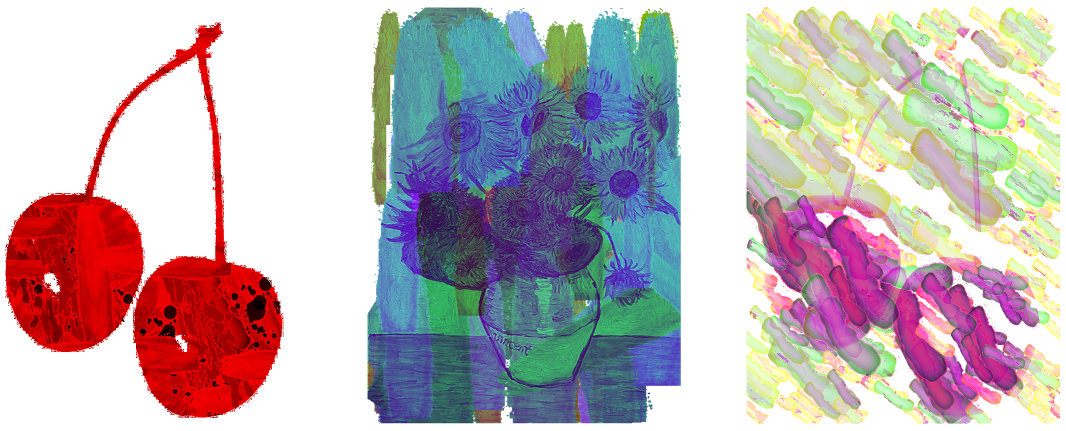


Figure 4.2: These are some of the interesting results we received from our experiments with CGP.

language subsets, and fitness formulas. Our experiments are focused on identifying how CGP results are different from tree-based GP results. Fig. 4.1 shows original input images that have been used for experiments throughout thesis. See a selection of interesting examples of the CGP results in Fig. 4.2

The biggest difference between the CGP and tree-based [5] implementations is that CGP has smaller and more concise expressions. In the tree-based system the paint function could be executed multiple times per paint stroke evaluation of the tree. With CGP, only one paint stroke is applied. CGP keeps expressions small, by encoding all required parameters for a single brushstroke application. For example, here are rendering expressions which we would typically see in CGP:

- R = nBld (round redInput) (neg greenCanvas) min9x9
- G = min std7x7 (round blueInput)
- B = + max5x5 (* redCanvas mean11x11)

All of these are small, and use very little inputs and functions. But the expressions do change between generations. They do not change a lot in structure (unlike trees), but they change often. Usually, either one function or input gets switched to some other and that influences the outcome of each generation. It is possible to see bigger expressions at early generations, but they usually reduce in size.

It is also interesting how CGP tries to satisfy multiple fitness objectives at the same time. It is important to note, that we are using 5 objectives with the sum-of-ranks fitness evaluation. The 5 objectives can be in conflict with each other. This means that when an individual has a good score for one objective, some other score becomes worse. For example, LUM and DFN are always in conflict; LUM tries to match the images perfectly, but DFN tries to apply brushstrokes, which blurs the shape. The CGP search will fluctuate in terms of which objective(s) are optimized at different times in the run (see Fig. 4.3). This effect is not as noticeable in tree-based many-objective NPR [5], where there is a more steady convergence of objectives during a run.

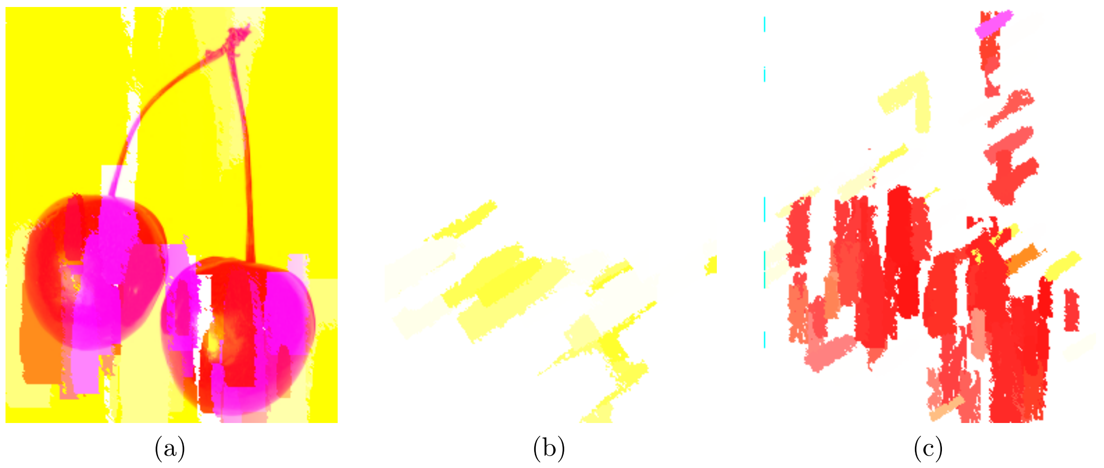


Figure 4.3: Examples from a single run. (a) Individual from generation 3386 has very high LUM objective match. (b) Individual from generation 3387 starts to satisfy a different objective. (c) Individual from generation 3786 has better colour match (CHISTQ), more visible brushstrokes (DFN), and a legible shape of cherries (LUM), thus satisfying more than one objective compared to (a) and (b).

CGP uses $\mu + \lambda$ evolution, which means it does not use tournament selection and crossover (see Table 3.1 for parameters used in runs). Thus, it favors exploration, and not strong convergence as in tree-based GP. The CGP system is therefore more exploratory, which is appropriate for an evolutionary art application. The performance graphs in Fig. 4.4 to Fig. 4.6, represent averages of each objective's scores. Fig. 4.4 shows average values per generation in one run. Fig. 4.5 shows average scores among 7 runs with 800 generations averaged by 10 generations. The graph in Fig. 4.6 shows all 5 objectives in one graph, for easier comparison, but all scores are normalized from previous 5 graphs, so they fit the scale of 1. 800 generations is not enough to obtain optimized results, but it is viable to illustrate how CGP acts in terms of convergence and many-objective fitness. We can observe that some objectives are getting higher scores (higher scores mean worse performance), but others are improving at the same time. It clearly shows the tradeoff of how CGP tries to satisfy one objective, but at the same time, fitness for the other objective becomes worse. The system will try to optimize fitness scores for all objectives in the long term, and that is why we require at least 5000 generations.

Another important point we can see from the graphs is how scores are chaotic in CGP with many-objective fitness. This is due to how mutation produce constant change between individuals.

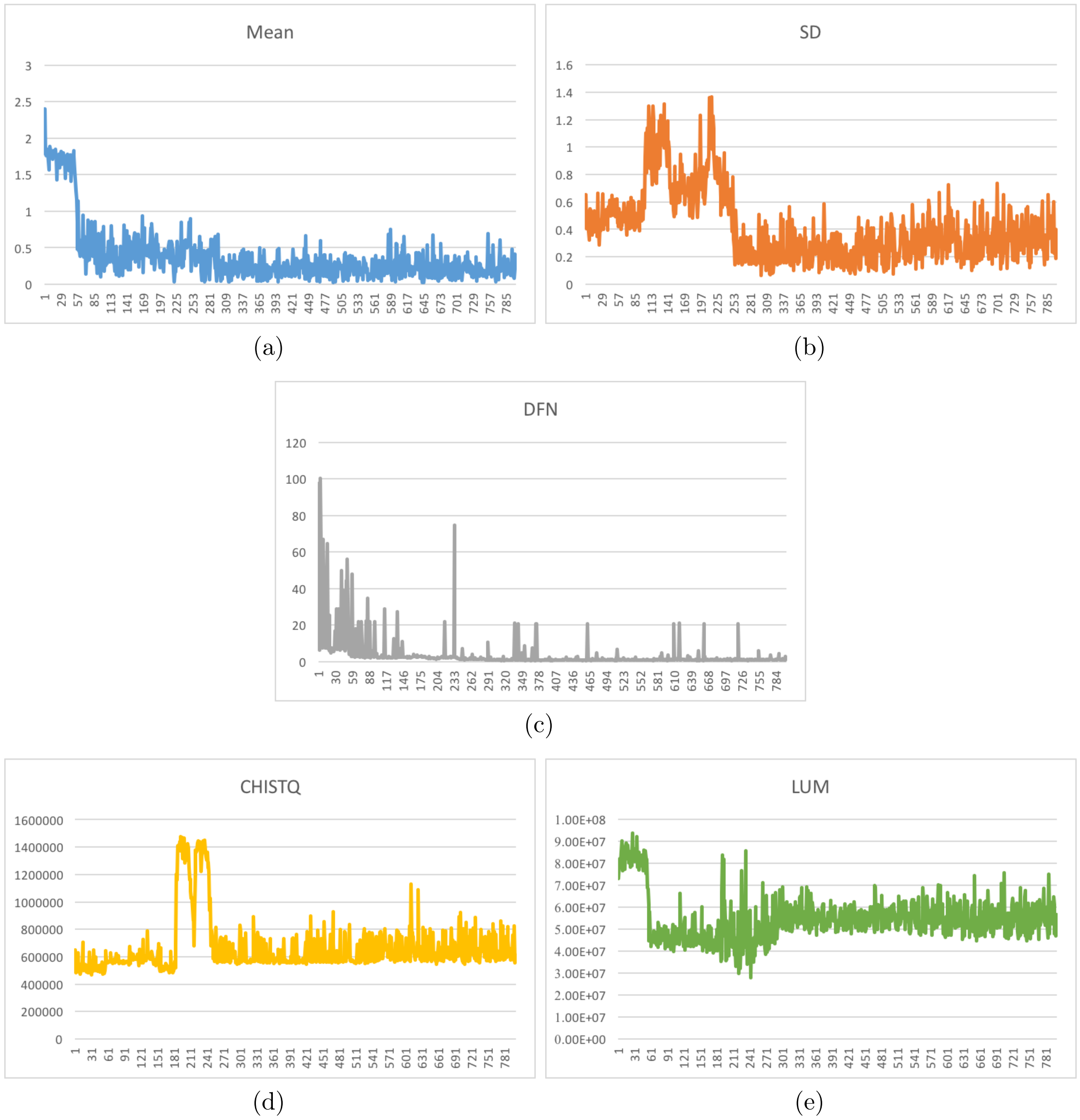
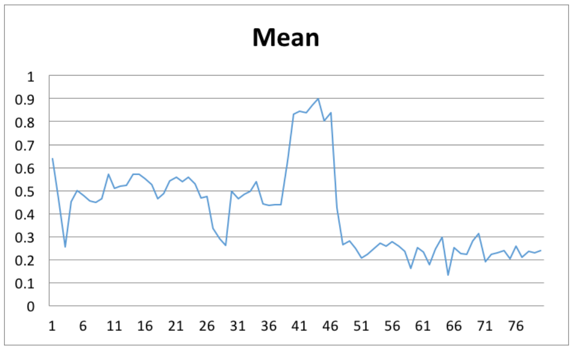
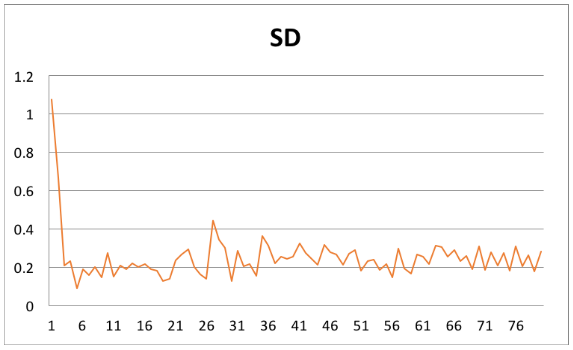


Figure 4.4: Mean, SD, DFN, CHISTQ and LUM scores in 1 run with 800 generations and averaged per generation.

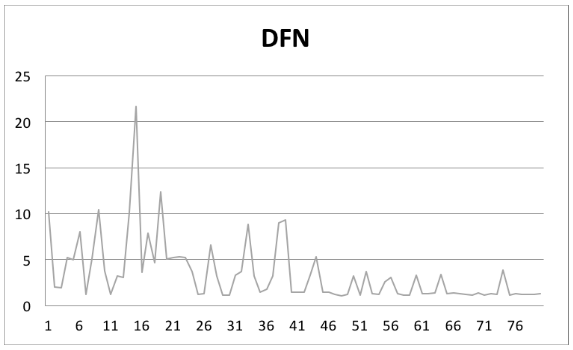
This seems to be a characteristic of $\mu + \lambda$ evolution on many-objective optimization problems using the sum-of-ranks scoring. Because the CGP search is frequently changing the objective which is being optimized, the results of the CGP change



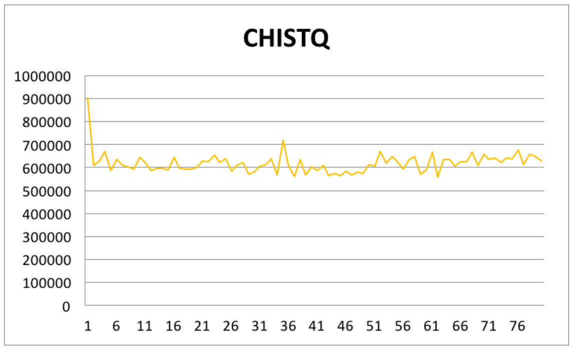
(a)



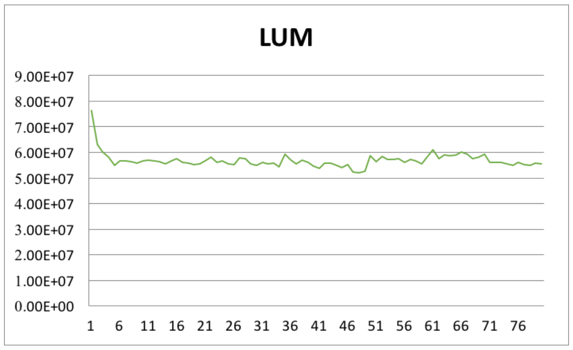
(b)



(c)



(d)



(e)

Figure 4.5: Mean, SD, DFN, CHISTQ and LUM scores in 7 runs with 800 generations each and averaged per every 10 generations.

dramatically through the run. The sum-of-ranks is unstable with a population of 5 individuals, and it is difficult to satisfy all scores simultaneously. There is more study required to examine whether the population is too small which leads to more chaos throughout the run. Moreover, there is more study required to identify if sum-of-ranks is a good optimization tool with $\mu + \lambda$ evolution strategy, even though it is good for art exploration. It also means that the best individual of the run is not necessarily the best NPR image from an aesthetic or subjective standpoint. This means that the best ranked individual from all generations is not necessarily the most desirable

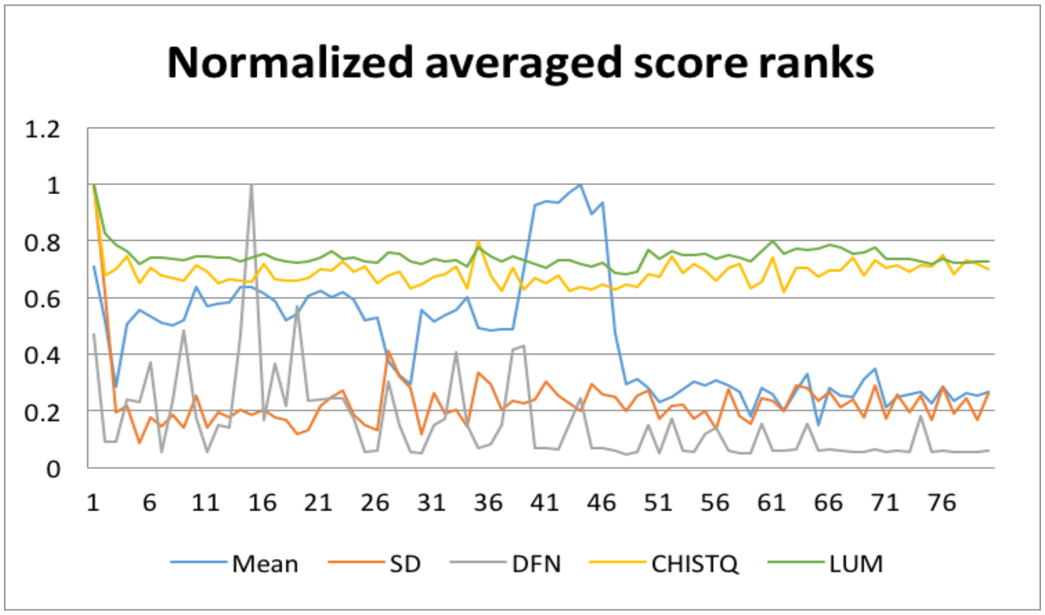


Figure 4.6: The graph with scores for all 5 objectives, averaged on 7 runs with 800 generations, then averaged over a moving window of 10 generations, and normalized so they all appear in one scale from 0 to 1.

image (see Fig. 4.7 and Fig. 4.8). Our preferred strategy was to examine solutions throughout the run. The system could greatly benefit from an interactive GUI in this case, described in Chapter 6).

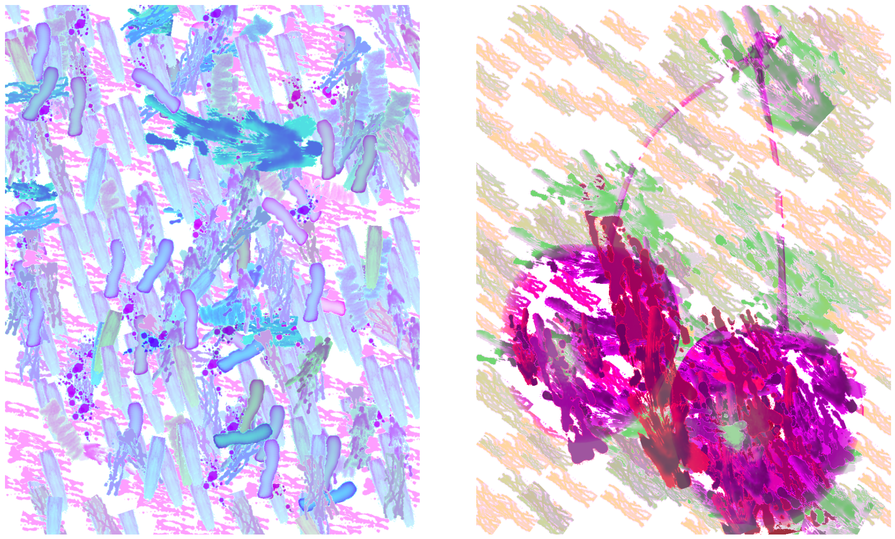
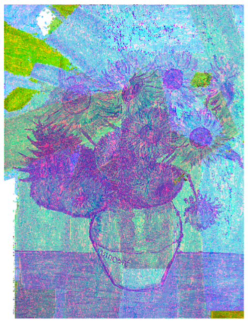
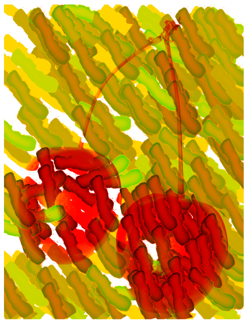
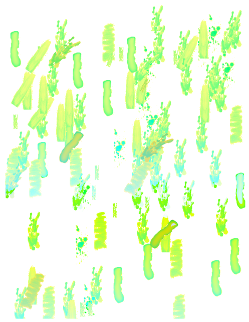
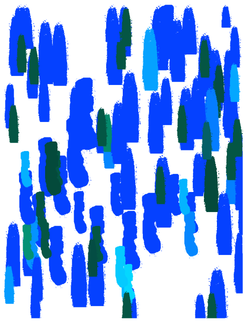


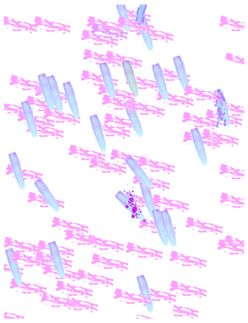
Figure 4.7: (a) The best individual out of all generations, but not the most aesthetically pleasing. (b) Not the fittest individual, but arguably a preferable result in which the shape of a cherry is clearly visible.

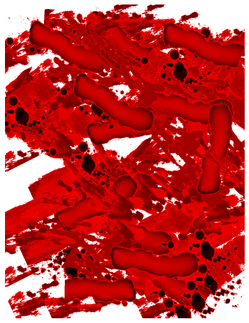

(a)

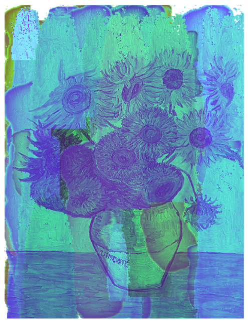

(b)

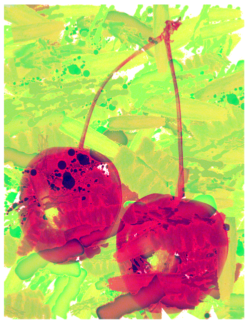

(c)

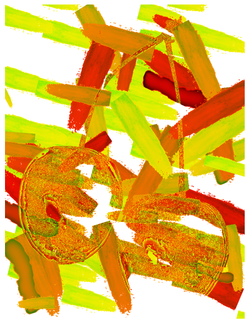

(d)

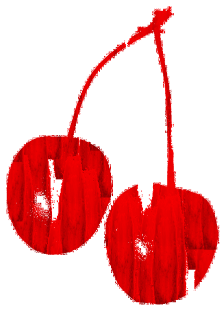

(e)

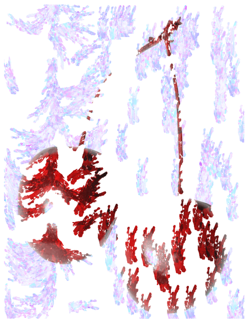

(f)


(g)


(h)


(i)


(j)


(k)

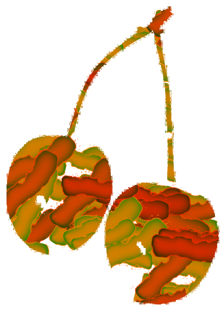
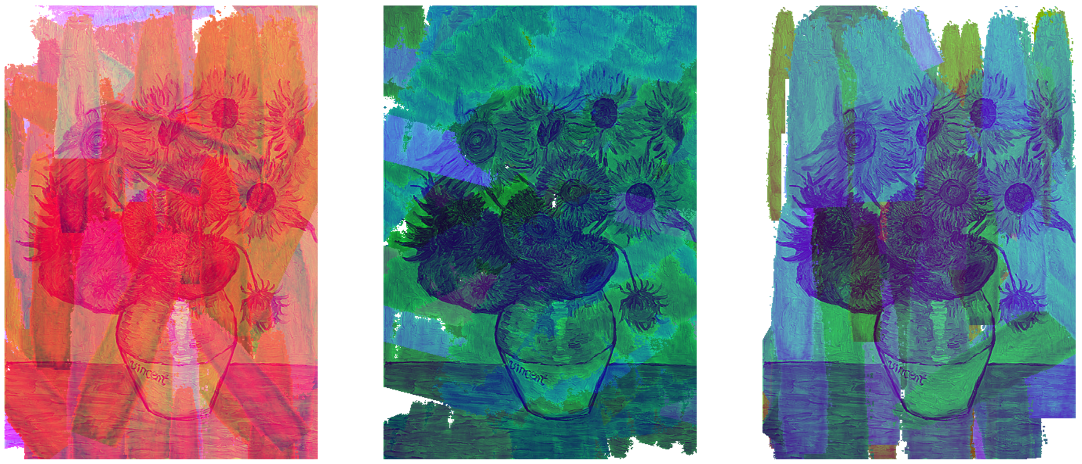
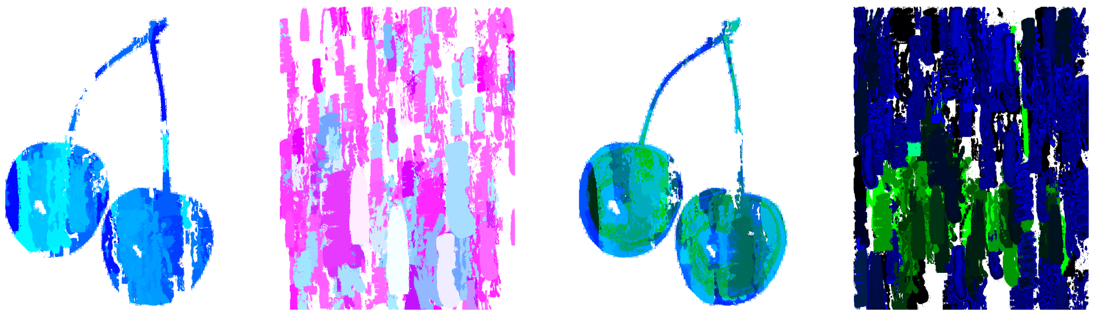

(l)

Figure 4.8: (a)-(f) are 6 images which show the best individuals selected by the system out of 5000 generations from 6 different runs. (g)-(l) are 6 hand picked images by the user from the same 6 runs. (g) is from generation 1753, (h) - 1369, (i) - 3322, (j) - 1200, (k) - 3884, (l) - 40.

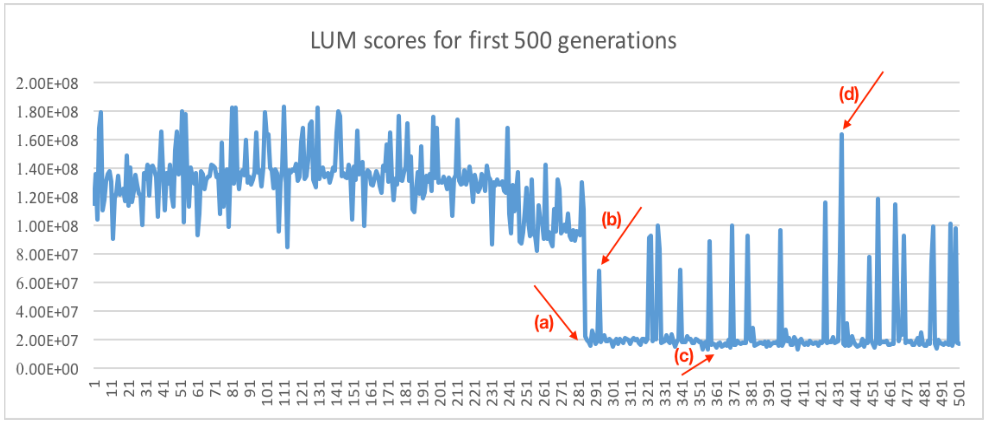


(a) (b) (c)

Figure 4.9: Results from the run with images with a non-white background.



(a) (b) (c) (d)



(e)

Figure 4.10: (a) is the first individual (285) in the run which has a shape of the cherries visible. The graph (e) shows how fitness improved once the shape became visible. (b) is the later individual (293) in which the shape of the object is not visible and the graph (e) shows a spike in the fitness. Same goes with (c) (362) and (d) (433), where the former one has the visible shape again, and the later one does not have cherries on it. The correspondent drops and spikes are shown in the graph (e).

In Fig. 4.2(a, b), and Fig. 4.3(a), notice how precisely CGP preserves the content of the target image. It still applies brushstrokes to it, but it does not always draw them outside of the shape. The luminance objective is responsible for this result, because evolved expressions can easily exploit the white background of target images, in order to optimize LUM scores (see Fig. 4.9 for examples of images without white backgrounds). In these cases, the system still tries to preserve details from the target image, but brushstrokes are applied everywhere else as well. It becomes harder for the system to identify what is background and what is the main object, and thus CGP is having difficulty preserving the target image. To show how LUM is important to preserve details from the target image, consider four individuals from the different stages of the run (see Fig. 4.10). Examining the plot for the LUM objective, we can observe that when the shape is matched closely to the original, the LUM objective is satisfied better. After the program switches to satisfy a different objective, LUM scores are becoming worse, and the rendered image does not match the target image. Note that if LUM is removed, then target image details are unlikely to be preserved. We also tried two different experiments. One run used the LUM objective and another run did not use it. In the later experiment, the cherry shape is never preserved.

4.2 $\mu + \lambda$ compared to population-based search

We also briefly compared different evolutionary strategies used in CGP. We compared $\mu + \lambda$ and the conventional population-based evolutionary algorithm. In case of $\mu + \lambda$, we used $1 + 4$ evolutionary strategy, and only mutation with probability of 4%. Generations were kept to 5 individuals, and the total run used 5000 generations. With population-based search we used both crossover and mutation, and we set number of generations and individuals closer to what is used in tree-based GP: 500 individuals in the population and 60 generations. The results in these runs were different. Population-based evolution is more stable (see graphs in Fig. 4.14), and leads to uniformity among results, especially towards the end of the run. It does not switch between different objectives which it tries to satisfy, because fitness pressure is stronger, and thus the exploratory advantage of the CGP is gone. See example of results in Fig. 4.13. It is likely the case that the normalized sum of ranks we use for many-objective fitness is more stable with a large populations. On the other hand, it is more chaotic with a population of 5, as used in $\mu + \lambda$ search. But this chaos means it is also more exploratory, which is not a bad thing for an art application. Fig. 4.11 shows that individuals by the end of the run with population-based search look the same and has similar features, such as style and colour palette. Fig. 4.11, on the other hand, shows how $\mu + \lambda$ search keeps individuals diverse even in the latest generations. It also keeps trying different types of brushstrokes and styles.

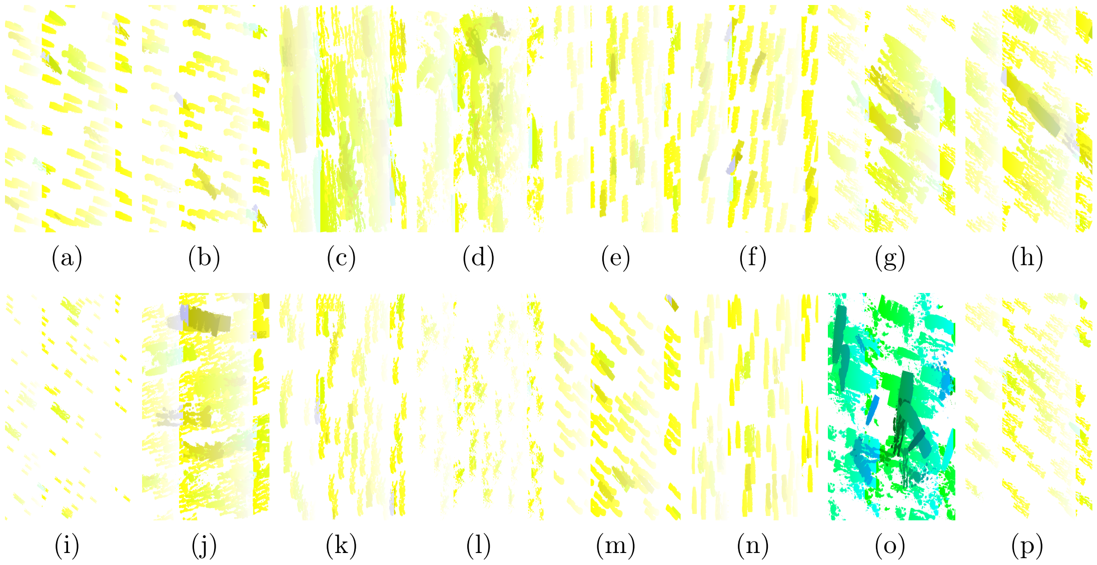


Figure 4.11: These images are the last 16 individuals from population based run.

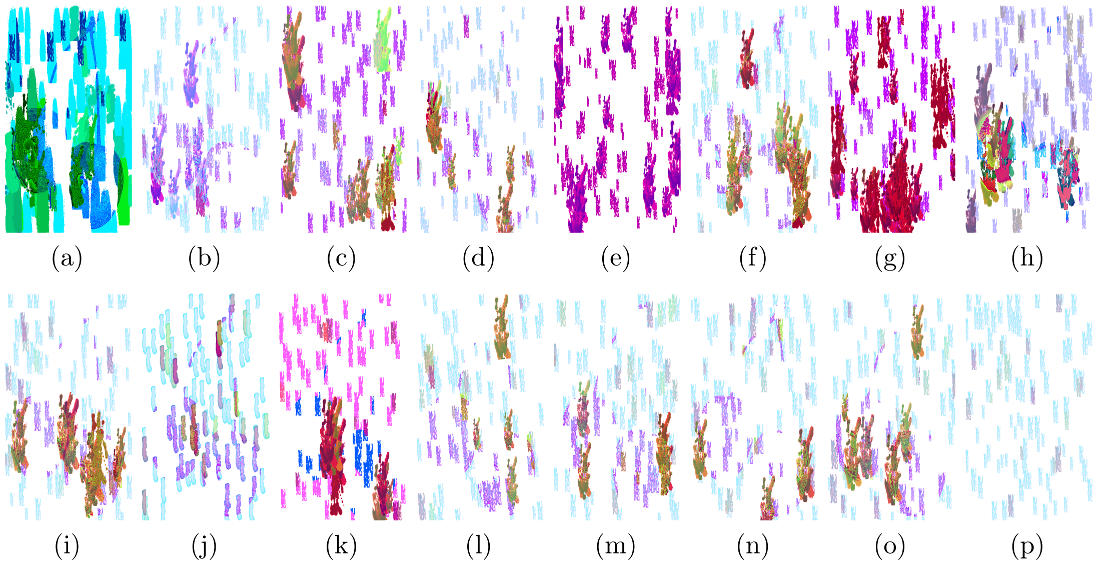


Figure 4.12: These images are the last 16 individuals from $\mu + \lambda$ run.

Dynamics of convergence which were observed during the experiments are possible due to factors such as:

- Small number of individuals in the population with $\mu + \lambda$ strategy, versus big populations in population-based strategy (more common in general GP)

- The sum-of-ranks effect is stronger on smaller generations then the big ones.
- Mutation and crossover operators versus only mutation reproduction operator with $\mu + \lambda$ strategy.

More studies are required to identify and understand the effect of all factors that have an impact on dynamics of convergence.

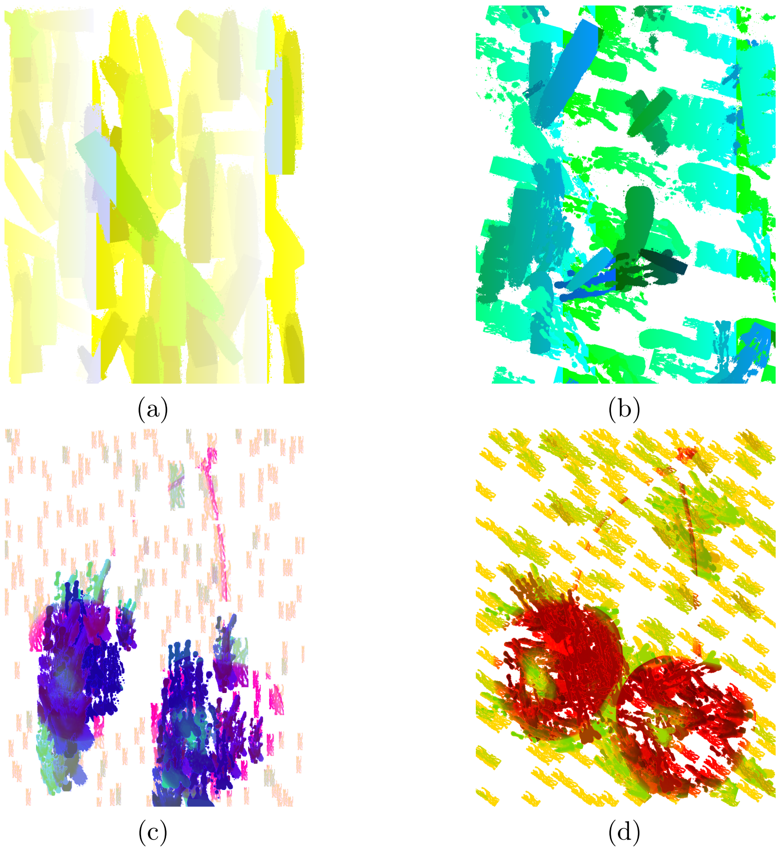


Figure 4.13: (a) & (b) are end of a run solutions from the population-based search strategy. (c) & (d) are from $\mu + \lambda$ $(1 + 4)$ strategy.

4.3 Conclusions

CGP system with many-objective fitness evaluation is a good fit for exploratory search of NPR effects. Due to mutation, and how CGP tries to satisfy multiple objectives at

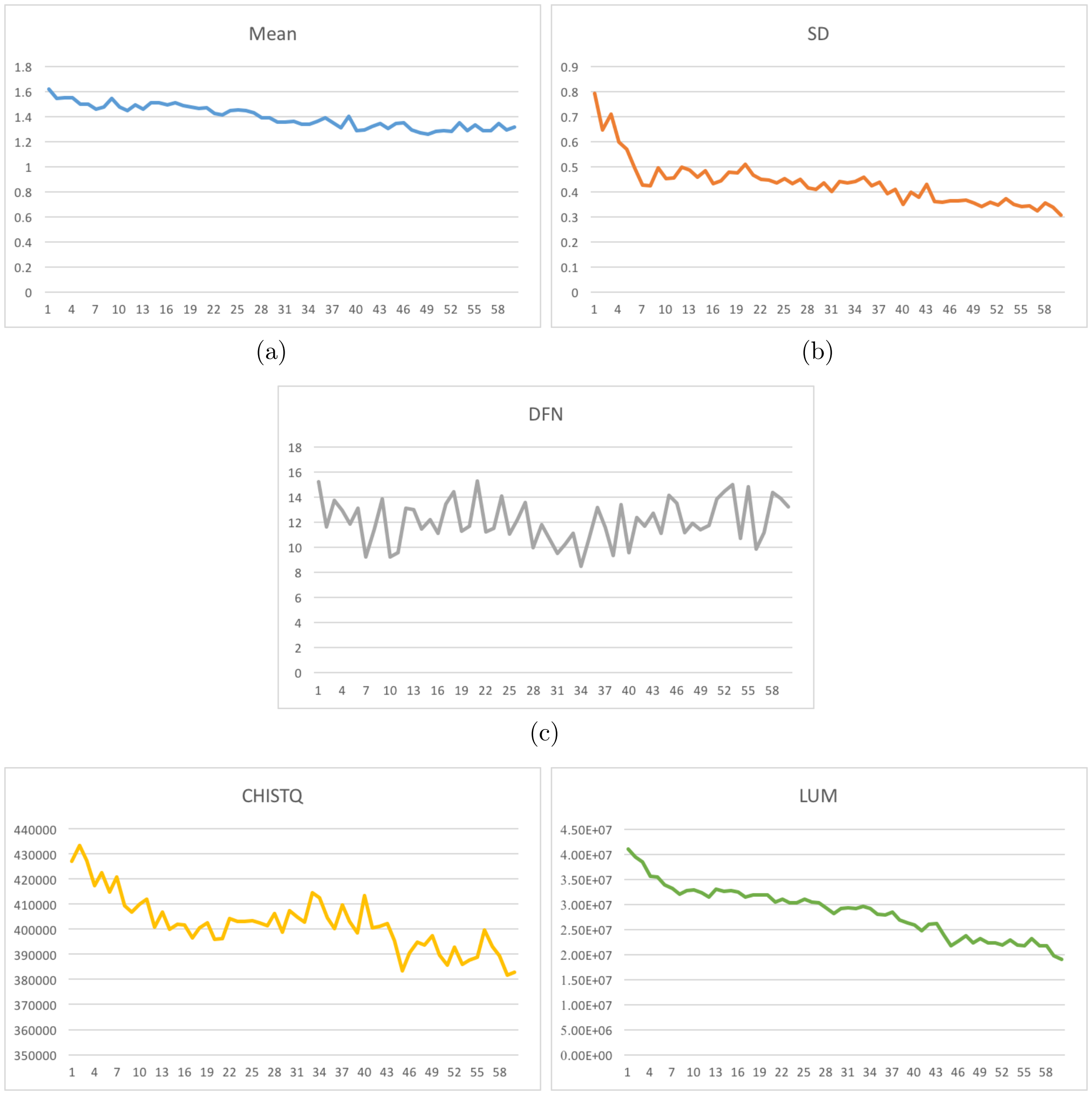


Figure 4.14: Mean, SD, DFN, CHISTQ and LUM scores in 1 run with 60 generations, averaged per generation with population based strategy.

the same time, the system generates a large variety of NPR results. However, more study is required to identify pros and cons of sum-of-ranks with $\mu + \lambda$, in problem outside of NPR applications. It should be noted that we were running each experiment for 10 runs due to time constraints, but this number of runs is inadequate for definitive conclusions such as statistical significance of the results. Running the system 30 times would be preferred for statistical analyses.

To conclude the chapter, Fig. 4.15 and Fig. 4.16 show two favorite results we obtained with the CGP NPR system.

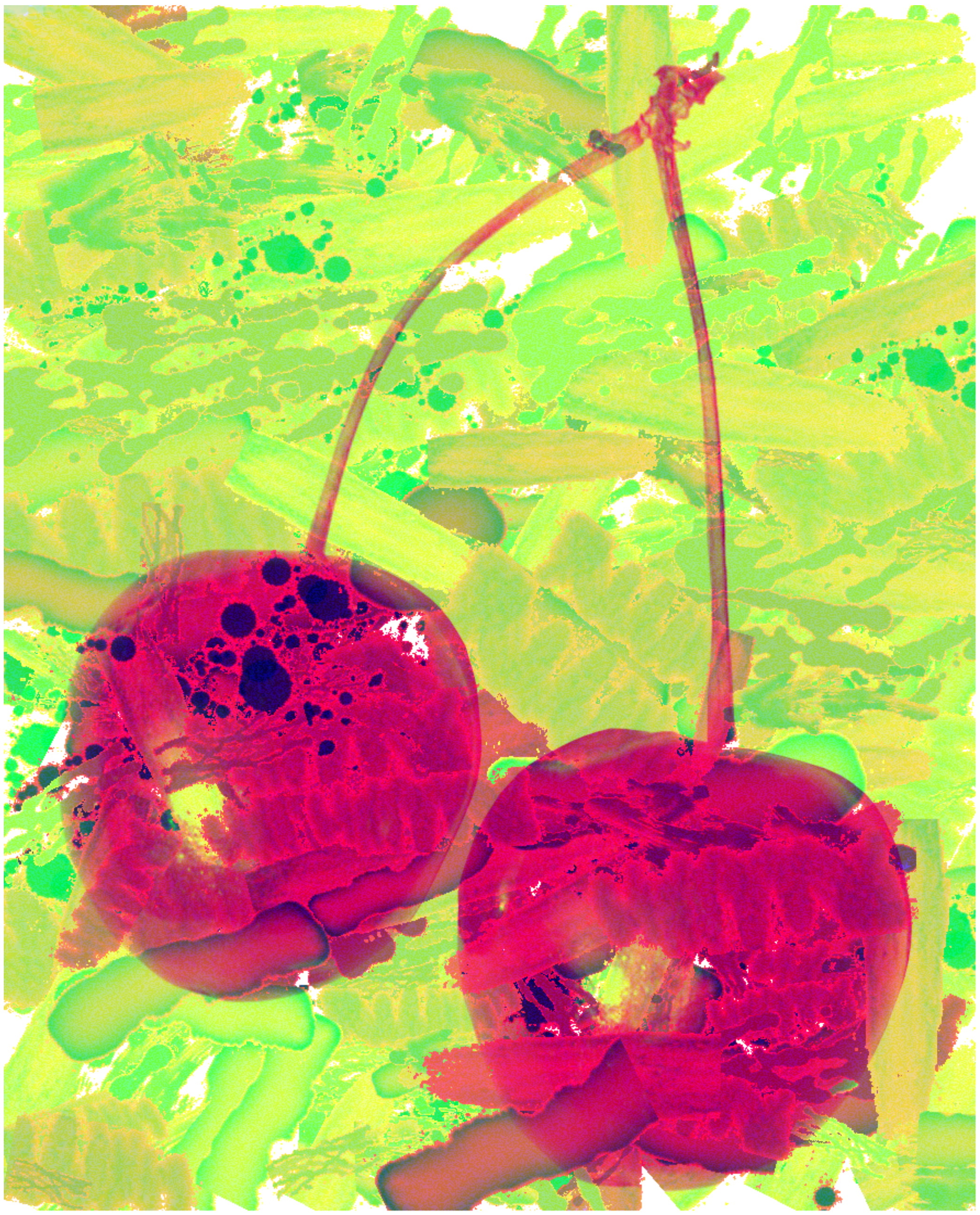


Figure 4.15: Cherries.

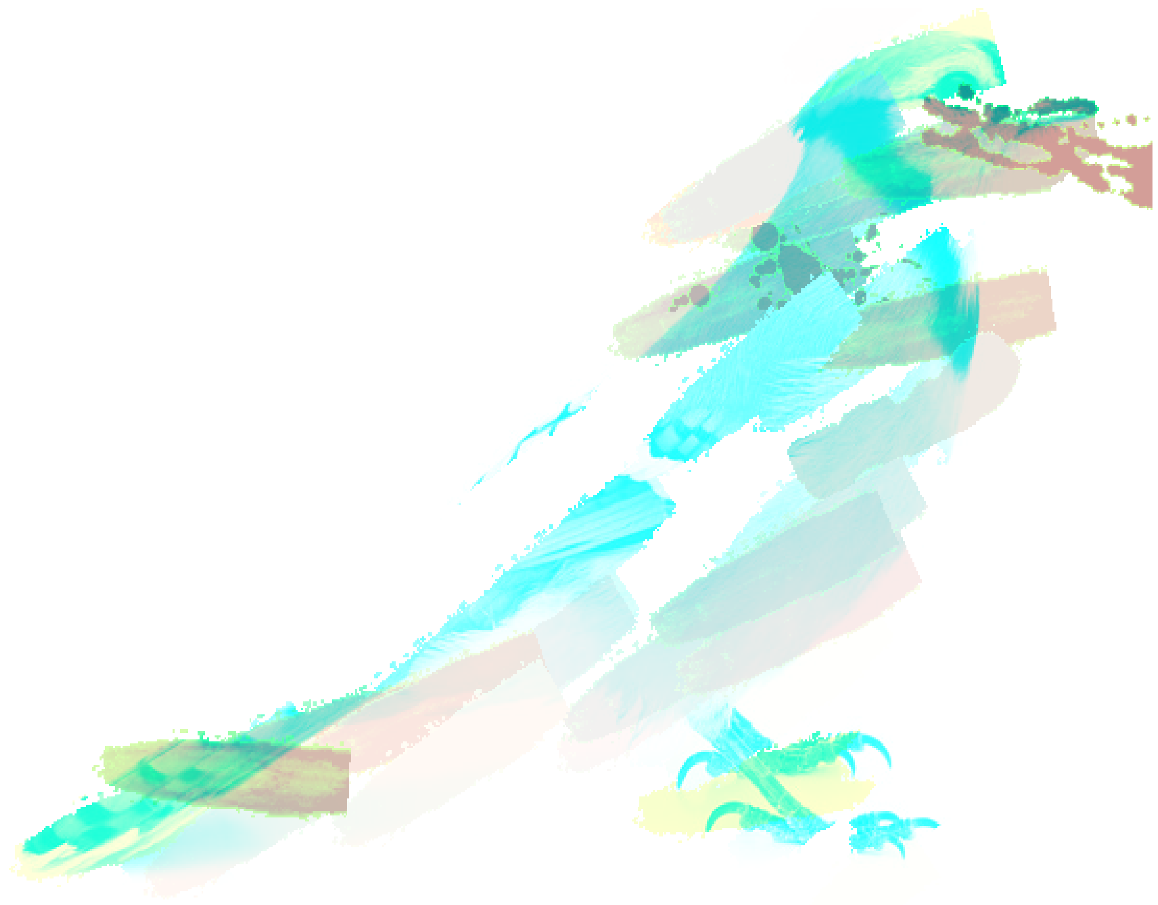


Figure 4.16: Blue Jay (CHISTQ target of a rainbow image is used; see Fig. 2.4(b)).

Chapter 5

GPU acceleration of CGP NPR

The main goal of implementing CGP with GPUs is to speed-up program execution, and in particular, canvas rendering. In Chapter 2, we discussed why GPUs are a good tool to accelerate genetic programs. In this chapter we will go into further details on how exactly we take advantage of GPUs and CUDA in our system, and what parts of the CGP program were enhanced with CUDA.

5.1 Use of GPU

We are not implementing the whole system on the GPU. Instead, we do the following steps for evaluating a CGP graph:

1. Run CGP on the CPU to obtain the bitmap number, scale, and rotation angle for the brushstroke we are going to apply.

2. We calculate the brushstroke position on the canvas.

3. We collect all the pixel data which will be covered by the brushstroke on the canvas.

4. We send all the pixel data and 3 colour rendering expressions to the GPU. These expressions will calculate the red, green, and blue components of all pixels' colours.

5. GPU calculates all RGB colours in parallel.

6. GPU returns array of calculated RGB colours to CPU.

7. CPU applies those colours to the canvas.

8. Process is repeated until termination criteria set by user is met (for example, all pixels coloured, some specific number of pixels is coloured, or some specific number of brushstrokes is applied).

The first and second steps in this process are described in Chapter 3. Thus, in this chapter, we will focus on the other steps and explain how we implemented them.

Algorithm 2 is pseudocode, which is similar to Algorithm 1 in Chapter 3.2, but includes changes to use the GPU. The changes can be seen on lines 8-10, and also the introduction of new function at line 11. The applyRGB function is the one which is going to be called after the GPU execution is done for all threads. Details are below.

Algorithm 2 Brush stroke application function

```
 1: procedure APPLYBRUSHSTROKE(BITMAPNUMBER, SCALE, ROTATION, PIXEL)
 2:     bitmap ← retrieve bitmap based on bitmapNumber
 3:     scale the bitmap appropriately
 4:     rotate the bitmap appropriately
 5:     find (x, y) coordinates for all 4 corners of the rectangle of the bitmap
 6:     for $x = minimumX; x < maximumX; x++$ do
 7:         for $y = minimumY; y < maximumY; y++$ do
 8:             aggregatedData ← collect data about pixel on (x, y) coordinate
 9:     add 3 CGP expressions to aggregatedData
10:     send aggregatedData to GPU to execute expressions in parallel
11: procedure APPLYRGB(RESULTFROMGPU)
12:     for $x = minimumX; x < maximumX; x++$ do
13:         for $y = minimumY; y < maximumY; y++$ do
14:             if $bitmap.opacity != 255$ then
15:                 RGB ← unwrap resultFromGPU into RGB colour
16:                 apply RGB to the pixel
```

5.1.1 Collecting data and sending it to GPU

As seen in Algorithm 2, we do not evaluate CGP individuals within for loops; rather, we collect the data about them into arrays. See Table 3.2 for the full list of terminals which are gathered and sent to a GPU. We instantiate arrays with the number of pixels which are going to be evaluated on GPU. That amount could be different for different brushstrokes. This is due to the fact that brushstrokes, after scale and rotation, will be of different sizes. Sometimes they may go out of the canvas's bounds, and we will not process the part of a brushstroke which is out of the bounds of the canvas. The number of pixels which we are going to process also corresponds to the number of threads we are going to request from the GPU to execute our task.

After all the pixel data is gathered into arrays, we need to get expressions from the CGP genotype and send them to the GPU. We take the CGP genotype, before is it converted into phenotype of any kind, and convert every sub-expression within it, to reverse Polish notation. The genotype represents terminals, functions, and connections between them as integer numbers. We go through the graph connections, from node to node, and retrieve the number which represents either a terminal or a function. We save that number into our translated expression which is going to be sent to the GPU. Terminals are saved with the same index, but incremented by 1. Functions, however, are saved as a total number of terminals and the index of the function, incremented by 1. One important thing to note here is that we add 0 to the end of the expression (we will discuss importance of it in the next section). This way we end up with three (R, G, B) expressions in reverse Polish notation. These expressions are ready to be sent to the GPU (see Fig. 5.1).

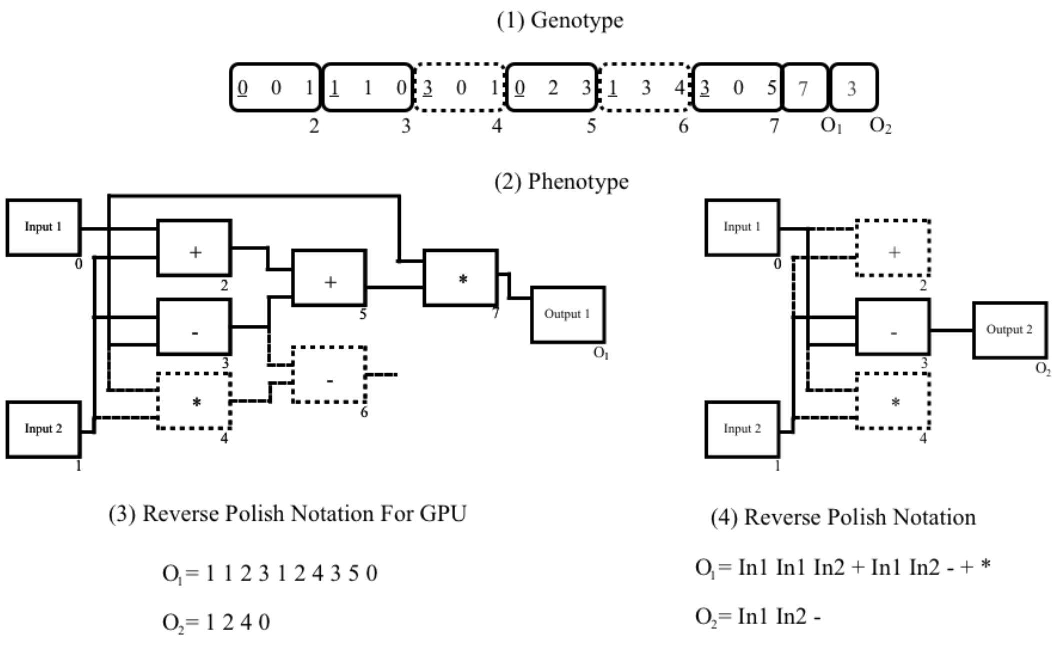


Figure 5.1: Conversions of genotype into phenotype and genotype into reverse Polish notation. Note how terminal and function indices are increased by 1 in (3). Also, note an extra 0 (exit symbol) at the end. (4) is reverse Polish notation with numbers replaced by terminals and functions, and without exit symbol.

5.1.2 GPU execution

We use jCuda framework [24] to use CUDA with Java. It is used only to port all code for setup and preparation of the GPU, and not the actual execution of a GPU code. The GPU code is written in CUDA on C++. Files with C++ code are saved with the .cu extension. Since we need to run them in Java, we convert them into .ptx files with nvcc compiler. jCuda is capable of running .ptx files from the Java environment.

The GPU code implements all functions and terminals from the CGP language. First, we calculate a unique id, which represents the thread and block we are in. This id is going to be an index of the data in the arrays we passed to the GPU earlier. Then we check if the opacity of the bitmap's pixel is not white. If it is white, we don't need to process this pixel, because it lies out of the brushstroke's bounds, or is a space within the brush shape itself.

We initialize the stack which will hold the terminals and results from function computations, which we will interpret in reverse Polish notation. We iterate through each expression in a while loop. The while loop has a switch condition which decides what action should be taken based on the integer number. If it is a terminal, it is going to be pushed into the stack. If it is a function, it is going to pop values from the stack for its arguments. It is important that implementation of the language on the device (GPU) is in-sync with CGP implementation on host (CPU), otherwise there are going to be two different results when program is run on CPU or on GPU. Because we added 0 to the end of the expression, we will go through the expression in while loop until we hit that 0, which represents the end of the expression. After this happens, the stack should hold only one final value. This value is the result of evaluation of the expression and represents one of reg, green or blue values of the pixel's colour.

RGB values are collected into the array, which is shared between all threads. We have our unique id of the thread, and we store values into array according to it. There is never going to be a situation when two threads write into the same array's indices. The array is returned to CPU once all threads are finished. The Algorithm 3 is pseudocode for the GPU interpreter which was based on Maghoumi's interpreter for texture generation [33].

5.1.3 Rendering the RGB data

Once the data is evaluated and the array with RGB values is returned to the CPU, we copy the calculated colours to the canvas. While we render the colours, we also record that the pixel has been coloured and should not be selected in the future.

Algorithm 3 GPU interpreter

```
1: procedure CALCULATERGBONGPU(EXPRESSIONS, DATA)
2:     pixelId ← unique id based on threads and blocks
3:     if data[pixelId].opacity == 255 then
4:         skip this pixel
5:     stack ← init empty stack
6:     for expression in expressions do
7:         l = 0
8:         while expression[l] != 0 do
9:             switch expression[l] do
10:                case 1:
11:                    stack.push(data[pixelId])
12:                case 2:
13:                    a ← stack.pop()
14:                    b ← stack.pop()
15:                    stack.push(b+a)
16:            l += 1
17:        RGB[pixelId] = stack.pop()
```

We continue the rendering process until the termination criteria set by user is met. We use the number of brushstrokes, which should be applied to the canvas, and the number of pixels, which should be coloured, as the final criteria to stop the rendering process for the current individual.

5.2 Discussion

Images generated with the GPU have identical characteristics to those from the CPU; the only visual difference is where the brushstrokes are placed on the canvas, due to the randomization of brush locations. The main aim of the GPU use is in speeding up the system, and not generating better quality images.

As seen in the graph on Fig. 5.2, the GPU implementation speeded up our system 6 to 10 times compared to the CPU implementation. With faster GPUs we can expect even further speed up. There are a couple of parameters which should be taken into account. First, more cores means more data can be computed in parallel. Second, wider bridges and better on-device memory management means faster data transfer, and more data can be transferred without loss in speed.

However, there is an opposite effect when using GPUs with small-sized images. It takes around an extra 30 minutes to 1 hour to run the system with the same

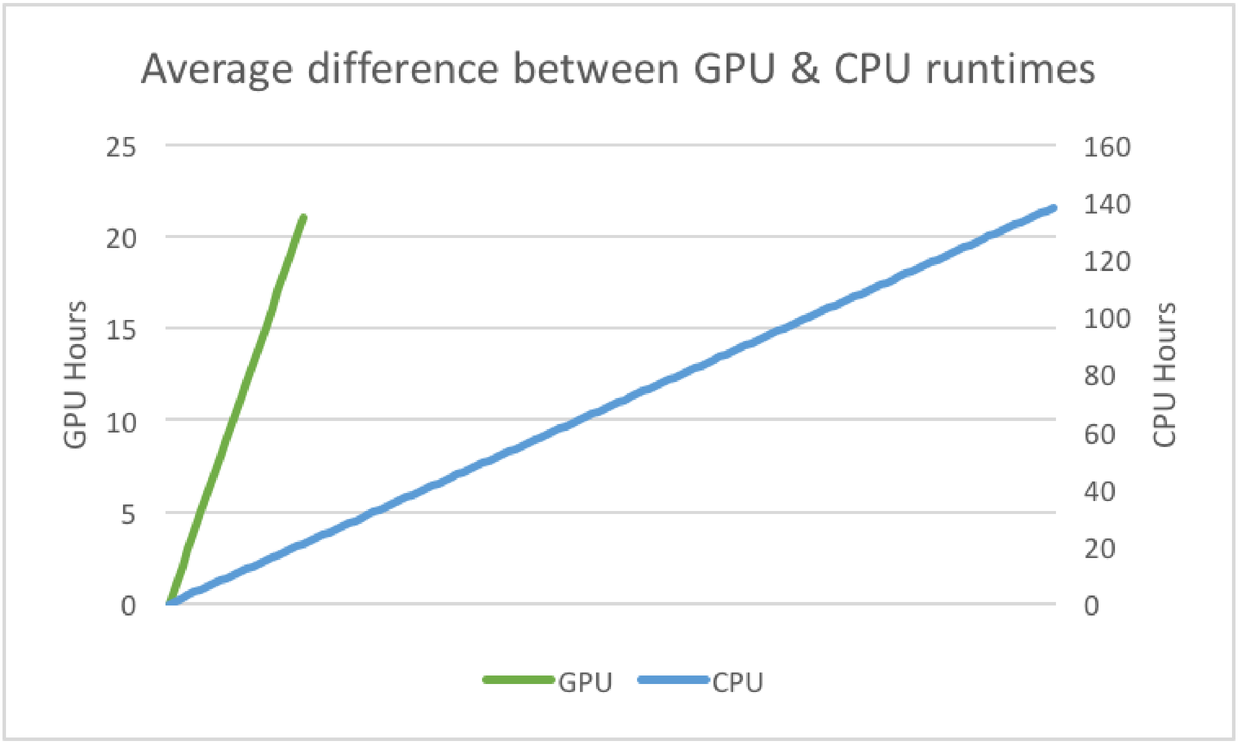


Figure 5.2: Estimated speed comparison: GPU versus CPU.

parameters and number of generations on the GPU than on the CPU (ie. 10% longer). This is due to a data transfer bottleneck (see Chapter 2). This issue can slow down execution if the GPU is not used properly. Small images have small brushstrokes applied to them, and that means that there is less data passed to the GPU during the rendering. When this happens, the CPU is more efficient in calculating this small amount of data. We do not get a big enough pay off from calculating the data in parallel, because of the overhead in transferring data to the GPU, and then back to the CPU, as well as collection of that data. This encourages us to use the GPU with big images if we want to see a speed up of the system.

Since usage of the GPU shows a sufficient improvement in speed, it has opened up multiple future advantages we could exploit. First, it is possible for us to run the system with big canvases for all individuals, and not only for the best ones. This is very important due to how CGP results are different from tree-based GP - the best final individual is not always the most aesthetically pleasing one (see Chapter 4). Second, we can use more varieties of brushstrokes. Because we have bigger images, we don't lose details of the brushstrokes. Finally, our DFN fitness became more accurate, due to the quality and colours which were rendered on the large canvases. These latter advantages follow from the usage of bigger canvases, and this is only practical due to use of the GPU. Otherwise, the runs would take too long to execute.

Chapter 6

Conclusion

We have successfully implemented NPR with CGP by porting it from tree-based GP implementation [5]. We simplified the NPR language without any loss of quality. Brushstroke application has been successfully implemented. All these elements can be run on CPU, as well as GPUs to accelerate the whole system. With GPUs we saw a 6 times speed-up, which allowed us to run the system on big images through the whole run. Overall, the system has been simplified and built as a core, so that it is easy to extend in the future.

Dozens of experiments were done, and as a result, we have seen a great potential in CGP's exploratory way to generate evolutionary art results. In comparison to tree-based GP [5], it uses simplified expressions and keeps individuals concise. CGP iterates through generations by mutating individuals, which provides agility in exploring multi-objective search spaces, and keeping populations diverse. The mutation-based evolution fits in well as a means of exploratory art generation.

6.1 Future work

There are many ways to improve the system we have created, which are discussed in detail below.

6.1.1 Further GPU acceleration

As discussed in Chapter 5, we accelerated the application of a brushstroke with GPUs. However, there are more components of the program which can be parallelized, leading to even greater improvements in speed.

We could apply all brushstrokes to the image at once. This approach is a very good fit for GPU acceleration. Currently, we calculate RGB colour of all pixels inside of one brushstroke, apply them at once, and move to the next brushstroke. But we could also calculate all brushstrokes at the same time. This could speed up the whole process significantly. However, there are a couple of pitfalls in this implementation which should be thoroughly thought through.

The first point is that it could be too much data to transfer between the CPU and GPU. This bottleneck could be solved by sending all the data for the source, target, and canvas images at the beginning of the run, and then calculate proper scale, rotation and positioning of a brushstroke on the GPU. In this way, we will not need to collect all the data about pixels on the CPU and send it to the GPU, because it is collected on the GPU right away. Moreover, after all the data is calculated, we could apply it to the canvas image on GPU, and afterwards send the final result to CPU for fitness evaluation.

A second issue is to properly handle the situation when two brushstrokes overlap on the same pixels. Because pixel colour can be changed after the first brushstroke is applied, it could render differently with the second brushstroke applied on top of it. When we do it brushstroke-by-brushstroke, it is no problem, but when all brushstrokes are applied at the same time, this problem will arise. As a possible solution, we could suggest monitoring if two brushstrokes overlap, and if necessary, pause those threads which require waiting for one brushstroke to be applied, and proceed when that brush is finished. Maybe indices of those threads could be calculated in advance, so that when the thread is fired, it already knows that it should check if the previous brushstroke has been applied and if it is safe for it to apply its changes.

Beside accelerating brushstroke application, we could parallelize population evaluation, so that all individuals in the population are evaluated at the same time. If this approach is combined with other techniques, the speed-up could be even higher. This seems to be a straight forward idea, as there is not too much data which must be transferred to the GPU. The hardest part of the implementation would be coordinating fitness evaluation. Potentially, we could ask the GPU to generate canvases for all individuals, and then these canvases are sent back to CPU and evaluated as they are now. Fitness evaluation is not a heavy task, and so we don't think it will benefit too much with GPU acceleration. There might be parts of fitness evaluation which could be parallelized such as spatial filter calculations. Other than that it is hard to justify using GPUs.

Finally, the speed up of the system with the GPU gives the future possibility to convert it into an interactive system. This could be a system where the user launches a CGP program accelerated with the GPU, and it applies brushstrokes to the canvas in real time. The user could see how they are applied and where, and also pause the system, do some manual adjustments, and resume it. This could in an interactive way to create NPR effects.

6.1.2 Fitness evaluation and solution image rendering

As discussed in Chapter 2, we were utilizing 5 objectives for fitness evaluation: mean, standard deviation, deviation from normality, luminance, and colour matching. Although they proved to be a good set of fitness formulas to assess individuals on being aesthetically pleasing, there is always a room for further exploration in this field. There might be additional objectives to evaluate shapes on an image, or to make colour matching more strict, so that it would force colours to be at the specific locations on the canvas. Also, new experiments might be done with different weights of fitness objectives, which would lead to different results than we obtained.

Because of the subjective nature of the problem, we could not always rely on the fitness formula to pick the individual which is indeed the best-looking painting out of all the run. Moreover, the system selects only one individual as the best one out of all the run. The user, however, might pick more than one solution as satisfactory. In future, we can add possibility for the user to select one specific genotype, and reload it into the system and re-apply it to high-definition canvases. This is especially important, if system uses small images for training and big canvas only for last final individual.

6.1.3 Advanced NPR ideas

The NPR techniques used in the CGP system could be enhanced with more advanced image processing and computer vision. For example, image segmentation and face/object detection. If we could find faces on an image, we could process them differently inside of the system. We could use different brushstrokes for the face area, or different parameters for scale, etc.. We could also assess these areas differently in the fitness evaluation. Detection of faces/objects is not the only way to apply this approach to the images, as we could also partition images based on colours, shapes, sizes, and distances of objects on them. Such technologies as segmentation or saliency are quite promising to consider. These features could be incorporated into the NPR system, resulting in higher-level NPR effects.

Bibliography

[1] D. Ashlock and J. Davidson. Texture synthesis with tandem genetic algorithms using nonparametric partially ordered markov models. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99*, pages 1157–1163, Washington, DC, USA, 1999. IEEE Press.

[2] Laurence Ashmore and Julian Francis Miller. Evolutionary art with cartesian genetic programming, 2004. "http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.571.9476", Retrieved on September 20, 2016.

[3] S. Baluja, D. Pomerleau, and T. Jochem. Towards automated artificial evolution for computer-generated images. *Connection Science*, 6(2/3):325–254, 1994.

[4] M. Baniasadi and B. Ross. Exploring non-photorealistic rendering with genetic programming. *Genetic Programming and Evolvable Machines*, 16(2), June 2015.

[5] Maryam Baniasadi. Exploring non-photorealistic rendering with genetic programming. Master's thesis, Brock University, 2014.

[6] P. Bentley and D.W. Corne. *Creative Evolutionary Systems.* Morgan Kaufmann, 2002.

[7] P.J. Bentley and J.P. Wakefield. Finding acceptable solutions in the pareto-optimal range using multiobjective genetic algorithms. In *Soft Computing in Engineering Design and Manufacturing.* Springer Verlag, 1997.

[8] S. Bergen and B. Ross. Aesthetic 3d model evolution. *Genetic Programming and Evolvable Machines*, 4(3):339–367, 2013.

[9] S. Bergen and B.J. Ross. Evolutionary art using summed multi-objective ranks. In *Genetic Programming - Theory and Practice VIII*, pages 227–244, Ann Arbor, MI, 2010. Springer.

[10] D. M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In *GECCO '07: Proceedings of the 9th annual conference on genetic and evolutionary computation.*, pages 1566–1573, London, England, UK, 2007. ACM Press.

[11] S. Colton, M. Valstar, and M. Pantic. Emotionally aware automated portrait painting demonstration. In *2008 8th IEEE International Conference on Automatic Face & Gesture Recognition*, pages 304–311, Amsterdam, Netherlands, 2008. IEEE Press.

[12] D. Corne and J. Knowles. Techniques for highly multiobjective optimisation: Some nondominated points are better than others. In *GECCO '07: Proceedings of the 9th annual conference on genetic and evolutionary computation.*, pages 773–780, London, England, UK, 2007. ACM Press.

[13] Richard Dawkins. *The Blind Watchmaker.* W. W. Norton & Company, Inc., New York, NY, USA, September 1986.

[14] A. Dorin. Aesthetic Fitness and Artificial Evolution for the Selection of Imagery from the Mythical Infinite Library. In *Advances in Artificial Life – Proc. 6th European Conference on Artificial Life*, pages 659–668. Springer-Verlag, 2001.

[15] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison Wesley, 1989.

[16] Bruce Gooch and Amy Gooch. *Non-Photorealistic Rendering.* A K Peters, 2001.

[17] D. Graham and C. Redies. Statistical regularities in art: relations with visual coding and perception. *Vision. Res.*, 50(16):1503–1509, 2010.

[18] S. Harding and W. Banzhaf. Fast genetic programming on gpus. In *Proceedings of the 10th European conference on genetic programming.*, pages 90–101, Valencia, Spain, April 11-13 2007.

[19] Simon Harding, Jürgen Leitner, and Jürgen Schmidhuber. Cartesian genetic programming for image processing. In *Genetic Programming Theory and Practice X*, pages 31–44, New York, NY, 2013. Springer New York.

[20] Simon Harding and Julian F. Miller. Cartesian genetic programming on the gpu. In *Massively Parallel Evolutionary Computation on GPGPUs*, pages 249–266, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[21] A. Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. In *SIGGRAPH '98 Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pages 453–460, New York, 1998.

[22] A. Hertzmann, C. Jacobs, N. Oliver, and B. Curless. Image analogies. In *SIGGRAPH '01 proceedings of the 28th annual conference on computer graphics and interactive techniques*, pages 327–340, New York, 2001.

[23] J.H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* University of Michigan Press, 1975.

[24] Marco Hutter. jcuda framework. `"http://www.jcuda.org"`, Retrieved on July 31, 2017.

[25] A.E.M. Ibrahim. Genshade: an evolutionary approach to automatic and interactive procedural texture generation. Phd thesis, Texas A&M University, December, 1998.

[26] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[27] W. B. Langdon. Graphics processing units and genetic programming: an overview. *Soft Computing*, 15(8):1657–1669, Aug 2011.

[28] W. B. Langdon. Performing with cuda. In *GECCO '11 Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 423–430, Dublin, Ireland, 2011. ACM Press.

[29] Sean Luke. Ecj framework. `"https://cs.gmu.edu/~eclab/projects/ecj/"`, Retrieved on June 18, 2016.

[30] P. Machado and A. Cardoso. Computing aesthetics. In *Proc. XIVth Brazilian Symposium on AI*, pages 239–249. Springer-Verlag, 1998.

[31] P. Machado and A. Cardoso. All the truth about nevar. *Applied Intelligence*, 16(2):101–118, 2002.

[32] Penousal Machado and Juan Romero. *The Art of Artificial Evolution.* Springer, 2008.

[33] Mehran Maghoumi. Real-time automatic object classification and tracking using genetic programming and nvidia cuda. Master's thesis, Brock University, 2014.

[34] Julian F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *GECCO'99 Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*, pages 1135–1142, Orlando, Florida, USA, 1999. Morgan Kaufmann.

[35] Julian F. Miller. *Cartesian Genetic Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[36] C. Neufeld, B. Ross, and W. Ralph. *The evolution of artistic filters, in The art of artificial evolution*. Springer, Berlin, 2008.

[37] Craig Neufeld. The evolution of artistic filters. Master's thesis, Brock University, 2005.

[38] Long Nguyen, Daniel Lang, Nico van Gessel, Anna K. Beike, Achim Menges, Ralf Reski, and Anita Roth-Nebelsick. *Evolutionary Processes as Models for Exploratory Design*, pages 295–318. Springer International Publishing, Cham, 2016.

[39] NVIDIA. Cuda enabled gpus. `"https://developer.nvidia.com/cuda-gpus"`, Retrieved on June 18, 2017.

[40] NVIDIA. Geforce gt 750m. `"https://www.geforce.com/hardware/notebook-gpus/geforce-gt-750m"`, Retrieved on July 15, 2017.

[41] NVIDIA. Kepler architecture. `"http://www.nvidia.com/object/nvidia-kepler.html"`, Retrieved on July 15, 2017.

[42] NVIDIA. Nvidia gpus. `"http://www.geforce.com/hardware"`, Retrieved on June 18, 2017.

[43] NVIDIA. Nvidia's next generation cuda compute architecture, 2014. `"http://la.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf"`, Retrieved on July 31, 2017.

[44] David Oranchak. Cgp framework. `"http://www.oranchak.com/cgp/doc/"`, Retrieved on June 18, 2016.

[45] Pawel Pomorski. Programming gpus with cuda. Presentation, 2016. `"http://ppomorsk.sharcnet.ca/summer_school_gpu/CUDA_summer_school_2015_central.pdf"`, Retrieved on June 20, 2016.

[46] W. Ralph. Painting the bell curve: The occurrence of the normal distribution in fine art. Unpublished, 2006.

[47] B. Ross, W. Ralph, and H. Zong. Evolutionary image synthesis using a model of aesthetics. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 3832–3839, Vancouver, B.C., Canada, 2006. IEEE Press.

[48] B.J. Ross and H. Zhu. Procedural texture evolution using multiobjective optimization. *New Generation Computing*, 22(3)(271-293), 2004.

[49] Jimmy Secretan, Nicholas Beato, David B. D'Ambrosio, Adelein Rodriguez, Adam Campbell, Jeremiah T. Folsom-Kovarik, and Kenneth O. Stanley. Picbreeder: A case study in collaborative evolutionary exploration of design space. *Evolutionary Computation*, 19(3):373–403, 2011. PMID: 20964537.

[50] M. Shiraishi and Y. Yamaguchi. An algorithm for automatic painterly rendering based on local source image approximation. In *Proceedings of NPAR 2000*, pages 53–58, Annecy, France, 2000. ACM.

[51] K. Sims. Interactive evolution of equations for procedural models. *The Visual Computer*, 9:466–476, 1993.

[52] J. Smith and S.-F. Chang. Visualseek: a fully automated content-based image query system. In *Proc. ACM-MM*, pages 87–98, 1996.

[53] B. Spehar, C.W.G. Clifford, B.R. Newell, and R.P. Taylor. Universal aesthetic of fractals. *Computer and Graphics*, 27:813–820, 2003.

[54] Thomas Strothotte and Stefan Schlechtweg. *Non-Photorealistic Computer Graphics; Modeling, Rendering, and Animation.* Morgan Kaufmann, 2002.

[55] S. Todd and W. Latham. *Evolutionary Art and Computers.* Academic Press, 1992.

[56] A.L. Wiens and B.J. Ross. Gentropy: Evolutionary 2d texture generation. *Computers and Graphics Journal*, 26(1):75–88, February 2002.

[57] Wikipedia. Greyscale. Wikipedia, 2017. "`https://en.wikipedia.org/wiki/Grayscale`", Retrieved on August 18, 2017.