Procedural Content Generation via Diversity Engines

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Paden Rumsey

Major Professor: Terence Soule, Ph.D.

Committee Members: Robert B. Heckendorn, Ph.D.; Barrie Robison, Ph.D.

Department Administrator: Terence Soule, Ph.D.

May 2020

Authorization to Submit Thesis

This thesis of Paden Rumsey, submitted for the degree of Master of Science with a Major in Computer Science and titled "Procedural Content Generation via Diversity Engines," has been reviewed in final form. Permission, as indicated by the signatures and dates below is now granted to submit final copies for the College of Graduate Studies for approval.

Major Professor:	Terence Soule, Ph.D.	Date
Committee Members:	Robert B. Heckendorn, Ph.D.	Date
	Barrie Robison, Ph.D.	Date
Department Chair:	Terence Soule, Ph.D.	Date

Abstract

Asset generation in video games and simulations is an expensive process that takes considerable resources to do well. Procedural Content Generation (PCG) is a technique that can be used to generate this content for video games and simulations. Although, there are certain flaws related to content quality and diversity that need to be addressed in order to utilize this technique generally. The Innovation Engine is a relatively new theoretical framework that is incrementally attempting to encapsulate human creativity by computational means. Optimizing techniques such as Genetic Algorithms (GAs) are inadequate when trying to accomplish goals related to creativity. A new software framework presented in this work, the Diversity Engine, acts as a practical application of the ideas found in Innovation Engines and shows procedurally generated flowers can be created, with mixed results, using a new Quality Diversity (QD) algorithm Centroidal Voronoi Tessalation MAP-Elites (CVT-MAP-Elites).

Acknowledgements

This work was the compilation of several years of research and investigation. I would first and foremost like to thank my major professor, Terry Soule, Ph.D. Without his guidance and support, this would not have been possible. I would also like to thank my committee, Robert B. Heckendorn, Ph.D. and Barrie Robison, Ph.D., for their unending patience and involvement. Credit also goes to Homaja Pydi Kumar for performing the initial investigation that led to this work. Others who have had an influence on the work herein include:

Robert Hiromoto, Ph.D., Min Xiang, Ph.D., and Don Scoffield

Dedication

To my friends: Zach, Mason, Ryan, and Chance.

A smooth sea never made a skilled sailor, but it was nice not to sail alone.

Table of Contents

List of Tables

List of Figures

List of Acronyms

AE	Autoencoders
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CPPN	Compositional Pattern Producing Networks
CRC	Computational Resources Core
CVT	Centroidal Voronoi Tessalations
CVT-MAP-Elites	Centroidal Voronoi Tessalation MAP-Elites
DeLeNoX	Deep Learning Novelty Explorer
DNN	Deep Neural Network
EA	Evolutionary Algorithm
EC	Evolutionary Computation
FINS	Feasible-Infeasible Novelty Search
GA	Genetic Algorithm
GAN	Generative Adversarial Network
ILSVRC	Imagenet Large Scale Visual Recognition Challenge
KD-tree	K-Dimensional tree
LSTM	Long-Short Term Memory
MAP-Elites	Multi-Dimensional Archive of Phenotypic Elites
MCTS	Monte Carlo Tree Search
ML	Machine Learning
MOO	Multi-Objective Optimization
MMFO	Multi-Modal Function Optimization
NEAT	NeuroEvolution of Augmenting Topologies
NSLC	Novelty-Search with Local Competition
PCG	Procedural Content Generation
PCGML	Procedural Content Generation via Machine Learning

QD	Quality Diversity
ReLU	Rectified Linear Units
RNN	Recurrent Neural Network
TRNG	Tina's Random Number Generator
YOLO	You Only Look Once

Chapter 1: Introduction

On October 26, 2018, *Red Dead Redemption II*, a video game developed by Rockstar Games was released to the public. The game was lauded by critics and players alike as one of the largest, most realistic worlds that has ever been developed in a video game [40]. However, this extraordinary game world is speculated to have come at huge monetary and labor costs. Conservative estimates state that the game cost $170 million over 8 years with some employees working 60-80 hour weeks [52][78]. While these estimates are pure speculation on the part of media analysts, it is no secret that crunch and poor work-life balance are commonplace in the video game industry. The expectations posed by modern AAA gaming have become so extreme that employees are suffering from burnout and mental health issues as a result of frequency and duration of hours worked [65][67][68]. Discussions of industry reform aside, there is clearly a financial and medical need for the development of tools that could provide a reduction in labor demands during the development of large-scale AAA games. These tools could also be of potential use to smaller "indie" developers that may not currently have the resources to create a product that is comparable to the AAA experience.

1.1 Objectives

There are two main objectives of this thesis. The first is to develop a lightweight, parallel framework derived from the ideas found in the Innovation Engine research project [35][48]. This new framework, called the Diversity Engine, is intended to be a pragmatic application of Innovation Engines that will help solve challenges related specifically to diversity in search-based PCG. The term Diversity Engine is derivative and intended to distinguish its narrower scope from the grander vision of Innovation Engines proper. The Diversity Engine will enable rapid prototyping and research in the domain of PCG, specifically creating assets using PCG. The goal of the framework is to primarily allow users to test various evaluation and generation methods with ease. The second objective of this thesis is to test the efficacy of generating PCG assets using a new QD technique, called CVT-MAP-Elites, that enables the generation of many high-performing solutions that possess a high number of feature, or behavior, descriptors [46]. These objectives are intended to help overcome a significant challenge in the PCG domain, the automatic generation of diverse, high-quality digital content.

1.2 Organization of Thesis

This thesis is organized into four parts, which are divided into their own respective subsections. Chapter 2 presents the basic backgrounds of all the fields that are utilized in the Diversity Engine or its subsequent experiments. Section 2.1 covers the basics of PCG, including search-based PCG, contemporary techniques, as well as challenges currently faced in the field. Section 2.2 covers the basics of Evolutionary Algorithms (EAs), Machine Learning (ML) as it relates to the field, and Quality Diversity (QD) algorithms including a core algorithm of this work, CVT-MAP-Elites. Section 2.3 addresses the very basics of Artificial Neural Networks (ANNs), popular deep learning techniques and architectures, and includes a discussion on the architecture of the model used in this work, Inception V3. Section 2.4 discusses the specifics of the perceptual hashing algorithms found in this work. Section 2.5 discusses the foundational work that lead to the creation of Innovation Engines as well as the techniques used in Innovation Engines proper. Section 2.6 covers alternative research being performed in the domains of computational creativity and PCG. Chapter 3 has two sections. The first, section 3.1, covers the high level design of the whole system and how it is intended to be used, and the second, section 3.2, covers more of the technical details for those interested in performance. Chapter 4 contains the experimental setup, results, and discussion when using the Diversity Engine, via CVT-MAP-Elites, to generate diverse populations of 2D and 3D flowers. Chapter 5 addresses the results of the experiments performed and the significance of the work performed here. It also elaborates on future challenges for the Diversity Engine and computational creativity.

Chapter 2: Background

2.1 Procedural Content Generation

Procedural Content Generation (PCG) is defined as the automatic creation of content (usually gaming related) by automatic, algorithmic techniques [81]. The usage of PCG in video games is by no means a recent phenomenon. In 1980, the ASCII-text, dungeon-crawling game *Rogue* was released. Instead of the game world being crafted by hand, the rooms were created with PCG. The room layout, which monsters resided in what room, as well as the treasures within were all decided algorithmically [82]. In the forty years since *Rogue* was released, PCG has made great leaps in the video game industry. Two contemporary titles to use the technique are *Minecraft* and *No Man's Sky*. In both titles, the worlds are procedurally generated. However, there is an extreme difference in scope. While the worlds in *Minecraft* are exceptionally large, the game has a limited number of biomes, flora, and fauna that exist in the game. Taking a step beyond these simple procedural worlds of *Minecraft*, *No Man's Sky* generates the dialogue, non-player characters, and even the music via PCG [19][44]. PCG is a popular technique in games like these for many reasons. The first being it's a form of data compression. The worlds in *Minecraft* and *No Man's Sky* are much too large to be stored directly on disk. Instead, their worlds are generated through algorithms via stored numerical seeds. When a player approaches a section of the game world, that section is rendered using the seed specified. Another typical use case for PCG is to generate varied experiences over multiple play throughs or to extend the play time of the game. In the case of *Rogue* and *Minecraft*, no play through will be the same, because the game world is generated differently every time a new world is created. In *No Man's Sky* the practical combination of all the assets means that every planet should have some unique combination of game assets, meaning two planets are extremely unlikely to be the exact same. Finally, and perhaps most importantly to this work, is that PCG can be a generative shortcut for developers. Developing assets takes time, and PCG is a method by which large amounts of assets can be generated quickly. For example, Perlin noise and graphical L-systems are typical ways developers deploy PCG to generate vegetation and terrain. The fractal nature of the algorithms is well suited to generating assets that have repeating sub-patterns or seemingly random textures [50][53]. These types of PCG are called constructive in that they "run in a fixed time, without iteration, and does not perform any search" [58].

There are several terms related to PCG that will be discussed. They are closely related, so some clarifications of their respective hierarchy will be made here. Search-based PCG is a type of PCG that utilizes search algorithms, such as Genetic Algorithms (GAs) to find optimal PCG content. Procedural

Content Generation via Machine Learning (PCGML) is a broad class of PCG that occurs when Machine Learning (ML) algorithms are used to generate the actual content that is desired. This differs from PCG that uses ML as an evaluatory component. If ML is merely used to evaluate solutions, then it is not PCGML. PCGML and PCG with an ML evaluatory component can both be used in a search-based PCG algorithm, as the "search" in "search-based" PCG relates to how optimal content can be found, not how it is generated or evaluated. Likewise, PCGML and PCG with an evaluatory component do not have to be performed during the course of search-based PCG. They could be used in constructive PCG.

PCGML is an extremely popular approach amongst contemporary PCG researchers. There are many examples of this. Generative Adversarial Networks (GANs) have created unique opportunities in PCG due to their role as machine learning generators, but they are not the only technique that is viable. Markov chains, Monte Carlo Tree Search (MCTS), and Long-Short Term Memory (LSTM) Recurrent Neural Networks (RNNs) have all been utilized to generate *Super Mario Bros.* levels with varying degrees of success [13][14][73][74][75]. In addition, LSTM RNNs have been used to generate cards in the popular card game *Magic: The Gathering* [59]. In this work, although deep learning is deployed, it is not officially recognized as PCGML. Since it only uses ML as the evaluative component, it does not qualify as PCGML, only PCG. Regardless of how it is used, machine learning is a popular area of research in PCG.

While ML can be an important tool when performing PCG, it is merely a way of creating or evaluating content. Search-based PCG is a method by which good content is selected. Search-based PCG is commonly implemented via Evolutionary Computation (EC). As such, much of the taxonomy found in the field of EC is also present in the field of search-based PCG [88]. This work implements search-based PCG with EC, and describing the EAs is analogous to describing the search-based PCG. It should be noted that, although EC is used as the basis of search-based PCG in this work, it is not the sole mechanism by which it can be accomplished. Local search algorithms, such as simulated annealing and hill climbing, are also common [30].

Search-based PCG is a sub-type of PCG known as generate-and-test. First, the content that is to be generated is defined via some form of encoding that establishes the bounds of a search space. Typically, these encodings are represented in a vector of real numbers called a *genotype*. This genotype acts like a blueprint for the generation of a corresponding *phenotype*, the "physical" expression of the genotype vector. For example, in this work, flowers are generated based on a vector of real numbers that determine various traits found in the generated corresponding flower image. These traits include: number of rings, color of petals, the height of the flower stalk, and color, among others. A search space is inherently created by the numerical bounds in such encodings and creates an N-dimensional plane that can be searched through by a variety of methods. The selection of this encoding is extremely important. Genotypes that are too small cannot wholly represent potentially complex solutions. Certain individuals may never be generated if the genotype size is insufficient. On the other end of the spectrum, there is the "curse of dimensionality." As the genotype vector grows, the search space itself also grows at an exponential rate. If the search space is too large, then finding optimal solutions could become an impossibility. This work seeks to limit the effects of such dimensionality explosion by employing techniques such as Centroidal Voronoi Tessalation MAP-Elites (CVT-MAP-Elites), which is discussed in section 2.2.2. These encodings are referred to as the individual's representation [30][62].

Once representations are selected for the problem, individuals are then randomly generated within the search space, and genotype values are tweaked, through a variety of methods, to generate different individuals throughout the search space over the course of many iterations. Individuals are then judged based on an evaluation, or fitness, function which assigns a real number, or a vector of real numbers, to be the individual's fitness. This evaluation could be performed in a variety of ways. One popular research problem for PCG researchers is the generation of *Super Mario Bro.* levels. In one such study, researchers performed a dual experiment testing the generation of *Super Mario Bro.* levels via a Generative Adversarial Network (GAN) [22]. In their first experiment, they measured the fitness of their levels by measuring the fraction of ground tiles to total number of tiles generated. In the second experiment, they measured the amount of ground tiles generated as a fraction of total ground tiles but also factored in the number of enemy tiles that were generated. In order to prevent levels that created uncross-able "gaps", the researchers also included a negative offset for levels that were incomplete [85]. This example highlights the importance of nuance when deciding on a fitness function. Although the fitness may be correspondingly high given a certain set of parameters, if those parameters can generate poor individuals, then the fitness function function itself may be poor. Individuals' genotypes are then passed on to the next generation based on the evaluation function's reported score [81].

Search-based PCG has been successful at generating a breadth of different content within the research community. Aside from the *Super Mario Bro.* example mentioned previously, search-based PCG has been able to generate puzzles for various puzzle games [1][49], maps for RTS strategy games such as *Starcraft* [6][79][80], and in-game weapons [24]. Despite its successes, search-based PCG is still faced with many challenges. Some of this can be explained by the underlying mechanisms used during PCG. Many techniques within EC are optimization techniques, specializing in producing a single solution, or a small set of optimal solutions. It is possible that a solution may never be found, or the solution could be poor. Even when a solution(s) is found, it tends to saturate (or converge) the population in the final iterations of the algorithm. One genome (or a similar genome) will come to dominate all the individuals in a population [30]. One of the large problems within PCG that this work seeks to address is the identical, or similar, nature amongst the content that is generated. Constructive fractal techniques are good at generating structures with "branching" patterns like trees and bushes, but when generating entire forests based on such techniques, there is a degree of similarity that can affect the quality of the content that is generated. This problem is not unique to constructive techniques and is present within search-based techniques as well. To counter this problem, there is growing trend within the search-based PCG community to prefer a different subset of algorithms known as Quality Diversity (QD), or illumination algorithms [54]. QD algorithms focus on creating a variety of different solutions. One popular algorithm that is utilized heavily in this work is Multi-Dimensional Archive of Phenotypic Elites (MAP-Elites) [11][46]. It is an algorithm that specializes in creating and retaining a set of high performing individuals over the course of the entire search. This algorithm is explained in section 2.2.2, but it forms the basis of much of the research that the Diversity Engine was derived from.

2.2 Evolutionary Computation

Evolutionary computation is a sub-field of computer science that uses the principles of Darwinian evolution in algorithms. A Darwinian evolutionary system can be defined by four principles:

- a population or group of populations comprised of individuals competing against one another for resources.
- population change in the form of birth or death of individuals within the population.
- a metric, or general concept, of fitness judging the individual on its efficacy, survivability, and ability to reproduce.
- a variation mechanic operating on the population enabling change from generation to generation.

These four ideas form the basis of all the Evolutionary Algorithms (EAs) present in this work. From them a common framework is identified and can be applied to any algorithm used in this work:

- populations of size m are evolved over time through selection, variation operators, and reproduction.
- previous individuals in the population are used as the starting point for any selection or variation. Thus, all parents must have existed in previous populations.
- newly generated individuals have the potential to take over for previous individuals in the population, but the population will always be constrained by some limit m.

The benefit of using EAs over other optimization methods, such as Artificial Neural Networks (ANNs), are many. EAs can perform on both discrete and continuous problems. In addition, since they are less concerned with exact solutions, rather generating an optimal estimate, they are less susceptible to noisy domains. Finally, their focus on generating individual solutions to form a population make them available to aggressive parallel computing strategies [30]. Because of their flexibility, EAs have been used to address a wide variety of problems outside of PCG such as robot locomotion, scheduling problems, and evolving neural network architectures [30][46][71].

2.2.1 Standard Genetic Algorithm

A Genetic Algorithm (GA) is the most basic algorithm that exists within the EA family. They contain almost the same elements as an algorithm for search-based PCG discussed in section 2.1. Algorithms 1 and 2 specify the basic outline of a GA. First, a population is established with a list of corresponding fitnesses. Random individual genotypes are then generated within the bounds of the representation chosen for the particular problem. Phenotypes are then expressed, and the fitness function assigns a fitness to the phenotype of each individual in the population. These individuals are then placed into the population until S individuals have been generated. Then, for I iterations, the process of generating new individuals takes place. In this process, two individuals are chosen from the population. Usually, a subset of individuals are stochastically chosen from the population, and the two highest fitness individuals from that subset are “bred” together to produce a new child. This reproduction usually happens via two mechanisms called crossover and mutation. Crossover is the exchange of genetic material between the two different individuals. Random amounts of the parent genomes are swapped between individuals producing a new genome, or sometimes several genomes if multiple; new individuals are being produced simultaneously. After crossover, the new genome is subject to mutation where each gene has a low probability to change via some, typically, small value. Mutation is the only mechanism for introducing new genes into the population as crossover only swaps genome values that exists between individuals in the population. These two processes are the variation operators. After this variation has taken place, the individual(s) is placed in the population. This can happen one of two ways. A steady-state GA replaces the individuals into the existing population if the new individuals outperform the old ones. In a generational GA, all the individuals are spawned from the generation that directly preceded it. After all the new individuals have been generated, the new generation replaces the old generation.

The difference between standard and generational GAs is in their selective pressure. Selective pressure is an aspect of a solution that causes certain individuals to be preferred over others. In a steady-state GA, the parents in the population will become more fit as time progresses. This forces the children

Algorithm 1 Standard Generational Genetic Algorithm (from [30])

```
1:  procedure GeneticAlgorithm($S$)
2:      $(\mathcal{X}, \mathcal{P}) \leftarrow$ create_empty_population()      ▷ Create: $\mathcal{X}$ empty population, $\mathcal{P}$ their fitnesses
3:      for $s = 1 \rightarrow S$ do                                    ▷ Initialize: population of $S$ individuals
4:          $\mathrm{x} \leftarrow$ random_solution()
5:          $p \leftarrow$ evaluate(x)
6:          $\mathcal{P}(s) \leftarrow p,\ \mathcal{X}(s) \leftarrow \mathrm{x}$
7:      for $i = 1 \rightarrow I$ do                                    ▷ Main loop: For I iterations update the population
8:          for $s = 1 \rightarrow S$ do
9:              $\mathrm{x}_1, \mathrm{x}_2 = $ selection($\mathcal{X}$)
10:             $\mathrm{x}' = $ variation($\mathrm{x}_1, \mathrm{x}_2$)
11:             $p \leftarrow$ evaluate($\mathrm{x}'$)
12:             $\mathcal{P}'(s) \leftarrow p,\ \mathcal{X}'(s) \leftarrow \mathrm{x}'$        ▷ Store Temporary: $\mathcal{X}'$ individuals, $\mathcal{P}'$ their fitnesses
13:         $\mathcal{P} \leftarrow \mathcal{P}',\ \mathcal{X} \leftarrow \mathcal{X}'$                     ▷ Update whole population
```

Algorithm 2 Standard Steady-State Genetic Algorithm (from [30])

```
1:  procedure GeneticAlgorithm($S$)
2:      $(\mathcal{X}, \mathcal{P}) \leftarrow$ create_empty_population()      ▷ Create: $\mathcal{X}$ empty population, $\mathcal{P}$ their fitnesses
3:      for $s = 1 \rightarrow S$ do                                    ▷ Initialize: population of $S$ individuals
4:          $\mathrm{x} \leftarrow$ random_solution()
5:          $p \leftarrow$ evaluate(x)
6:          $\mathcal{P}(s) \leftarrow p,\ \mathcal{X}(s) \leftarrow \mathrm{x}$
7:      for $i = 1 \rightarrow I$ do                                    ▷ Main loop: For I iterations update the population
8:          $\mathrm{x}_1, \mathrm{x}_2 = $ selection($\mathcal{X}$)
9:          $\mathrm{x}' = $ variation($\mathrm{x}_1, \mathrm{x}_2$)
10:         $p \leftarrow$ evaluate($\mathrm{x}'$)
11:         $s \leftarrow$ select_lowest_fitness($\mathrm{x}'$)
12:         if $\mathcal{P}(s) < p$ then
13:             $\mathcal{P}(s) \leftarrow p,\ \mathcal{X}(s) \leftarrow \mathrm{x}'$    ▷ Update: new replaces lowest performing if p is more fit
```

to be high in fitness if they are to replace the parent, which places more selective pressure to be of a higher fitness. It should be noted in Algorithm 2 that the lowest fitness individual is replaced with the highest performing child. This is only one way to select a potential individual for replacement. Stochastic methods are also a valid way of selecting this individual. In a generational GA, there is smaller selective pressure to be better than parents from the previous generation. Poor individuals generated from the variation operators will be transferred over to the new population regardless of their fitness. The only selective pressure comes from the initial selection of individuals in the prior population. This generation of new individuals is an EAs way of traversing through the search space, or fitness landscape, as it is known in EC. By continually generating new individuals throughout the search space, a GA attempts to discover an optimum [30][62].

2.2.2 Quality Diversity Algorithms

These standard GAs solve a problem known as objective optimization. As stated previously, they are designed to return a single optimal solution, or a set of optimal solutions [31]. An idea, growing greatly in popularity in the last decade, has been to utilize algorithms that, instead of returning a converged, optimal solution, return a group of diverse, high-performing solutions. These are collectively known as *Quality Diversity (QD)* algorithms, or *illumination* algorithms in some domains [54]. This interest is driven by the idea that natural evolution is not necessarily an optimizer. Natural evolution has a tendency to display a wide variety of solutions that operate in a variety of niches. Research suggests an explanation of this observation. High-performing structures found in organisms can sometimes only arise through simpler structures that were rewarded through evolution [38]. Using QD, EAs can become more than just an alternative to optimizing techniques but, instead, can act in entirely new problem domains where the goals are to generate diverse or creative solutions [55]. QD algorithms are ideological descendants of several ideas such as Island Models, Multi-Modal Function Optimization (MMFO), and Multi-Objective Optimization (MOO).

In an Island Model, many populations, each performing a different instance of an identical GA, are run in parallel. Members from the different populations migrate to each other intermittently throughout the run-time of the algorithm. Results suggest that, although each algorithm is identical, starting the search from a different point in the search space on each island yields different overall results [87].

MMFO techniques are designed to discover multiple optima of the search space simultaneously, resulting in diversity. Unlike Island Models, this diversity of solutions is one of the goals of MMFO. One of the first instances of this was in 1975 [29]. The goals of MMFO and QD algorithms are almost identical, but in literature, they are not seen as identical fields. Literature describes the difference between QD and MMFO as, "a later shift in interest toward *behavioral* diversity ... where the relationship between genome and behavior is complex" [54]. Genotypes may not always correspond in a one-to-one behavior with their phenotypes. This interest in the expression of the genotype in the form of its phenotype, as opposed to the genotype itself, marks the difference between MMFO and its later change into QD [54]. This is discussed further on in this section.

Multi-Objective Optimization (MOO), while ideologically further from QD algorithms than MMFO, can also be considered as a precursor to QD algorithms. MOO algorithms are trying to optimize towards a set of different criterion (multiple fitnesses), instead of a single fitness. The algorithm attempts to discover a trade off of different objectives, called a Pareto front, when trying to find out what the "best" solution is [15][54]. While convergence is mainly the goal of MOO, considerable effort has been made to generate diversity along the Pareto front. This emphasis of diversity along the Pareto front is an idea that can be considered a precursor to QD [8][33].

Although the ideological history of QD algorithms goes back many decades, one of the first defining algorithms in the field, Novelty-Search with Local Competition (NSLC), wouldn't be published until 2011. NSLC is an algorithm that rewards individuals for unique behavior but also rewards competition amongst individuals within similar morphological niches. Instead of rewarding only exploration of the best niches, the exploration of a variety of different niches was emphasized [36]. This idea of optimizing for a set of high-performing, diverse solutions would grow within the community and result in new algorithms. One of these was an algorithm known as MAP-Elites [11][23][46].

Multi-Dimensional Archive of Phenotypic Elites

Algorithm 3 gives a description of default MAP-Elites. First, the search space is divided into a N-dimensional grid by splitting each of the different traits in a phenotype into N sections. These traits could be number of petals, number of leaves, choice of color, etc. This results in an archive made up of P^N different cells, or niches, where each cell is a different permutation of the number of phenotypes, P, and number of discretizations N. Each cell represents a unique permutation of phenotype combinations. For example, one cell could represent any flower that has three to five petals, four to six leaves, and is blue. Another cell could represent a flower with six to nine petals, two to four leaves, and is red. A key part of the algorithm is that only one individual can be placed in a cell at a time. Individuals with phenotype ranges that correspond to a particular cell must compete with each other to be that cell's chosen elite. Individuals with the same phenotype ranges can only compete with each other, and there can only be one elite in each cell. From this, the name arises. The algorithm creates an archive with cells based on permutations of different phenotype ranges and individuals with phenotype values inside those ranges compete to be the elite individual in each cell.

This discretization of phenotypes, and inter-niche competition, also naturally creates discretizations in the search space. Suppose there were six colors for a flower and the number of discretizations for each trait in the phenotype was three. This would mean that flowers of two colors would be competing for the same cells. If the number of discretizations were six, then flowers of a specific color would be guaranteed to compete against flowers of the same color. In the first case, the archive would have the best individuals from either color be present. If red and blue flowers were competing for the same cell and red flowers were always superior to blue flowers, then blue flowers would never be present in the archive. In the second case, this does not hold true. Blue flowers do not have to compete against red flowers. They are in their own cells and have a possibility of showing up in the final archive. This is an example of how splitting

Algorithm 3 MAP-Elites algorithm (from [46])

```
1: procedure MAP-ELITES($[n_1, ..., n_d]$)
2:     $(\mathcal{X}, \mathcal{P}) \leftarrow$ create_empty_archive($[n_1, ..., n_d]$)    ▷ Create: $\mathcal{X}$ empty archive, $\mathcal{P}$ their fitnesses
3:     for $i = 1 \rightarrow G$ do    ▷ Initialize: create G individuals
4:         x $\leftarrow$ random_solution()
5:         ADD_TO_ARCHIVE(x, $\mathcal{X}$, $\mathcal{P}$)
6:     for $i = 1 \rightarrow I$ do    ▷ Main loop: For I iterations
7:         x = selection($\mathcal{X}$)
8:         x$'$ = variation
9:         ADD_TO_ARCHIVE(x, $\mathcal{X}$, $\mathcal{P}$)
10: procedure ADD_TO_ARCHIVE(x, $\mathcal{X}$, $\mathcal{P}$)
11:     $(p, \mathbf{b}) \leftarrow$ evaluate(x)    ▷ Evaluate: get fitness p, calculate archive index b
12:     $c \leftarrow$ get_cell_index($\mathbf{b}$)
13:     if $\mathcal{P}(c) = null$ or $\mathcal{P}(c) < p$ then
14:         $\mathcal{P}^(c) \leftarrow p$, $\mathcal{X}(c) \leftarrow$ x
```

the phenotype traits also splits the search space.

To initialize the archive, a number of individuals are generated to create a base population from which more individuals can be generated. In the main loop of the algorithm, a single individual is chosen at random from the archive and its genome manipulated by some variation operators. Afterwards, the archive cell corresponding to the individual's new phenotype ranges is calculated. If the cell accessed with the index is empty, the individual is stored at that index. If an elite currently resides at that index, a comparison of fitnesses occurs and the individual with the highest fitness is the one that is placed at the current index. Variation and fitness assignment can be identical to the techniques found in regular GAs. In algorithm 3, **b** (behavior or phenotype), is used to calculate the cell's index, but the genome can be used as well. In some experiments, the genome does not have a one to one correlation to features in the behavior space. For example, in the robot locomotion experiments used in the original paper unveiling MAP-Elites, the amount of fuel consumed by the robot is calculated from the genome, but there is no gene that directly corresponds to fuel usage. These are known as *indirect encodings.* Some of the encodings used in this work are *direct encodings.* Each part of the genome directly corresponds to some part of the phenotype generated [18][72]. Thus, the genome can be used as **b** in this instance as well as used to index cells in the archive. That is for 3D flowers. For 2D flowers, the the rules of QD algorithms are violated slightly. The genotypes of 2D flowers can be similar but produce distinctly different flowers. Attempting to simplify the technical process, the genotype is still used as the behavior metric in this instance, since to do anything different would require us to be able to recognize flowers of different dimensionality. To be consistent with prior literature, the term behavior will be used in place of genome when discussing how archives are structured (for both types of flowers) and how individuals map to archives, but they are synonymous. This applies to both MAP-Elites and CVT-MAP-Elites.

MAP-Elites has a few distinct advantages. First, the user of the algorithm can determine the granularity of the behavior space. This allows for more control when navigating the search space. This is shown in the example with red and blue flowers. The selective pressure for flowers was higher in the first instance because flowers were competing with more individuals. The more individuals competing in a single niche (equivalent to the niche being larger), the more likely a superior behavior exists within that niche. In addition, much like the different populations in Island Models, each cell is simultaneously navigating the search space. Unlike Island Models, different areas of the search space are (almost) guaranteed to be explored, because each cell in the grid strictly corresponds to a different permutation of behavior ranges. This is beneficial because a cell might be generating a morphology or mechanism that could be useful to another cell in a different part of the grid. This idea of "goal switching" is further explored in section 2.5 [48]. Finally, this technique reduces the effect of local optima, areas in the search space that appear as the optimal solution, but are only optimal in that localized area of the space [62]. Local optima, also known as local minima in some problems, have less of an effect on MAP-Elites because more of the search space is guaranteed to be explored. A local optima may impede a particular cell, but this won't prevent optimum in other cells from being discovered. Unlike NSLC, the complexity of lookup for evaluation in the MAP-Elites algorithm is *O(1).* Novelty-Search with Local Competition (NSLC) has to calculate the feature distance to all other individuals in each generation The nearest-neighbor calculations required for this make the complexity *O(n(log(n)).* Although the algorithm specializes in exploring the search space, a result for each cell is not guaranteed. It is possible that the algorithm will never search any number of cells, either because the opportunity did not arise or that the combination of phenotypes required for the cell simply cannot be made [46].

MAP-Elites has a few major disadvantages that have to be considered when evaluating the algorithm. The first is that its bounds are fixed. Techniques that seek to use EA as a form of endless creativity, such as Innovation Engines, are bound by this constraint as no new niches can be discovered during the course of the algorithm's run time. The second, is a problem of dimensionality. The size of the grid is P^N, which can become prohibitively expensive as dimensions increase. This is seen in prior literature, during an experiment performed on hexapod locomotion where it would have cost 4096TB of RAM just to store the pointers for a 50-dimensional discretized genome in a grid [84]. This is of particular consequence to the problems found in this work, as the dimensions of some of the genomes in the Diversity Engine can be over 50 features. In the case of this work, it is required that another form of MAP-Elites called Centroidal Voronoi Tessalation MAP-Elites (CVT-MAP-Elites) is used [84].

Algorithm 4 CVT approximation (adapted from [12] in [84])

```
1: procedure CVT($k$)
2:     $\mathcal{C} \leftarrow$ sample_points($k$)                ▷ $k$ random centroids
3:     $\mathcal{S} \leftarrow$ sample_points($k$)                ▷ $K$ random samples
4:     for $i = 0 \rightarrow max\_iter$ do
5:         $\mathcal{I} \leftarrow$ get_closest_centroid_indices($\mathcal{S}$, $\mathcal{C}$)
6:         $\mathcal{C} \leftarrow$ update_centroids($\mathcal{I}$)
7:     return centroids $\mathcal{C}$
```

Centroidal Voronoi Tessalation MAP-Elites

Centroidal Voronoi Tessalation MAP-Elites (CVT-MAP-Elites) was created to allow the MAP-Elites algorithm to scale to genomes of large dimensionality, where a uniform discretization amongst all the cells is intractable. The same basic principles of niching that hold in MAP-Elites are applied to CVT-MAP-Elites as well. Discretionary boundaries are drawn in the N-dimensional search space to promote competition within morphological niches. An empty archive is created and filled using the same principles of EC previously discussed with no modifications. The only difference between the algorithms is how the boundaries are created in the search space. Instead of splitting each gene in the genome into equal ranges, a technique known as Centroidal Voronoi Tessalations (CVT) is used [17].

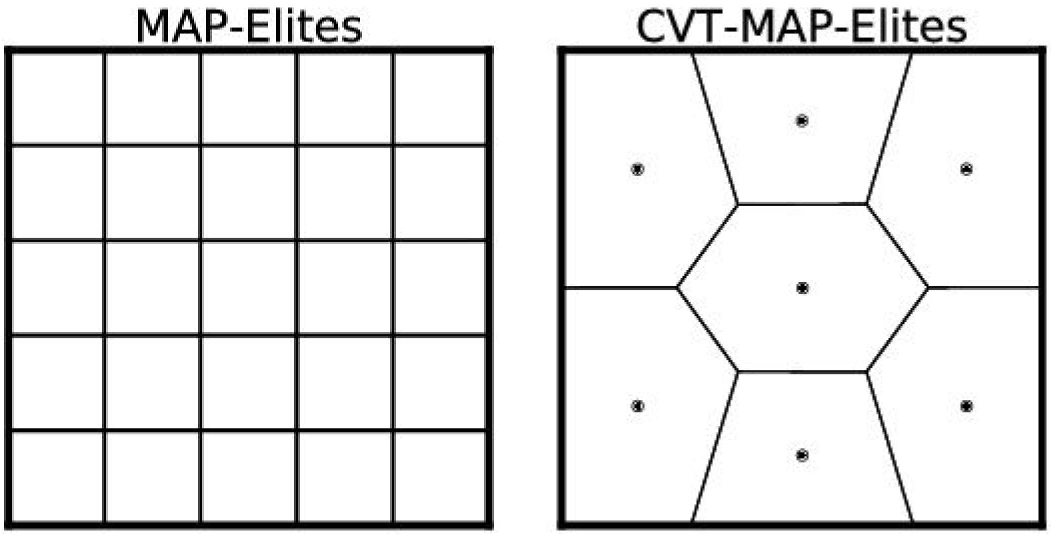


Figure 2.1: Conceptual Image of MAP-Elites versus CVT-MAP-Elites (from [46])

CVT can be thought of as splitting the search space into unequal tiles, or tessalations. Each grid cell in CVT-MAP-Elites is determined by a centroid, which is chosen from a set of randomly generated individuals at the start of the algorithm. These centroids act as distance markers in k-dimensional space.

Algorithm 5 CVT-MAP-Elites algorithm (from [84])

```
1: procedure MAP-ELITES($k$)
2:     $\mathcal{C} \leftarrow$ CVT($k$)                                    ▷ Run CVT and get the centroids
3:     $(\mathcal{X}, \mathcal{P}) \leftarrow$ create_empty_archive($k$)     ▷ Create: $\mathcal{X}$ empty archive, $\mathcal{P}$ their fitnesses
4:     for $i = 1 \rightarrow G$ do                                          ▷ Initialize: create G individuals
5:         x $\leftarrow$ random_solution()
6:         ADD_TO_ARCHIVE(x, $\mathcal{X}$, $\mathcal{P}$)
7:     for $i = 1 \rightarrow I$ do                                          ▷ Main loop: For I iterations
8:         x $=$ selection($\mathcal{X}$)
9:         x$'$ $=$ variation
10:        ADD_TO_ARCHIVE(x, $\mathcal{X}$, $\mathcal{P}$)
11: procedure ADD_TO_ARCHIVE(x, $\mathcal{X}$ ,$\mathcal{P}$)
12:     $(p, \mathbf{b}) \leftarrow$ evaluate(x)          ▷ Evaluate: get fitness p, calculate archive index b
13:     $c \leftarrow$ get_index_of_closest_centroid($\mathbf{b}$, $\mathcal{C}$)    ▷ Access: using b, CVT retrieve centroid c
14:     if $\mathcal{P}(c) = null$ or $\mathcal{P}(c) < p$ then
15:         $\mathcal{P}^(c) \leftarrow p$, $\mathcal{X}(c) \leftarrow$ x
```

As new individuals are generated throughout the course of the algorithm's run-time, they are compared to genomes of the centroids. Whichever centroid's genome they are closest to becomes that individual's new cell. The centroids are determined through a process analogous to using clustering algorithms to classify points in k-dimensional space. This work uses K-means as the clustering algorithm of choice [42]. When the centroids are finalized, areas within the search space form morphological niches based on layout of the centroids in k-dimensional space. The centroids can be indexed in either a list or a K-Dimensional tree (KD-tree) [5]. This has an obvious time complexity disadvantage when compared to default MAP-Elites. The time complexity for a grid lookup in the default algorithm is $O(1)$ whereas CVT-MAP-Elites has a lookup of $O(n)$ on average for a list. For a KD-tree, the lookup is better on average at $O(log(n))$ but still performs at $O(n)$ at worst. In addition, the creation of the CVT is $O(ndki)$ where n is the number of samples in d-dimensional space that are going to be clustered, k is the number of clusters, and i is the number of iterations.

The clear advantage of using CVT-MAP-Elites over default MAP-Elites is it allows for the utilization of the MAP-Elites algorithm for solution representations of extremely high feature dimensions, as it allows for a more flexible choice in the number of niches. As is present in default MAP-Elites, this choice of niching allows the user to directly affect the amount of selective pressure when choosing new elites. In a CVT of fewer niches, there will be higher selective pressure for fitness because more potentially elite genomes exist in the same niche. Whereas in a CVT with a large number of niches, the selective pressure decreases because the number of different genomes that could potentially compete is lessened. It should be clear that using CVT-MAP-Elites puts a heavy reliance on the type of distribution used to generate the initial centroids. If the initial centroids are not generated uniformly across the search space, then the centroids in the CVT will not represent a uniform distribution of solutions. Instead, the centroids will form a dense concentration in k-dimensional space.[84].

The paper proposing CVT-MAP-Elites as an algorithm had several metrics that were measured against the original MAP-Elites algorithm. As such, they used the same experiments as the paper unveling MAP-Elites, maze navigation and robot locomotion. During the maze navigation experiment, robots were evaluated on their ability to traverse a maze based on varying length of feature descriptors (D) [45]. The algorithms were compared in the number of niches filled as well as the spread of solutions. The spread of solutions was calculated using:

$$s = \frac{\frac{1}{|\mathcal{A}|}\Sigma_{\mathbf{b}\in\mathcal{A}}\, d_{nn}(\mathbf{b},\, \mathcal{A})}{\max_{\mathbf{b}\in\mathcal{A}}\, d_{nn}(\mathbf{b},\, \mathcal{A})} \tag{2.1}$$

where $\mathbf{b}$ is the behavioral descriptor of the solution (in this work the genome), and A is the archive. $d_{nn}(\mathbf{b},\, A)$ is the Euclidean distance of of the individual's descriptor to its nearest centroid in the archive. Features were compared between the two algorithms for 2D, 10D, and 50D descriptor lengths. 50D, 250D, and 1000D were used for CVT-MAP-Elites but could not be used in default MAP-Elites due to memory requirements. The discretization of MAP-Elites was 71, 3 and 2 per dimension, which resulted in 5041, 59049, and 1048576 cells for each respective descriptor length of MAP-Elites. The number of centroids for CVT-MAP-Elites was 5000. The median percentage of niches filled for MAP-Elites was 0.72 (2D), 0.09 (10D), and 0.0005(20D). For CVT-MAP-Elites the median percentages were 0.73 (2D), 0.46(10D), 0.40 (20D), 0.38 (50D), 0.36 (250D), and 0.36 (1000D). The median spread of MAP-Elites was shown to be 0.34 (2D), 0.37 (20D), and 0.29 (20D). In CVT-MAP-Elites the median spreads were 0.34 (2D), 0.41 (10D), 0.40 (20D), 0.43 (50D), 0.44 (250D), and 0.45 (1000D). The fitness quality of the archives were extremely similar for each algorithm and CVT-MAP-Elites retained similar fitness quality at higher dimensions. When the original authors compared CVT-MAP-Elites and MAP-Elites for hexapod locomotion tasks, the fitness quality was shown to have similar results to those of the maze navigation tasks, although at higher feature descriptor lengths (12D+) the results of MAP-Elites were much poorer than that of CVT-MAP-Elites [11]. These results seem to suggest that CVT-MAP-Elites can perform at similar levels to MAP-Elites for problems of low-dimensionality and can scale to problems that default MAP-Elites cannot. It is for these reasons that CVT-MAP-Elites was selected as the algorithm of choice for the problems in this work [84].

2.3 Deep Learning

2.3.1 Deep Learning Overview

A main component of the currently established Innovation Engine, the predecessor to this work, is a machine learning technique known as an Artificial Neural Networks (ANNs), or, more specifically, Deep Neural Networks (DNNs). The literature behind DNNs is vast, so a terse explanation will be given here. For more information, consult [21]. DNNs are based loosely on the biology of the human brain. They are composed of many atomic units called nodes that, like biological neurons, are designed to fire when an input signal meets a given threshold. The threshold of a node is dictated by the chosen activation function. Historically S-shaped activation functions, such as the sigmoid function, have been used, but more recently linear activation functions, such as Rectified Linear Units (ReLU), are more popular due to their consistent distribution of error throughout the whole network. The output of this threshold is called an activation. A network is comprised of many of these nodes in layers. One or more layers are comprised of one or more nodes, and these layers produce activated output vectors that are used as input to the next layer. A vector of features is used as input to the first layer, and each feature is connected to each of the hidden nodes via a connection called a weight. The weights dictate how much of an effect an input will have during the calculation of the activation of a node. The output of each layer is a value, or a vector of values, that is the activations of the sum of all the weighted inputs plus a bias node. The bias node is an extra weighted value that is used to adjust the threshold function in each of the nodes. The output of a node is used as input to a subsequent layer where the same process is used to generate another output vector. The final layer is used to compute the output of the network. This process is collectively known as the feed forward phase. The output of the network is then compared with the real desired, or ground truth, values.

A network learns by adjusting its weights. One method of adjusting these weights is by distributing the error signal, generated by comparing the ground truth (real values) to the network's output, which is distributed throughout all the layers in the network. This error signal adjusts the weight values through a process called backpropagation [26]. By adjusting these weights, the network generalizes the distribution of the given input, or training, values. This entire process is known as the training phase. Once training is complete, the model can be used to predict values when given new data [21].

Many advances in contemporary technology have given rise to networks that are drastically larger than the first ANNs. These large networks with many layers are collectively known as Deep Neural Networks (DNNs). The number of weights in a network are generally referred to as the number of parameters. A typical pedagogical neural network (XOR solver) has 6 parameters. The network used in this work, Inception V3, has close to 25 million [77]. As the problems in this work are based in the image recognition domain, the type of DNN used is one that specializes in image recognition, a Convolutional Neural Network (CNN). CNNs are special DNNs that use the layers in their networks to extract high-level features from images, or any data that can be broken down into a grid-like structure, via an operator called convolution. Convolution operators are used similarly to activation functions in regular neural networks, although in CNN what they work over is a subsection, or filter, of the data that iteratively moves over the entire grid. This filter can be thought of as a flashlight that focuses on one section of an image. The flashlight moves from section to section of the image until the whole image has been scanned. The result of these convolutional operators is the generated feature maps that highlight important areas of the grid or image, which are used as input to subsequent layers. When applying convolution to the outer edges of a grid, it is common to apply padding (in the form of extra rows and columns of zeroes) around the external grid values to preserve dimensionality. CNNs also use a pooling operator, which is analogous to reducing the dimensionality of an image [21]. These pooling operators are applied in grid-like filters analogous to convolutional filters, but instead of activating functions in a grid, they are responsible for condensing groups of values into a singular value.

CNNs have been shown to excel in image recognition challenges. The first popular network used to perform image recognition was LeNet-5 in the late 90's. LeNet-5 was used to classify 32x32 images of numbers, now commonly known as the MNIST problem [34]. The next popular CNN model to be unveiled was not until 2012, when AlexNet, a revolutionary DNN architecture, was used during the 2012 Imagenet Large Scale Visual Recognition Challenge (ILSVRC). AlexNet had 8 layers and 60 million parameters in the network. AlexNet's error rate on the ImageNet challenge was significantly lower than the error rates from other techniques [32]. A few years later, another significant model, GoogleLeNet used a form of architecture called Inception, to perform in the 2014 ILSVRC challenge set, achieving similar performance to AlexNet [76]. GoogleLeNet is most relevant to this work, since it uses a very specific architecture called the Inception architecture. GoogleLeNet was build using Inception V1. This work uses Inception V3 for its image recognition component. A discussion of the Inception V3 architecture is given here.

2.3.2 Inception V3 Network

Inception V3 is comprised of a variety of Inception modules. There are four types of modules that are present in the network. All four are used during the training phase of the network, but only three modules (A, B, and C) are used during the actual prediction phase of the network. Each module is comprised of

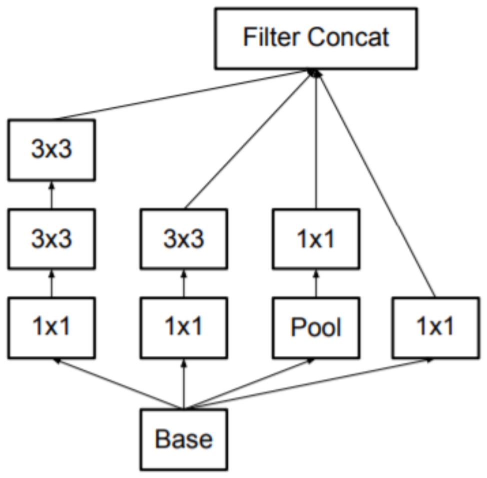


Figure 2.2: Inception V3 Module A Architecture. (from [77])

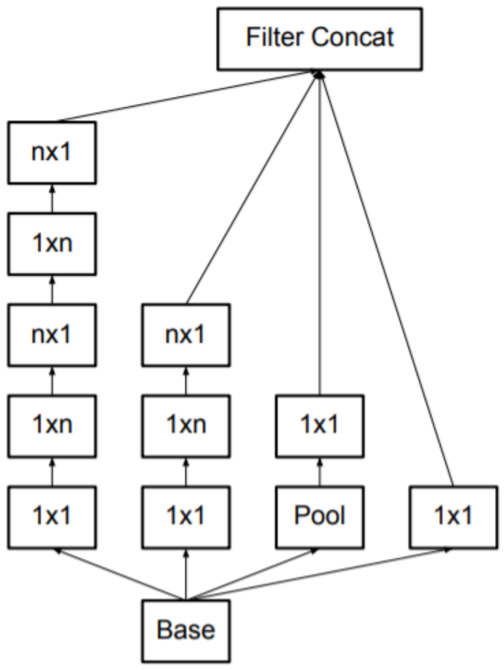


Figure 2.3: Inception V3 Module B Architecture (from [77]). In the proposed architecture n=7.

a different combination of layers of convolutional operations and pooling operations. Convolutions are represented as m x n operations meaning, the that features in the grid have an m x n convolutional filter applied iteratively over the whole grid. The stride represents how many columns of cells the filter adjusts every time it moves to a new section of the image. If a layer has a stride of 1, then the edge of the filter will begin in the next corresponding cell. If the layer has a stride of 2 then the filter will skip the next cell and its starting edge will be the one directly after the skipped cell [21]. After all the convolution has taken place, the resulting values are concatenated to create a single output vector that can be used as input to the next module or layer.

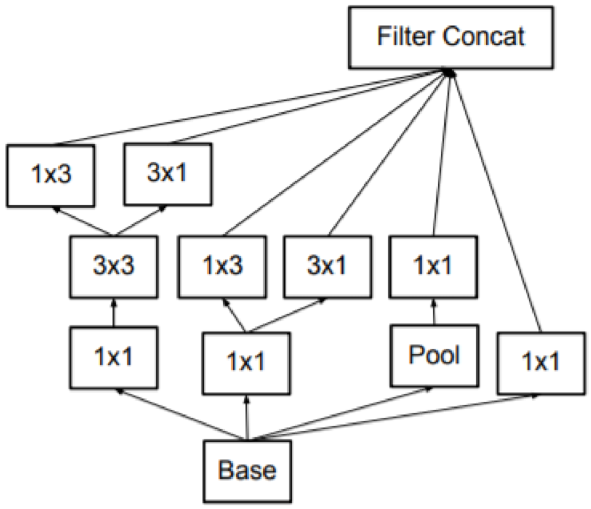


Figure 2.4: Inception V3 Module C Architecture. (from [77])

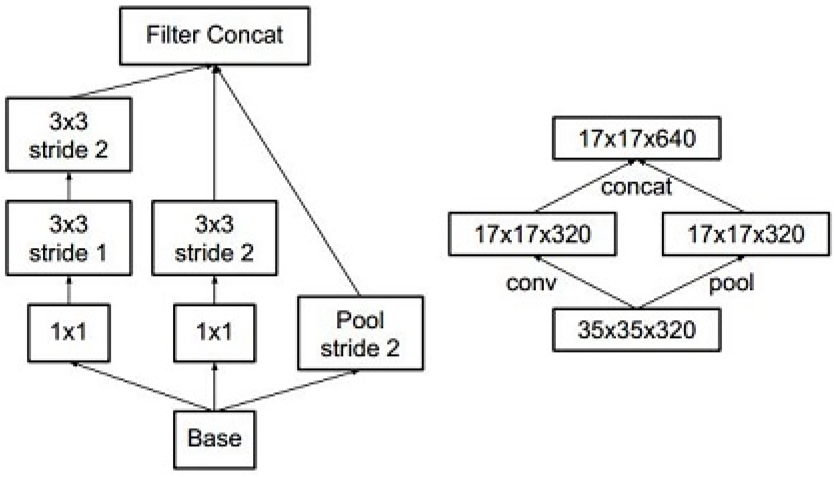


Figure 2.5: Inception V3 Grid Reduction. (from [77]) The figure on the right indicates a high level view of the grid reduction being performed, while the figure on the left indicates the specific operations being applied. It can be seen that dimensionality of each feature map is reduced while overall feature map number (representational expressiveness) increases.

Modules A, B, and C (figures 2.2-2.4) are all various combinations of convolution and pooling. Grid reduction (figure 2.5) is used whenever possible during the training process to reduce the dimensionality of future feature maps without reducing the overall representational quality of the current feature map, thus losing expressiveness in the network. The auxiliary classifier module (figure 2.6) is a way of retaining the loss signal in the middle of the network. Due to the depth of the network, the loss signal propagation

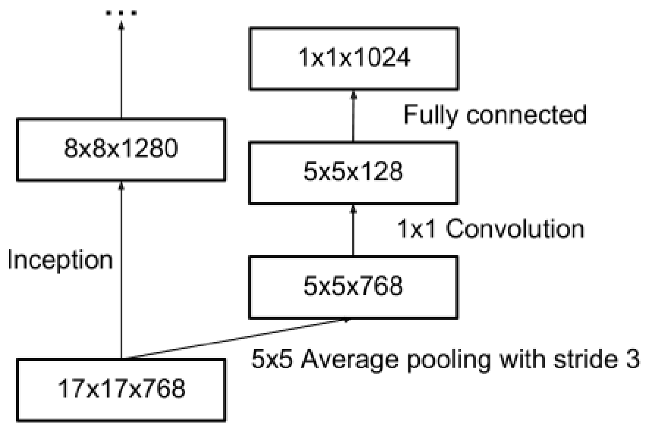


Figure 2.6: Inception V3 Module Auxiliary Classifier. (from [77])

becomes very weak when approaching the start of the network (as backpropagation starts at the end of the network and works its way to the starting layers). To strengthen the loss distribution so that it adequately reaches the first layers of the network, the auxiliary classifier predicts the values of all the training samples in the middle of the network. This intermediate loss is then added to the loss calculated at the very end of the network. This creates a stronger loss signal that is distributed throughout the CNN. The total network architecture is shown in table 2.1 [77].

Inception V3, as well as many of the models shown previously, are known to outperform humans when tasked to the same image recognition challenges, such as ILSVRC [25][32]. While their excellent performance is noted, it has been shown that DNNs can produce images that are graded in high confidence

type	**patch size/stride** or remarks	**input size**
conv	3 x 3/2	299 x 299 x 3
conv	3 x 3/1	149 x 149 x32
conv padded	3 x 3/1	147 x 147 x32
pool	3 x 3/2	147 x 147 x 64
conv	3 x 3/1	73 x 73 x 64
conv	3 x 3/2	71 x 71 x 80
conv	3 x 3/1	35 x 35 x 192
3 x Inception	As in figure 2.2	35 x 35 x 288
5 x Inception	As in figure 2.3	17 x 17 x 768
2 x Inception	As in figure 2.4	8 x 8 x 1280
pool	8 x 8	8 x 8 x 2048
linear	logits	1 x 1 x 2048
softmax	classifier	1 x 1 x 1000

Table 2.1: Outline of the inception architecture. 0 padding is explicitly stated and is also applied in Inception Modules that do not change the grid size (successive modules of the same type).

by the network but look nothing like the class in which the network places them. In fact, many times the images are not discernible to the human eye [47]. It is theorized that this occurs due to the classification regions that are created in N dimensional space by DNNs. The space that is created is theorized to encompass much more than the examples shown during a network's training, and bad examples can fall into these regions.

2.4 Image Hashing Using Perceptual Hashes

Hashing algorithms are transformative algorithms that can map one value into another. This mapped value can act as a form of compression or encryption of the original value. More recently, hashing has been used to do reverse look-ups of images on the web. Cryptographic hashes such as md5 or sha-1 are ill-suited to this task as small changes in the original data can produce extremely large changes in the resulting hash. This is useful in security applications but is undesirable when trying to compare similar, but not identical, images. In contrast, a perceptual hash is a family of hashing algorithms used to transform data, usually images, into a compressed format specifically for comparison. It does this by converting the image into a string of bits. This bit string can then be tested for similarity to other images by taking the difference in bits, or Hamming distance, of the two image hashes. The more similar the string, the more similar the image. Perceptual hashes are able to do this because, unlike encryption hashing algorithms, they are designed to be extremely resilient to changes in the original data [60]. They can be used to find images that have been subject to digital image processing techniques such as cropping, brightening, scaling, and Gaussian smoothing [7]. There are four perceptual hashing algorithms that used in this work.

- Average Hash (ahash)
- Difference Hash (dhash)
- Perceptive Hash (phash)
- Wavelet Hash (whash)

In an average hash, the image is converted to grayscale and then scaled down. The average grayscale value of the image is calculated and then compared with all the grayscale pixels in the image. If the grayscale pixel exceeds the average value, then a 1 bit is added to the hash string, else a 0 bit. In a difference hash the images are again converted to grayscale and scaled down. From left to right, the pixels are compared with their neighbor in the same process that is described in ahash. Perceptive and wavelet hashing are slightly different from these previous hashing algorithms as they operate in the frequency domain, but they both follow the original pattern of being converted to grayscale and scaled down. In a perceptive hashing, a discrete cosine transform is applied to the newly scaled image. This causes all the high frequency areas of the image (the useful parts with the most information) to be located in the upper left corner of the newly transformed image. The image is then cropped to those high frequency pixels, and the median of those pixels is calculated. All the pixels in this cropped image are then compared to this median and the bit string is formulated analogously to difference and average hashing. Wavelet hashing is very similar to perceptive hashing, but uses a different frequency transform algorithm. There are several different forms of the wavelet algorithm that can be used (the wavelet function can be shifted and scaled but it is still wavelet). In this work the haar-wavelet is used. The wavelet algorithm is applied to the image three times. After this, the median is calculated and compared as in perceptive hashing. The difference is the image is not cropped to the upper left corner as in perceptive hashing. The whole image is used. For more information on image hashing consult [7], [20], and [60]. For more information on transforming images into the frequency domain using discrete transforms and wavelets consult [51].

2.5 Innovation Engines

Diversity Engines are derivative of a research project known as Innovation Engines [35][48]. While the term "Innovation Engine" was officially announced in the official paper in 2016, primal ideas of the research project exist in several prior works. In 1994, researchers used an interactive session between a user and the computer to select between two parent images, where the goal was to select the image that the user found most aesthetically pleasing. An ANN was then used to generalize the the user's preferences and automatically evolve images, using EC, that would conform to the user's prior selections [2]. With the research advancements in ANN technology, researchers have been able to vastly improve upon this idea in the research discussed in this section, but the central idea of Innovation Engines (V1) has not changed. A DNN is still used to learn a distribution, and this DNN is then used to guide EC to some desired state. It should be noted that creativity, not image generation, is the overall goal of Innovation Engines, and further iterations intend to address challenges of creativity not related to image recognition.

2.5.1 Schmidhuber's Creativity Theory

Schmidhuber's theory of creativity is the framework from which the Innovation Engine, and its predecessor Deep Learning Novelty Explorer (DeLeNoX) derives some of its philosophical and theoretical ideas about creativity. In the theory, the potential origin and utility of creativity, discovery, and beauty are expounded on. It posits that appreciation of art, music, culture, and beauty are determined by how easily they can be compressed into a form that the human mind can understand. The example given was the ubiquitous appreciation of pop music over classical music. Pop music is generally created using basic, easy to compress chords and harmonies, while classical music is comprised of much more nuanced musical techniques. Appreciation of harder to compress data can be realized through the process of learning, which is synonymous with increasing data compression capabilities, according to this theory. A musical aficionado is much more likely to appreciate classical music than a pedestrian consumer of music due to their ability to understand, or compress, that which they are listening to. Logically, this follows that beauty is on a continuum which is determined by how easily something can be compressed. Furthermore, the theory states that something retains peak beauty only for the short amount of time that it is new, as learning, or in this case data compression enhancement, is limited when re-treading already compressed concepts. A tenet of this theory is that beauty, art, and creativity are all by-products of the mind attempting to increase its ability to compress data. This can extend to scientists as well. The discovery of new scientific principles, and understanding them, helps to quickly compress future observations in the world around us. It does this by introducing simple heuristics or formulas that can be used to compress potentially complex ideas into a simpler, easier-to-use explanation. To summarize, learning new, more complex subjects, whether they be art or science, is the brain's attempt to better its compression techniques. [63][64].

2.5.2 Delenox

Deep Learning Novelty Explorer (DeLeNoX) is a precursor to the Innovation Engines [39]. It is a more basic example of Schmidhuber's creative agent. As in Schmidhuber's theory, the creative agent attempts to find new types of information to compress. It operates through subsequent iterations of transformation and exploration. In the DeLeNoX experiment, the goal was to produce sprites of pixelated space ships for video games. The way the ships were generated was by Compositional Pattern Producing Networks (CPPN) [70]. A CPPN uses the composition of different activation functions (see section 2.3) to create different patterns in its output. A CPPN can be comprised of a variety of different node sequences, and the path the data takes through the network can change. Through the various combinations of nodes and activation functions, many different types of patterns can be created. This applies well to pixelated space ships, as CPPNs can produce many different types of ships. The generation of the space ships is through CPPN, but there has to be a mechanism to search for new archetypes. The transformation phase, or compression, of the sprites is done via Autoencoders (AEs) [26]. AEs are a Machine Learning (ML) technique used to transform the input data, P, into a new feature space, Q. Encoders ($Q = f^w(P)$) are used to transform the data into the new feature space, while decoders ($P' = g^w(Q)$) are used to transform them back into the original distribution P. In DeLeNoX, AEs are used to transform the sprites produced by the CPPN into a low dimensional feature vector encoding, this encoding is used to calculate the novelty heuristic used in Novelty Search. DeLeNoX utilizes a type of EA called Novelty Search [36][37]. More specifically, it uses a form of novelty search called Feasible-Infeasible Novelty Search (FINS). Novelty Search is a precursor to Novelty-Search with Local Competition (NSLC) mentioned in section 2.2.2. It operates by searching for behaviors different than those that exist in its current population. It technically does not qualify as a Quality Diversity (QD) algorithm because it does not search for quality at all. Its only concern is with the novelty of solutions. Furthermore, it does not retain any memory and can cycle back to old solutions. The type of novelty search used in DeLeNoX, FINS, keeps two populations of solutions. The feasible population contains individuals who fit within the constraints specific to the problem. For example, a ship must fit inside a certain boundary to be considered. The infeasible population contains individuals that fall outside the constraints, making them unusable solutions. In DeLeNoX, FINS optimizes for the Euclidean distance between the AEs representation of each of the sprites produced by the CPPN. Evolution is performed via the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [71]. DeLeNoX showed the early capabilities of combining an EA with an ANN. An idea that was developed further in Innovation Engines.

2.5.3 Innovation Engines

Innovation Engines greatly expand upon the ideas first proposed in DeLeNoX [35][48]. Like DeLeNoX, the goal of Innovation Engines is to encapsulate Schmidhuber's work in a technological agent enabling it to make creative decisions, or works, comparable to that of a human agent. Creative decision making is noted not only through the discovery of the new, but by the discovery of the *interestingly* new. Obviously, the creation of such a creative agent is a vast undertaking, so researchers are developing a series of iterative approaches toward the development of such an agent. The first experiments designed to test the idea of deploying such an agent, the ones this work derive from, were to test the automatic creation of recognizable images from the real world. These experiments were labeled as version 1.0. A major reason image generation was chosen was due to the efficiency and accuracy DNNs have in recognizing entities from images [21]. It is a domain where computers can mimic, or rival, the capabilities of human recognition [25][32]. Although excelling at recognition tasks, DNNs do not perform as well when used in creative tasks. As DNNs are a supervised technique, their optimization (during training) is based on a loss function that reports an error signal, which creativity inherently lacks. Creative functions also do not necessarily follow a differentiable curve, a key component used in the gradient descent methods when training DNNs. To make up for this weakness, the Innovation Engine uses EC as the creative component for the reasons discussed in section 2.2. A CPPN was used to generate 2D images in a style found on

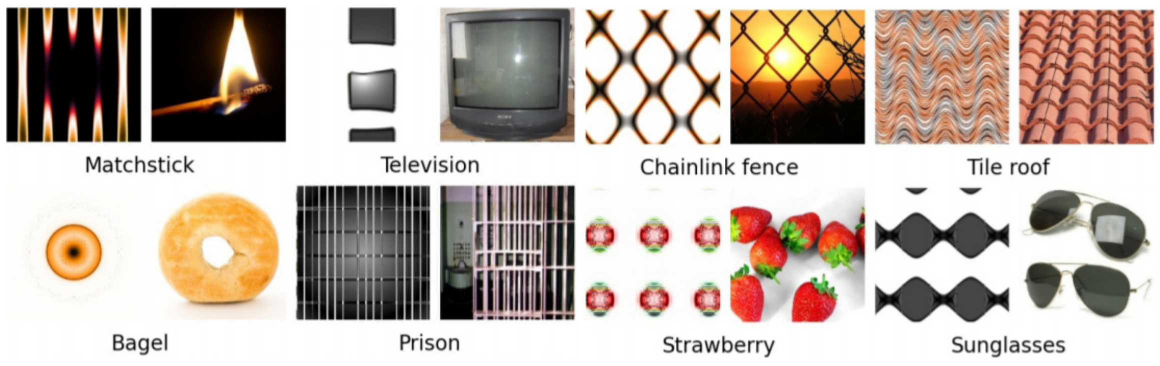


Figure 2.7: Results from the 2D Innovation Engine Generation. (from [48]). The image on the left represents the generated image when compared with it's real-life equivalent

PicBreeder, a website where users can breed evolutionary art [66]. The generated images were classified by a DNN (see section 2.3 AlexNet) into the 1000 different classes found in the ImageNet database [16][32]. The CPPN is then optimized based on the confidence that the DNN has in that image. The algorithms that made up the search for new solutions were Novelty Search and MAP-Elites (see section 2.2.2) [11][46]. For MAP-Elites, each class in the ImageNet set was a corresponding cell in the MAP-Elites grid. For Novelty Search, the final set of images were associated with their highest corresponding class in the ImageNet database. The CPPN was evolved using NEAT [71]. Images generated by this Innovation Engine are shown in figure 2.7.

The motivation behind using these specific search algorithms, as mentioned before, is the problem with single objective optimization. The morphologies in some class may only present themselves when they are rewarded in other classes that are also being evolved [36][37][38]. The results of the experiment supported this conclusion. The initial work coined this phenomenon *goal switching*, when a member of one class passes its genes to form an elite member of another class. For each class, there was a mean of 8.7 goal switches. For context, each class had a mean of 48.6 new champions arise during the course of the algorithm. This means 17.9% of the new members generated for each class arose out of *goal switching*. The experiment also tested the effect that the number of objectives, in this case classes, had on the overall performance. N classes were randomly selected from the ImageNet categories in amounts of 1, 50, 100, 500, and 1000. The median performance of the Innovation Engine improved monotonically as the number of classes increased. The researchers hypothesized that this happened because a larger number of classes enabled goal switching to occur more frequently. This was supported by the observation that the number of *goal switches* also increased monotonically with the number of objectives. A noted secondary explanation for increased performance is the increased diversity from a varied number of classes. Poor diversity means fewer varied genes in the pool. A lack of simpler building blocks could mean that

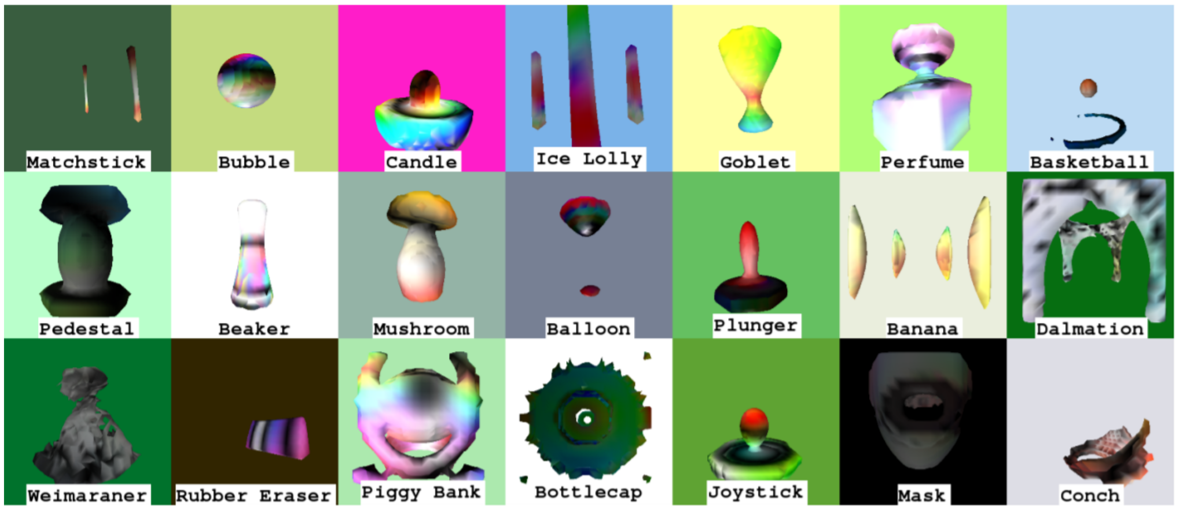


Figure 2.8: Results from the 3D Innovation Engine Generation. (from [35])

more complex structures that require those blocks will never arise [38]. This concept of more efficient evolvability via more objectives was tested using the Innovation Engine through a variety of methods. The results are not conclusive, but the experiments suggest that more objectives tend to increase the median performance of offspring of high-performing classes over all the classes as a *whole*. That is, the performance of offspring in their parent niche is poorer in relation to the parent, but their performance as it relates to the entire set of classes is higher [48].

Another application of Innovation Engines was published not long after the release of the initial paper. 3D images were created using encodings generated by EndlessForms.com, a web site similar to PicBreeder, but specializing in 3D constructs [9]. The forms are created, again, via CPPN. The MAP-Elites algorithm is used to create a grid of the 1000 classes in the ImageNet database. The DNN that is utilized is GoogLeNet [76]. The paper acts as a proof of concept in combining an Innovation Engine with a DNN that can produce recognizable 3D images of the various classes in ImageNet. EAs offer a mechanism by which 3D artefacts can be generated using a neural network. In the Innovation Engine, 3D artefacts were generated using the EndlessForms.com encoding and then images were taken at different angles of the 3D artefact. The confidences of these images, obtained from the DNN, were then multiplied together to create an overall score of the artefact. Different rendering improvements, such as lighting, were also evolved alongside the encoding as some artefacts, such as fish, might not be recognized by the network unless a blue background was included. Search improvements, such as adding more general objective (class) categories and biasing selection towards niches that innovated more often, were also included. Images generated by this Innovation Engine are shown in figure 2.8. The requirements of the algorithm were computationally expensive, so only two full runs were performed. 18.2% of classes achieved confidence greater than 0.85 (on a scale of [0,1]) where search and rendering improvements were included. Alternatively, only 15.1% of classes achieved confidence greater than 0.85 when only rendering improvements were added. The classes that performed the worst had multiple classes that were extremely similar to themselves. These classes usually were breeds of animals like cats and dogs. This is of particular relevance to this work, as the optimization of different types of flowers is attempted [35].

The overall driving force behind Innovation Engines is to generate an agent that can perform all sorts of tasks related to creativity and do so autonomously. The researchers stated goals are to keep applying the Innovation Engine to different domains in order to advance the work in a more a general way. Ultimately, the Innovation Engine should be able to innovate in all sorts of domains, addressing challenges found in the physical, scientific, and artistic realms. Although grand, these goals are beyond the scope of this work, and Diversity Engines attempt to apply themselves to a much more narrow set of problems.

2.6 Other Related Work

Innovation Engines are by no means the first technology to employ image generation technology. There are a number of other works that have attempted to apply EC to image generation tasks. This early work includes the generation of human faces and evolutionary art [10][41]. Other researchers have attempted to encapsulate the ideas of creativity by generating vases based on the abstract qualities of the environments around them as well as generating novel sculptures based on ML techniques [27].

More specific to this work is a subject of research known as Petalz. Petalz was a game that was designed to run on Adobe Flash systems and the Facebook platform. The goal of the game was to breed flowers on a virtual balcony. These flowers could be sold to other players of the games so that the buyer could breed that flower with flowers that the already own. Flowers could be classified into differing types and 3D models could be printed using the appropriate software. The flowers were encoded via a CPPN which was evolved using the NEAT [71][70]. It should be noted that the goal of Petalz was not to generate any specific flower, but more as a proof of concept for PCG as a core game mechanic.

Another game to utilize EC heavily is *Darwin's Demons* [69]. *Darwin's Demons* is a game where the player, who pilots a space ship, is tasked with defeating waves of enemies that progressively get closer while firing at the player's ship. Between each wave, the enemies are evolved using a GA. Although, the GA has some special modifications to the variation operators. Enemies are evaluated based on their lifetime over the course of the wave, their aggression, and their accuracy. Genetic anomalies in the form of "migrants" can appear over the course of play and represent difficult enemy archetypes. These archetypes were selected manually prior to play. The results from this work showed that evolution, and player choices throughout the evolutionary process, can have a significant effect on the difficulty level and the types of individuals that arise throughout the game.

Chapter 3: The Diversity Engine

Innovation Engines provide a solid foundation for automatic, asset-driven PCG. However, the results are somewhat abstract (refer to figures 2.7 and 2.8), and there is a lot more work to be performed for the quality of the generative solutions to be up to the standards of even a simple video game. As Machine Learning (ML) techniques improve, the potential quality of generative solutions may improve as well. There is a research-driven mandate for a lightweight and adaptable framework that can generate many different kinds of objects that can be evaluated by many different methods, all done quickly and with relative ease. The Diversity Engine is a response to those requirements. It is intended to be a pragmatic application of the image-based theoretical work of Innovation Engines. To be clear, the Diversity Engine is intended to be derivative of Innovation Engines proper and exists within their domain, albeit in a narrower scope. Designed via object oriented principles, the Diversity Engine is intended to be a modular, easily-extendable framework that allows for rapid prototyping of different PCG class types (flowers, trees, rocks, etc.) and evaluation of those classes via different methods, whether those be based on ML or not.

3.1 High Level Tool Architecture

This section is intended to be a high level overview of the Diversity Engine's architecture. For more specific software implementation, consult section 3.2. Figure 3.1 shows the high level design of the Diversity Engine. Generative objects get encapsulated in the Evolving Object class. Evolving Objects form the atomic unit of the entire framework. They represent, in C++, a solution that exists (or will exist) on disk in the form of an image. The Evolving Object is in a form that can be manipulated by the evolutionary facilities and carry the evaluation of their corresponding images. They are also responsible for writing relevant information to trait files when it comes time to execute PCG class generators. PCG classes can be hard coded as a C++ child class of Evolving Object, stored as Evolving Objects themselves with their specifics determined via a configuration file that specifies gene ranges and types. These Evolving Objects (and their children classes) are wrapped in another class, the Evolving Object Factory. This class serves as the Evolving Object class factory (not to be confused with PCG class generators) for the Evolutionary Algorithm classes, which allows them to create any user-defined C++ class. After being generated by the factories, the Evolving Objects are stored in populations or archives in the Evolutionary Algorithm classes and provide a mechanism to generate PCG class images and write them to disk. The internal Evolutionary Algorithm classes are currently limited to CVT-MAP-Elites and a standard GA. A Python wrapper is placed around their Pybind interfaces. For CVT-MAP-Elites, this wrapper allows for the creation of the centroid clusters that will act as archive cells, as well as allowing for hashing

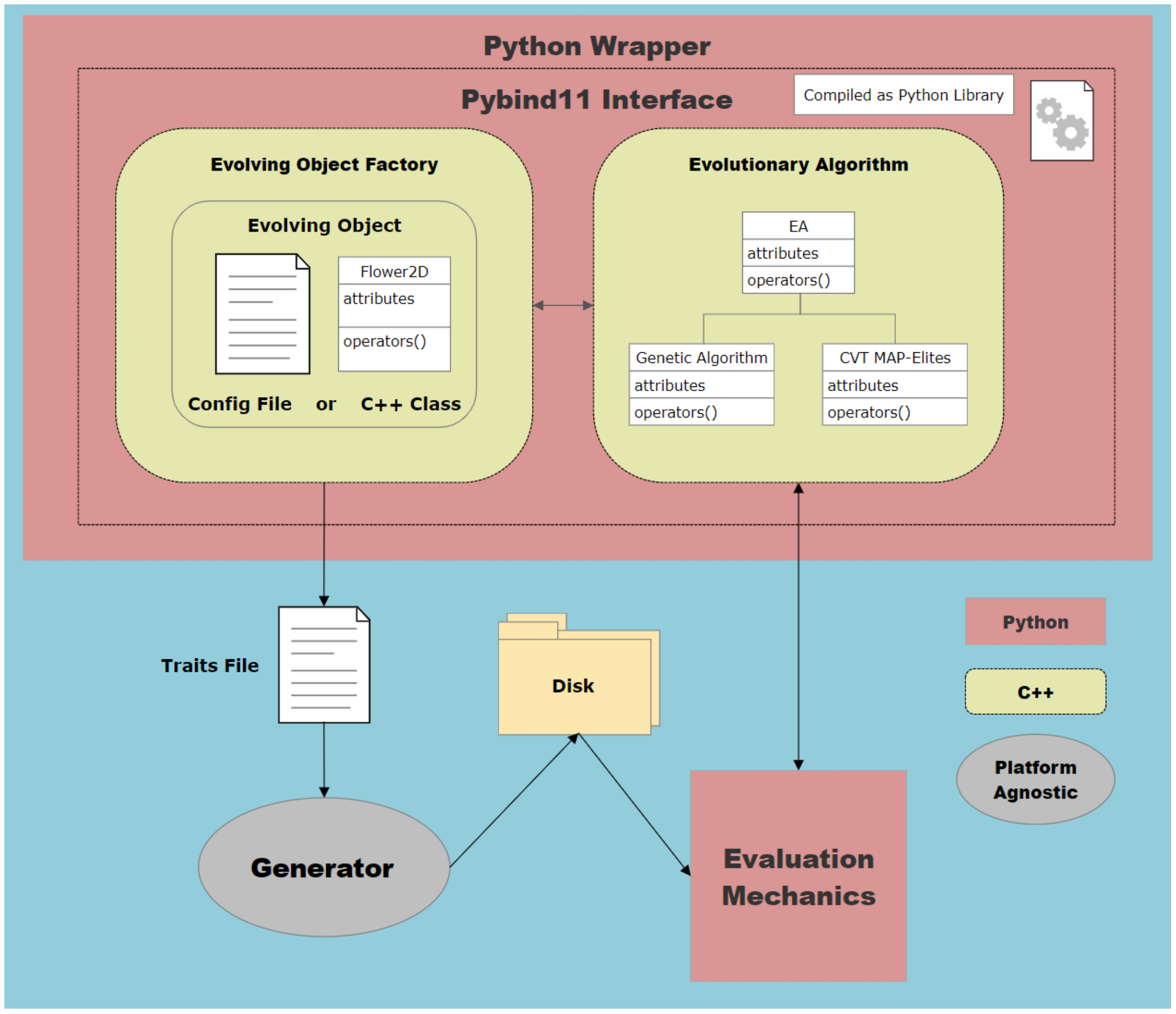


Figure 3.1: High Level Architecture of the Diversity Engine

genomes (via Python tuple) for archive cell placement. For both GAs and CVT-MAP-Elites, it allows for more sophisticated printing facilities. For more information about extending these facilities consult section 3.2. These classes are written in C++ to utilize the language's well-known speed and parallel frameworks, but they are wrapped in a Python3 interface to allow for Python's ease of use. This C++ to Python interface is implemented via Pybind11 and is written to a Python library which can be called from Python as a module [86]. The generator and evaluation mechanisms exist independently of this interface. The evaluation mechanisms could exist outside the Python environment with use of Python Pickling and sub-process calls, but this work utilizes mechanisms written in entirely in Python. The PCG class image generator is platform agnostic. The user specifies a sub-process system call prior to run time which, when activated during the PCG creation of an Evolving Object, calls an executable which generates the actual PCG class image. There are three requirements of any generator that is going to be used in the Diversity Engine. The first is that is should take a trait file as an argument. The second is that it must be able to read from this traits file that should contain the list of genes required to build the object as well as the image location (or any other relevant information). Finally, it must be able to save the image to disk in the same location provided in the traits file. As long as the generator follows those rules, it can act as a generator for the Diversity Engine. This enables the usage of generators written in any language, on any software platform, providing true decoupling of the generator from the interfaces. For example, this work utilizes two generators. One is written in the gaming software engine Unity (C#) [83]. The other is written using a platform called Processing and is written in Java [56].

3.2 Implementation

This section is intended to be a technical reference for the Diversity Engine. For a high level overview consult section 3.1. The Evolving Object class acts as the parent class for more specific object classes, although it can be used on its own. It retains all the important characteristics of the object images it corresponds to, such as genome length, file location, class type, and, obviously, the genome. Polymorphism is used to access user defined classes inside the Diversity Engine's evolutionary facilities and also in evaluation mechanisms. Through inheritance, all Evolving Object children classes receive the methods and members necessary to operate within the Evolutionary Algorithm classes and operate from inside an Evolving Object parent container. This enables users to define their own classes through one of two ways, creating their own C++ child class (and factory wrapper) or using Evolving Objects themselves and specifying PCG class traits via a configuration file. If the genome can be represented as a simple ordered list, it is not necessary for users to create their own C++ classes to represent their objects when using the Diversity Engine. Users can create the configuration files mentioned previously that can be passed to the Evolving Object Factory's constructor. All genes in the genome are normalized between 0 and 1 when created, and this configuration file specifies each gene's range so that each gene can be restored correctly when writing the genome to a traits file. The configuration file also specifies whether these genes should be floats or integers. If the genome required by a generator is more complex than an ordered list, users will be required to create their own C++ classes derived from the Evolving Object class. For example, the 2D flowers used in the upcoming Diversity Engine experiments have a variable length genome. A static configuration file is inappropriate in this case, and methods were created so that genomes of arbitrary length could be written to a traits file.

Since the Evolving Object type is used as a container for all Evolving Object children inside the Evolutionary Algorithm classes, it is necessary to initially generate the objects from factory classes that can be passed to the Evolutionary Algorithm classes and generate objects of the appropriate Evolving Object child class. The Evolving Object Factory, or its children classes, are created prior to the initial construction of the Evolutionary Algorithm classes. They are then passed as objects to the constructor of the Evolutionary Algorithm classes inside an Evolving Object Factory parent class container and used whenever new objects need to be generated. This polymorphism allows the framework to reference the Evolving Object container throughout the Evolutionary Algorithm classes while also generating the correct objects at run time.

Currently the Diversity Engine only offers two different Evolutionary Algorithm classes, a standard GA and CVT-MAP-Elites. These are children classes that derive from the parent Evolutionary Algorithm class. The parent class offers many of the different members you would find in a typical EA such as crossover and mutation, and some print facilities. Selection, population structure, population insertion maintenance, and some more algorithm specific print facilities are all the responsibility of the children classes. These operators can take vastly different forms across different algorithms and must be relegated to the children classes. A Python wrapper class has been placed around the Evolutionary Algorithm Pybind interfaces (which includes the factory and object classes by extension). A large portion of the Python wrapper for CVT-MAP-Elites was written in Python for [46]. It has been adapted to fit the Diversity Engine's framework. For the CVT-MAP-Elites, this allows access to Python's sci-kit learn package with its clustering facilities as well as the KD-tree facilities. The centroid map (archive) is handled by this Python wrapper where the centroids are initially generated using sci-kit learn's k-means clustering. After these centroids are found, they are stored in a in a KD-tree for $O(log(n))$ lookup. This wrapper allows for the hashing of an object's genome (via tuple) in the Python layer, which allows tuple-to-archive index mapping. The retrieved archive index and the object are then passed to the C++ layer, where the new object could be placed in the C++ map (archive). Due to the optimization and ease of use of sci-kit learn's packages it was determined that splitting CVT-MAP-Elites functionality between layers was the optimal choice at present. A wrapper is also placed around the GA class but is much less important to its core functionality. Like the CVT-MAP-Elites, it contributes to printing some of the algorithm's more important metrics. These wrapper classes are, at present, the interfaces that should be used when utilizing the system in Python. Due to the way that memory is managed between the C++ and Python layers, the C++ classes should not be used as standalone C++ libraries, but as source for the Pybind libraries.

OpenMP is used to generate PCG class images simultaneously via multi-threading. To clarify, it is only initial object generation, the writing to trait files, and actual generation via process calls and execution that is done in parallel, not the variation of the C++ Evolving Objects themselves. The number of threads can be specified upon compilation of the Pybind11 libraries. To ensure that pseudo-random number generation is still valid during parallel execution, this work utilizes Tina's Random Number Generator (TRNG) for all random number generation [3][4]. TRNG is specifically designed to generate numbers in parallel environments. Typically it is used in Monte Carlo simulations, but it translates quite well to EAs. TRNG allows for the "leap-frogging" of sequences of random numbers in the specified generator during parallel execution. This allows each thread to leap to a unique sequence of numbers for each Evolving Object generated. Reproduce-ability is an important design consideration for the Diversity Engine. So both the factory and EA classes are designed to accept a seed in their constructors that is used to initialize their TRNG generators. During run time, this seed is shifted every time a generator is used during the course of the EA, so a different sequence of numbers is used for the variation operators each time they are called. This seed shifting provides decent pseudo-randomness while also ensuring reproduce-ability via deterministic seed specification.

Chapter 4: Experimental Configurations and Results

Due to the comparative nature of the experiments, all experimental commentary will be left to chapter 5 to provide a more detailed, holistic overview of the experimental results. This chapter will merely provide the refined data collected from the experiments.

4.1 Generators

Generators exist independently of the Diversity Engine. They are responsible for the actual generation of object images. As they are not intrinsic to the overall schema of the framework, they will be discussed here. Flowers were the object being generated in the experiments, and there are two types, 3D and 2D. They are written in two different platforms and will be discussed independently.

4.1.1 2D Flowers

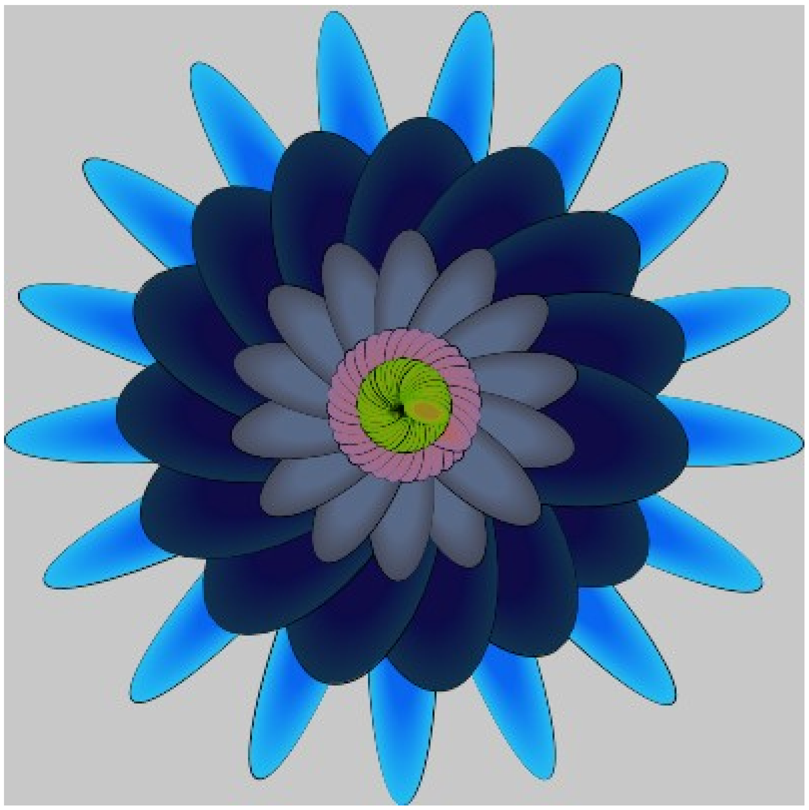


Figure 4.1: Example of a generated 2D flower.

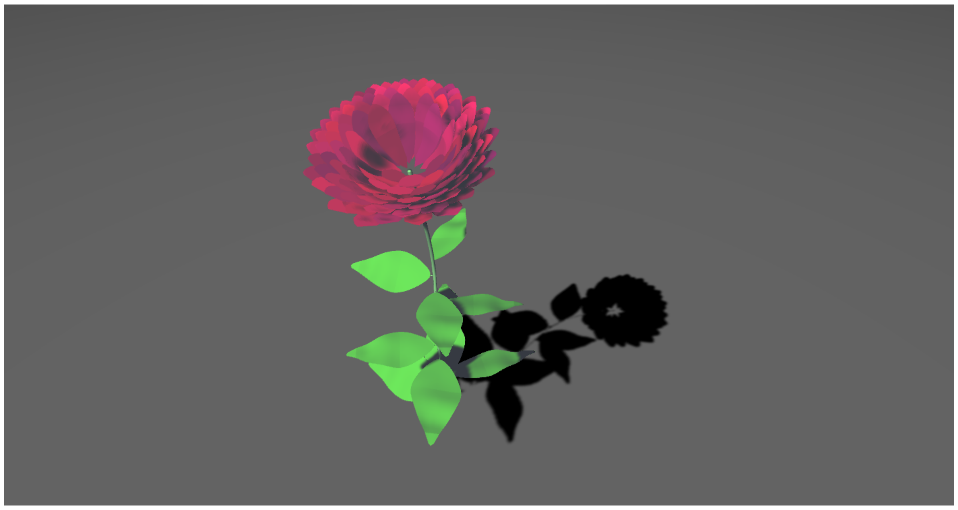


Figure 4.2: Example of a generated 3D flower.

The 2D flowers are written in a program called Processing [56]. Processing is a Java-based platform that specializes in the programmatic generation of artwork, or sketches. The flowers are generated by creating overlapping rings of petals that are varying elliptical shapes. These shapes can be thin enough to form lines, or round enough to form circles. Each ring has a separate list of traits that determines its size, shape, and color. The number of rings a flower can have is variable but its maximum is statically determined prior to run time. The number of maximum rings a flower can have in the trials is 5. That is, each flower can have up to a maximum of 5 rings of petals. There are a total of 52 genes in each flower's genome (52D). There are 10 different traits that vary with each ring, meaning there are 50 traits that correspond to the rings of petals. Two of the traits correspond to the whole flower, the number of rings, and the type of shading inside the petals. It should be noted that it is possible to generate objects that look nothing like flowers inside this program. An example flower is shown in Figure 4.1.

4.1.2 3D Flowers

The 3D flowers are written in Unity using the C# language. The software used to generate the flowers was originally written by GitHub user mattatz and is being used under the GitHub open source license [43]. Some small modifications have been made to procedurally generate flowers based on trait files, but aside from these modifications the code has not been changed. The software is specifically designed to create flowers and, because of this, generates procedural objects that tend to look more like flowers, on average, than the 2D flowers. Each flower has 13 genes that account for traits that would typically be seen in flowers such as petal number, flower stalk height, flower size, and petal color (all stalks are green). The image of the flower is derived from taking a screenshot of the flower above the flower itself and at an angle. An example flower is shown in Figure 4.2.

4.2 DNN Training Parameters and Test Performance

A DNN known as Inception V3 is used as an evaluation component for the Diversity Engine for the experiments shown here [76][77]. It is discussed at length in section 2.3.2 and network architecture specifics should be looked for there. Transfer learning was used to train the initial network on the 1000 classes of found in the ImageNet data base [61]. Transfer learning involves training a network on classes of images and then freezing all but the last few layers of the network and retraining these layers on new classes of data. This is typically done when there isn't enough data for a class to train the whole network. The network trains most of its weight layers to recognize general characteristics using the original training classes with plentiful data. Then, it learns the specifics of the new classes by retraining the last few layers on the set with less data [21]. Daisies, dandelions, roses, and sunflowers were the newly trained classes. Tulips were originally included, but they were eventually excluded due to the perceived difficulty of generating a 2D tulip - the 2D generator focused on creating topdown images of flowers, whereas almost all images of tulips from the database are side views. Prior to data augmentation, there were 633 daisy images, 898 dandelion images, 641 rose images, and 699 sunflower images. The images were augmented using flips, a 10% random crop, a 10% random scale, and a 20% random brightness alteration. After augmentation, there were roughly 1360 daisy images, 1660 dandelion images, 1240 rose images, and 1780 sunflower images. 10% of these were used as test images to benchmark the networks training performance. 300 epochs were used to train the model, and the mini-batches were in sizes of 100. After training, the network was able to identify images from the test set with a 91% accuracy on daisy, an 89% accuracy for dandelions, a 96% accuracy for roses, and a 93% accuracy for sunflowers. When used for prediction, the CNN gives a score from 0 to 1 that can be interpreted as a percentage.

4.3 Image Hashing Software and Comparison Images

This work uses the four perceptual hashing algorithms described in section 2.4. The algorithms are written from Python3 and are based on ImageHash library written by Johannes Buchner [28]. This repository can be downloaded using the Python Pip tool. Each hashing algorithm creates a 64-bit long string of the image being hashed. Different strings for different images are compared against one another. The total score is the percent, from 0 to 1, of bits that are identical in both strings. The hashing algorithms are used to extract individuals from the final populations generated by the EAs by comparing generated images with pre-made "accurate" images in the corresponding generators. In some experiments, they are used as part of the fitness score. When used as a fitness score, the percent correct is added to the total confidence from the DNN. For each of the generators, different comparison images are required. These are listed in table 4.1. Real images are supplied for references but are *not* used as hash images during the course of any experimental run.

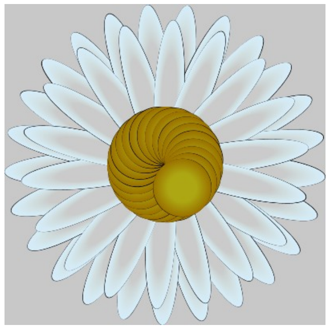
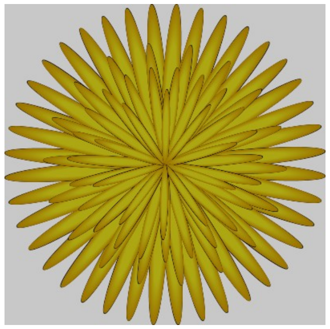
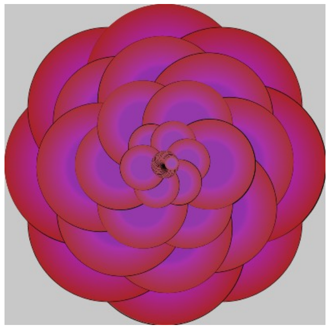
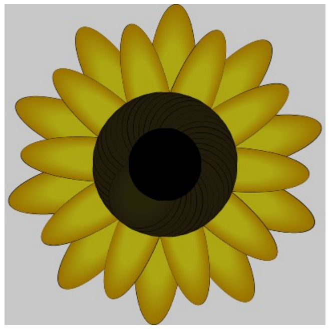
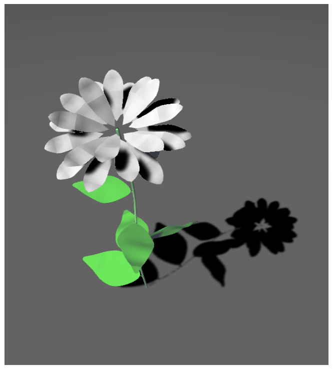
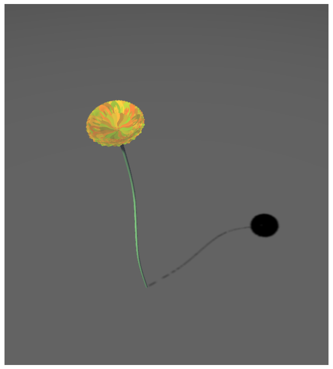
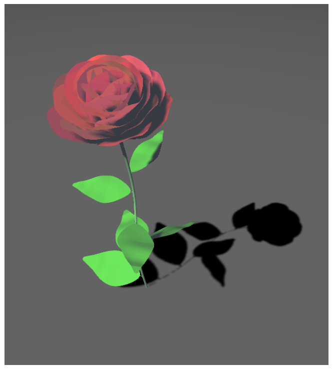
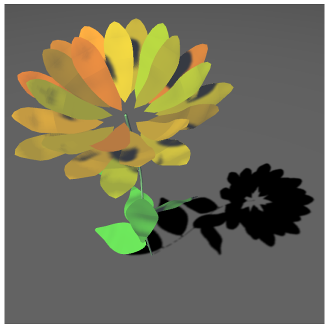
	Daisy	Dandelion	Rose	Sunflower
Real				
Processing				
mattatz-Unity				

Table 4.1: Images used for comparison when using hashing algorithms

4.4 Experimental Equipment

The 2D flower experiments were ran on the Computational Resources Core (CRC) located at the University of Idaho. Four different nodes were used. All the nodes were comprised of 2 Nvidia GTX 1080Ti graphics cards, 128GB of system RAM, and 2 Xeon 35-2623 v4 processors each with 8 cores. Eight threads were used to run the experiments. Due to rendering limitations, the 3D flowers could not be generated on the CRC. Instead, the 3D flower experiments were ran on a desktop home computer with an NVIDIA RTX 2070 graphics card, 64GB of system of RAM, and an i5 Intel processor with 4 cores. Only two threads were used during the course of these experiments.

4.5 A Note on Reproducibility

Appendices A.2 and A.3 contain the seeds necessary to reproduce each of these trial runs. Identical seeds are used in the 2D and 3D flower trials. This is made possible by the fact that the genomes and traits between 2D and 3D flowers vary wildly, so psuedo-randomness is still preserved. During each of the trials, due to lack of computational resources, there was a potential for a flower to get "corrupted" and not be produced. This had the potential to happen 0-5 times during the course of each run. While this does hamper the ability to do exact reproductions, keep in mind these are 0-5 flowers out of the production of many thousands. The overall predicted effect of these corruptions on any run is minimal.

4.6 Standard Genetic Algorithm

For the standard GA, uniform crossover and mutation were utilized. Crossover was performed at a rate of 30%. Mutation was performed at a rate of 5% for the 2D flowers and 10% for the 3D flowers. Tournament selection was used to select the individuals from the population with a tournament size of 3. The reproduction model was generational so all newly generated individuals were passed to the new population. The population size for both 2D and 3D flowers was 250. Due to hardware limitations, the 3D flowers were only generated for 100 generations, resulting in 25,000 flowers over the algorithm's life-cycle. The 2D flowers were generated for 200 generations, resulting in 50,000 flowers over the algorithm's life-cycle. Only the DNN confidence value was considered as a fitness. Therefore, the fitness is expressed as a confidence percentage from the DNN. Five trials were performed for both 2D and 3D flowers, with the metrics being averaged over those five trials.

4.6.1 2D Flower Generator

Out of the five trials, a rose was the best DNN-scored individual four out of five times. For the one other trial it was a daisy. In the final generation, the fitness, on average, of the best individual in the population was 99.4%. The average fitness of the final population was 99.0%. The average number of goal switches (changes from one class to another during reproduction) was 0.2. At approximately 40 generations, the algorithm converged on a single value. This image came to dominate the population and only it, along with slight variations, existed in the population. That means that in four out of five trials, only roses existed in the final population, and in one trial it was daisies. When filtering the population with whash-haar 95% threshold, only the fourth trial, which converged on daisies, was able to extract a single image. Figures 4.3 and 4.4 show the best and average fitnesses as generations increase, averaged over the five trials. Table 4.2 shows the best DNN-scored image (basically the only image) from the final populations, as well as hash-filtered sub-populations, from each trial, if any exist.

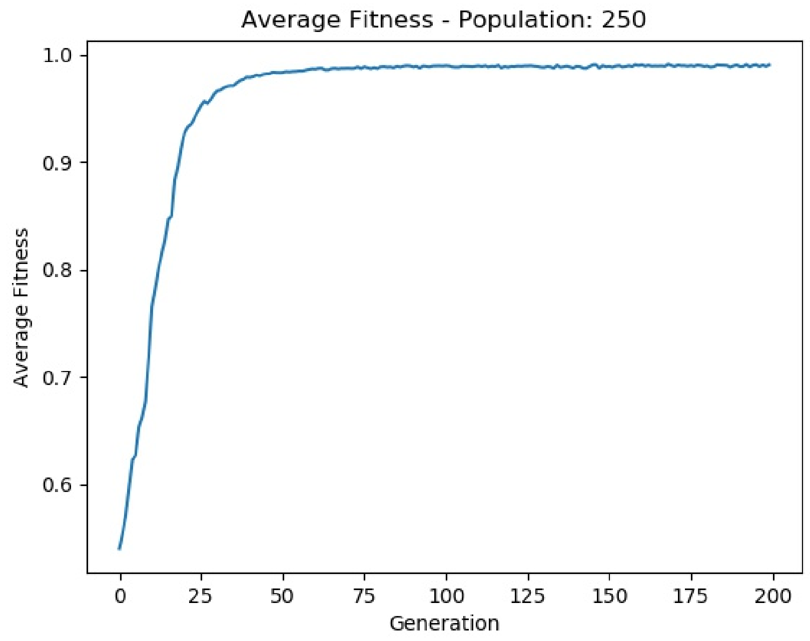


Figure 4.3: Average generational fitness for 2D flowers created using a standard GA

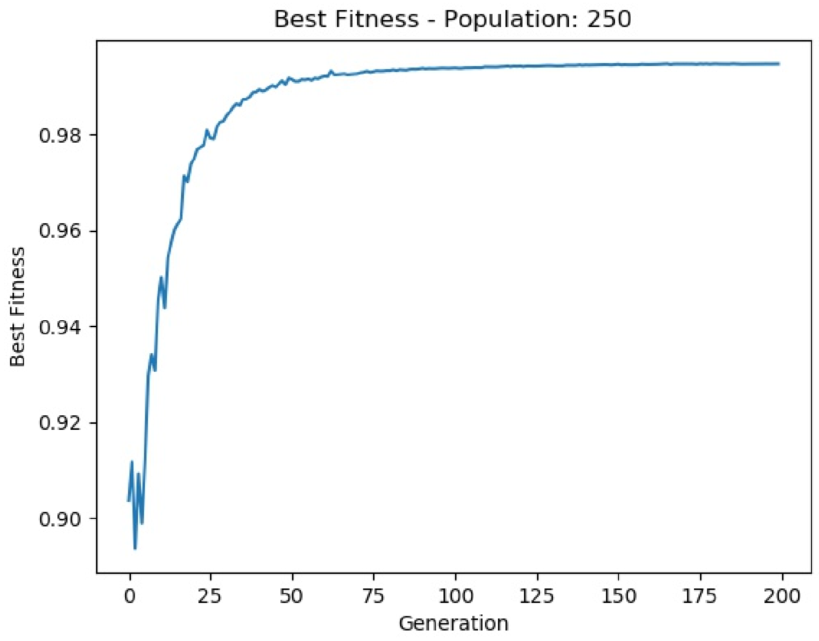


Figure 4.4: Best individual fitness for 2D flowers created using a standard GA

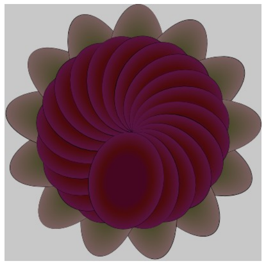
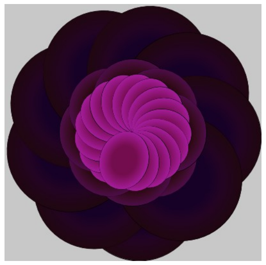
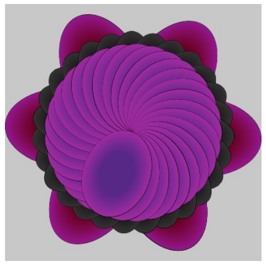
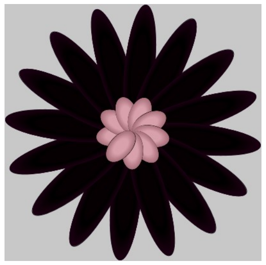
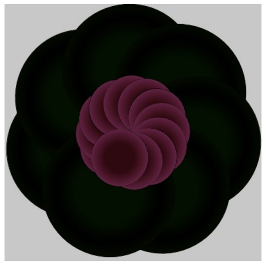
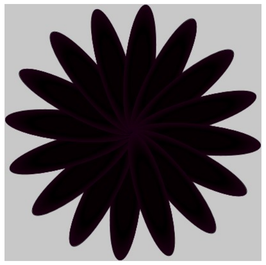
Best In	Trial 1 (Rose)	Trial 2 (Rose)	Trial 3 (Rose)	Trial 4 (Daisy)	Trial 5 (Rose)
Total Population					
Hash-Filtered	NA	NA	NA		NA

Table 4.2: Best 2D Flower in final/hash-filtered populations of standard GA

4.6.2 3D Image Generator

A rose was the best generated individual out of all five GA trials. In the final generation, the fitness, on average, of the best individual in the population was 99.9%. The average fitness of the final population was 99.5%. The average number of goal switches (changes from one class to another during reproduction) was approximately 0. At approximately 20 generations, the algorithm converged on a single value. Like the 2D flowers, a single rose, and its slight variations, came to dominate the whole population. When filtering the population with whash-haar 95% threshold, no images were extracted. Figures 4.5 and 4.6 show the best and average fitnesses as generations increase, averaged over the five trials. Table C.10 shows the best DNN-scored image (basically the only image) from the final populations. There are no hash filtered images as none were selected.

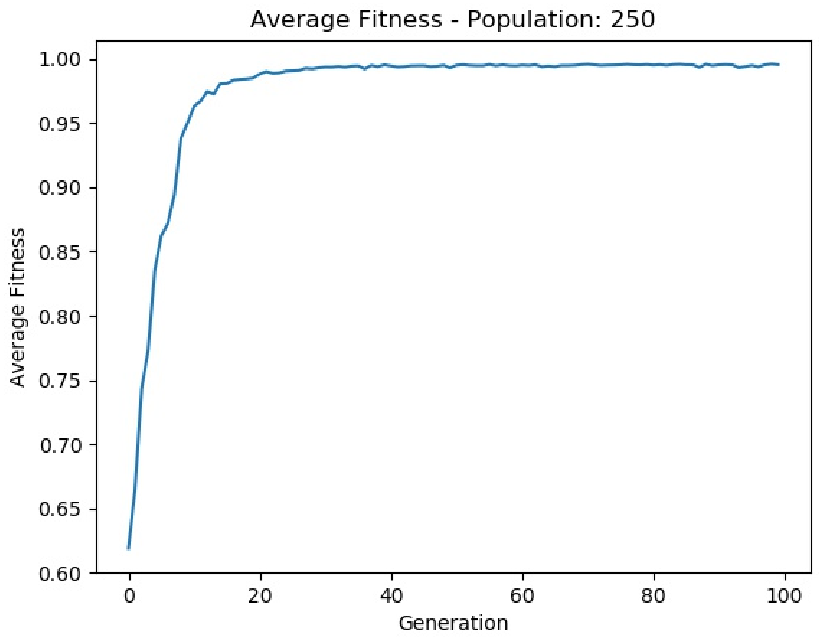


Figure 4.5: Average generational fitness for 3D flowers using a standard GA

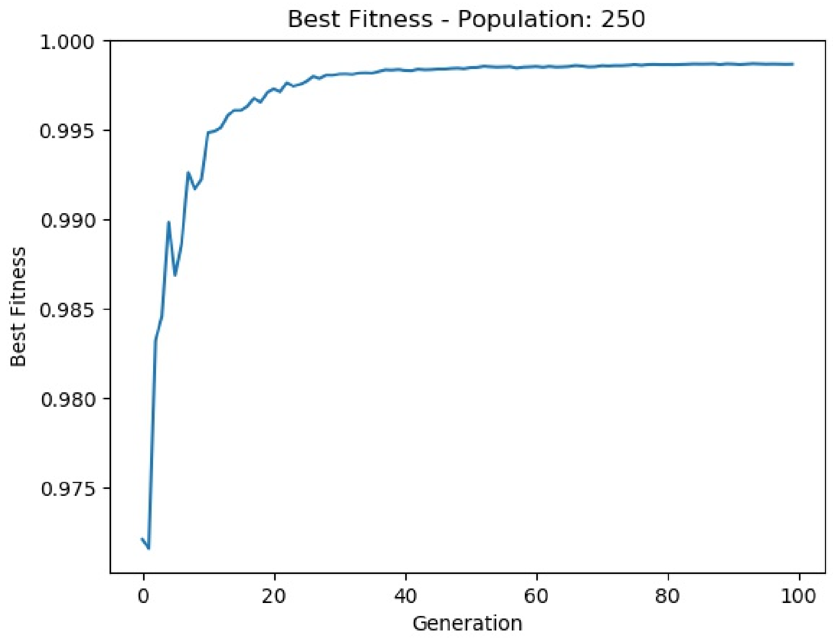


Figure 4.6: Best individual fitness for 3D flowers created using a standard GA

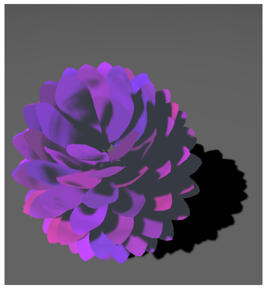
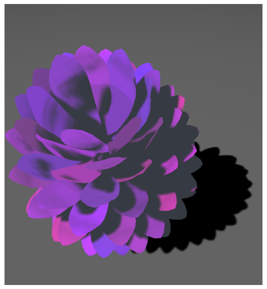
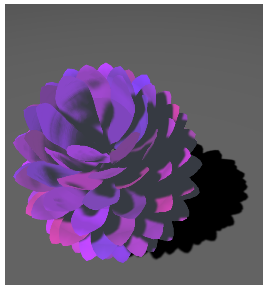
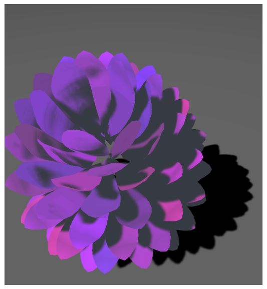
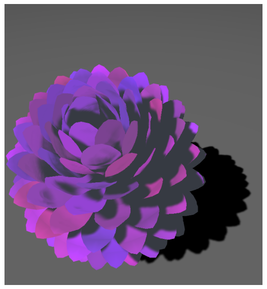
Best In	Trial 1 (Rose)	Trial 2 (Rose)	Trial 3 (Rose)	Trial 4 (Rose)	Trial 5 (Rose)
Total Population					

Table 4.3: Best 3D Flower in final populations of standard GA (no hash-filtered results)

4.7 CVT-MAP-Elites

There are several metrics used to define the diversity amongst solutions found in the CVT-MAP-Elites archives. The first is spread, s, as defined in equation 2.1 found in section 2.2.2. The diversity of an archive is defined as:

$$d = \frac{1}{|\mathcal{A}|} \sum_{i=1}^{\mathbf{b}_{n-1}}{}_{\mathbf{b}_i \in \mathcal{A}} \sum_{j=i+1}^{\mathbf{b}_n}{}_{\mathbf{b}_j \in \mathcal{A}} d_{nn}(\mathbf{b}_i,\, \mathbf{b}_j) \tag{4.1}$$

In equation 4.2 **b** is a behavior (in this case the genome) for any given solutions, A is the archive, n is the max number of solutions in the archive, and $\mathrm{d}_{nn}(\mathbf{b}_i,\ \ \mathbf{b}_j)$ is the Euclidean distance between behavior $\mathbf{b}_i$ and $\mathbf{b}_j$. Centroidal diversity, $\mathcal{C}_d$ is defined analogously as:

$$\mathcal{C}_d = \frac{1}{|\mathcal{C}|} \sum_{i=1}^{\mathbf{c}_{n-1}}{}_{\mathbf{c}_i \in \mathcal{C}} \sum_{j=i+1}^{\mathbf{c}_n}{}_{\mathbf{c}_j \in \mathcal{C}} d_{nn}(\mathbf{c}_i,\, \mathbf{c}_j) \tag{4.2}$$

c is a randomly generated real-number vector where $|\mathbf{c}|$ is equivalent to the length of behaviors to be generated in the problem set. $\mathcal{C}$ is the set of centroidal clusters generated by the CVT-MAP-Elites algorithm.

The Solution-Centroid Diversity Ratio $\mathcal{SC}_{dr}$, can be defined as

$$\mathcal{SC}_{dr} = \frac{d}{\mathcal{C}_d} \tag{4.3}$$

If it is assumed that the centroid clusters are uniformly distributed throughout the solution space (as they should be), then we can tell how diverse the solutions are by this score. Scores greater than 1 imply solution clustering in the search space.

This work defines another spread term, Niche Inhabitation Spread ($\mathcal{NI}_s$), defined as:

$$\mathcal{NI}_s = \frac{\frac{1}{|\mathcal{A}|} \Sigma_{\mathbf{b} \in \mathcal{A}}\, d_{nn}(\mathbf{b},\, \mathcal{A})}{d} \tag{4.4}$$

The top half of the equation calculates the average Euclidean distance from the behaviors of solutions in the archive to their centroid clusters, identically to numeration in equation 2.1. Instead of dividing by the max Euclidean distance of a solution's behavior to its centroid, the numerator is divided by the diversity of the archive. Similar to equation 4.3, this can be an indication of how well spread solutions are in space. Higher values indicate solution clustering in the search space.

Instead of the archive being transformed in generations, CVT-MAP-Elites is changed using batches.

The batch number is different for 2D flowers and 3D flowers. In a standard GA, individuals are pulled from previous generations to take part in the new generation. Generations do not really exist in MAP-Elites and CVT-MAP-Elites. Instead, batches of individuals are chosen and added to archive throughout the algorithm's run-time. Due to hardware limitations, for 3D flowers, the batch size is 50, and 500 different batches are created for a total of 25,000 different flowers generated over the algorithm's life-cycle. For 2D flowers, the batch size is 50, and 1000 different batches are generated for a total of 50,000 flowers over the algorithm's life-cycle. Selection is performed randomly from the archive as dictated by the original algorithm. Crossover is performed at rate of 30%. Mutation is performed at a rate of 10% for the 3D flowers, and 5% for the 2D flowers, with a 10% change to the gene when it is mutated.

4.7.1 2D Flower Generator Without Hashing Fitness

During this experiment, hashing was only used as a final filter after evolution had already taken place. Only the DNN score from 0 to 1, in this case interpreted as a percentage, is used. Based on preliminary experiments, the whash-haar perceptual hash algorithm appeared to perform better than its counterparts (ahash, dhash, whash-haar) when it came to filtering. When testing a filter was applied at thresholds of 85%, 90%, and 95% similarity, only the whash-haar algorithm allowed enough images to pass through the filter to be analyzed as a population. As such, the filtered hashing photos shown are the ones filtered from that algorithm. Five trials were performed. The metrics shown have been averaged over the five trials.

An average of 80.0% of the 5000 niches are filled over the five trials. When analyzing the whole population, the best individual is a rose three out of five times. Two out of five times it is a daisy. As this metric keeps getting reported, remember that although the best flowers are roses and daisies, this does not mean there are no good sunflowers or dandelions. Consult the class distribution histogram and the class fitness figures, such as figures 4.7 and 4.8, to get a better understanding of the distribution makeup and fitness. These are available for all the experiments. The average fitness of the best flower is 98.2%. Average fitness, for the whole archive, over five trials, is 76%. Breaking down the class distributions, 10.6% of the archive is daisies, 19.4% is dandelions, 68.0% is roses, and 2.0% is sunflowers. Daisies have an average fitness (confidence value) of 82%, dandelions an average of 81%, roses an average of 74%, and sunflowers an average of 61%. The average number of goal switches is 0.67. $\mathcal{SC}_{dr}$ is 1.04. s is equal to 0.83 and $\mathcal{NI}_s$ is approximately 0.

The top 5% of all solutions are also analyzed as a sub-population. The average fitness of the top 5% of all solutions is 93%. The average number of goal switches is 0.8. Daisies make up approximately 50.8% of the population, dandelions make up approximately 6.3%, roses make up approximately 42.6%, and sunflowers make up approximately 3%. The average fitness of daisies is 93.5%, dandelions is 91.3%, roses is 93%, and sunflowers is 92%. $\mathcal{SC}_{dr}$ is 0.05. s is equal to 0.86 and $\mathcal{NI}_s$ is approximately 0.

Images that were 95% similar, and above, to their appropriate class comparison image in table 4.1, when using a whash-haar filter, were isolated and analyzed independently as a whole population. On average, 399.6 images were extracted from the archive population when using the whash-haar hash as a filter. The average fitness of the best individual is 97.4%. The average number of goal switches is 1.1. In this extricated population, 34.5% are daisies, 64.9% are dandelions, 0% are roses, and 53% are sunflowers. Daisies have an average fitness of 82.5%, dandelions an average fitness of 81.6%, and sunflowers an average of 53%. $\mathcal{SC}_{dr}$ is 0.1. s is equal to 0.86 and $\mathcal{NI}_s$ is approximately 0.

Figure 4.7 shows the percentile class distributions from the different sub-populations. Figure 4.8 shows the normalized frequencies of the different ranges of fitness values from all the solutions in the final archives across the five trials. Tables 4.4 - 4.7 contain the the best scored (from the DNN) images from the overall population and the sub-population filtered with a whash-haar 95% filter threshold. Flower class types not found in the population, or sub-population, after filtering using the hash values, are indicated with an "NA."

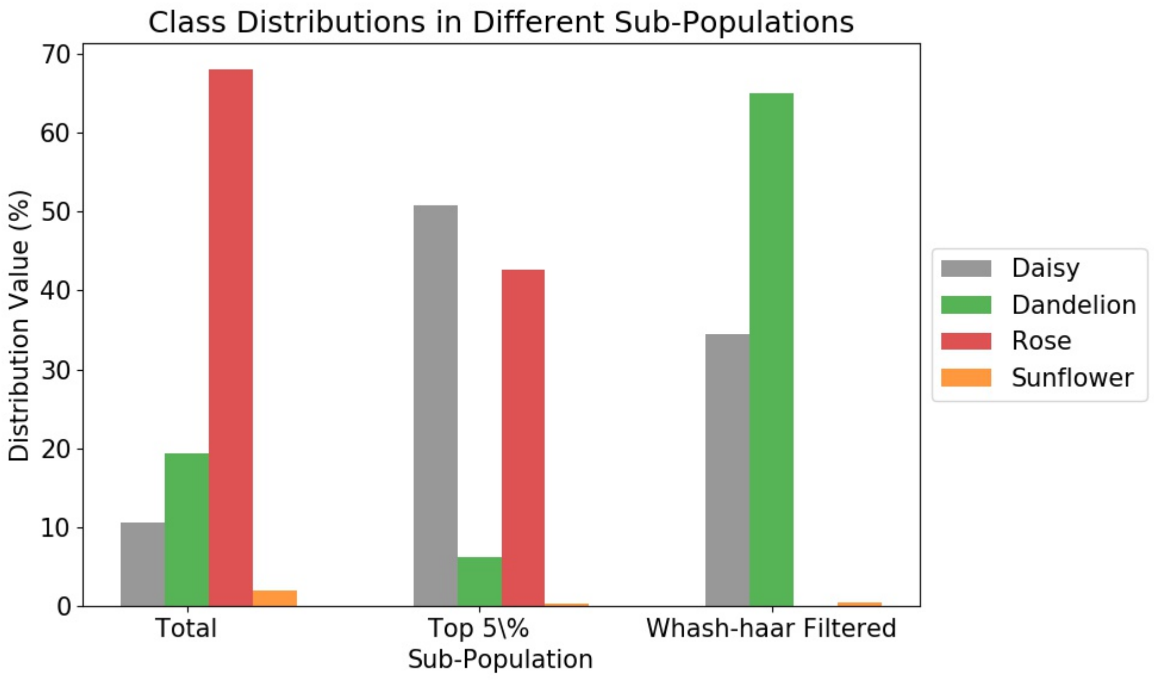


Figure 4.7: The distribution (in %) of different class types in different 2D flower sub-populations (what percentage of the sub-population is flower type x?)

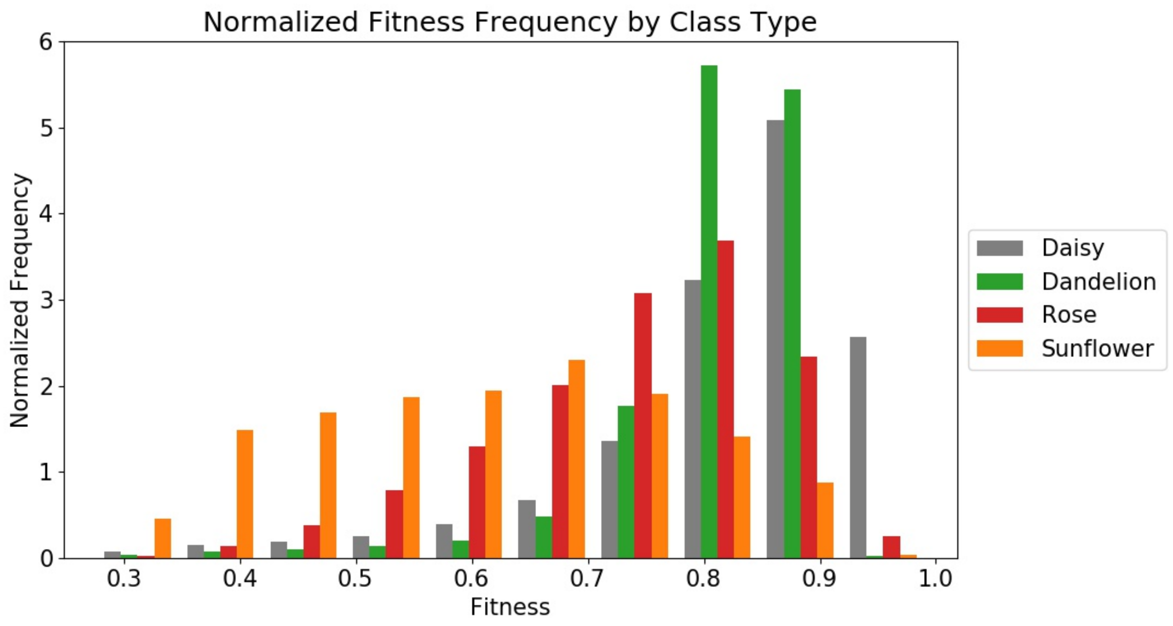


Figure 4.8: Normalized frequencies of fitness values for 2D flowers in each class type (all flowers over five trials)

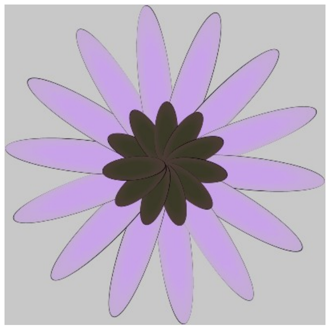
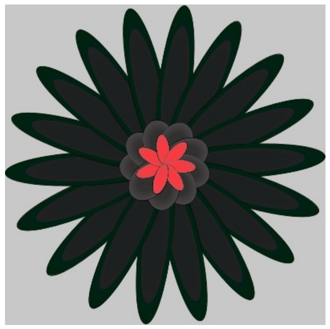
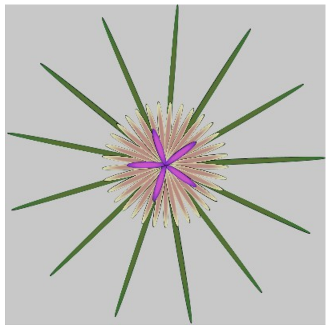
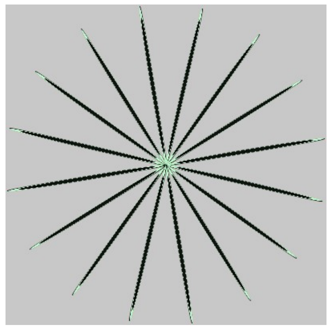
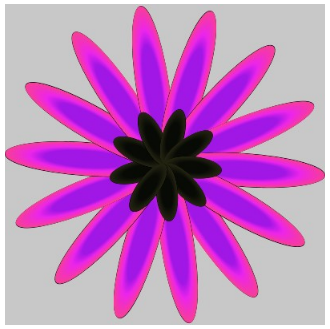
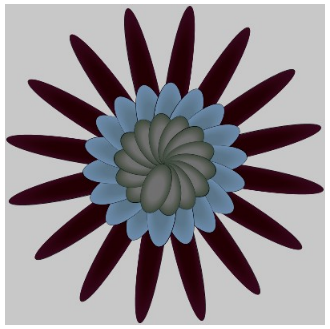
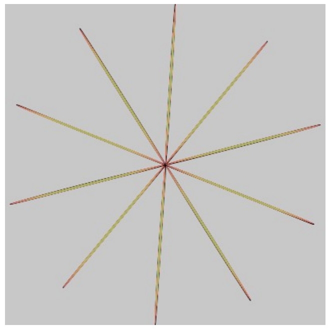
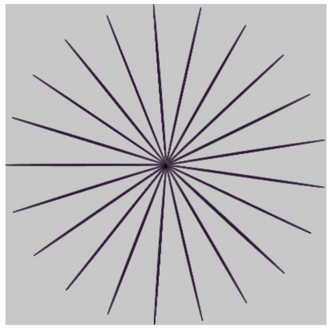
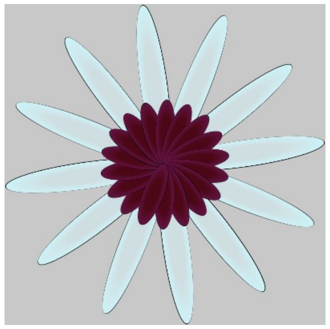
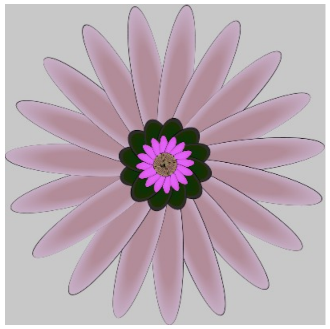
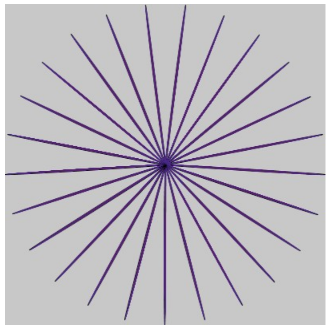
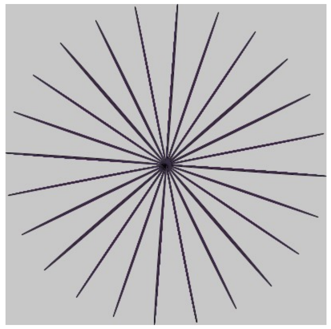
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 1				
Trial 2				
Trial 3				

Table 4.4: Best DNN scored 2D flower of each class type from the total and hash-filtered populations

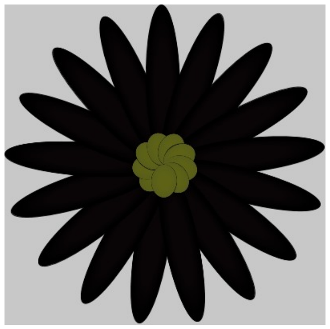
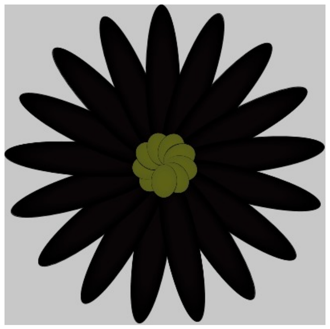
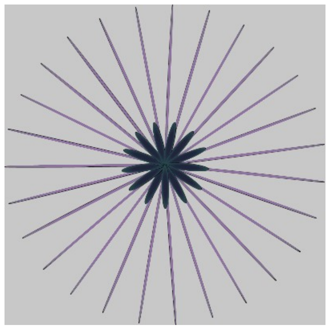
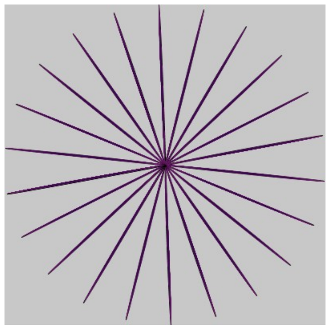
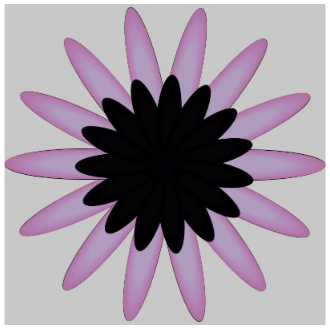
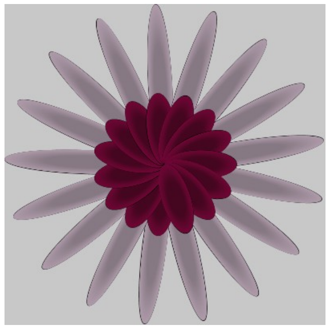
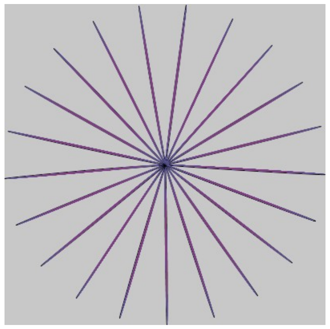
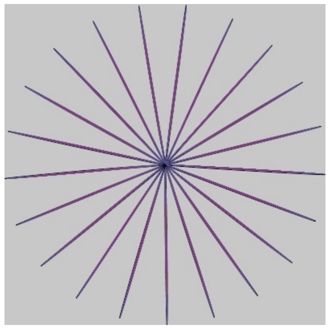
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 4				
Trial 5				

Table 4.5: Best DNN scored 2D flower of each class type from the total and hash-filtered populations

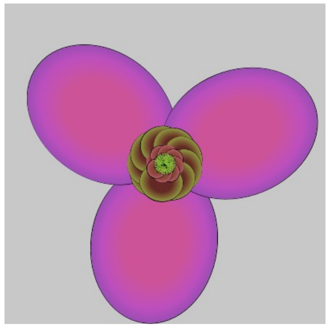
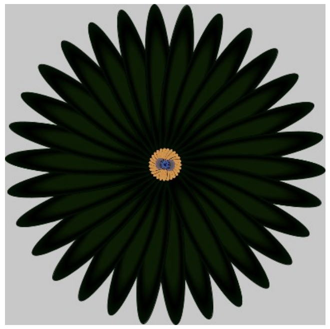
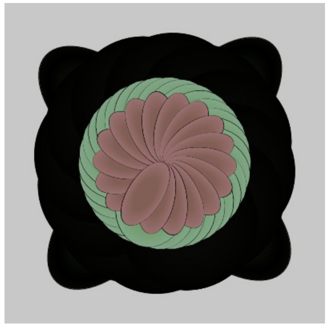
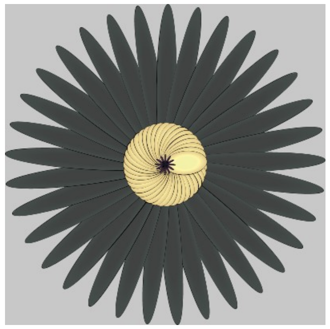
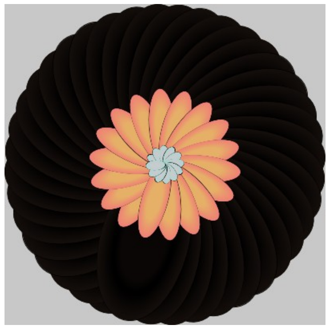
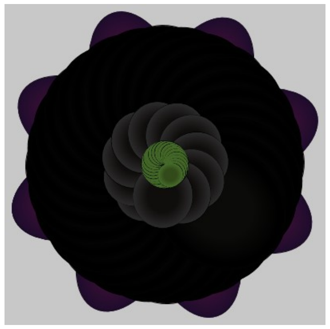
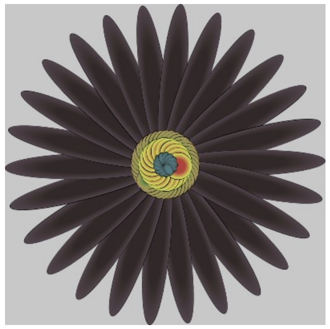
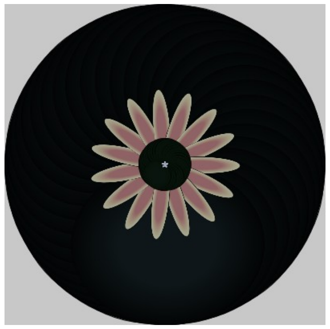
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 1		NA		NA
Trial 2		NA		
Trial 3		NA		

Table 4.6: Best DNN scored 2D flower of each class type from the total and hash-filtered populations

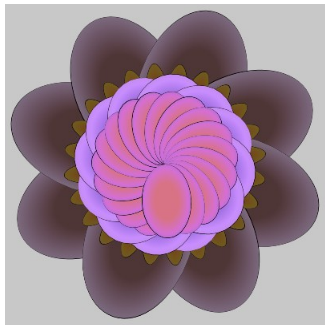
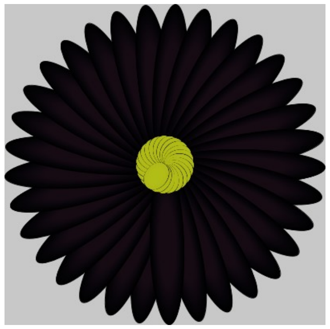
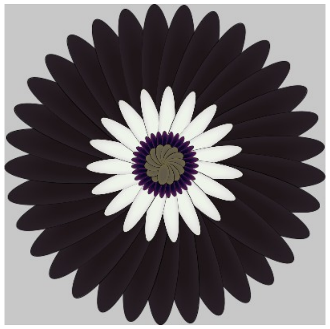
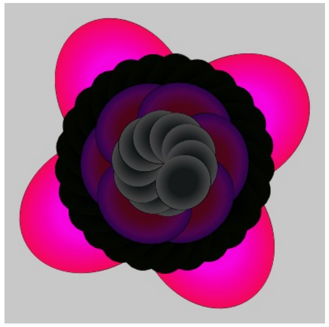
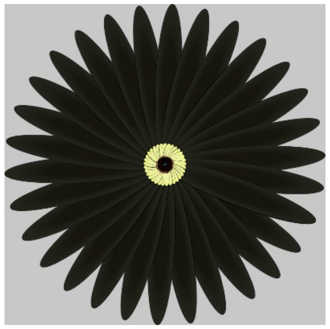
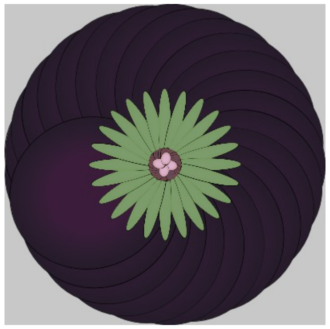
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 4		NA		
Trial 5		NA		

Table 4.7: Best DNN scored 2D flower of each class type from the total and hash-filtered populations

4.7.2 3D Flower Generator Without Hashing Fitness

Hashing was utilized in exactly the same way as section 4.7.1. In addition, the same sub-populations types were analyzed using the same metrics. Five trials were performed with metrics being averaged over those five trials.

An average of 96.8% of 5000 niches were filled over each of the different trials. When analyzing the whole population, the fitness of the best individual is, on average, 99.3%. All five times that individual is a rose. The average fitness for the whole population is 75.8%. The average number of goal switches is 0.96. The archive is comprised of 0.2% daisies, 42.1% dandelions, 52.3% roses, and 5.3% sunflowers. Daisies have an average fitness of 50.9%, dandelions an average of 76.4%, roses an average of 76.6%, and sunflowers an average of 64.9%. $\mathcal{SC}_{dr}$ is 1.03. s is equal to 0.67 and $\mathcal{NI}_s$ is approximately 0.

The average fitness for the top 5% of individuals is 96.7%. An average of 0.85 goal switches occur. Of the top 5% of solutions, 26.7% are dandelions and 73.3% are roses. There are no daisies or sunflowers. Roses have an average fitness of 96.7%, and dandelions have an average fitness of 96.1%. $\mathcal{SC}_{dr}$ is 0.05, s is equal to 0.72, and $\mathcal{NI}_s$ is approximately 0.

When filtered using a 95% whash-haar threshold, an average of 6.6 images were extracted from the archive. The average fitness, over five trials, is 74.3%. The average number of goal switches is 0.93. Dandelions make up 92.1% of all these hashed images. Roses make up the other 7.9%. The average fitness value for a dandelion in this extract is 75%, and for roses the fitness is 67.6%. $\mathcal{SC}_{dr}$ is 0.001. s is equal to 0.87 and $\mathcal{NI}_s$ is approximately 0.

Figure 4.9 shows the percentile class distributions from the different sub-populations. Figure 4.10 shows the normalized frequencies of the different ranges of fitness values from all the solutions in the final archives across the five trials. Tables 4.8 - 4.11 contain the the best scored (from the DNN) images from the overall population and the sub-population filtered with a whash-haar 95% filter threshold. Flower class types not found in the population or sub-population are indicated with an “NA.”

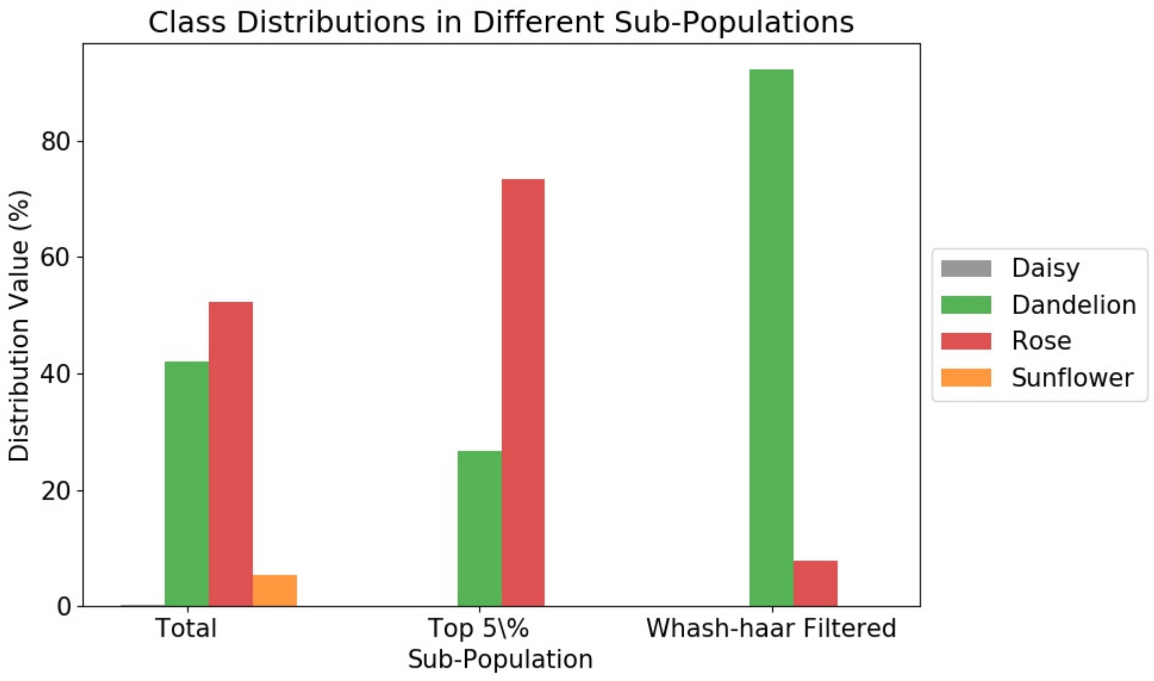


Figure 4.9: The distribution (in %) of different class types in different 3D flower sub-populations (what percentage of the sub-population is flower type x?)

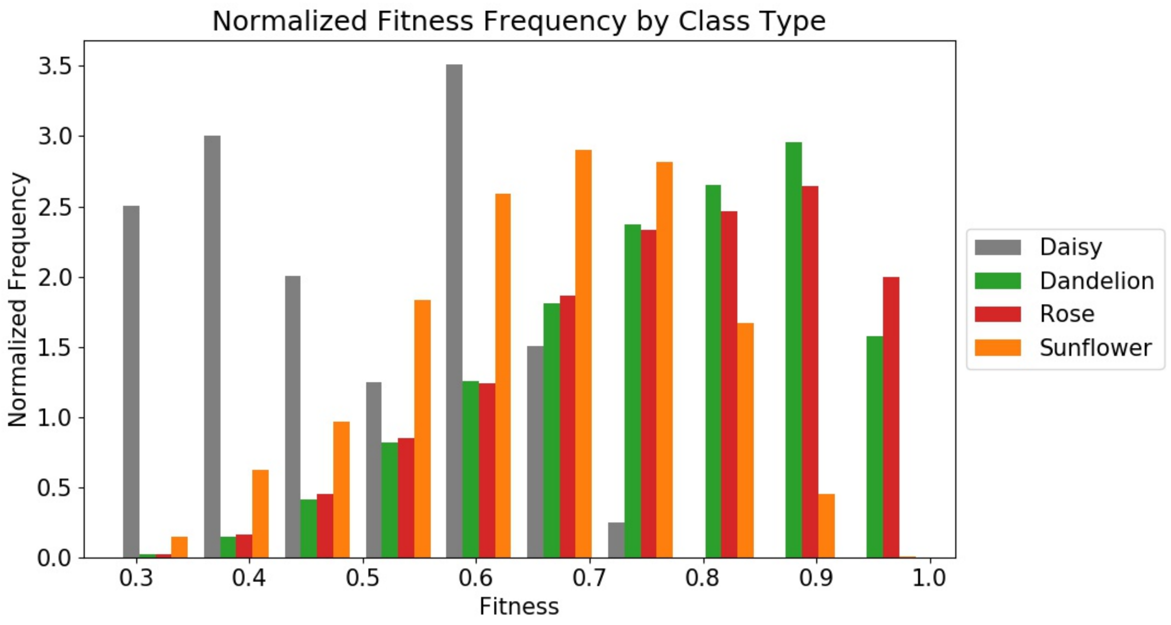


Figure 4.10: Normalized frequencies of fitness values for 3D flowers in each class type (all flowers over five trials)

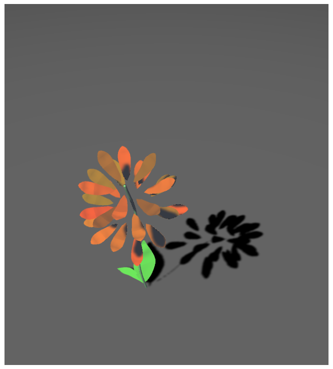
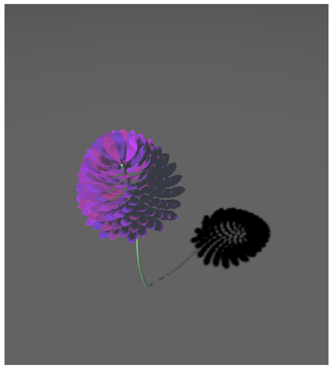
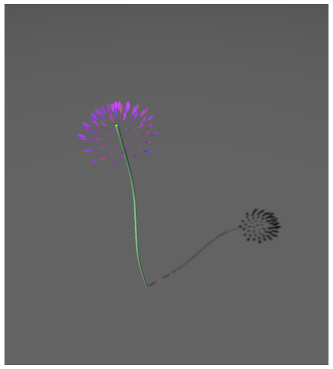
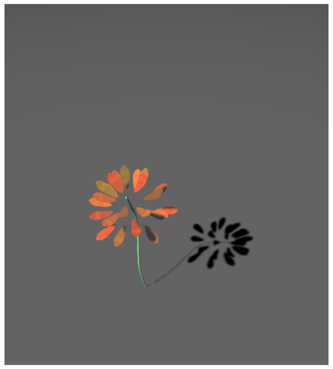
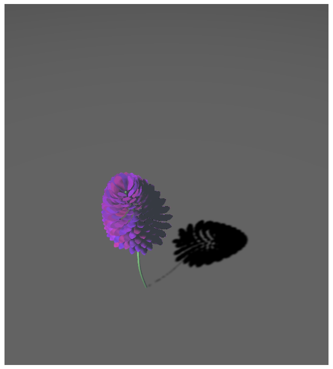
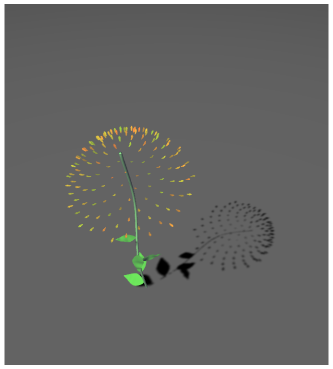
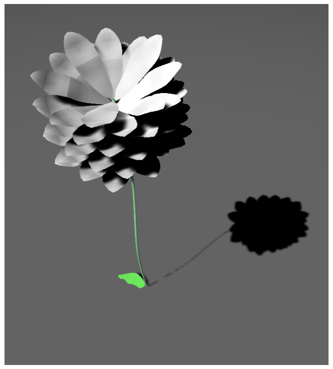
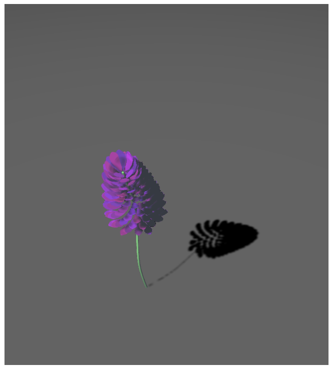
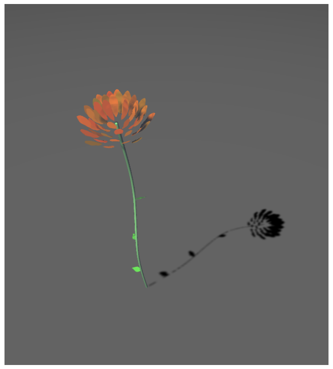
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 1		NA		
Trial 2		NA		
Trial 3		NA		

Table 4.8: Best DNN scored 3D flower of each class type from the total and hash-filtered populations

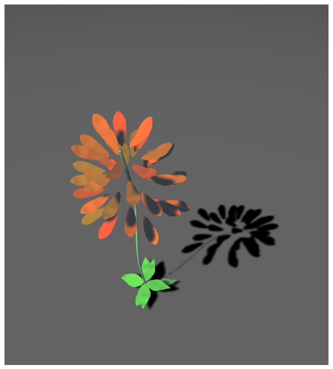
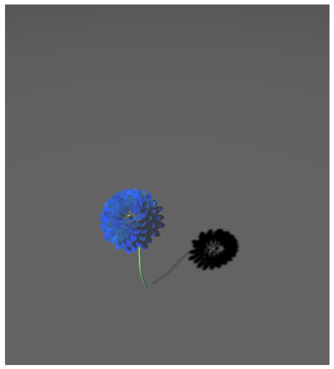
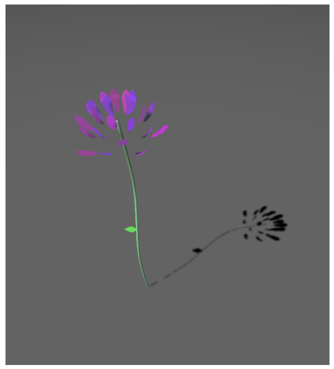
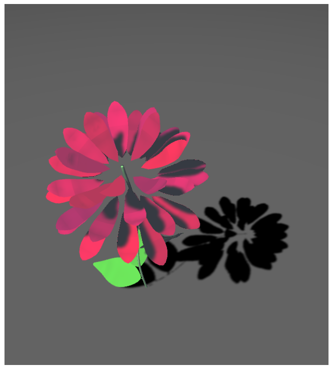
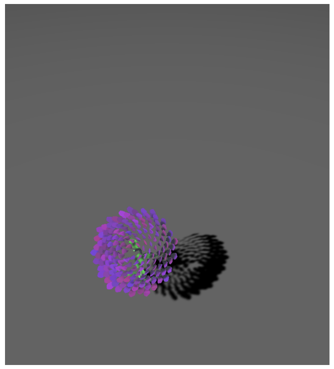
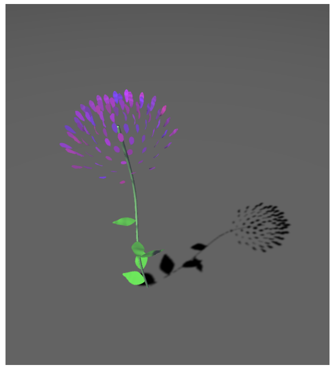
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 4		NA		
Trial 5		NA		

Table 4.9: Best DNN scored 3D flower of each class type from the total and hash-filtered populations

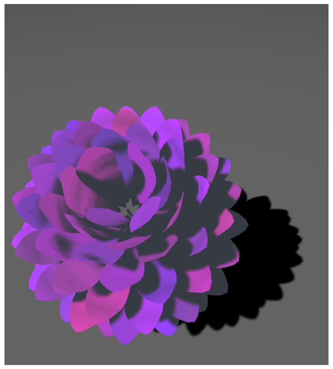
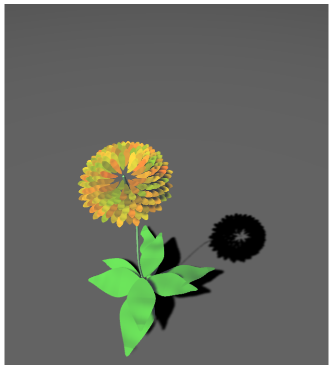
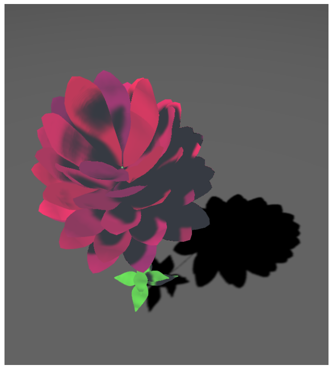
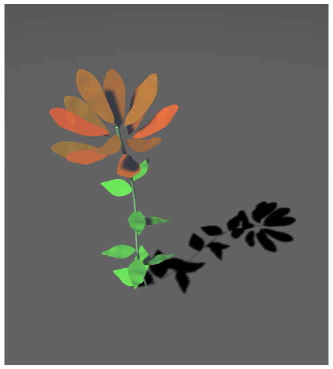
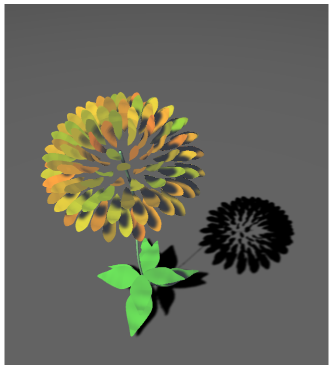
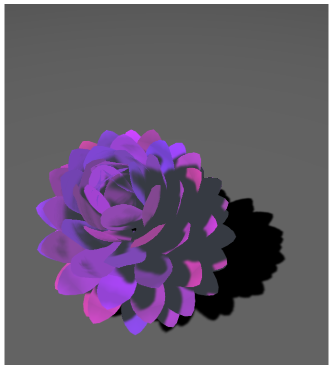
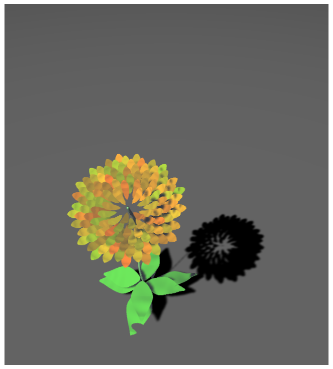
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 1		NA		NA
Trial 2				NA
Trial 3		NA		NA

Table 4.10: Best DNN scored 3D flower of each class type from the total and hash-filtered populations

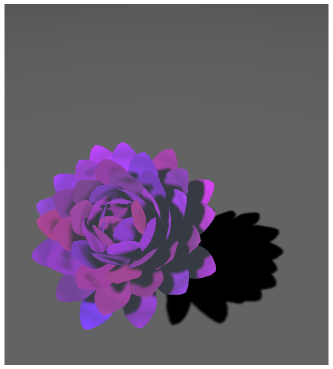
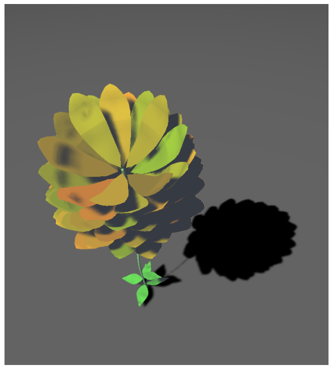
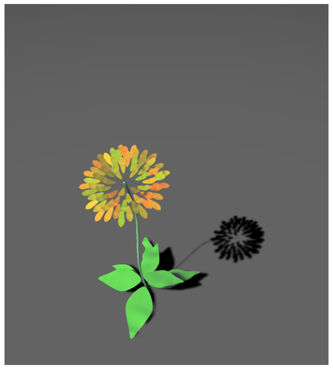
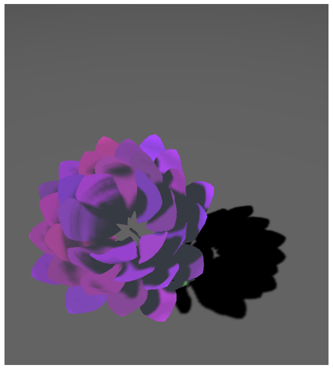
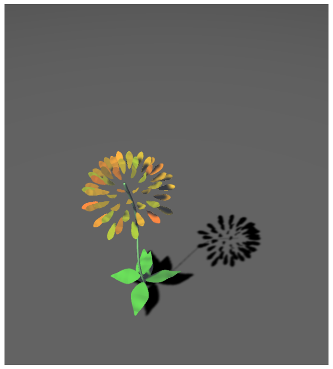
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 4				NA
Trial 5		NA		NA

Table 4.11: Best DNN scored 3D flower of each class type from the total and hash-filtered populations

4.7.3 2D Flower Generator With Composite Hashing Score

In the next four experiments, the four hashing algorithms from section 2.4 were added to the fitness value produced by the DNN. The added hashing score is the percentage of identical bits when comparing two different 64-bit image hash strings. This score is from 0 to 1. The total fitness, no longer a percentage but a composition of percentages, can be from 0 to 2.0. The DNN score is from 0 to 1.0 and the hashing score can also be from 0 to 1.0, resulting in the range stated previously. Aside from this hashing score, all the same parameters are used from section 4.7.1. Whash-haar is still used to filter images from the final archive. Five trials were implemented with experimental metrics being averaged over those five trials.

Ahash Similarity Score Added

An average of 80% of 5000 niches were filled over each of the different trials. When analyzing the whole population, the fitness of the best individual is, on average, 1.93. Four out of five times, the best individual is a rose. One out of the five it is a dandelion. The average fitness for the whole population is 1.40. The average number of goal switches is 1.16. The archive is comprised of 3.2% daisies, 41.4% dandelions, 46.7% roses, and 8.7% sunflowers. Daisies have an average fitness of 1.16, dandelions an average of 1.54, roses an average of 1.30, and sunflowers an average of 1.33. $\mathcal{SC}_{dr}$ is 1.06. s is equal to 0.83 and $\mathcal{NI}_s$ is approximately 0.

The average fitness for the top 5% of individuals is 1.82. An average of 1.12 goal switches occur. Of the top 5% of solutions, 0.1% are daisies, 62.2% are dandelions, 32.8% are roses, and 4.9% are sunflowers. Daisies have an average fitness of 1.78, dandelions an average fitness of 1.81, roses an average of 1.83, and sunflowers an average of 1.82. $\mathcal{SC}_{dr}$ is 0.05. s is equal to 0.86 and $\mathcal{NI}_s$ is approximately 0.

When filtered using a 95% whash-haar threshold, an average of 1356.4 images were extracted from the archive. The average fitness of the best individual is 1.91. The average number of goal switches is 1.46. In this extricated population, 3.0% are daisies, 96.5% are dandelions, 0.01% are roses, and 0.4% are sunflowers. Daisies have an average fitness of 1.11, dandelions an average fitness of 1.59, roses an average of 1.47, and sunflowers an average of 1.18. $\mathcal{SC}_{dr}$ is 0.35. s is equal to 0.85 and $\mathcal{NI}_s$ is approximately 0.

Figure 4.11 shows the percentile class distributions from the different sub-populations. Figure 4.12 shows the normalized frequencies of the different ranges of fitness values from all the solutions in the final archives across the five trials. Tables 4.12 - 4.15 contain the images with the best composite score from both the DNN and the ahash algorithm. Images are filtered into a sub-population with a whash-haar 95% threshold. Flower class types not found in the population or sub-population are indicated with an "NA."

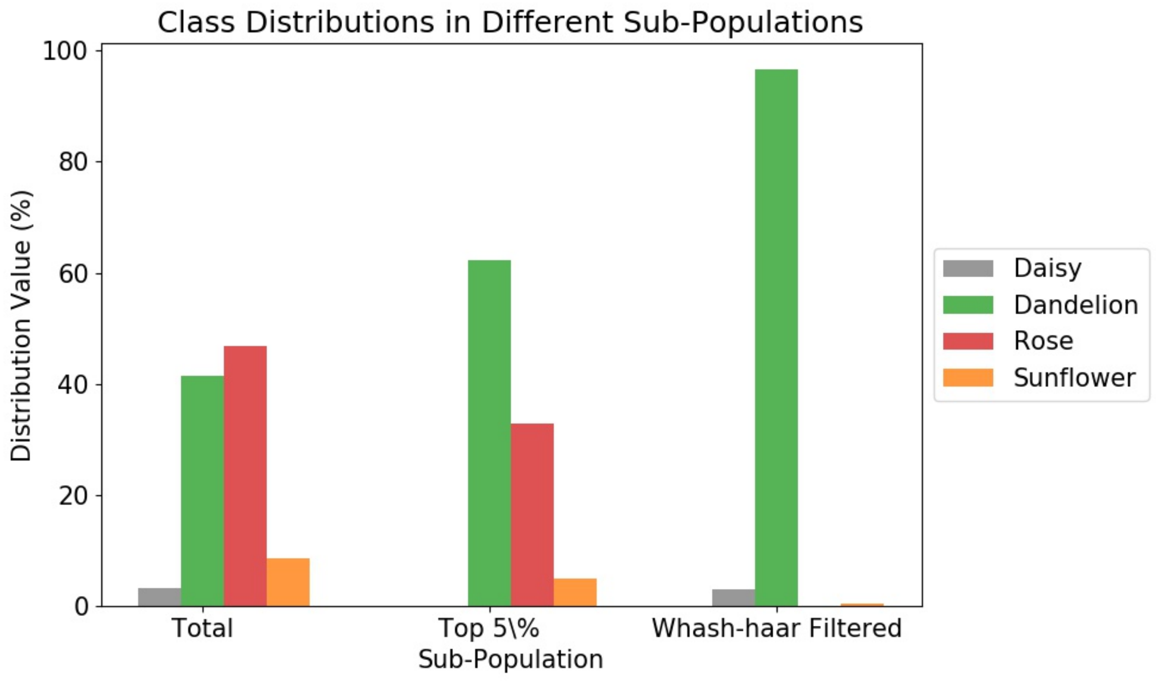


Figure 4.11: The distribution (in %) of different class types in different DNN-ahash composite scored 2D flower sub-populations (what fraction of the sub-population is flower type x?)

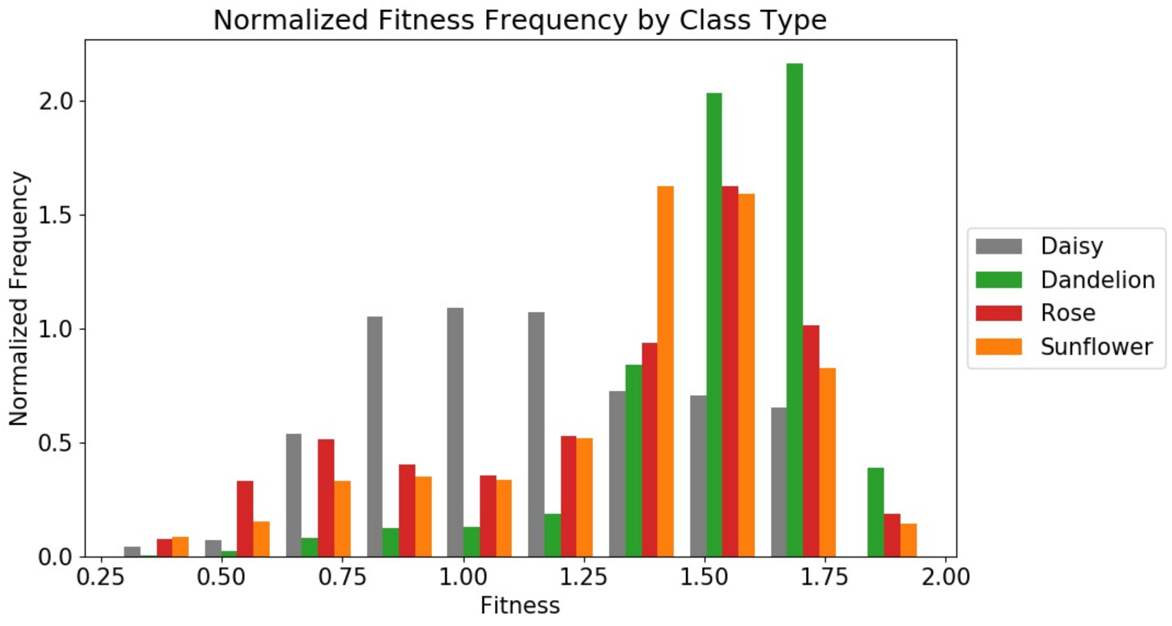


Figure 4.12: Normalized frequencies of fitness values for DNN-ahash composite-scored 2D flowers in each class type (all flowers over five trials)

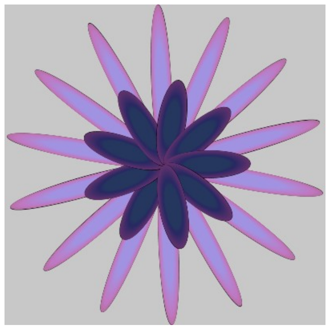
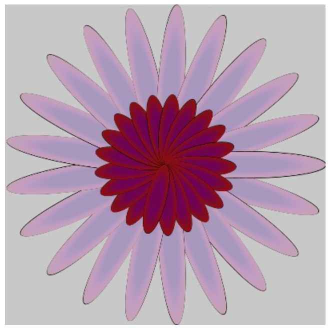
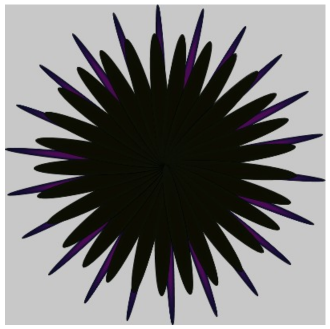
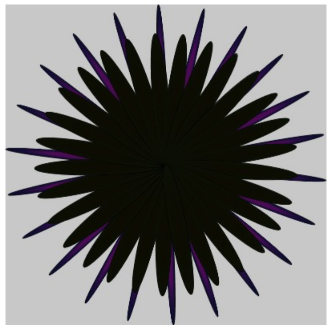
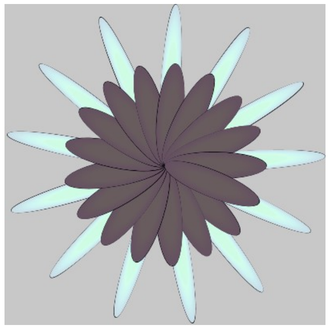
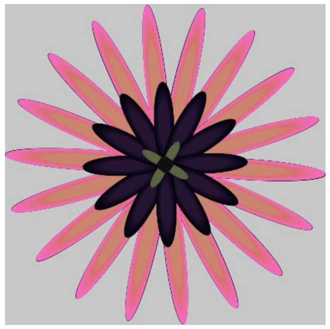
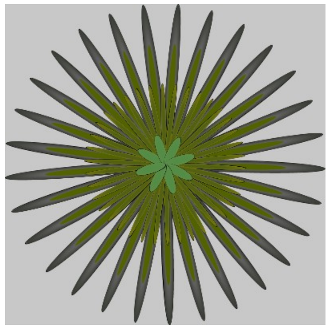
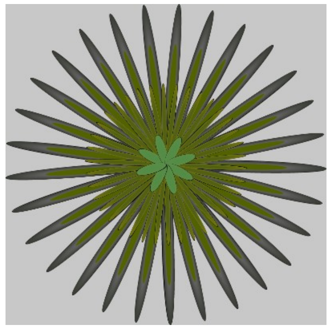
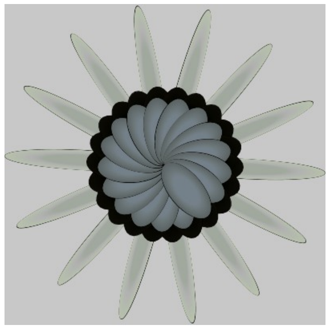
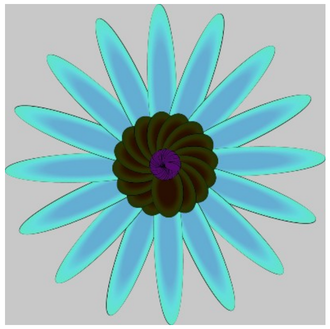
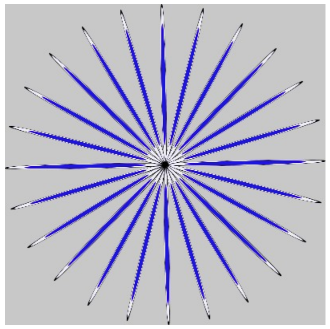
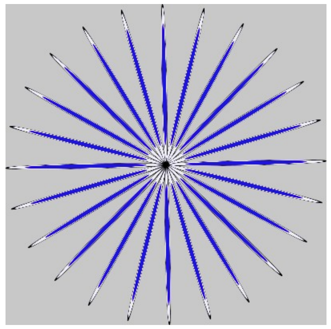
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 1				
Trial 2				
Trial 3				

Table 4.12: Best DNN-ahash scored 2D flower of each class type from the total and hash-filtered populations

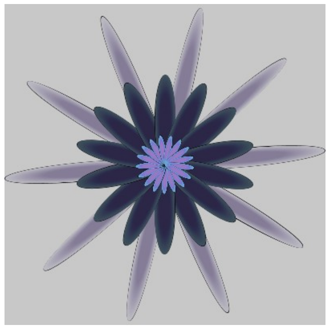
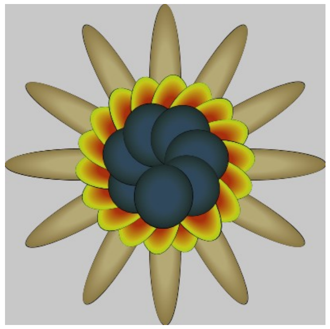
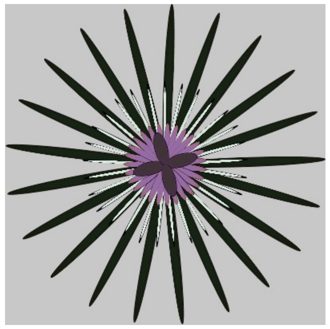
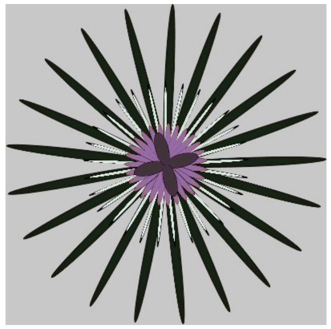
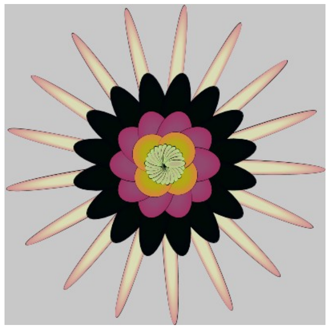
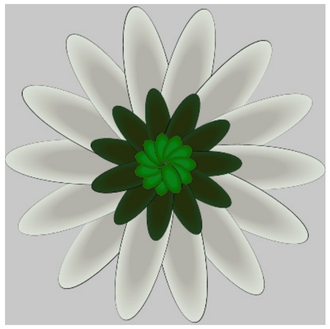
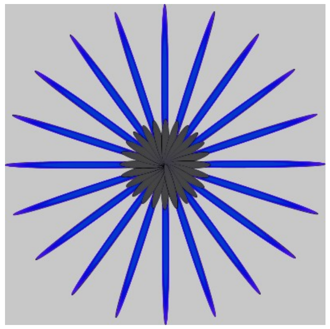
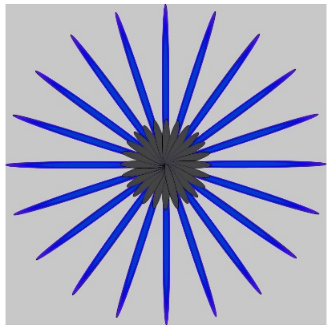
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 4				
Trial 5				

Table 4.13: Best DNN-ahash scored 2D flower of each class type from the total and hash-filtered populations

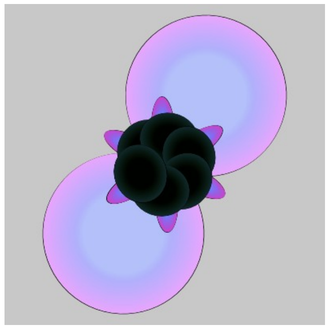
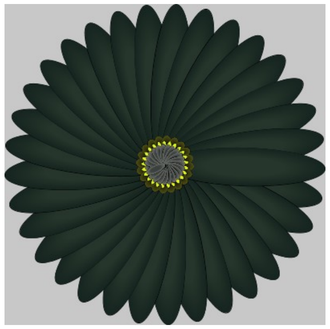
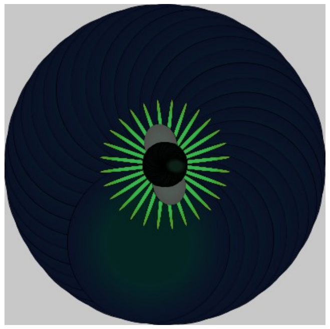
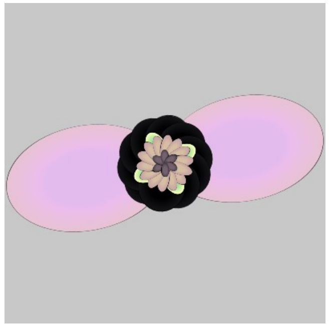
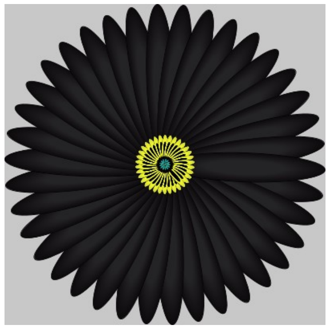
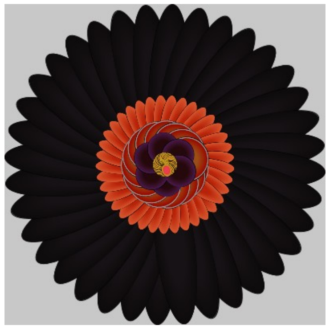
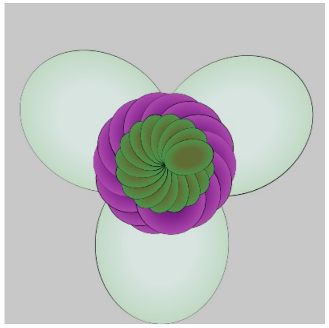
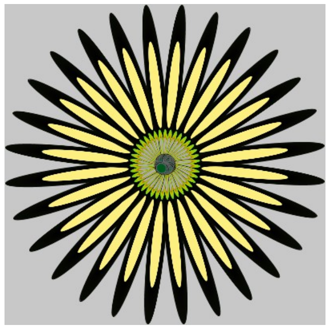
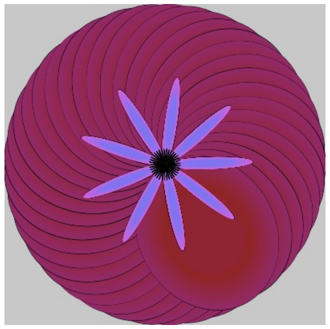
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 1		NA		
Trial 2		NA		
Trial 3		NA		

Table 4.14: Best DNN-ahash scored 2D flower of each class type from the total and hash-filtered populations

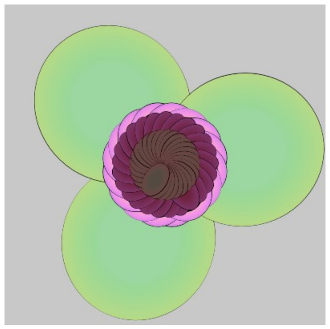
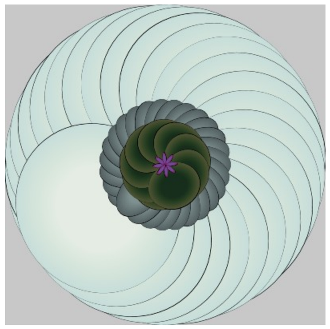
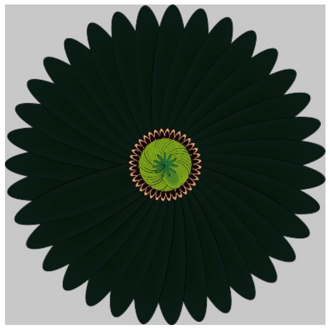
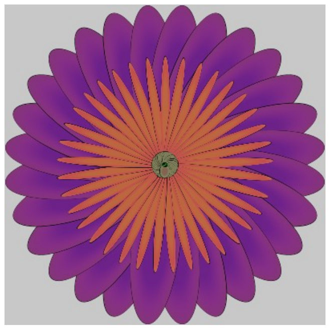
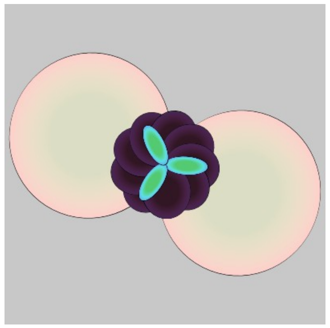
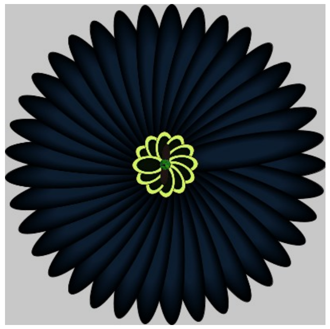
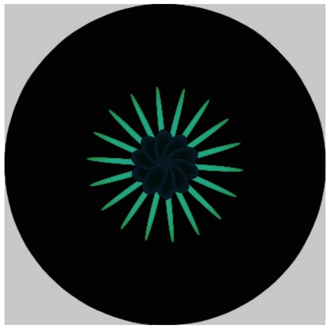
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 4				
Trial 5		NA		

Table 4.15: Best DNN-ahash scored 2D flower of each class type from the total and hash-filtered populations

Dhash Similarity Score Added

An average of 80% of 5000 niches were filled over each of the different trials. When analyzing the whole population, the fitness of the best individual is, on average, 1.58. Two of of five times the best flower is a daisy, two out of five times it is dandelion, and the final time it is a sunflower. The average fitness for the whole population is 0.99. The average number of goal switches is 1.25. The archive is comprised of 12.3% daisies, 46.8% dandelions, 28.4% roses, and 12.4% sunflowers. Daisies have an average fitness of 1.06, dandelions an average of 1.09, roses an average of 0.79, and sunflowers an average of 0.99. $\mathcal{SC}_{dr}$ is 1.06. s is equal to 0.82 and $\mathcal{NI}_s$ is approximately 0.

The average fitness for the top 5% of individuals is 1.34. An average of 1.33 goal switches occur. Of the top 5% of solutions, 37.6% are daisies, 40.8% are dandelions, 7.7% are roses, and 13.9% are sunflowers. Daisies have an average fitness of 1.36, dandelions an average fitness of 1.33, roses an average of 1.33, and sunflowers an average of 1.34. $\mathcal{SC}_{dr}$ is 0.05. s is equal to 0.86 and $\mathcal{NI}_s$ is approximately 0.

When filtered using a 95% whash-haar threshold, an average of 1270.4 images were extracted from the archive. The average fitness of the best individual is 1.10. The average number of goal switches is 1.4. In this extricated population, 20.0% are daisies, 76.7% are dandelions, and 3.4% are sunflowers. There are no roses. Daisies have an average fitness of 1.10, dandelions an average fitness of 1.10, and sunflowers an average of 1.01. $\mathcal{SC}_{dr}$ is 0.33. s is equal to 0.84 and $\mathcal{NI}_s$ is approximately 0.

Figure 4.13 shows the percentile class distributions from the different sub-populations. Figure 4.14 shows the normalized frequencies of the different ranges of fitness values from all the solutions in the final archives across the five trials. Tables 4.16 - 4.19 contain the images with the best composite score from both the DNN and the dhash algorithm. Images are filtered into a sub-population with a whash-haar 95% threshold. Flower class types not found in the population or sub-population are indicated with an "NA."

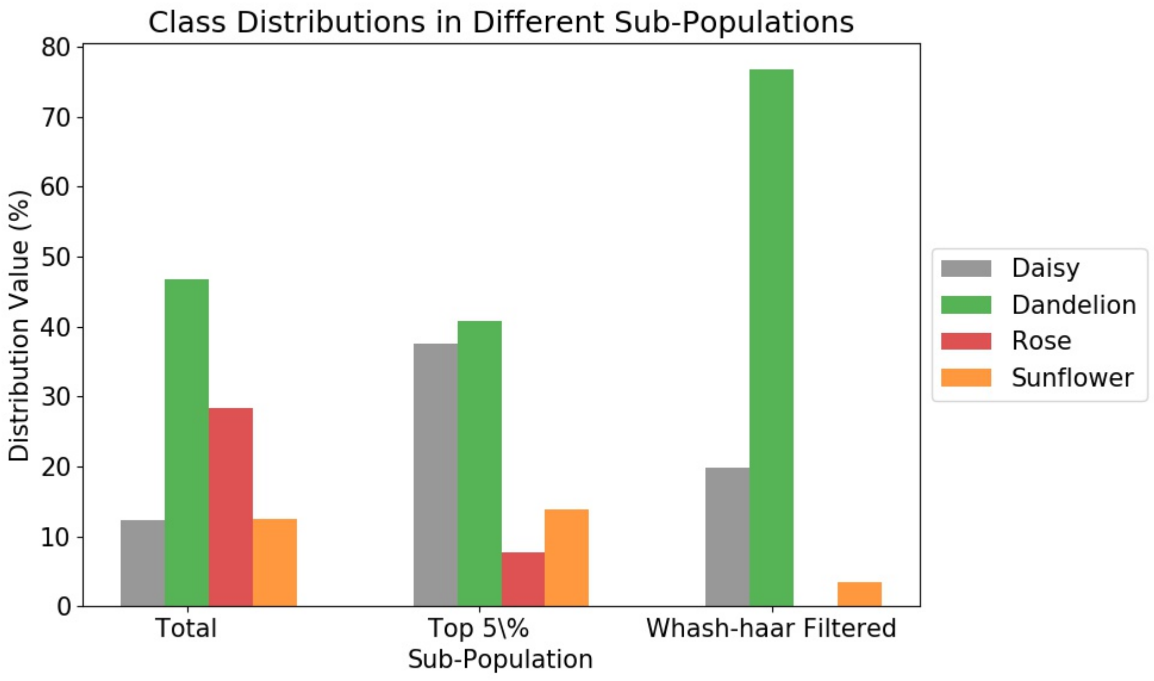


Figure 4.13: The distribution (in %) of different class types in different DNN-dhash composite scored 2D flower sub-populations (what fraction of the sub-population is flower type x?)

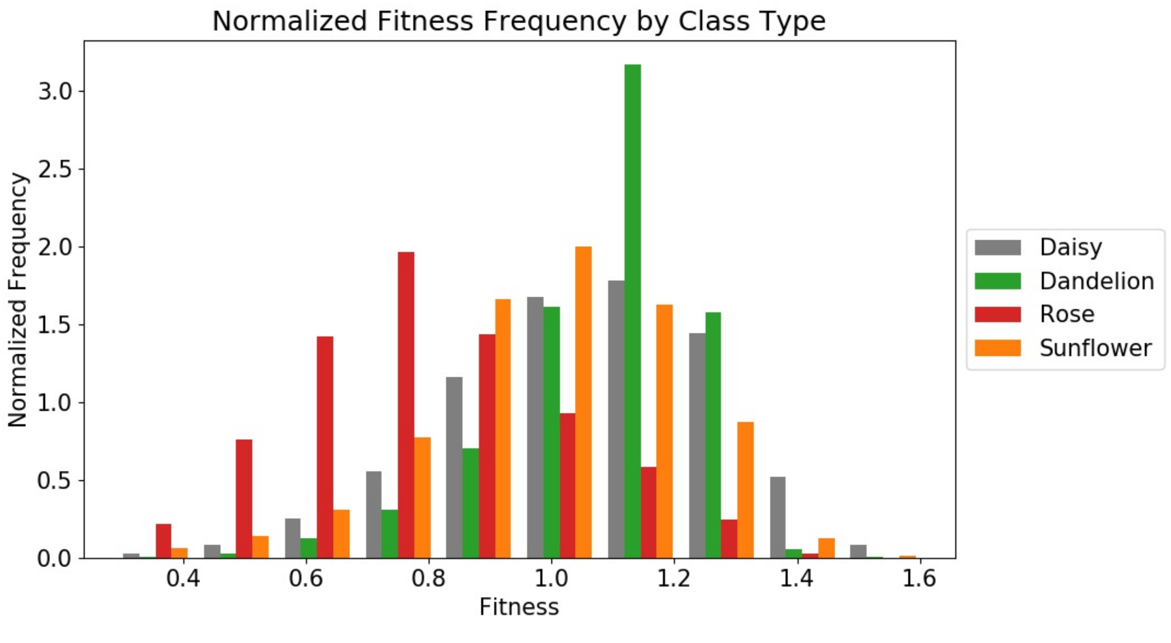


Figure 4.14: Normalized frequencies of fitness values for DNN-dhash composite-scored 2D flowers in each class type (all flowers over five trials)

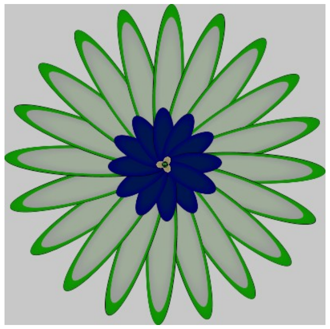
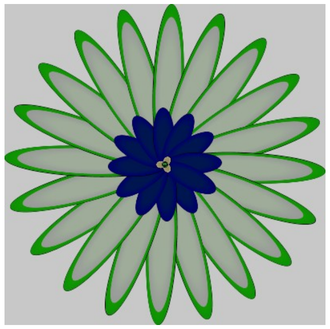
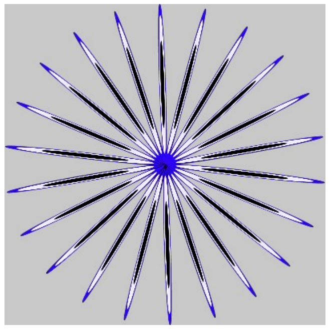
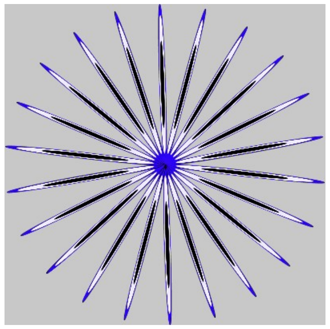
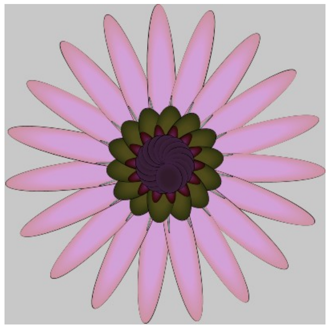
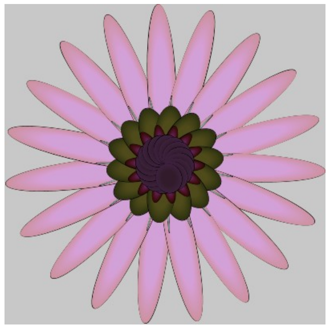
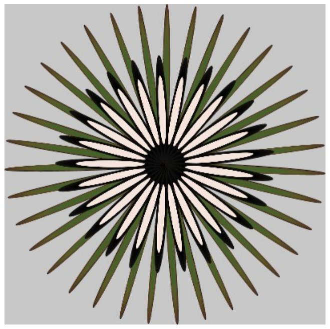
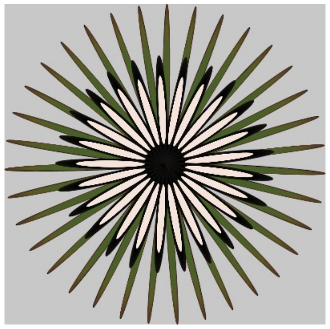
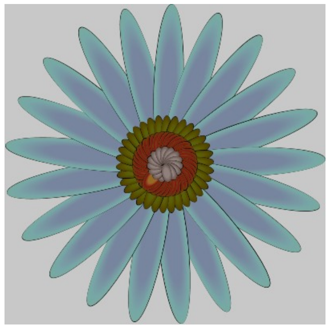
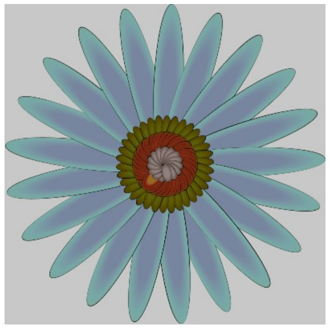
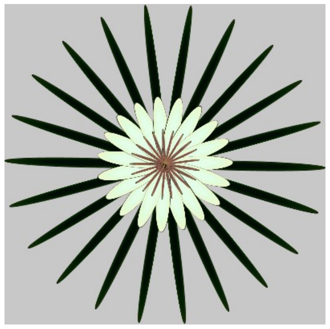
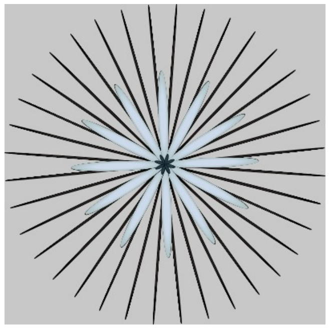
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 1				
Trial 2				
Trial 3				

Table 4.16: Best DNN-dhash scored 2D flower of each class type from the total and hash-filtered populations

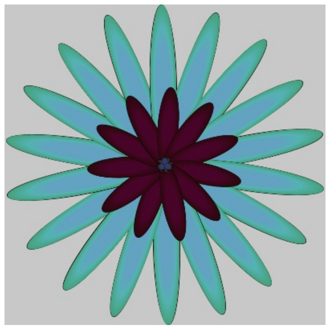
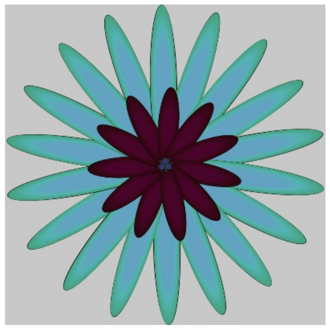
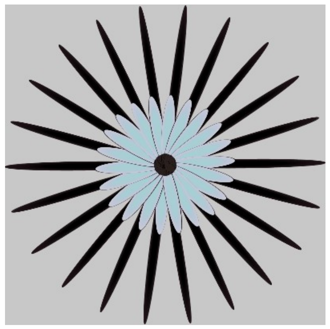
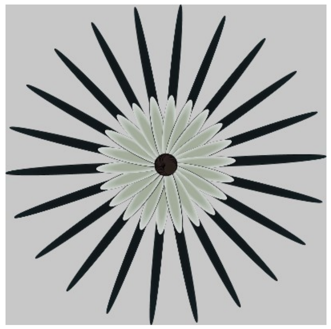
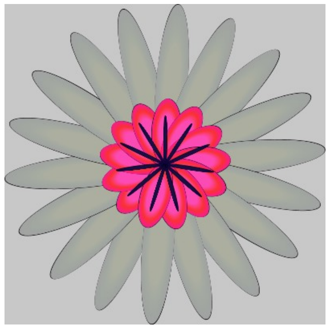
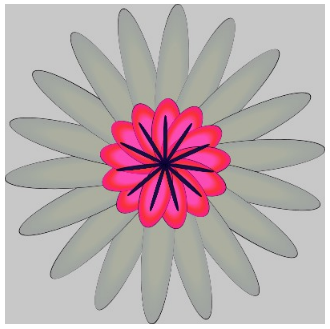
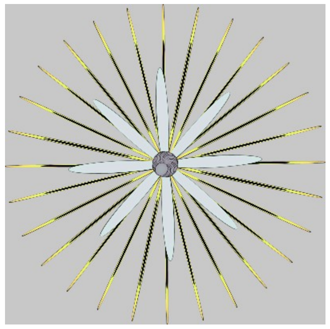
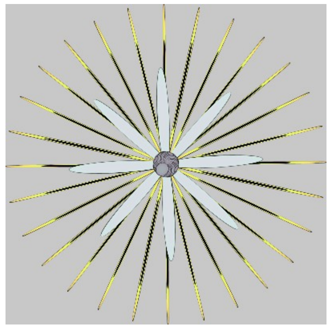
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 4				
Trial 5				

Table 4.17: Best DNN-dhash scored 2D flower of each class type from the total and hash-filtered populations

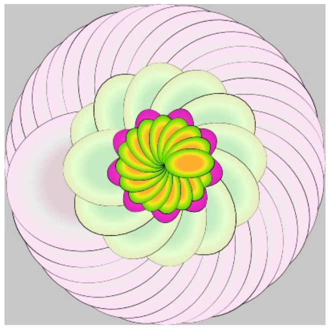
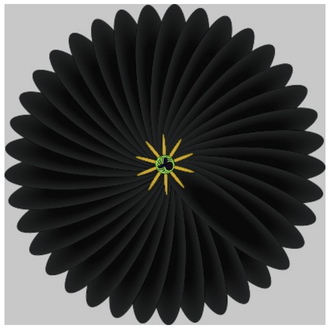
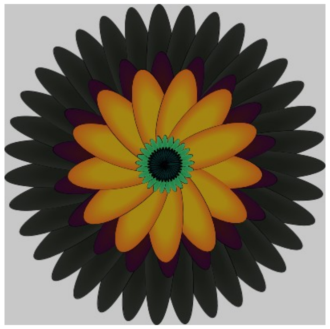
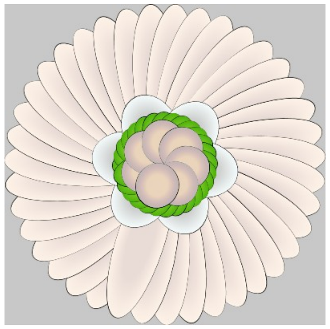
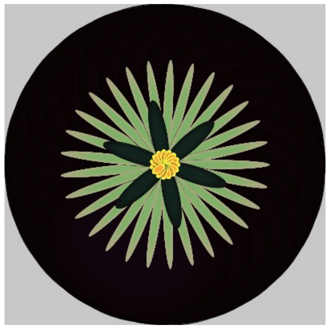
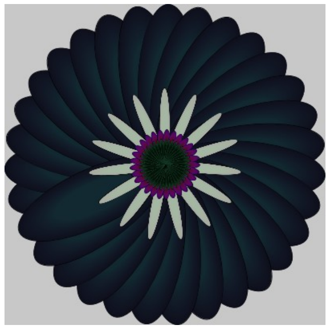
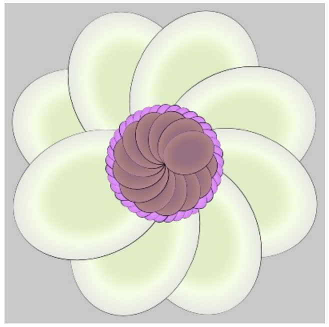
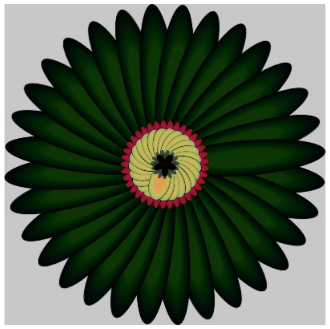
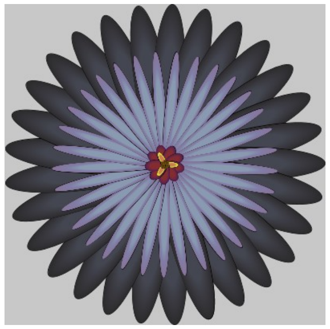
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 1		NA		
Trial 2		NA		
Trial 3		NA		

Table 4.18: Best DNN-dhash scored 2D flower of each class type from the total and hash-filtered populations

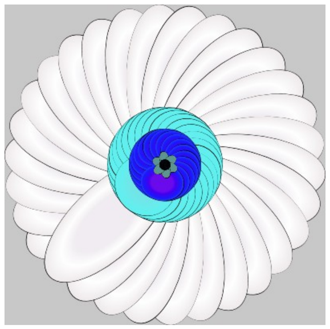
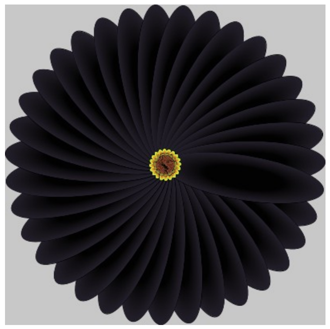
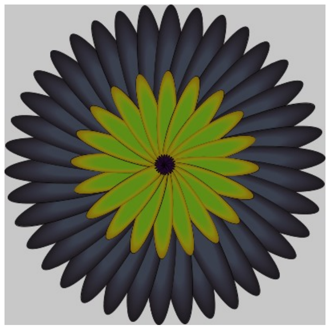
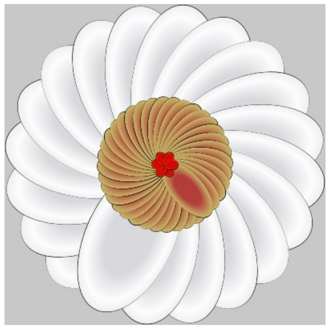
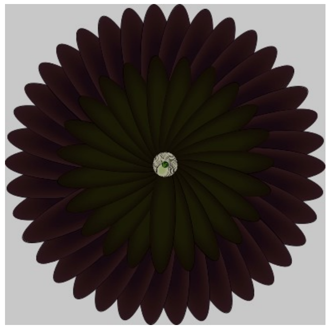
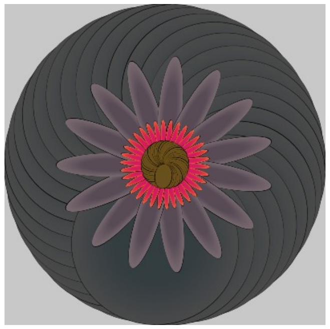
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 4		NA		
Trial 5		NA		

Table 4.19: Best DNN-dhash scored 2D flower of each class type from the total and hash-filtered populations

Phash Similarity Score Added

An average of 80% of 5000 niches were filled over each of the different trials. When analyzing the whole population, the fitness of the best individual is, on average, 1.48. All five times, that individual is a rose. The average fitness for the whole population is 0.94. The average number of goal switches is 0.58. The archive is comprised of 3.4% daisies, 9.7% dandelions, 84.9% roses, and 2.0% sunflowers. Daisies have an average fitness of 0.91, dandelions an average of 0.95, roses an average of 0.94, and sunflowers an average of 0.78. $\mathcal{SC}_{dr}$ is 1.04. s is equal to 0.83 and $\mathcal{NI}_s$ is approximately 0.

The average fitness for the top 5% of individuals is 1.23. An average of 0.48 goal switches occur. Of the top 5% of solutions, 1.9% are daisies, 6.7% are dandelions, 91.2% are roses, and 0.2% are sunflowers. Daisies have an average fitness of 1.19, dandelions an average fitness of 1.21, roses an average of 1.23, and sunflowers an average of 1.23. $\mathcal{SC}_{dr}$ is 0.05. s is equal to 0.87 and $\mathcal{NI}_s$ is approximately 0.

When filtered using a 95% whash-haar threshold, an average of 235.8 images were extracted from the archive. The average fitness of the best individual is 0.94. The average number of goal switches is 1.23. In this extricated population, 20.5% were daisies, 77.6% are dandelions, and 1.8% are sunflowers. There are no roses. Daisies have an average fitness of 0.9, dandelions an average fitness of 0.96, and sunflowers an average of 0.75. $\mathcal{SC}_{dr}$ is 0.06. s is equal to 0.87 and $\mathcal{NI}_s$ is approximately 0.

Figure 4.15 shows the percentile class distributions from the different sub-populations. Figure 4.16 shows the normalized frequencies of the different ranges of fitness values from all the solutions in the final archives across the five trials. Tables 4.20 - 4.23 contain the images with the best composite score from both the DNN and the phash algorithm. Images are filtered into a sub-population with a whash-haar 95% threshold. Flower class types not found in the population or sub-population are indicated with an "NA."

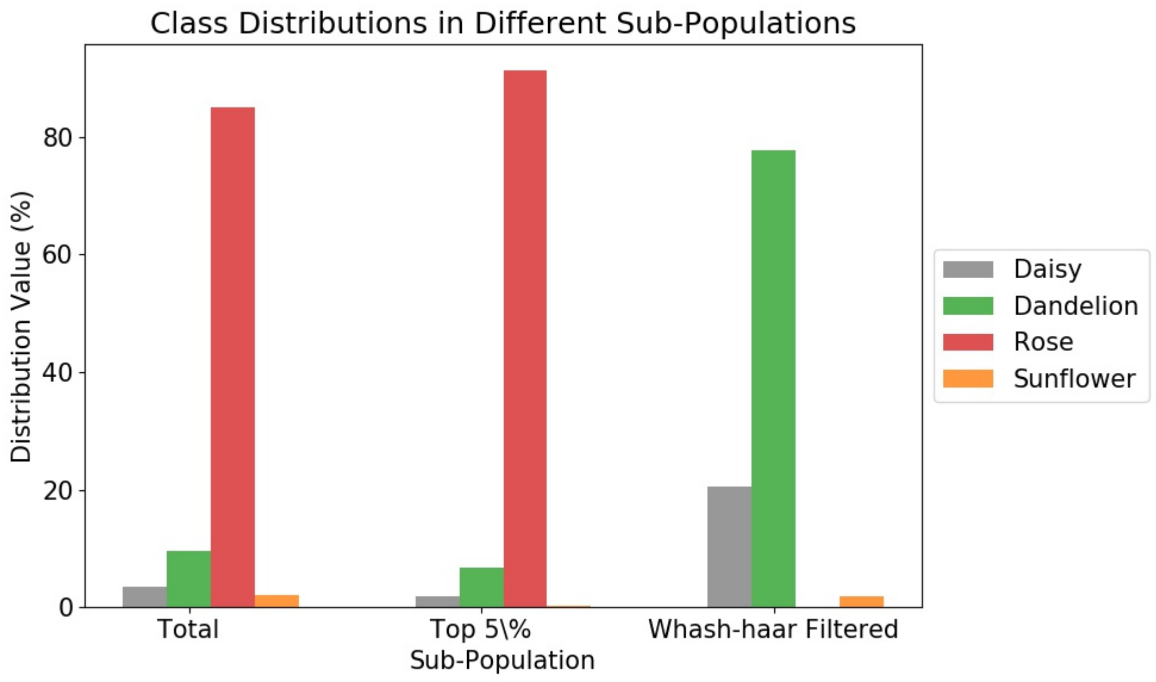


Figure 4.15: The distribution (in %) of different class types in different DNN-phash composite scored 2D flower sub-populations (what fraction of the sub-population is flower type x?)

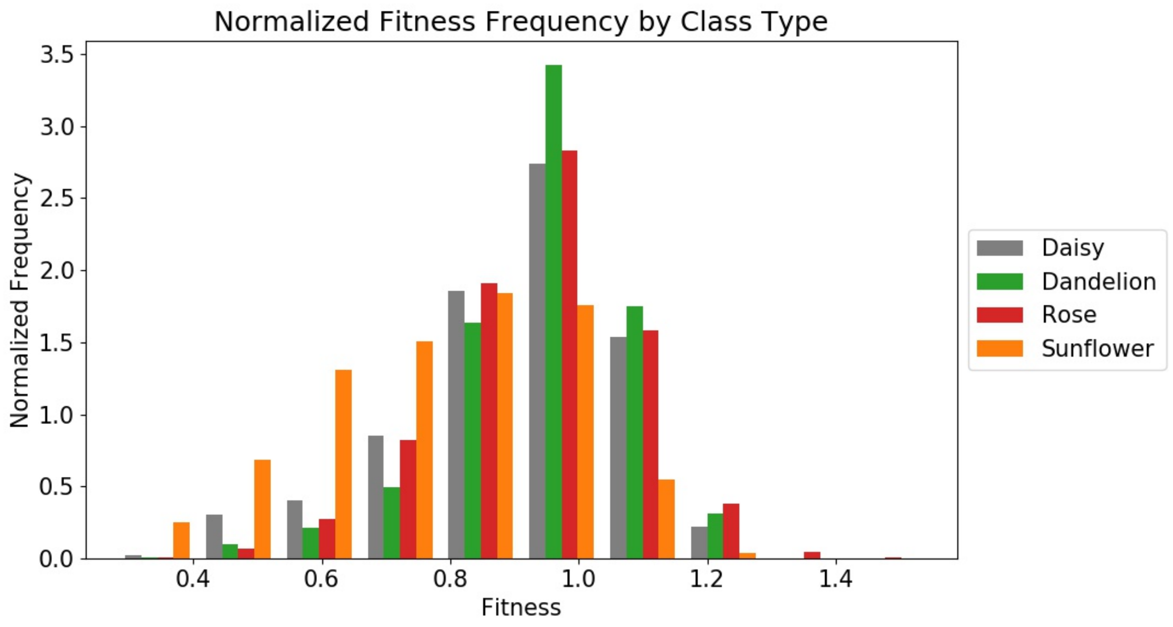


Figure 4.16: Normalized frequencies of fitness values for DNN-phash composite-scored 2D flowers in each class type (all flowers over five trials)

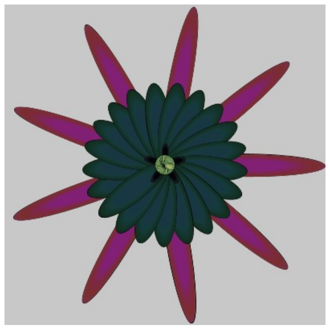
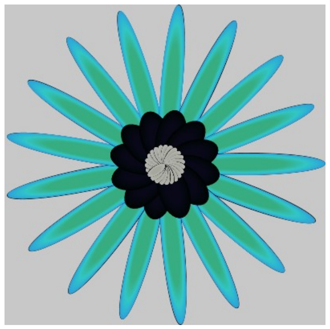
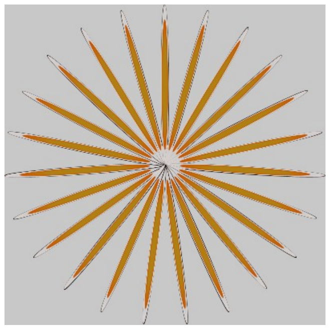
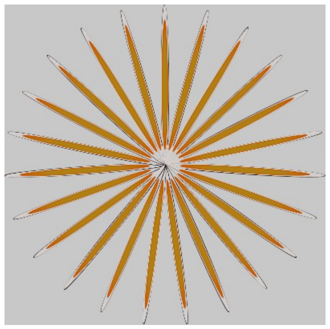
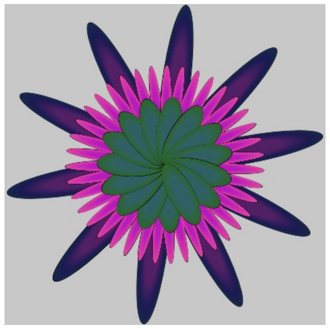
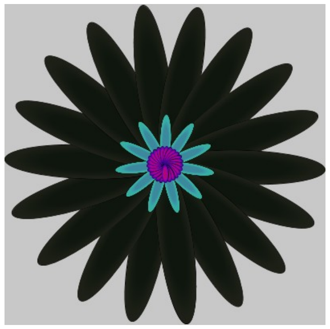
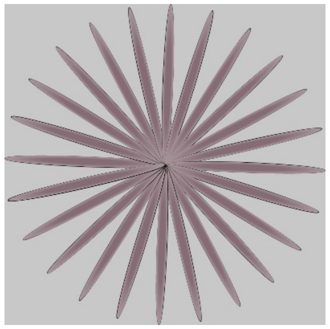
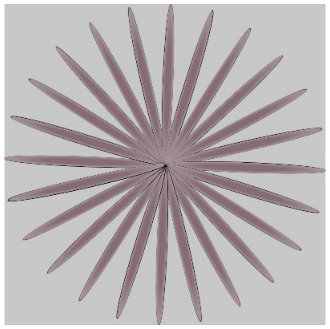
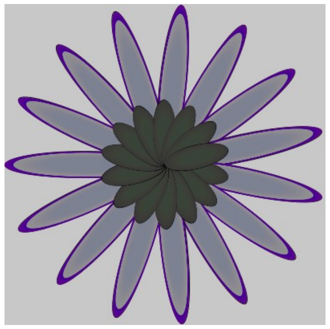
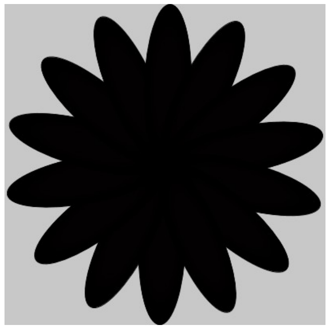
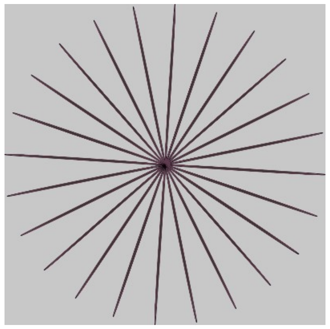
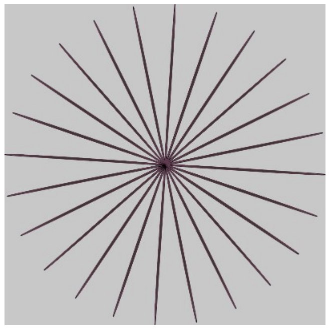
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 1				
Trial 2				
Trial 3				

Table 4.20: Best DNN-phash scored 2D flower of each class type from the total and hash-filtered populations

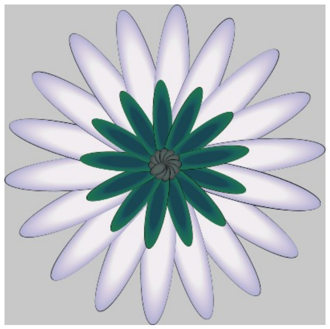
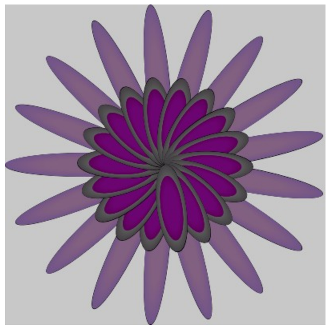
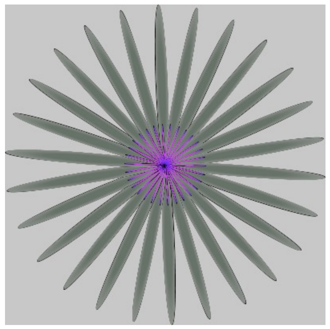
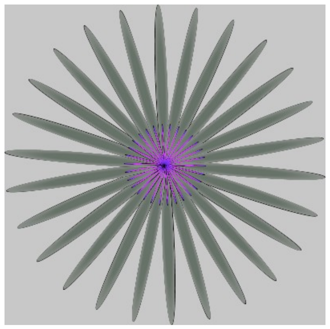
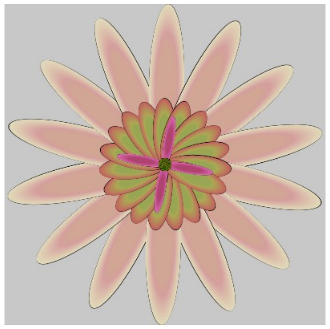
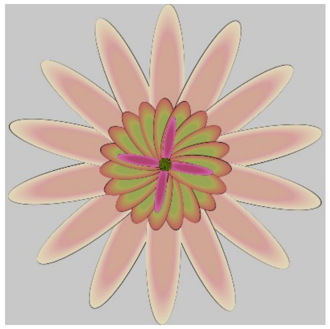
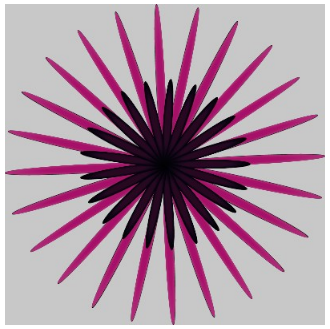
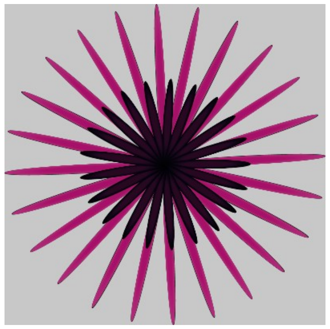
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 4				
Trial 5				

Table 4.21: Best DNN-phash scored 2D flower of each class type from the total and hash-filtered populations

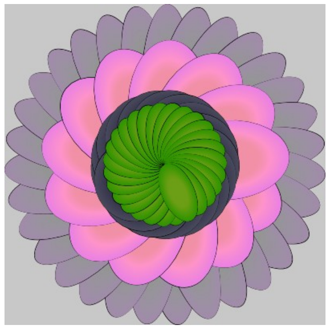
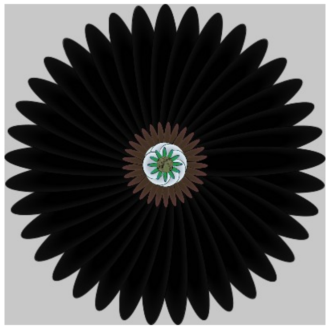
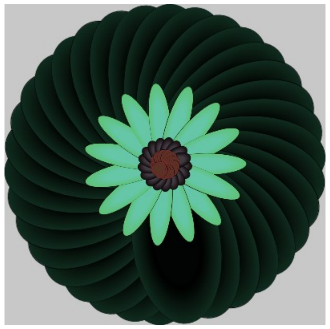
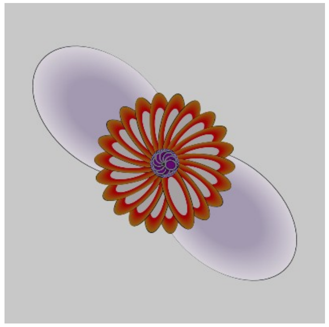
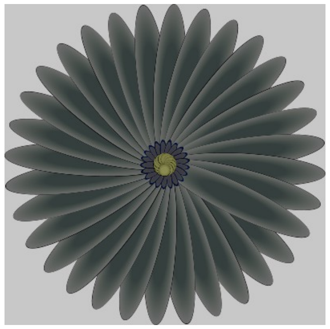
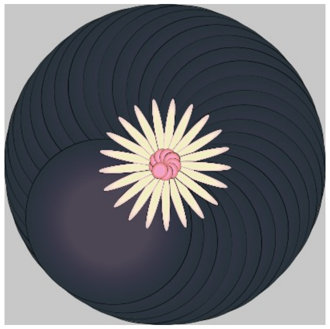
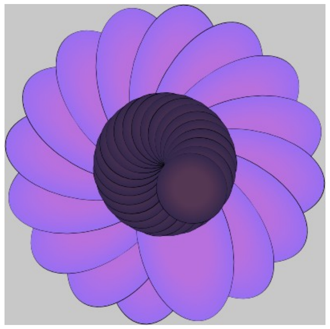
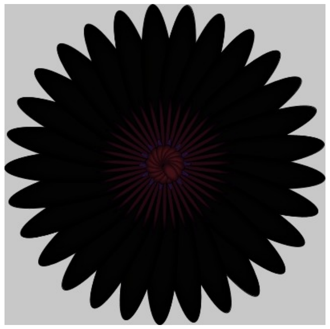
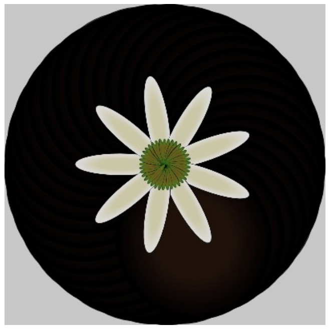
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 1		NA		
Trial 2		NA		
Trial 3		NA		

Table 4.22: Best DNN-phash scored 2D flower of each class type from the total and hash-filtered populations

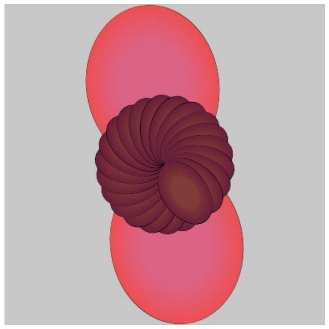
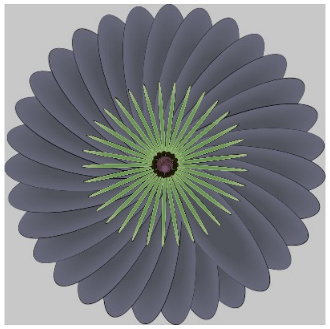
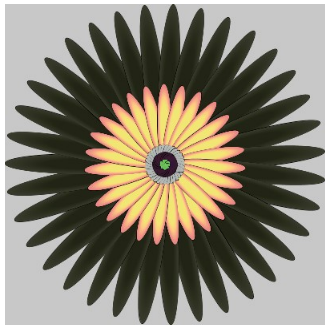
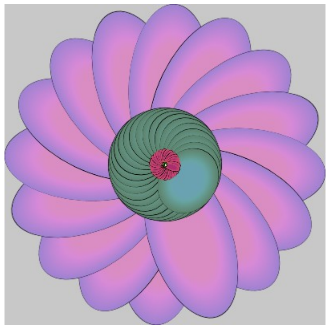
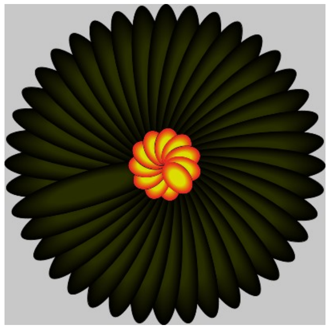
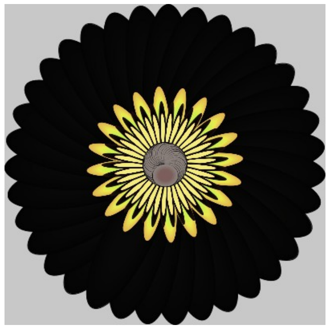
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 4		NA		
Trial 5		NA		

Table 4.23: Best DNN-phash scored 2D flower of each class type from the total and hash-filtered populations

Whash-haar Similarity Score Added

An average of 80% of 5000 niches were filled over each of the different trials. When analyzing the whole population, the fitness of the best individual is, on average, 1.98. The average fitness for the whole population is 1.50. The average number of goal switches is 1.36. The archive is comprised of 20.0% daisies, 61.1% dandelions, 15.9% roses, and 2.9% sunflowers. Daisies have an average fitness of 1.64, dandelions an average of 1.68, roses an average of 0.70, and sunflowers an average of 0.94. $\mathcal{SC}_{dr}$ is 1.06. s is equal to 0.83 and $\mathcal{NI}_s$ is approximately 0.

The average fitness for the top 5% of individuals is 1.92. An average of 1.34 goal switches occur. Of the top 5% of solutions, 54.6% are daisies and 45.4% are dandelions. There are no sunflowers or roses. Daisies have an average fitness of 1.93 and dandelions an average fitness of 1.91. $\mathcal{SC}_{dr}$ is 0.05. s is equal to 0.86 and $\mathcal{NI}_s$ is approximately 0.

When filtered using a 95% whash-haar threshold, an average of 2778.8 images were extracted from the archive. The average fitness of the best individual is 1.98. The average number of goal switches is 1.43. In this extricated population, 23.9% are daisies, 75.0% are dandelions, and 1.1% are sunflowers. There are no roses. Daisies have an average fitness of 1.73, dandelions an average fitness of 1.73, and sunflowers an average of 1.50. $\mathcal{SC}_{dr}$ is 0.07. s is equal to 0.84 and $\mathcal{NI}_s$ is approximately 0.

Figure 4.17 shows the percentile class distributions from the different sub-populations. Figure 4.18 shows the normalized frequencies of the different ranges of fitness values from all the solutions in the final archives across the five trials. Tables 4.24 - 4.27 contain the images with the best composite score from both the DNN and the whash-haar algorithm. Images are filtered into a sub-population with a whash-haar 95% threshold. Flower class types not found in the population or sub-population are indicated with an "NA."

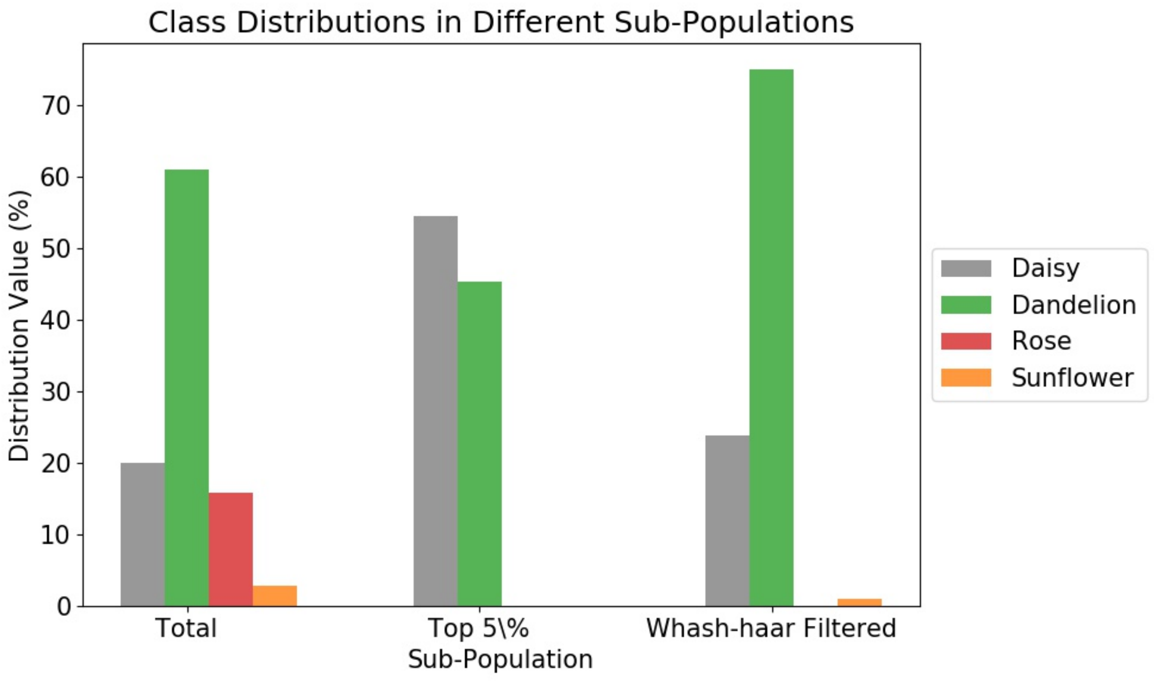


Figure 4.17: The distribution (in %) of different class types in different DNN-whash-haar composite scored 2D flower sub-populations (what fraction of the sub-population is flower type x?)

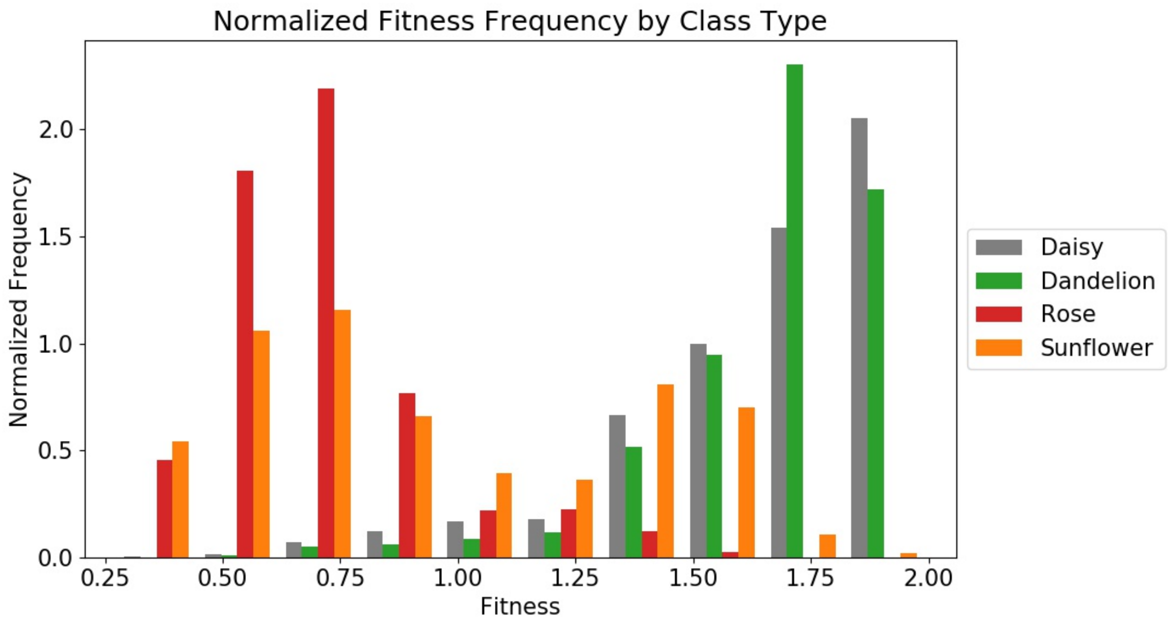


Figure 4.18: Normalized frequencies of fitness values for DNN-whash-haar composite-scored 2D flowers in each class type (all flowers over five trials)

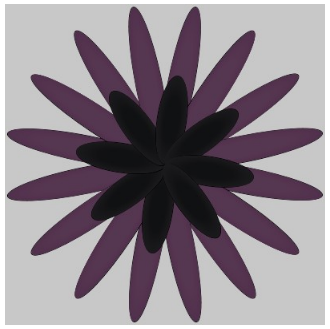
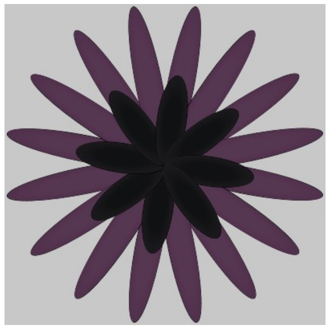
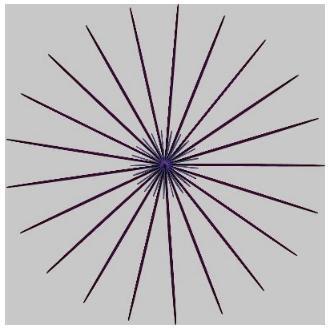
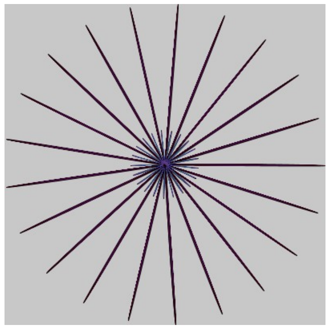
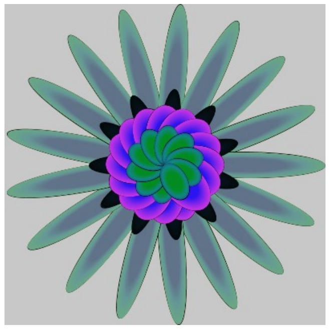
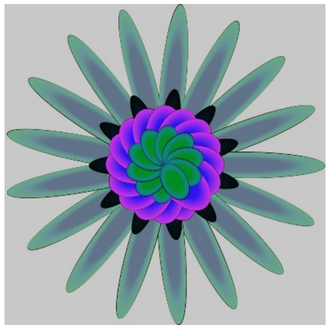
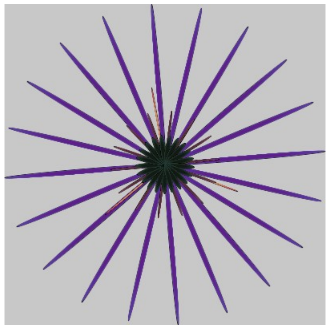
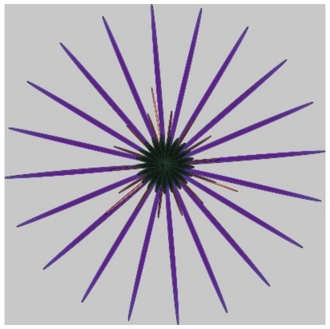
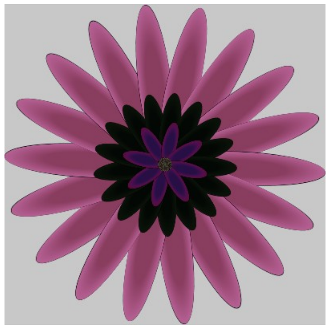
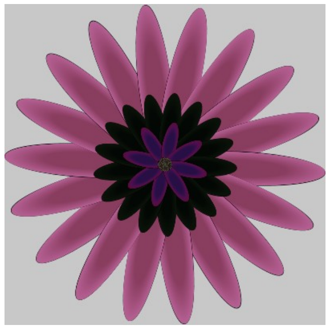
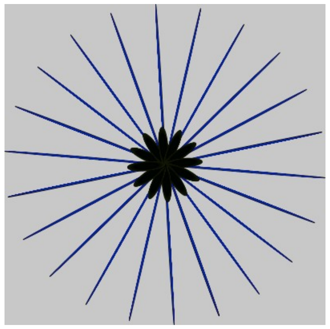
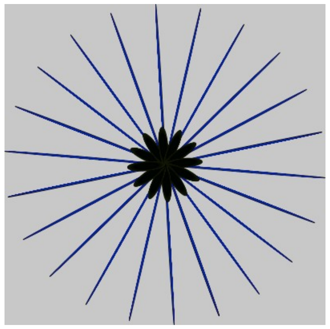
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 1				
Trial 2				
Trial 3				

Table 4.24: Best DNN-whash-haar scored 2D flower of each class type from the total and hash-filtered populations

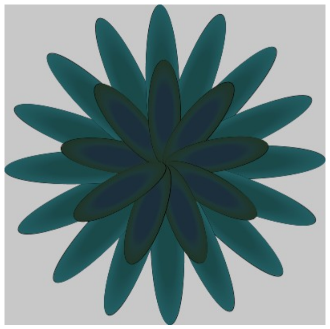
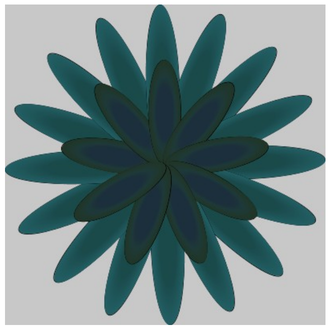
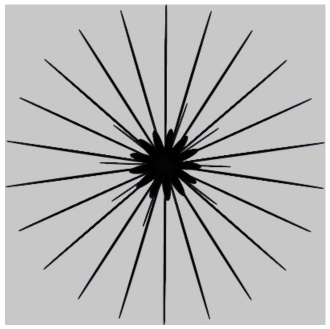
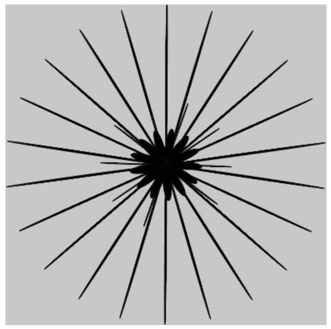
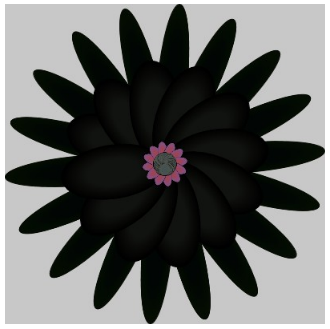
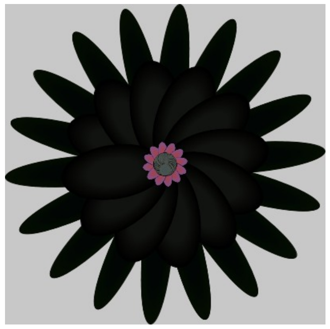
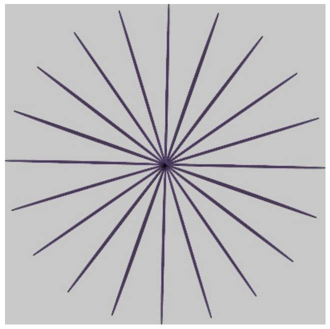
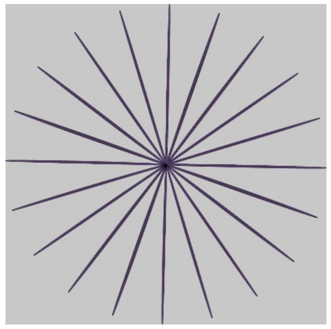
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 4				
Trial 5				

Table 4.25: Best DNN-whash-haar scored 2D flower of each class type from the total and hash-filtered populations

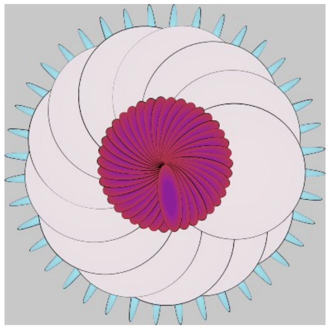
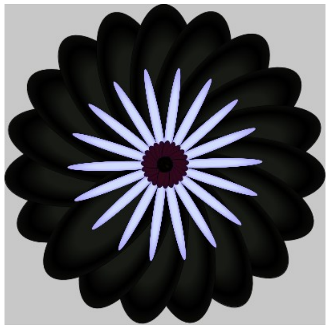
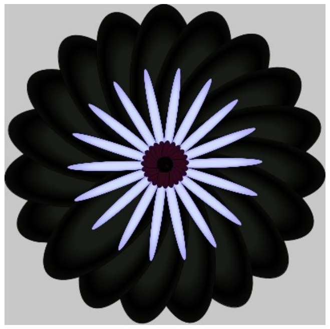
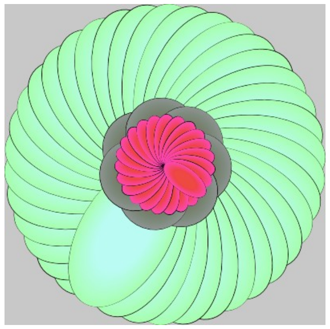
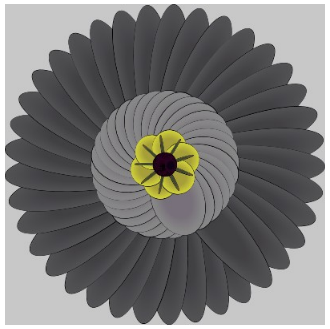
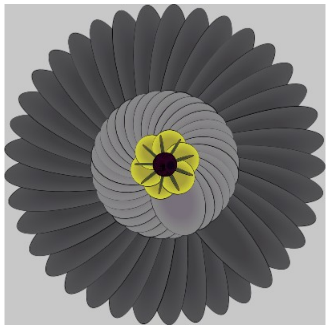
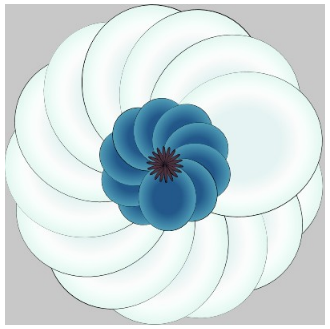
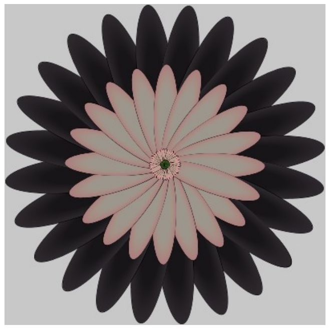
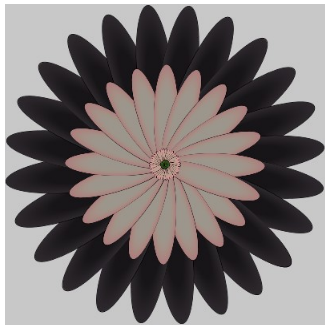
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 1		NA		
Trial 2		NA		
Trial 3		NA		

Table 4.26: Best DNN-whash-haar scored 2D flower of each class type from the total and hash-filtered populations

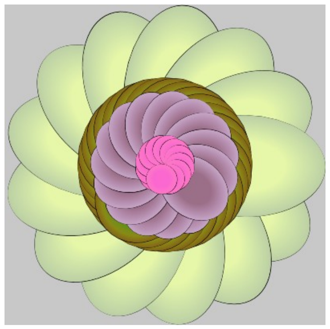
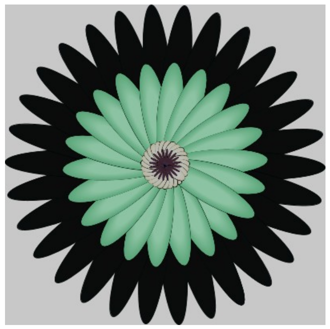
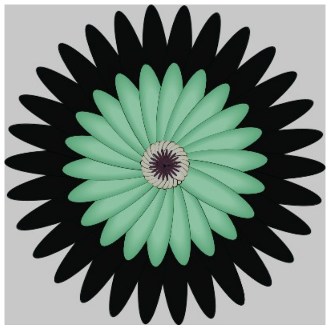
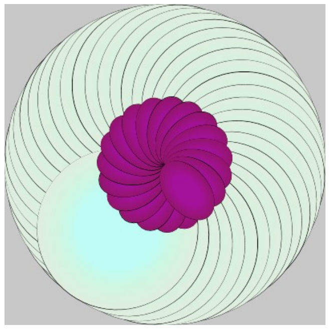
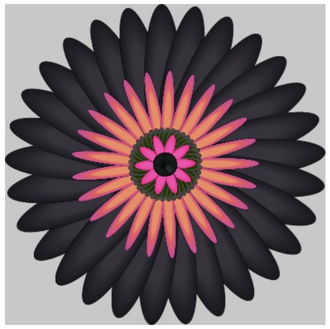
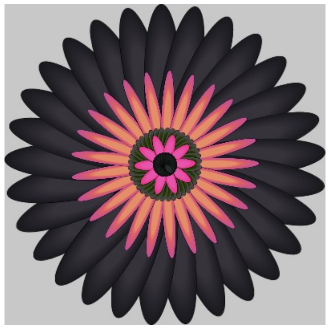
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 4		NA		
Trial 5		NA		

Table 4.27: Best DNN-whash scored 2D flower of each class type from the total and hash-filtered populations

4.7.4 3D Flower Generator With Composite Hashing Score

In the next four experiments, the hash scoring is analogous to section 4.7.3. Aside from this hashing score, all the same parameters are used from section 4.7.2. Whash-haar is still used to filter images from the final archive. Due to hardware and time constraints, only two trials were conducted for each of the different hashing algorithms.

Ahash Similarity Score Added

An average of 96.7% of 5000 niches were filled over each of the two trials. When analyzing the whole population, the fitness of the best individual is, on average, 1.84. Both times, that individual is a dandelion. The average fitness for the whole population is 1.23. The average number of goal switches is 1.19. The archive is comprised of 0.06% daisies, 56.9% dandelions, 20.3% roses, and 22.7% sunflowers. Daisies have an average fitness of 0.58, dandelions an average of 1.30, roses an average of 0.99, and sunflowers an average of 1.26. $\mathcal{SC}_{dr}$ is 1.04. s is equal to 0.65 and $\mathcal{NI}_s$ is approximately 0.

The average fitness for the top 5% of individuals is 1.63. An average of 0.98 goal switches occur. Of the top 5% of solutions, 90% are dandelions, and 9.9% are sunflowers. There are no daisies or roses. Dandelions have an average fitness of 1.63 and sunflowers an average of 1.61. $\mathcal{SC}_{dr}$ is 0.05. s is equal to 0.77 and $\mathcal{NI}_s$ is approximately 0.

When filtered using a 95% whash-haar threshold, an average of 13 images were extracted from the archive. The average fitness of the best individual is 1.60. The average number of goal switches is 1.33. In this extricated population, 73.2% are dandelions and 26.8% are roses. There are no daisies or sunflowers. Dandelions had an average fitness of 1.39, and roses an average fitness of 1.16. $\mathcal{SC}_{dr}$ is approximately 0. s is equal to 0.81 and $\mathcal{NI}_s$ is approximately 0.

Figure 4.19 shows the percentile class distributions from the different sub-populations. Figure 4.20 shows the normalized frequencies of the different ranges of fitness values from all the solutions in the final archives across the two trials. Tables 4.28 - 4.29 contain the images with the best composite score from both the DNN and the ahash algorithm. Images are filtered into a sub-population with a whash-haar 95% threshold. Flower class types not found in the population or sub-population are indicated with an "NA."

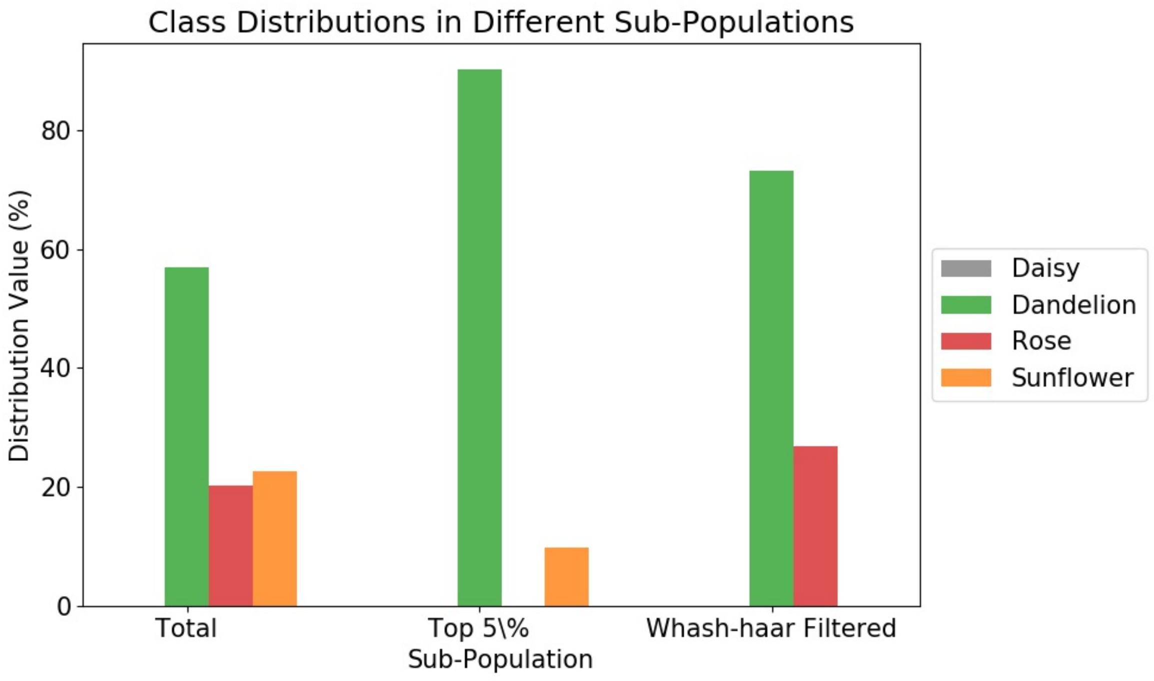


Figure 4.19: The distribution (in %) of different class types in different DNN-ahash composite scored 3D flower sub-populations (what fraction of the sub-population is flower type x?)

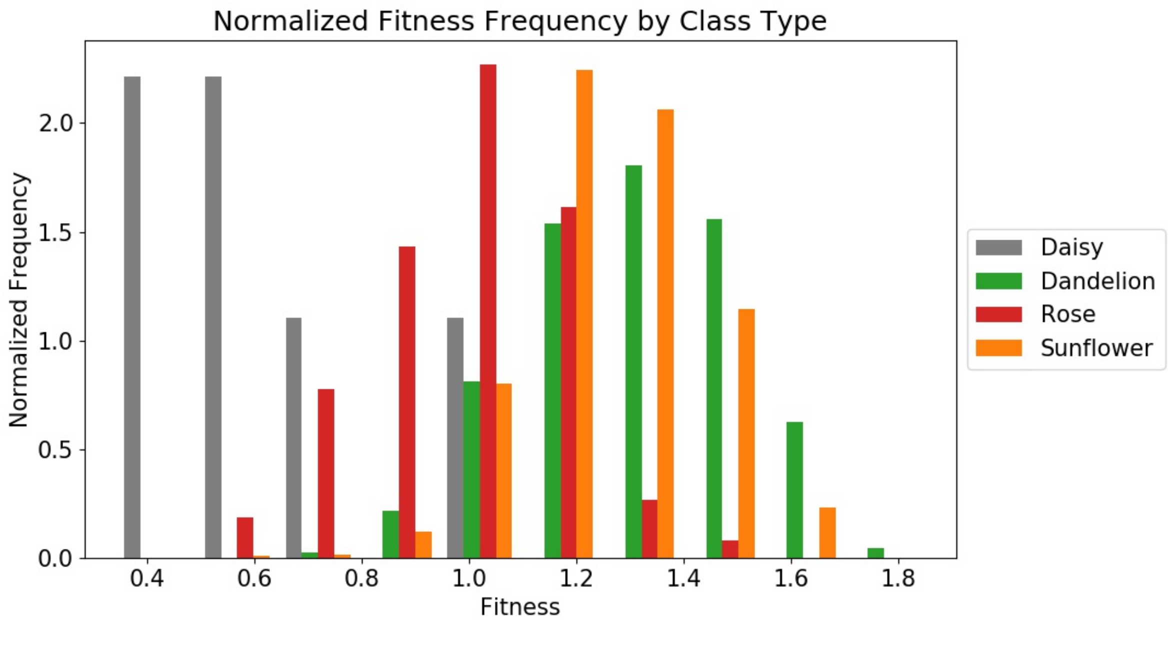


Figure 4.20: Normalized frequencies of fitness values for DNN-ahash composite-scored 3D flowers in each class type (all flowers over two trials)

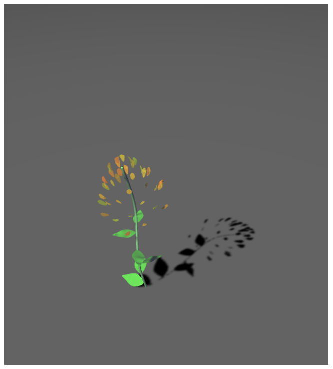
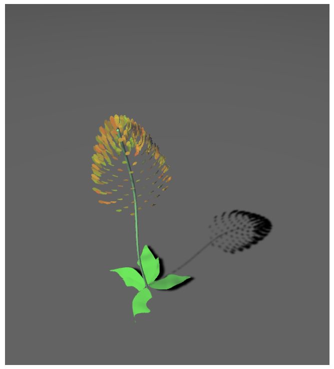
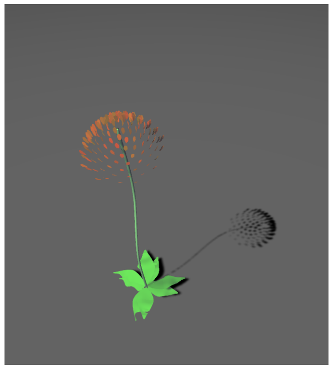
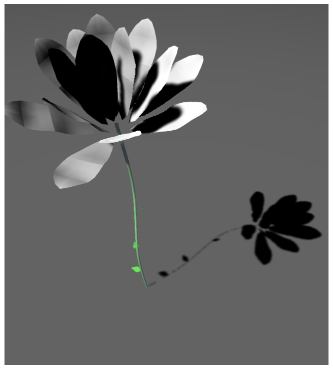
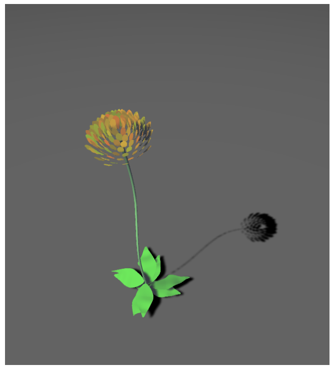
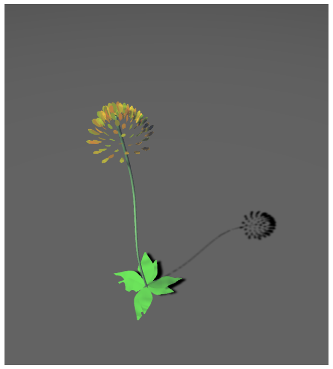
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 1		NA		
Trial 2		NA		

Table 4.28: Best DNN-ahash scored 3D flower of each class type from the total and hash-filtered populations

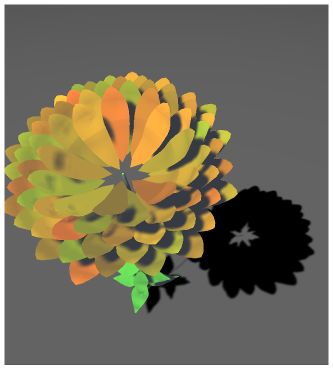
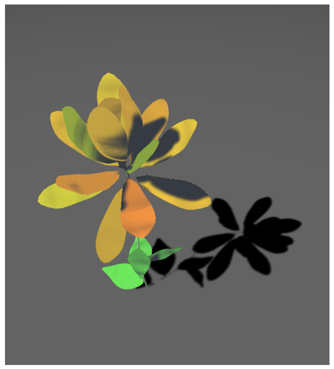
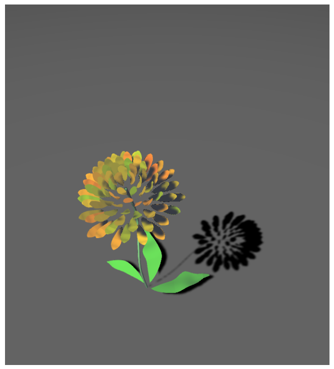
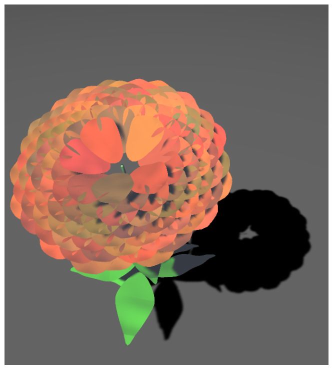
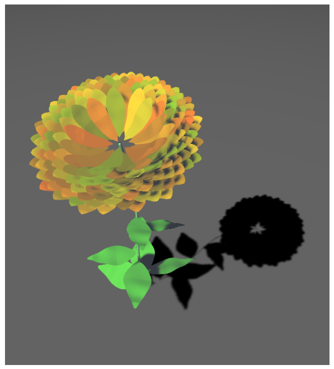
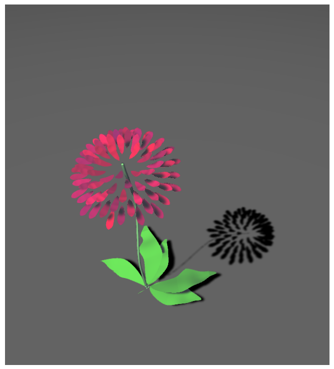
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 1				NA
Trial 2				NA

Table 4.29: Best DNN-ahash scored 3D flower of each class type from the total and hash-filtered populations

Dhash Similarity Score Added

An average of 96.9% of 5000 niches were filled over each of the two trials. When analyzing the whole population, the fitness of the best individual is, on average, 1.79. Both times that individual is a dandelion. The average fitness for the whole population is 1.17. The average number of goal switches is 1.03. The archive is comprised of 0.1% daisies, 53.8% dandelions, 35.3% roses, and 10.8% sunflowers. Daisies have an average fitness of 0.77, dandelions an average of 1.24, roses an average of 1.09, and sunflowers an average of 1.10. $\mathcal{SC}_{dr}$ is 1.05. s is equal to 0.63 and $\mathcal{NI}_s$ is approximately 0.

The average fitness for the top 5% of individuals is 1.55. An average of 0.95 goal switches occur. Of the top 5% of solutions, 98.8% are dandelions, 0.8% are roses, and 0.4% are sunflowers. There are no daisies. Dandelions have an average fitness of 1.55, roses an average of 1.51, and sunflowers an average of 1.50. $\mathcal{SC}_{dr}$ is 0.05. s is equal to 0.70 and $\mathcal{NI}_s$ is approximately 0.

When filtered using a 95% whash-haar threshold, an average of 18 images were extracted from the archive. The average fitness of the best individual is 1.66. The average number of goal switches is 1.03. In this extricated population, 63.6% are dandelions and 36.4% are roses. There are no daisies or sunflowers. Dandelions have an average fitness of 1.42, and roses an average fitness of 1.20. $\mathcal{SC}_{dr}$ is approximately 0. s is equal to 0.82 and $\mathcal{NI}_s$ is approximately 0.

Figure 4.21 shows the percentile class distributions from the different sub-populations. Figure 4.22 shows the normalized frequencies of the different ranges of fitness values from all the solutions in the final archives across the two trials. Tables 4.30 - 4.31 contain the images with the best composite score from both the DNN and the dhash algorithm. Images are filtered into a sub-population with a whash-haar 95% threshold. Flower class types not found in the population or sub-population are indicated with an "NA."

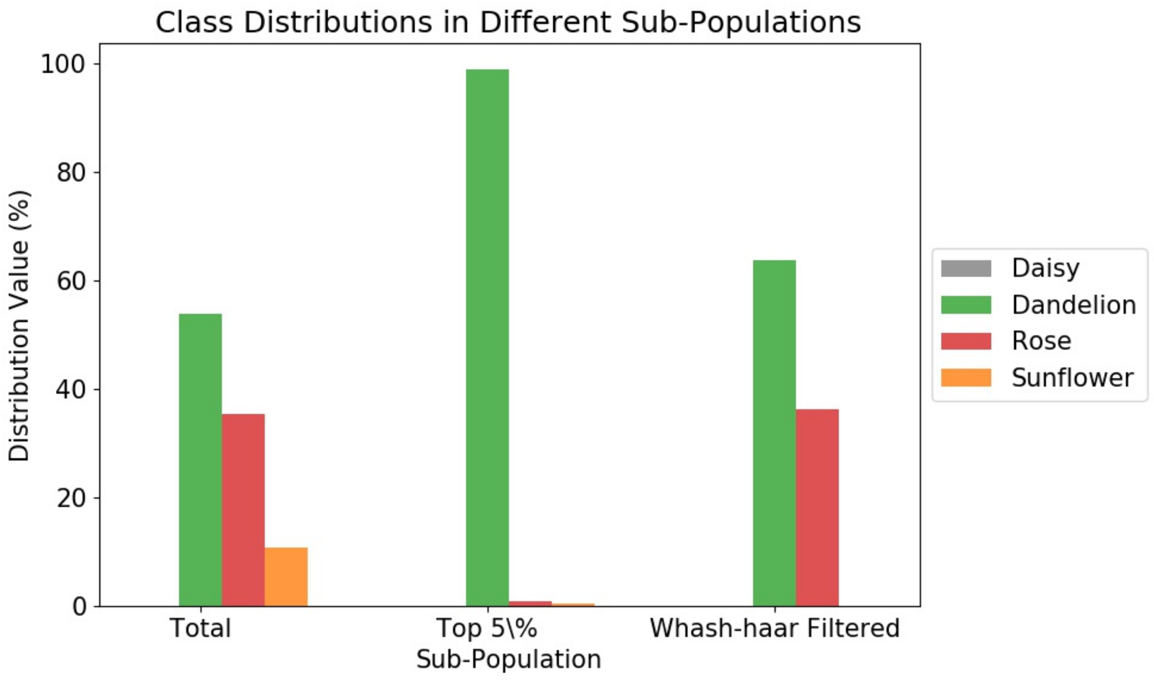


Figure 4.21: The distribution (in %) of different class types in different DNN-dhash composite scored 3D flower sub-populations (what fraction of the sub-population is flower type x?)

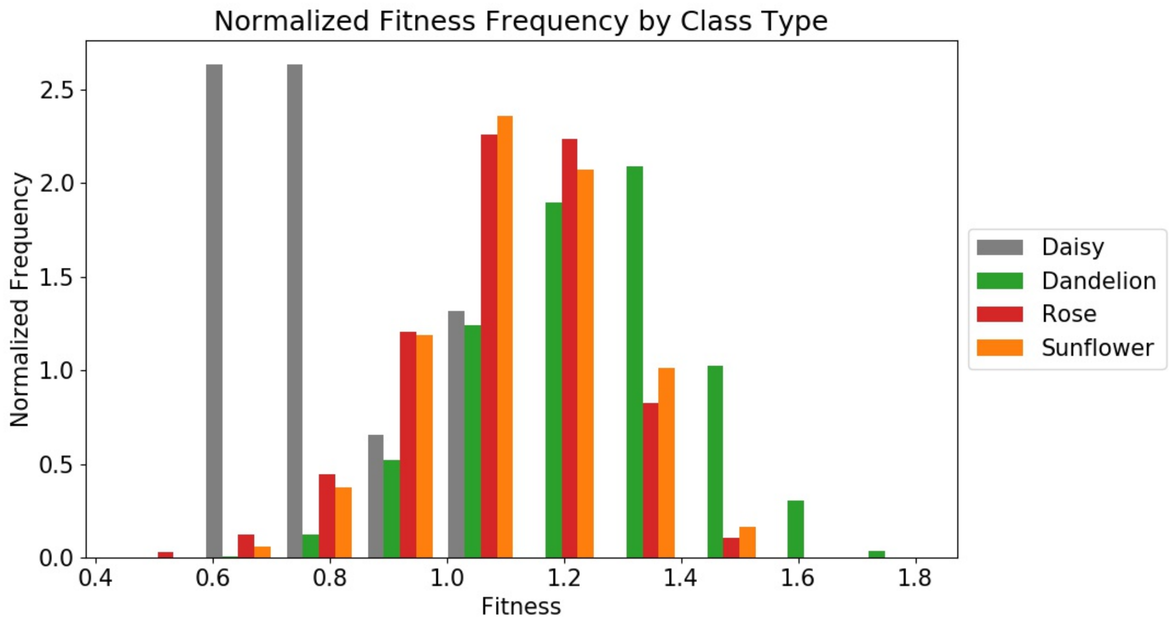


Figure 4.22: Normalized frequencies of fitness values for DNN-dhash composite-scored 3D flowers in each class type (all flowers over two trials)

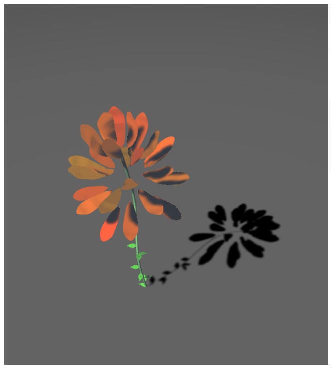
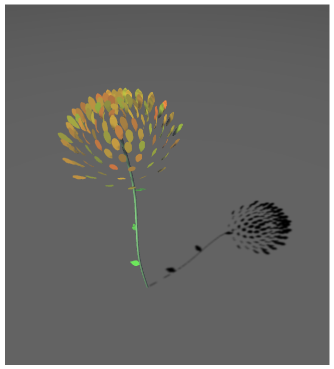
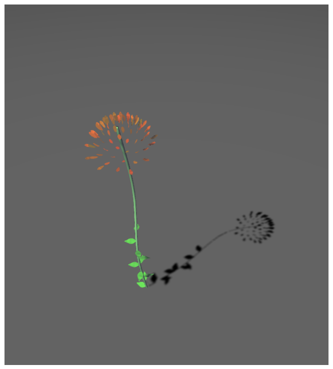
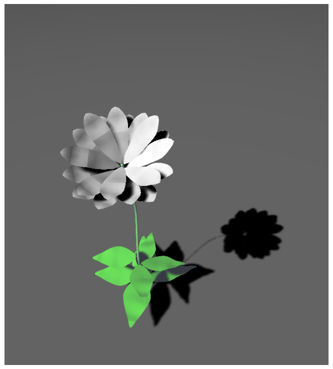
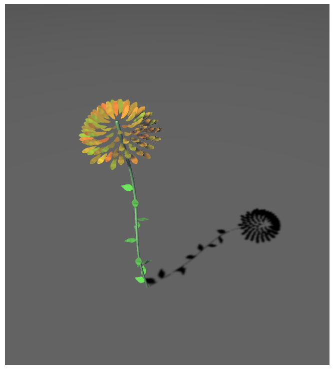
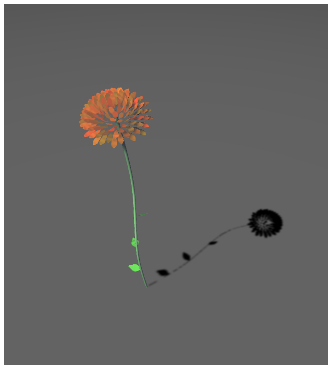
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 1		NA		
Trial 2		NA		

Table 4.30: Best DNN-dhash scored 3D flower of each class type from the total and hash-filtered populations

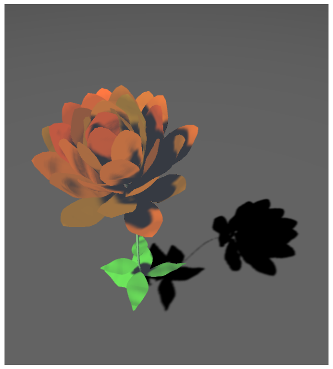
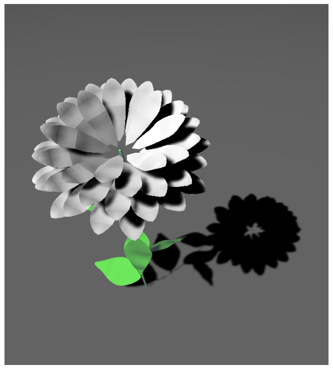
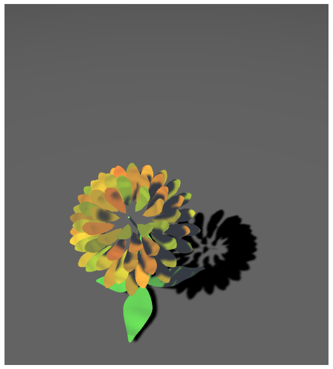
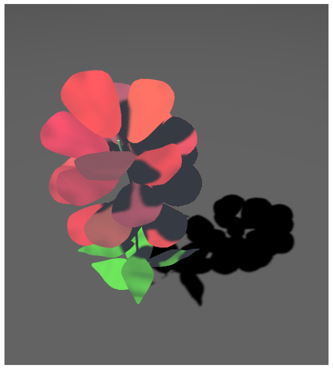
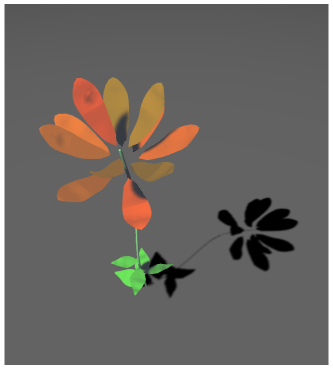
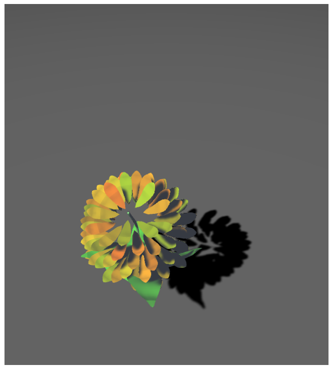
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 1				NA
Trial 2				NA

Table 4.31: Best DNN-dhash scored 3D flower of each class type from the total and hash-filtered populations

Phash Similarity Score Added

An average of 96.8% of niches were filled over each of the two trials. When analyzing the whole population, the fitness of the best individual is, on average, 1.67. One time that individual is a rose; the other is a dandelion. The average fitness for the whole population is 1.03. The average number of goal switches is 1.09. The archive is comprised of 36.2% daisies, 22.0% dandelions, 52.6% roses, and 25.0% sunflowers. Daisies have an average fitness of 0.87, dandelions an average of 0.93, roses an average of 1.04, and sunflowers an average of 1.09. $\mathcal{SC}_{dr}$ is 1.03. s is equal to 0.64 and $\mathcal{NI}_s$ is approximately 0.

The average fitness for the top 5% of individuals is 1.38. An average of 1.05 goal switches occur. Of the top 5% of solutions, 0.2% are daisies, 9.7% are dandelions, 52.9% are roses, and 37.2% are sunflowers. Daisies have an average fitness of 1.33, dandelions an average of 1.41, roses an average of 1.36, and sunflowers an average of 1.39. $\mathcal{SC}_{dr}$ is 0.05. s is equal to 0.67 and $\mathcal{NI}_s$ is approximately 0.

When filtered using a 95% whash-haar threshold, an average of 10 images were extracted from the archive. The average fitness of the best individual is 1.12. The average number of goal switches is 1.17. In this extricated population, 59.9% are dandelions and 40.1% are roses. There are no daisies or sunflowers. Dandelions have an average fitness of 1.12, and roses an average fitness of 1.16. $\mathcal{SC}_{dr}$ is approximately 0. s is equal to 0.84 and $\mathcal{NI}_s$ is approximately 0.

Figure 4.23 shows the percentile class distributions from the different sub-populations. Figure 4.24 shows the normalized frequencies of the different ranges of fitness values from all the solutions in the final archives across the two trials. Tables 4.32 - 4.33 contain the images with the best composite score from both the DNN and the phash algorithm. Images are filtered into a sub-population with a whash-haar 95% threshold. Flower class types not found in the population or sub-population are indicated with an "NA."

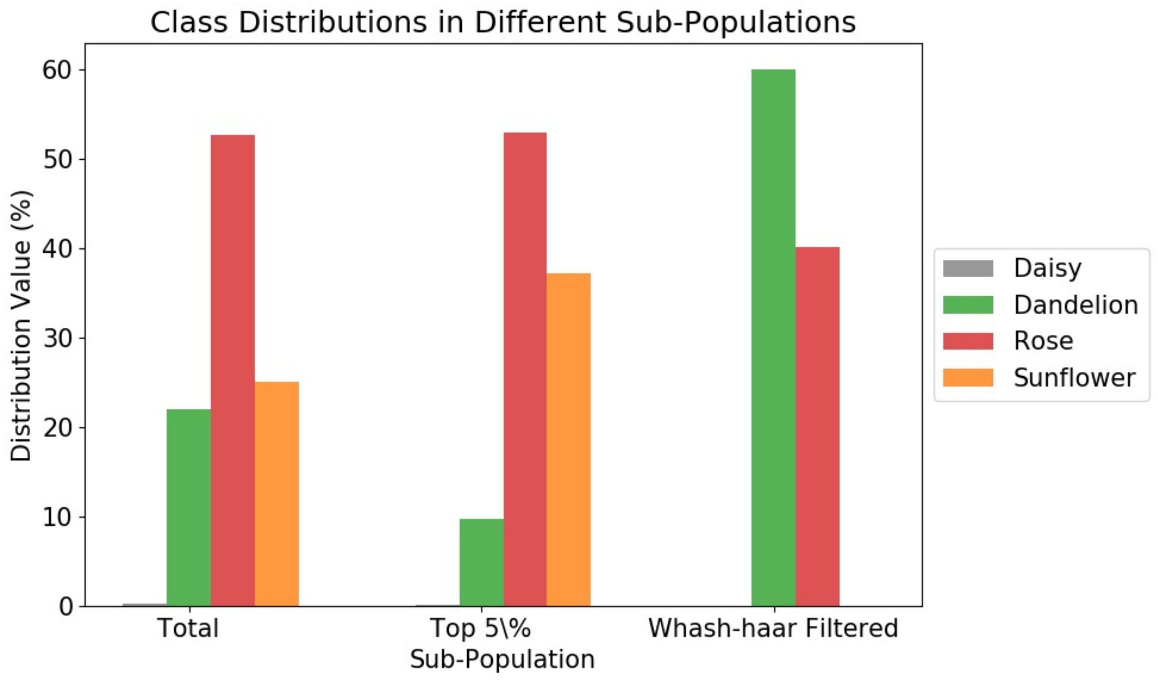


Figure 4.23: The distribution (in %) of different class types in different DNN-phash composite scored 3D flower sub-populations (what fraction of the sub-population is flower type x?)

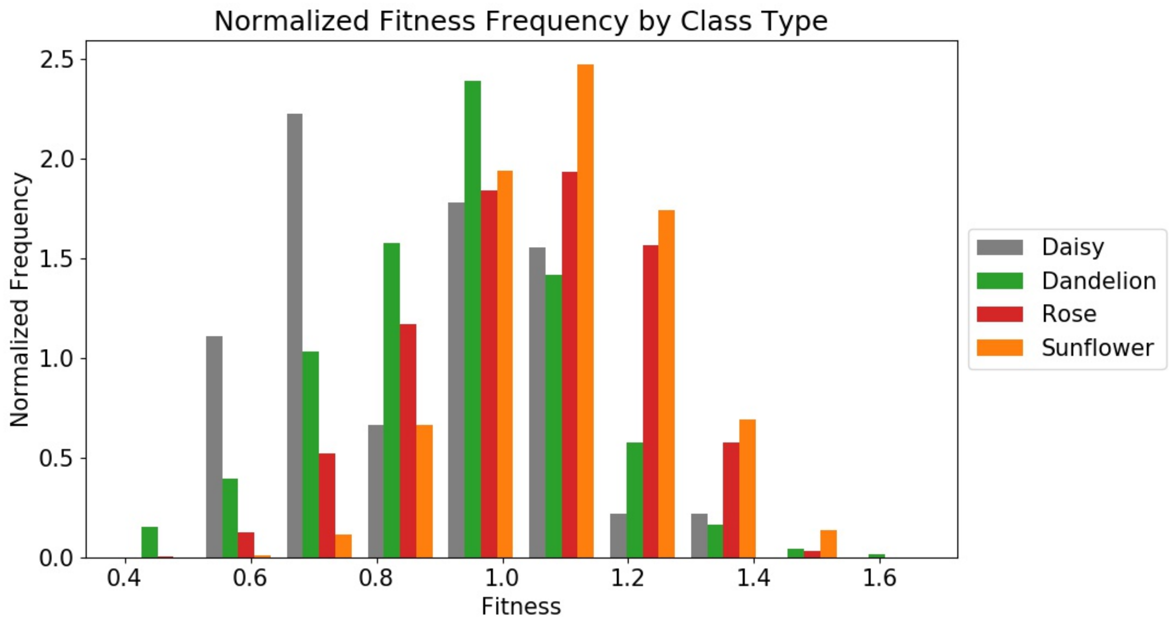


Figure 4.24: Normalized frequencies of fitness values for DNN-phash composite-scored 3D flowers in each class type (all flowers over two trials)

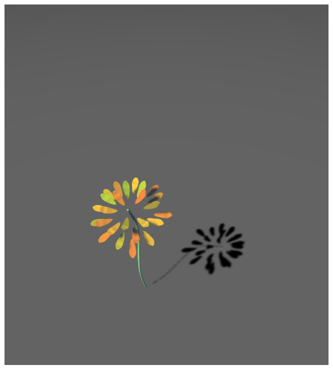
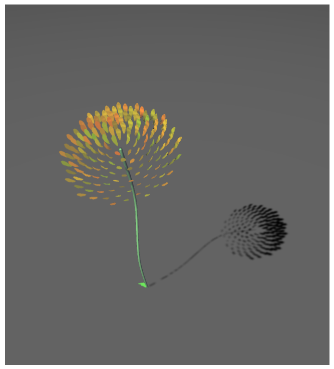
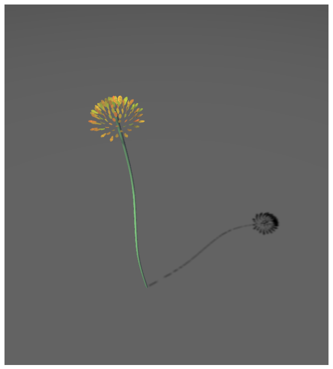
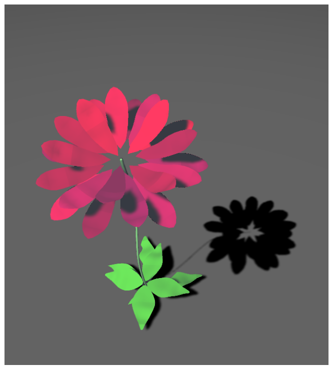
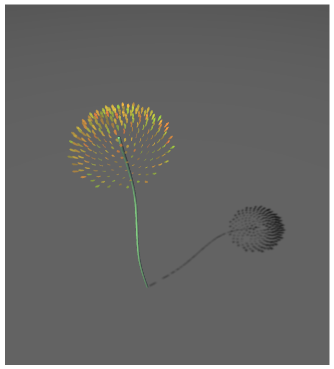
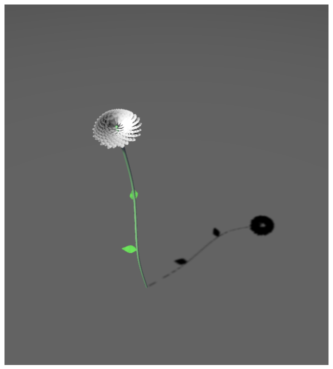
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 1		NA		
Trial 2		NA		

Table 4.32: Best DNN-phash scored 3D flower of each class type from the total and hash-filtered populations

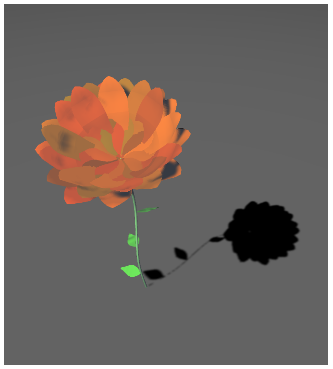
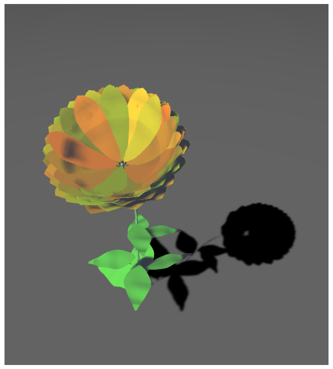
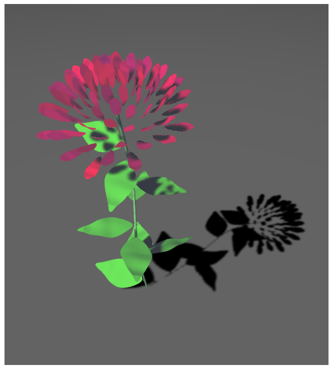
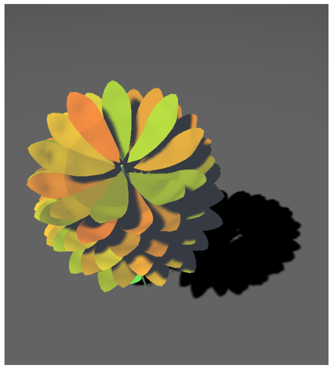
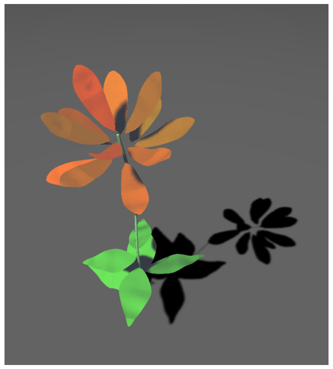
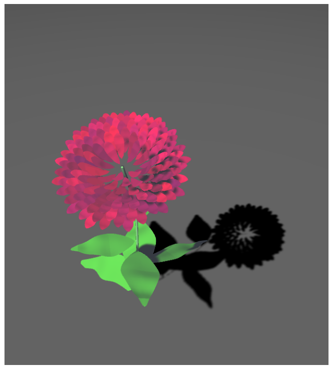
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 1				NA
Trial 2				NA

Table 4.33: Best DNN-phash scored 3D flower of each class type from the total and hash-filtered populations

Whash Similarity Score Added

An average of 96.9% of 5000 niches were filled over each of the two trials. When analyzing the whole population, the fitness of the best individual is, on average, 1.92. Both times, that individual is a dandelion. The average fitness for the whole population is 1.38. The average number of goal switches is 1.09. The archive is comprised of 0.08% daisies, 51.6% dandelions, 38.6% roses, and 9.7% sunflowers. Daisies have an average fitness of 0.76, dandelions an average of 1.44, roses an average of 1.33, and sunflowers an average of 1.32. $\mathcal{SC}_{dr}$ is 1.04. s is equal to 0.67 and $\mathcal{NI}_s$ is approximately 0.

The average fitness for the top 5% of individuals is 1.73. An average of 0.90 goal switches occur. Of the top 5% of solutions, 89.3% are dandelions, 9.3% are roses, and 1.4% are sunflowers. There are no daisies. Dandelions have an average fitness of 1.73, roses an average of 1.71, and sunflowers an average of 1.72. $\mathcal{SC}_{dr}$ is 0.05. s is equal to 0.74 and $\mathcal{NI}_s$ is approximately 0.

When filtered using a 95% whash-haar threshold, an average of 51 images were extracted from the archive. The average fitness of the best individual is 1.70. The average number of goal switches is 0.96. In this extricated population, 84.3% are dandelions and 15.7% are roses. There are no daisies or sunflowers. Dandelions have an average fitness of 1.72, and roses an average fitness of 1.59. $\mathcal{SC}_{dr}$ is approximately 0. s is equal to 0.76 and $\mathcal{NI}_s$ is approximately 0.

Figure 4.25 shows the percentile class distributions from the different sub-populations. Figure 4.26 shows the normalized frequencies of the different ranges of fitness values from all the solutions in the final archives across the two trials. Tables 4.34 - 4.35 contain the images with the best composite score from both the DNN and the whash-haar algorithm. Images are filtered into a sub-population with a whash-haar 95% threshold. Flower class types not found in the population or sub-population are indicated with an "NA."

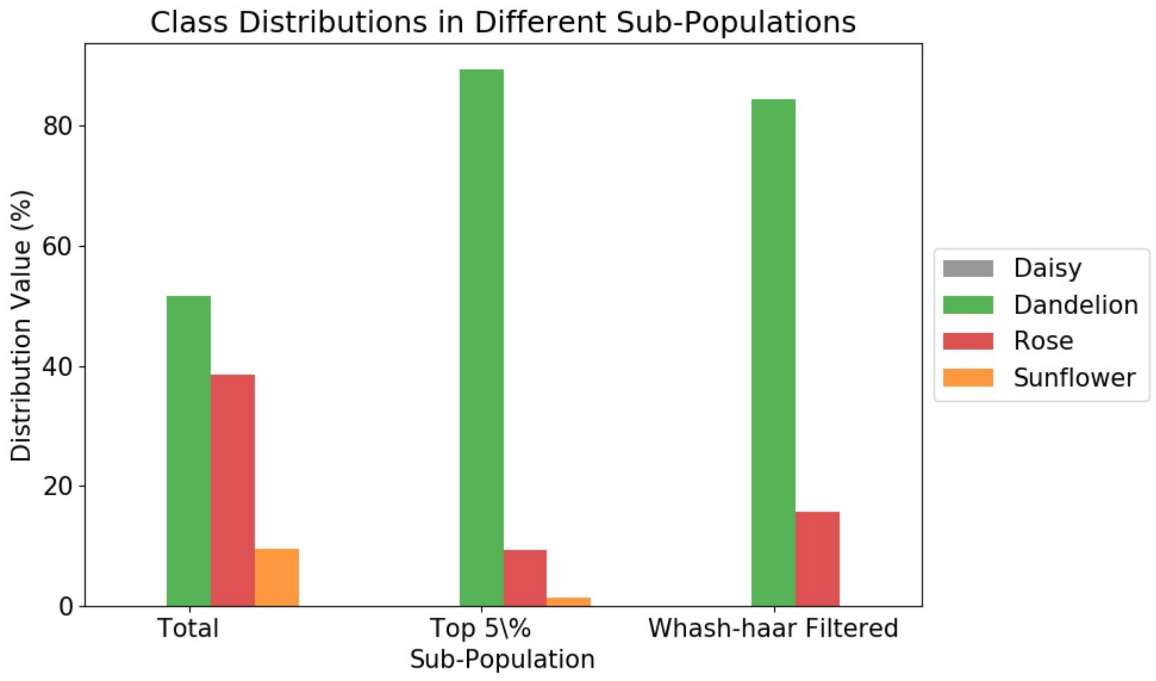


Figure 4.25: The distribution (in %) of different class types in different DNN-whash-haar composite scored 3D flower sub-populations (what fraction of the sub-population is flower type x?)

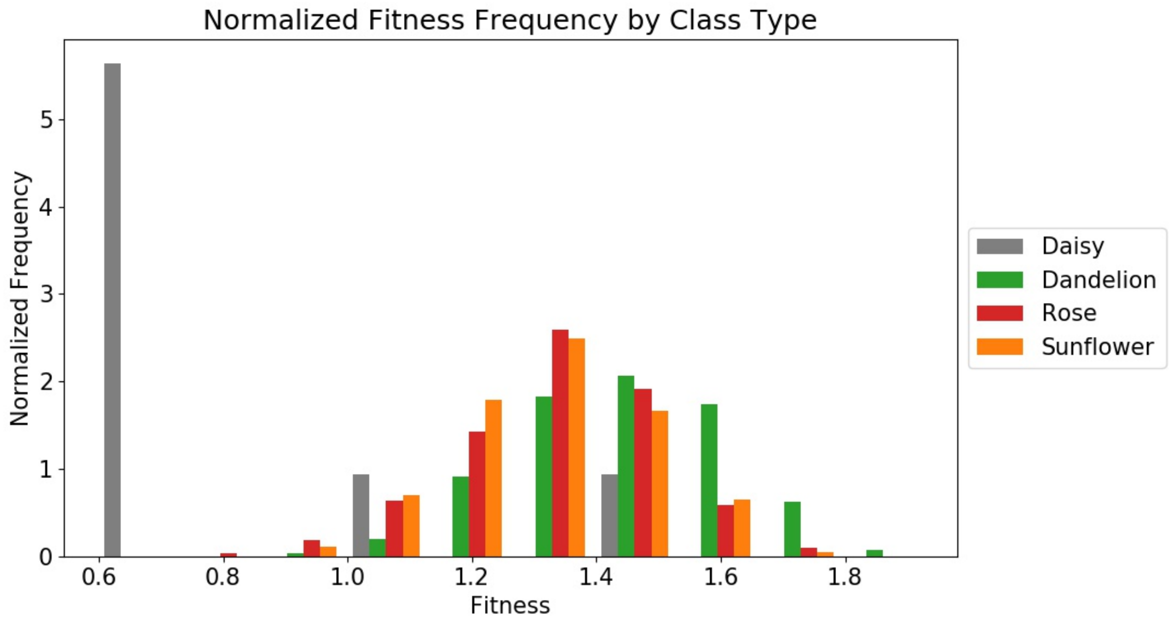


Figure 4.26: Normalized frequencies of fitness values for DNN-whash composite-scored 3D flowers in each class type (all flowers over two trials)

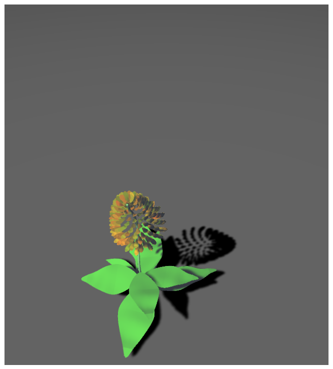
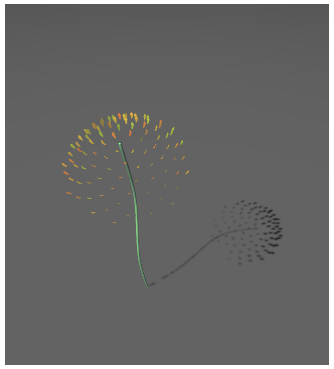
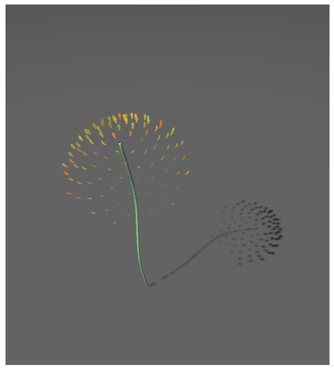
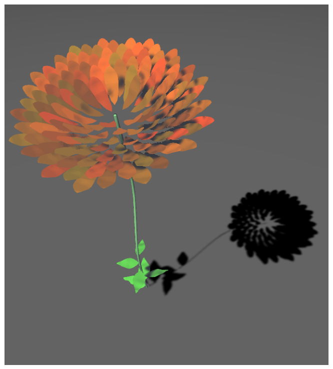
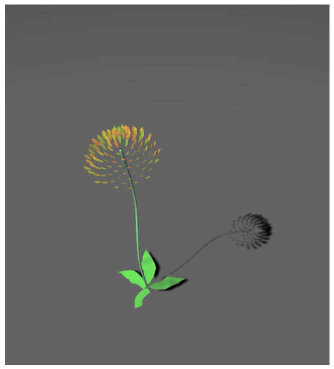
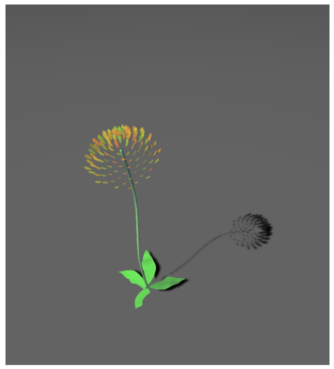
	Daisy	Hash-Filtered Daisy	Dandelion	Hash-Filtered Dandelion
Trial 1		NA		
Trial 2		NA		

Table 4.34: Best DNN-whash-haar scored 3D flower of each class type from the total and hash-filtered populations

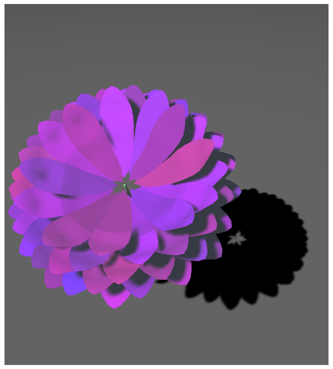
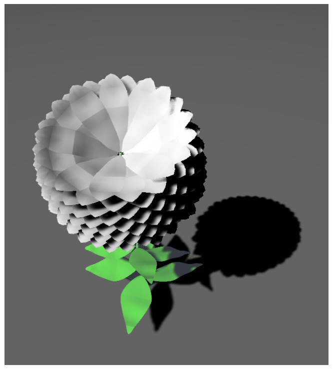
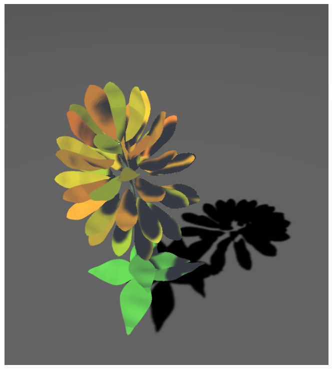
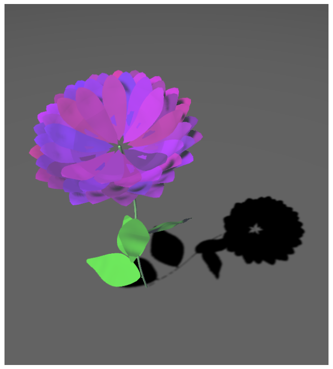
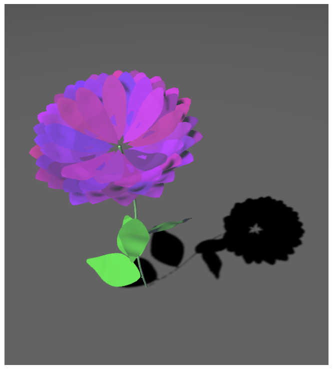
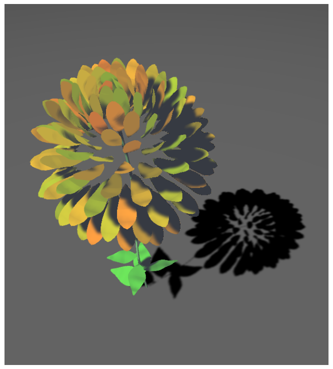
	Rose	Hash-Filtered Rose	Sunflower	Hash-Filtered Sunflower
Trial 1				NA
Trial 2				NA

Table 4.35: Best DNN-whash-haar scored 3D flower of each class type from the total and hash-filtered populations

Chapter 5: Conclusions and Future Work

For the state of the art techniques in PCG, generating multiple classes of objects with close visual similarity to representative target objects is still a difficult problem [35][48]. This is extremely relevant to asset generation, since many assets in simulation can share basic overall characteristics but be distinct visually. Being able to generate them diversely, and accurately, is extremely important. The goal of this work was to investigate whether, when utilizing generators more specific to the classes themselves, better results could be produced. To reduce the effects of dimensionality explosion encountered when creating objects that have lots of available parameters, CVT-MAP-Elites was used. The developed approach was tested on both 2 and 3 dimensional objects, specifically several types of flowers. The results are mixed with some promising outcomes, but clearly there are additional hurdles to overcome.

When using a standard GA, for both 2D and 3D flowers, the population converged very early relative to the algorithm's run-time, as shown by figures 4.3, 4.4, 4.5, and 4.6. As expected, it also converged to a population of identical, or extremely similar, solutions, and it did so only within one or two of the possible classes of flowers. The 3D flowers are remarkable in the sense that every trial produced almost the exact same rose (as shown in table C.10). This seems to indicate that this flower is an easily found, locally optimal solution that the network deems to be superior to most other generated solutions, as the same amount of flowers are generated, totally over the algorithm's life-cycle, as the CVT-MAP-Elites experiments. These solutions had an average fitness of 99% for 2D flowers and 99.5% for 3D flowers. Although high-performing, the solutions are not diverse (tables 4.2 and C.10). These results confirm the results in prior literature, that standard GAs are *not* the technique to use when trying to produce a large, diverse set of solutions. It should not be discounted that the results from the 3D rose in table C.10 are, subjectively, quite good - they look like roses. They just are not diverse.

CVT-MAP-Elites, as reported in other work, shows promise for producing a diverse set of high performing solutions. When generating 2D flowers with the DNN score alone, the average fitness of an individual in the top 5% of solutions was 93%. For 3D flowers, it was 96.7%. This is consistent with prior work in Innovation Engines, where many high-performing solutions were generated. CVT-MAP-Elites is also extremely consistent when it comes to filling the niches in the archive. For 2D flowers, 80% of 5000 niches are filled in all cases. 96.8% or 96.9% are filled in the case of 3D flowers. This is remarkably consistent, and it seems to suggest that the search space of the generator is being consistently explored with the current algorithm parameters, such as crossover and mutation rate. This suggests high reproducibility. It also shows the CVT-MAP-Elites algorithm is exploring a broad number of classes, resulting in the generation of many different high-fitness individuals across all classes. Contrasting this to the standard GA where (essentially) only a single individual of a single class was produced (tables 4.2 and C.10). When the hashing score was excluded in the fitness during CVT-MAP-Elites experiments, the top individual nearly reached peak fitness in all cases. When the hashing score was included, the average score of the best individual could fall, such as the case for 2D flower generation with the added phash score (see figures 4.16 and 4.16). So CVT-MAP-Elites can generate "good" solutions, but this notion of good is relative.

The fitnesses of the best individuals tend to be at maximum for many trials, with some exceptions that will be explored later. However, when looking at the images produced across all the trials, some of them are, subjectively, poor. They do not conform to what the human eye would consider to be a "good" dandelion or rose. Roses tend to dominate populations throughout many of the experiments (figures 4.7, 4.9, 4.11, 4.15, 4.23). Although they dominate the populations, in 2D flowers the results are poor. While clearly attempting to generate layered individuals of pink, purple, or reddish hues, the defined shape of a rose is never present. Sunflowers, as a class, are underrepresented in quantity and quality in all the experiments. None of the trial runs produced a sunflower that, subjectively, can be considered good. Many times, the sunflower is the reverse of its real life equivalent, a dark outer ring with a yellow interior. 2D dandelions are poor in general, resulting in images with long, thin tendrils as petals instead of the rounder shape of a typical dandelion. The 3D generator struggles to generate daisies or sunflowers that are easily distinguishable to the human eye. Some of the elements are there, like petal shape or color, but overall, the entire pattern doesn't get produced. While there are many poor results, there are other, more positive results. While 2D models fail to embody the color of a daisy, the models accurately generate the structure of a daisy. This is logical, as they tend to represent themselves in 2D populations prominently. Dandelions, of all scoring types, represent very well in structure in the 3D trials. The roses produced by the 3D generator experiments, shown in tables 4.10 and 4.11, are subjectively quite good. They are also, subjectively, much better than their 2D counterparts.

Adding a hashing score to the fitness, much like the results across all experiments, tends to produce mixed results. Whash-haar might have some implicit bias against 2D roses, as they rarely get selected in the hashing populations. However, considering 2D roses are quite poor, this might be by design. This is also suggested by tables 4.4-4.27. Roses are rarely filtered into the hashing sub-populations, if ever. Hashing algorithms seem to make 3D daisies better in some cases, such as dhash and phash (figures 4.30 4.31, 4.32, and 4.32), but they show no marked improvement for the other two hashing algorithms, ahash and whash-haar (figures 4.28 4.29, 4.34, and 4.34). For the other flowers, this is a lot less clear. For 2D flowers, there seems to be almost no improvement to any of the classes. For roses, it seems to make them worse. Dandelions and sunflowers do not appear to have any marked changes. The hashing algorithm and the DNN seem to have different criteria for what it takes to be a within a certain class, shown in the class distributions of different experiments like in figures 4.7 and 4.15. Within the hashing algorithms, there are different criteria for what it means to be close to a given image. This idea is supported by the varied results between hashing experiments.

Overall, 3D flowers without the hashing score added (tables 4.8-4.11) tend to perform, subjectively, the best for dandelions and roses. Moreover, the 2D generator performs consistently well for daisies (tables 4.4 and 4.7). The 3D generator is better at creating the multi-dimensional shapes required in a rose or a dandelion, but it lacks a way to generate the "circular center" that is present in the 2D generator. This would make the 3D generator naturally better at creating roses and dandelions and much less suited to generating daisies or sunflowers. It is more difficult to explain the 2D generator's bias. The images in figure 4.1 suggest that the 2D generator is capable of generating images that look very much like daisies, dandelions, roses, and sunflowers. Yet these images never present themselves in the main population. Daises are, structurally, made very well. However, a similar flower, the sunflower, doesn't make a similar appearance. The "circular center" is definitely capable of being generated, so this lack of sunflowers seems to be caused by the selection bias of the DNN rather than a limitation of the generator.

Color is also a uniquely problematic facet of all flowers generated. It seems that in most cases, but not all, color is disregarded as an important feature in flower generation. There are no purple dandelions. Yet, they continue to exist in experimental populations. On the other hand, sunflowers tend to bias far more to yellow, their natural color, than other classes. Pinkish hues are also common in roses, and the generated roses had pink or red flowers exclusively. Likewise, yellow, sometimes more orange, is the only color a sunflower can have.

A majority of the flowers generated exhibit a high fitness value, at least for the pure DNN scores. Although, subjectively they do not represent the class of flowers to which they belong. This might be because the DNN is being fooled, prior work has shown that DNN can be 'fooled' fairly easily [47]. It could also be a lack of data samples. While the data was augmented via cropping, scaling, and brightness changes, the amount of training data is limited, which could be detrimental to accurate classification. Lack of class diversity might also be a problem. The average goal switches for the CVT-MAP-Elites experiments range somewhere between 0.6 and 1.2. Consider the results in chapter 2.5, where the average goal switches for a particular experiment was 8.7. Prior literature suggests that large class diversity is helpful in generating better individuals. Four classes may not be enough to generate the subjectively good results for reasons related to morphology, or phenotype structure. More classes means more opportunities for unique morphologies that might be required in specific classes but are difficult to generate directly in that class. This idea is expanded upon when discussing goal switching (or lack of) for the standard GA.

Additionally, the number of goal switches for standard GAs are significantly lower than for CVT-MAP-Elites. In the two GA experiments, the average goal switches was 0.2 for 2D flowers and 0 for 3D flowers. The lack of goal switches could indicate that the algorithm converges faster, giving flowers less of an opportunity for goal switching. Important morphologies for generating good individuals may present themselves first in poorer, easier-to-find solutions. However, if these morphologies are not rewarded in some capacity (as they are not in a standard GA) then they may not be used in future generations [35][38][48].

Clearly, there are many different avenues future research could take. The Diversity Engine is designed to make the implementation of research for these questions much easier. The modularity of the framework allows for the usage of many different types of evaluation mechanisms as well as generation mechanisms. Some of these questions include:

- The reproduction of this experiment with different rendering techniques. Prior work suggests that the addition of different rendering/lighting techniques, and the addition of multiple evaluation angles of 3D objects has significant effect on performance. It was attempted for the 3D flowers seen here, but due to technical limitations, it was dismissed in favor of a single angle.

- A comparison of different evaluation mechanisms, such as real-time object detection networks like You Only Look Once (YOLO) [57]. These would enable the discovery of what the parts of the image that the DNN considers relevant. There are many flowers in the populations that seem to contain flowers inside black rings. Is the network recognizing these "interior flowers" and, somehow, the black backdrop acts as backdrop for the network?

- The comparison of CVT-MAP-Elites with other QD algorithms, such as NSLC or regular MAP-Elites. This would clarify how much CVT-MAP-Elites contributes to generating objects with a higher number of features, such as 2D flowers.

- The addition of many more PCG classes such as rocks, trees, and terrain. Does building all the assets for a digital world all at once actually create better PCG assets?

- It is possible that color is more integral to sunflowers and roses than it is to daisies and dandelions. The dandelion data set contained images of yellow dandelions and its corresponding seedling, gray form. And the data set also contained many daisies that were multi-colored. Color could be a far more important facet of image detection for roses and sunflowers than daises and dandelions for these reasons. Dandelions can be yellow or grey, so it is possible that this lack of color consistency drives down the importance of color in pattern recognition in the DNN. Training a network to test this hypothesis could be valuable for both DNN and PCG research.

As image recognition continues to improve, more opportunities for automatically generating diverse, quality PCG will continue to present themselves. As with previous Innovation Engine research, the results here are mixed. Some genuinely good solutions were generated, but they are intermixed with many more, subjectively, poor solutions. A more consistent generation of objects is the goal. This is absolutely necessary to achieve one of the original goals of this work, which is to generate interestingly diverse, high-fidelity assets for games and simulation. The Diversity Engine, built on a modular framework, can serve as a launch point for the rapid prototyping of experiments in pursuit of that goal. This framework will hopefully act as one of the first practical applications existing under the intellectual domain of Innovation Engines.

In conclusion, results show that:

- Diversity Engines that utilize Quality Diversity (QD) algorithms provide a better pathway to high-fidelity, diverse object generation than typical optimization methods such as standard GAs.
- The results themselves, while not subjectively good in all cases, contain some extremely good examples of diverse class generation. Especially when compared to the standard GA.
- As QD algorithms, and DNN, continue to improve, the opportunity for diverse, high-fidelity PCG generation will expand. With those expanding opportunities, the utility of this framework as demand for a modular, easily-customizable tool will also increase.

References

[1] D. Ashlock. Automatic generation of game elements via evolution. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG2010*, pages 289–296, 09 2010.

[2] S. Baluja, D. Pomerleau, and T. Jochem. Towards automated artificial evolution for computer-generated images. *Connection Science*, 6(2-3):325–354, 1994.

[3] H. Bauke. Tina's random number generator library, 2014. https://www.numbercrunch.de/trng.

[4] H. Bauke and S. Mertens. Random numbers for large-scale distributed monte carlo simulations. *Physical Review E*, 75:1–14, June 2007.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[6] Blizzard. Starcraft, 1998.

[7] Content Blockchain. Testing different image hash functions. https://content-blockchain.org/research/testing-different-image-hash-functions.

[8] O. Ciftcioglu and M. S. Bittermann. Solution diversity in multi-objective optimization: A study in virtual reality. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 1019–1026, 2008.

[9] J. Clune and H. Lipson. Evolving 3d objects with a generative encoding inspired by developmental biology. *SIGEVOlution*, 5(4):2–12, November 2011.

[10] J. Correia, P. Machado, J. Romero, and A. Carballal. Evolving figurative images using expression-based evolutionary art. In *ICCC*, pages 24–31, 2013.

[11] A. Cully, J. Clune, D. Tarapore, and J.B. Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, May 2015.

[12] A. Cully and Y. Demiris. Quality and diversity optimization: A unifying modular framework. *IEEE Transactions on Evolutionary Computation*, 22:245–259, 2017.

[13] S. Dahlskog and J. Togelius. In *EvoApplications*, 2014.

[14] S. Dahlskog, J. Togelius, and M. J. Nelson. Linear levels through n-grams. In *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content Services,*

AcademicMindTrek '14, pages 200–206, New York, NY, USA, 2014. Association for Computing Machinery.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[16] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, June 2009.

[17] Q. Du, V. Faber, and M. Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM Rev.*, 41(4):637–676, December 1999.

[18] D. Floreano and C. Mattiussi. *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. The MIT Press, 2008.

[19] Hello Games. No man's sky, 2016.

[20] R. C. Gonzalez and R. E Woods. *Digital Image Processing (Fourth Edition)*. Pearson, 2018.

[21] I. J. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016.

[22] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, and S. Ozair. Generative adversarial networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS '14, pages 2672–2680, Cambridge, MA, USA, 2014. MIT Press.

[23] D. Gravina, A. Khalifa, A. Liapiss, J. Togelius, and G. Yannakakis. Procedural content generation through quality diversity. In *Proceedings of the IEEE Conference on Games 2019*, pages 1–8, August 2019.

[24] E. J. Hastings, R. K. Guha, and K. O. Stanley. Automatic content generation in the galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):245–263, December 2009.

[25] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 1026–1034, USA, 2015. IEEE Computer Society.

[26] G. E. Hinton and R. S. Zemel. Autoencoders, minimum description length and helmholtz free energy. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS '93, pages 3–10, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[27] B. Horn, G. Smith, R. Masri, and J. Stone. Visual information vases: Towards a framework for transmedia creative inspiration. In *Proceedings of the Sixth International Conference on Computational Creativity*, pages 182–188, July 2015.

[28] J. Buchner. Imagehash. https://pypi.org/project/ImageHash/2.2/.

[29] K. A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, USA, 1975. AAI7609381.

[30] K. A. De Jong. *Evolutionary Computation - A Unified Approach.* January 2006.

[31] L. Ju, Q. Du, and M. Gunzburger. Probabilistic methods for centroidal voronoi tessellations and their parallel implementations. *Parallel Comput.*, 28(10):1477–1500, October 2002.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS '12, pages 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[33] M. Laumanns, L. Thiele, K. Deb, and E. Zitzler. Combining convergence and diversity in evolutionary multiobjective optimization. *Evol. Comput.*, 10(3):263â282, September 2002.

[34] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio. Object recognition with gradient-based learning. In *Shape, Contour and Grouping in Computer Vision*, page 319, Berlin, Heidelberg, 1999. Springer-Verlag.

[35] J. Lehman, S. Risi, and J. Clune. Creative generation of 3d objects with deep learning and innovation engines. In *Proceedings of the Seventh International Conference on Computational Creativity*, pages 180–187, Paris, France, 2016. UPMC.

[36] J. Lehman and K. O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19:189–223, June 2011.

[37] J. Lehman and K. O. Stanley. Novelty search and the problem with objectives. In *Genetic Programming Theory and Practice IX*, pages 37–56. Springer New York, New York, NY, 2011.

[38] R. Lenski, C. Ofria, R. Pennock, and C. Adami. The evolutionary origin of complex features. *Nature*, 423:139–144, June 2003.

[39] A. Liapis, H. P. Martínez, J. Togelius, and G. N. Yannakakis. Transforming exploratory creativity with delenox,. In *Proceedings of the Fourth International Conference on Computational Creativity*, pages 56–63, Sydney, Australia, June 2013.

[40] K. MacDonald. Get real! behind the scenes of red dead redemption 2 - the most realistic video game ever made, October 2018. https://www.theguardian.com/games/2018/oct/24/get-real-behind-the-scenes-of-red-dead-redemption-2-the-most-realistic-video-game-ever-made.

[41] P. Machado, J. Correia, and J. Romero. Expression-based evolution of faces. In *International Conference on Evolutionary and Biologically Inspired Music, Sound, Art and Design*, pages 187–198, April 2012.

[42] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[43] mattatz. unity-procedural-flower. https://github.com/mattatz/unity-procedural-flower.

[44] Mojang. Minecraft, 2009.

[45] J. B. Mouret. Novelty-based multiobjectivization. In Stéphane Doncieux, Nicolas Bredèche, and Jean-Baptiste Mouret, editors, *New Horizons in Evolutionary Robotics*, pages 139–154, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[46] J.B. Mouret and J. Clune. Illuminating search spaces by mapping elites. *ArXiv*, 2015.

[47] A. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 427–436, 2015.

[48] A. Nguyen, J. Yosinski, and J. Clune. Understanding innovation engines: Automated creativity and improved stochastic optimization via deep learning. *Evolutionary Computation*, 24(3):545–572, September 2016.

[49] D. Oranchak. Evolutionary algorithm for generation of entertaining shinro logic puzzles. In *Applications of Evolutionary Computation*, pages 181–190, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[50] K. Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[51] D. Petrov. Wavelet image hash in python, July 2016. https://fullstackml.com/wavelet-image-hash-in-python-3504fdd282b5.

[52] T. Phillips. The human cost of red dead redemption 2, May 2019. https://www.eurogamer.net/articles/2018-10-25-the-human-cost-of-red-dead-redemption-2.

[53] P. Prusinkiewicz. Graphical applications of l-systems. In *Proceedings on Graphics Interface '86/Vision Interface '86*, pages 247–253, CAN, 1986. Canadian Information Processing Society.

[54] J. K. Pugh, L. Soros, and K. O. Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:40, July 2016.

[55] J. K. Pugh, L. B. Soros, P. A Szerlip, and K. O. Kenneth. Confronting the challenge of quality diversity. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 967–974, New York, NY, USA, 2015. Association for Computing Machinery.

[56] C. Reas and B Fry. Processing: Programming for the media arts. *AI Soc.*, 20(4):526–538, August 2006.

[57] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[58] S. Risi and J. Togelius. Procedural content generation: From automatically generating game levels to increasing generality in machine learning. *ArXiv*, November 2019.

[59] RoboRosewater. Roborosewater (@roborosewater), July 2015. https://twitter.com/roborosewater?lang=en.

[60] Adrian Rosebrock. Image hashing with opencv and python, November 2017. https://www.pyimagesearch.com/2017/11/27/image-hashing-opencv-python/.

[61] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115:211–252, October 2015.

[62] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, USA, 3rd edition, 2009.

[63] J. Schmidhuber. Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts. *Connect. Sci.*, 18:173–187, June 2006.

[64] J. Schmidhuber. Simple algorithmic principles of discovery, subjective beauty, selective attention, curiosity creativity. In *Discovery Science*, pages 26–38, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[65] J. Schreier. The horrible world of video game crunch, September 2016. https://kotaku.com/crunch-time-why-game-developers-work-such-insane-hours-1704744577.

[66] J. Secretan, N. Beato, D. D'Ambrosio, A. Rodriguez, A Campbell, and K. O. Stanley. Picbreeder: Evolving pictures collaboratively online. In *Proceedings of Computer Human Interaction Conference*, pages 1759–1768, April 2008.

[67] A. Semuels. Video game creators are burned out and desperate for change, June 2019. https://time.com/5603329/e3-video-game-creators-union/.

[68] M. Snider. Video game makers must manage crunch time, increase diversity to improve mental health, July 2019. https://www.usatoday.com/story/tech/gaming/2019/07/09/layoffs-crunch-time-hurting-video-game-workers-mental-health/1681531001/.

[69] T. Soule, S. Heck, T. Haynes, N. Wood, and B. Robison. Darwin's demons: Does evolution improve the game? In *Applications of Evolutionary Computation*, pages 435–451. Springer International Publishing, 2017.

[70] K. O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8:131–162, June 2007.

[71] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, June 2002.

[72] K. O. Stanley and R. Miikkulainen. A taxonomy for artificial embryogeny. *Artificial life*, 9:93–130, February 2003.

[73] A. J. Summerville, M. Guzdial, M. Mateas, and M. Riedl. Learning player tailored content from observation: Platformer level generation from video traces using lstms. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2016.

[74] A. J. Summerville, S. Philip, and M. Mateas. Mcmcts pcg 4 smb : Monte carlo tree search to guide platformer level generation. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2015.

[75] A. J. Summerville, S. Snodgrass, M. Guzdial, C. HolmgÃrd, A. K. Hoover, A Isaksen, A. Nealen, and J. Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018.

[76] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[77] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and ZB. Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[78] D. Takahashi. The deanbeat: How much did red dead redemption 2 cost to make? (updated), Oct 2019. https://venturebeat.com/2018/10/26/the-deanbeat-how-much-did-red-dead-redemption-2-cost-to-make/.

[79] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. N. Yannakakis. Multiobjective exploration of the starcraft map space. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 265–272, Aug 2010.

[80] J. Togelius, M. Preuss, and G. Yannakakis. Towards multiobjective procedural map generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, New York, NY, USA, June 2010. Association for Computing Machinery.

[81] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, September 2011.

[82] M. Toy, G. Wichmann, and Ken Arnold. Rogue, 1980.

[83] Unity Technologies. Unity. https://unity.com.

[84] V. Vassiliades, K. Chatzilygeroudis, and J. Mouret. Using centroidal voronoi tessellations to scale up the multidimensional archive of phenotypic elites algorithm. *IEEE Transactions on Evolutionary Computation*, 22(4):623–630, August 2018.

[85] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, and S. Risi. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 221–228, New York, NY, USA, 2018. Association for Computing Machinery.

[86] J. Wenzel, J. Rhinelander, and D. Moldovan. pybind11 - seamless operability between c++11 and python, 2016. https://github.com/pybind/pybind11.

[87] D. Whitley, S Rana, and R. B. Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7(1):33–47, 1999.

[88] G.N. Yannakakis and J. Togelius. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, 2:147–161, 2011.

Appendix A: Experimental Parameters and Seeds

A.1 DNN Training Parameters and Accuracy

Network: Inception V3 [77]

Classes: Daisies, Dandelions, Roses, Sunflowers

Training Epochs: 300 - **Training Batches:** 100

Random Crop: 10% - **Random Scale** - 10% - **Random brightness**: 20%

Test Percentage After Data Augmentation: 10%

Data Samples Prior to Data Augmentation:

- Daisies: 633
- Dandelions: 898
- Roses: 641
- Sunflowers: 699

Approximate Data Samples After Data Augmentation:

- Daisies: 1360
- Dandelions: 1660
- Roses: 1240
- Sunflowers: 1780

Class Accuracy After Training:

- Daisies: 91%
- Dandelions: 89%
- Roses: 96%
- Sunflowers: 93%

A.2 Standard GA Experimental Parameters and Seeds

A.2.1 2D Flowers

Population Size: 250 - Number of Generations - 200

Crossover Rate: 30% - Mutation Rate - 5% - Mutation Change Interval [-10%/10%]

A.2.2 3D Flowers

Population Size: 250 - Number of Generations - 100

Crossover Rate: 30% - Mutation Rate - 10% - Mutation Change Interval [-10%/10%]

A.2.3 No Hashing Fitness

- **Trial 1 (2D and 3D Flowers)**: Factory Seed: 275 - CVT-MAP-Elites Seed: 275
- **Trial 2 (2D and 3D Flowers)**: Factory Seed: 250 - CVT-MAP-Elites Seed: 250
- **Trial 3 (2D Flowers)**: Factory Seed: 92 - CVT-MAP-Elites Seed: 92
- **Trial 3 (3D Flowers)**: Factory Seed: 292 - CVT-MAP-Elites Seed: 292
- **Trial 4 (2D and 3D Flowers)**: Factory Seed: 1011 - CVT-MAP-Elites Seed: 1011
- **Trial 5 (2D and 3D Flowers)**: Factory Seed: 6033 - CVT-MAP-Elites Seed: 6033

A.3 CVT-MAP-Elites Experimental Parameters and Seeds

A.3.1 2D Flowers

Archive Max Size: 5000

Batch Size: 50 - Number of Batches - 1000

Crossover Rate: 30% - Mutation Rate - 5% - Mutation Change Interval [-10%/10%]

A.3.2 3D Flowers

Archive Max Size: 5000

Batch Size: 50 - Number of Batches - 50

Crossover Rate: 30% - Mutation Rate - 10% - Mutation Change Interval [-10%/10%]

A.3.3 No Hashing Fitness

- **Trial 1 (2D and 3D Flowers)**: Factory Seed: 3 - CVT-MAP-Elites Seed: 1
- **Trial 2 (2D and 3D Flowers)**: Factory Seed: 6 - CVT-MAP-Elites Seed: 6
- **Trial 3 (2D and 3D Flowers)**: Factory Seed: 124 - CVT-MAP-Elites Seed: 124
- **Trial 4 (2D and 3D Flowers)**: Factory Seed: 623 - CVT-MAP-Elites Seed: 623
- **Trial 5 (2D and 3D Flowers)**: Factory Seed: 555 - CVT-MAP-Elites Seed: 555

A.3.4 ahash Fitness Added

- **Trial 1 (2D and 3D Flowers)**: Factory Seed: 663 - CVT-MAP-Elites Seed: 663
- **Trial 2 (2D and 3D Flowers)**: Factory Seed: 547 - CVT-MAP-Elites Seed: 547
- **Trial 3 (2D Flowers Only)**: Factory Seed: 915 - CVT-MAP-Elites Seed: 915
- **Trial 4 (2D Flowers Only)**: Factory Seed: 877 - CVT-MAP-Elites Seed: 877
- **Trial 5 (2D Flowers Only)**: Factory Seed: 499 - CVT-MAP-Elites Seed: 499

A.3.5 dhash Fitness Added

- **Trial 1 (2D and 3D Flowers)**: Factory Seed: 743 - CVT-MAP-Elites Seed: 743
- **Trial 2 (2D and 3D Flowers)**: Factory Seed: 599 - CVT-MAP-Elites Seed: 599
- **Trial 3 (2D Flowers Only)**: Factory Seed: 213 - CVT-MAP-Elites Seed: 213
- **Trial 4 (2D Flowers Only)**: Factory Seed: 719 - CVT-MAP-Elites Seed: 719
- **Trial 5 (2D Flowers Only)**: Factory Seed: 269 - CVT-MAP-Elites Seed: 269

A.3.6 phash Fitness Added

- **Trial 1 (2D and 3D Flowers)**: Factory Seed: 201 - CVT-MAP-Elites Seed: 201
- **Trial 2 (2D and 3D Flowers)**: Factory Seed: 700 - CVT-MAP-Elites Seed: 700

- **Trial 3 (2D Flowers Only)**: Factory Seed: 353 - CVT-MAP-Elites Seed: 353
- **Trial 4 (2D Flowers Only)**: Factory Seed: 467 - CVT-MAP-Elites Seed: 467
- **Trial 5 (2D Flowers Only)**: Factory Seed: 163 - CVT-MAP-Elites Seed: 163

A.3.7 whash-haar Fitness Added

- **Trial 1 (2D and 3D Flowers)**: Factory Seed: 33 - CVT-MAP-Elites Seed: 33
- **Trial 2 (2D and 3D Flowers)**: Factory Seed: 72 - CVT-MAP-Elites Seed: 72
- **Trial 3 (2D Flowers Only)**: Factory Seed: 301 - CVT-MAP-Elites Seed: 301
- **Trial 4 (2D Flowers Only)**: Factory Seed: 67 - CVT-MAP-Elites Seed: 67
- **Trial 5 (2D Flowers Only)**: Factory Seed: 223 - CVT-MAP-Elites Seed: 223

Appendix B: Subjectively Good CVT-MAP-Elites Generated Images

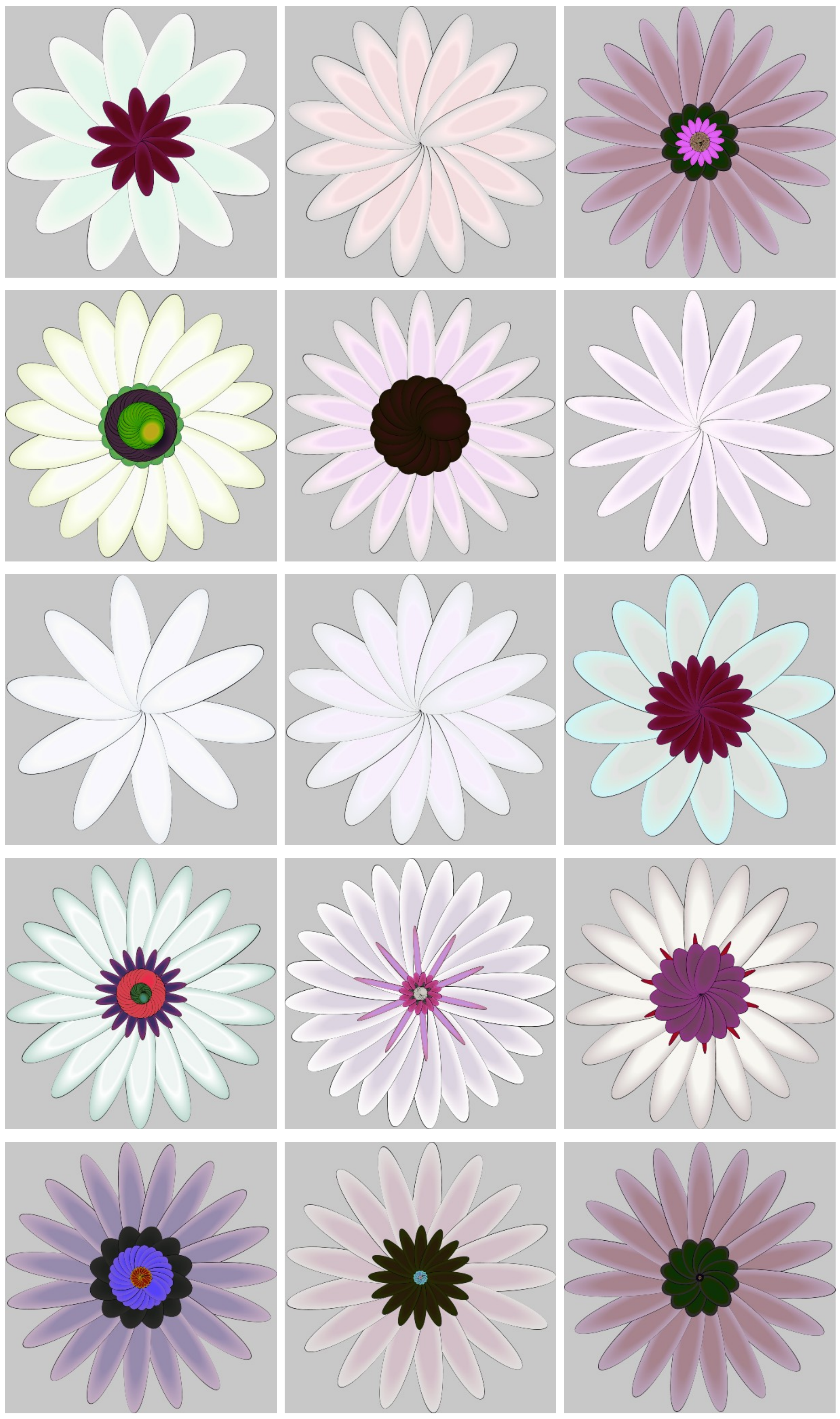


Table B.1: Some of the subjectively best 2D flowers in the CVT-MAP-Elites no-hash score experiments (all five trials) - all are daisies

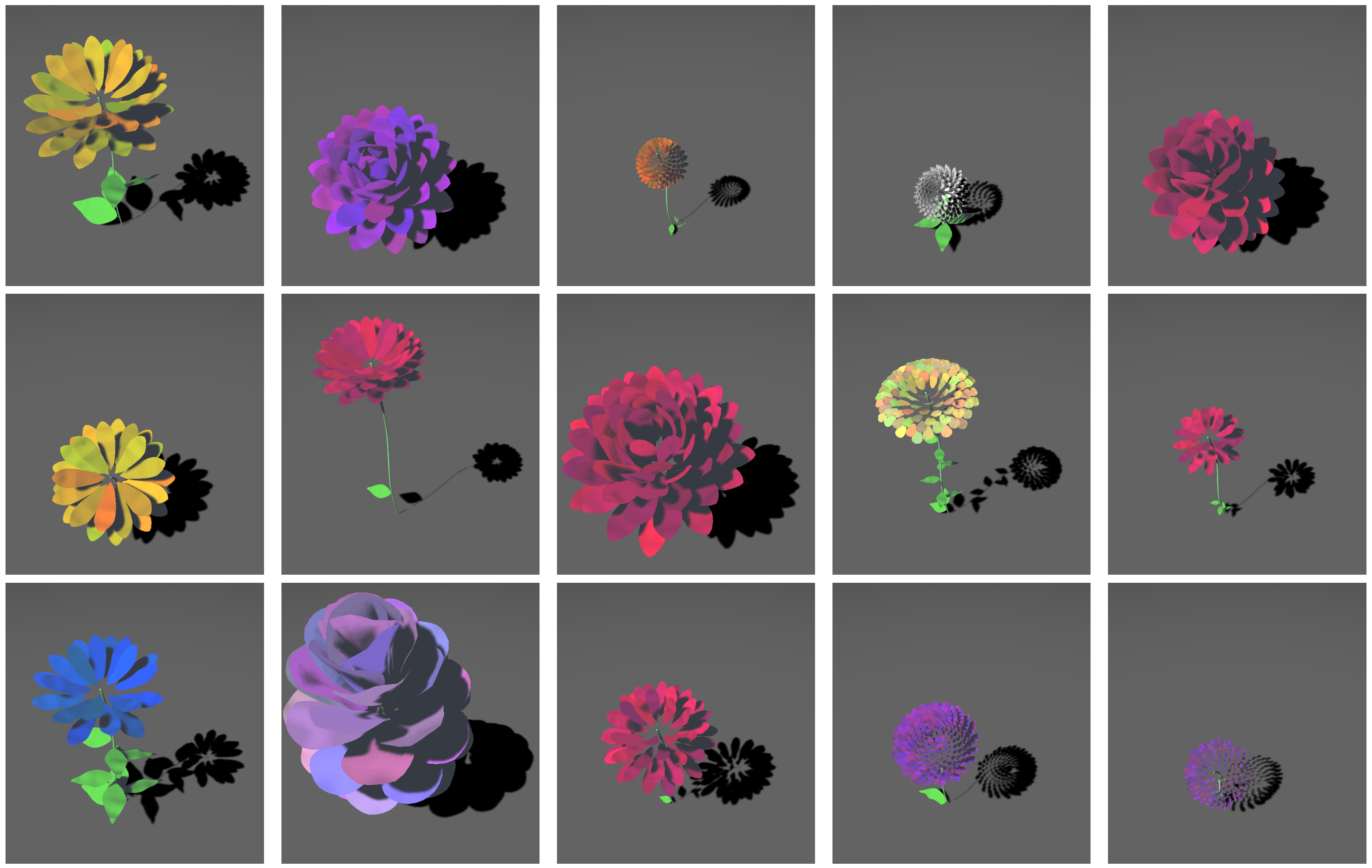


Table B.2: Some of the subjectively best 3D flowers in the CVT-MAP-Elites no-hash score experiments (all five trials) - Top row: sunflower, rose, dandelion, dandelion, rose - Middle row: sunflower, daisy, rose, sunflower, daisy - Bottom row: daisy, rose, daisy, dandelion, dandelion

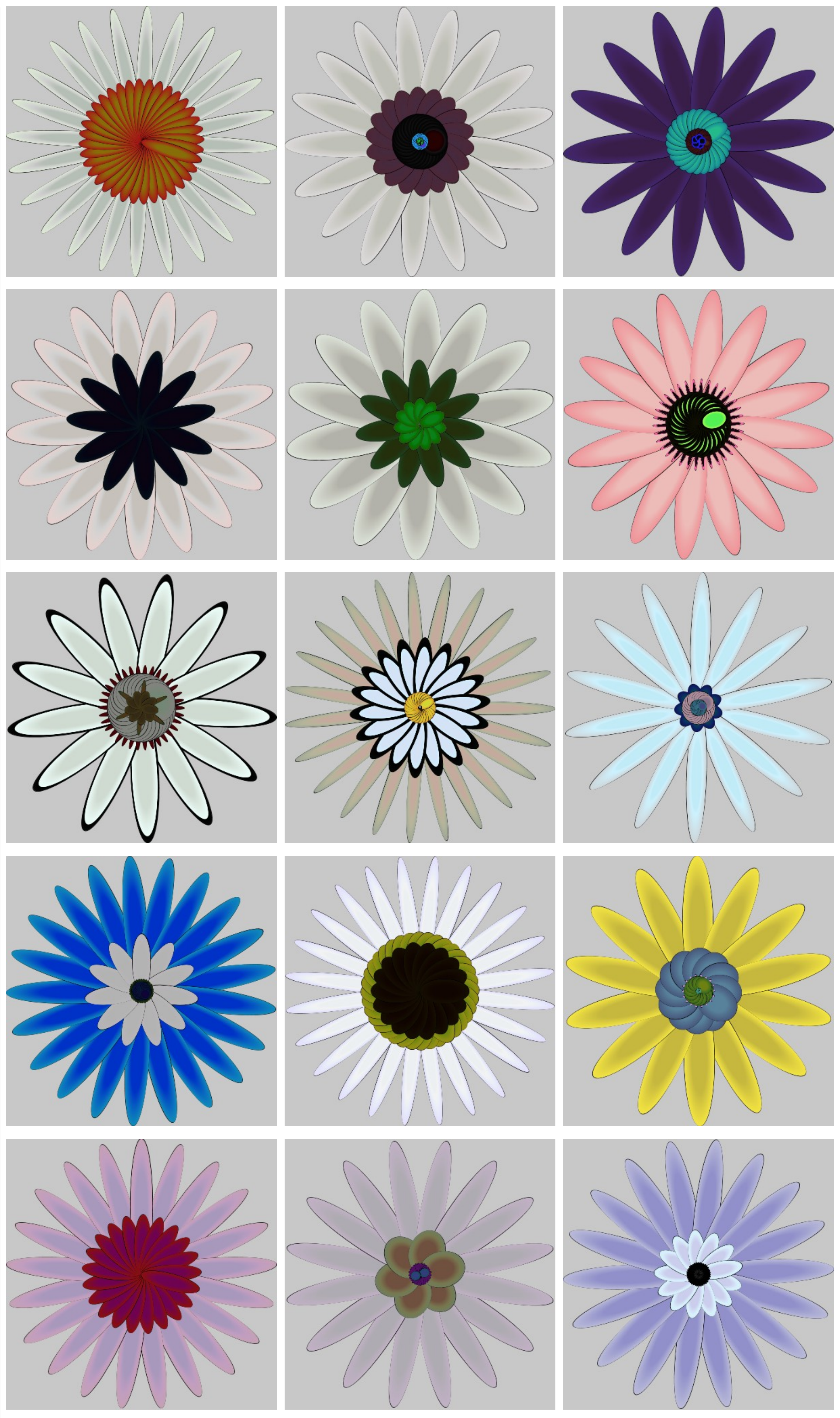


Table B.3: Some of the subjectively best 2D flowers in the CVT-MAP-Elites DNN-ahash score experiments (all five trials) - all are daisies

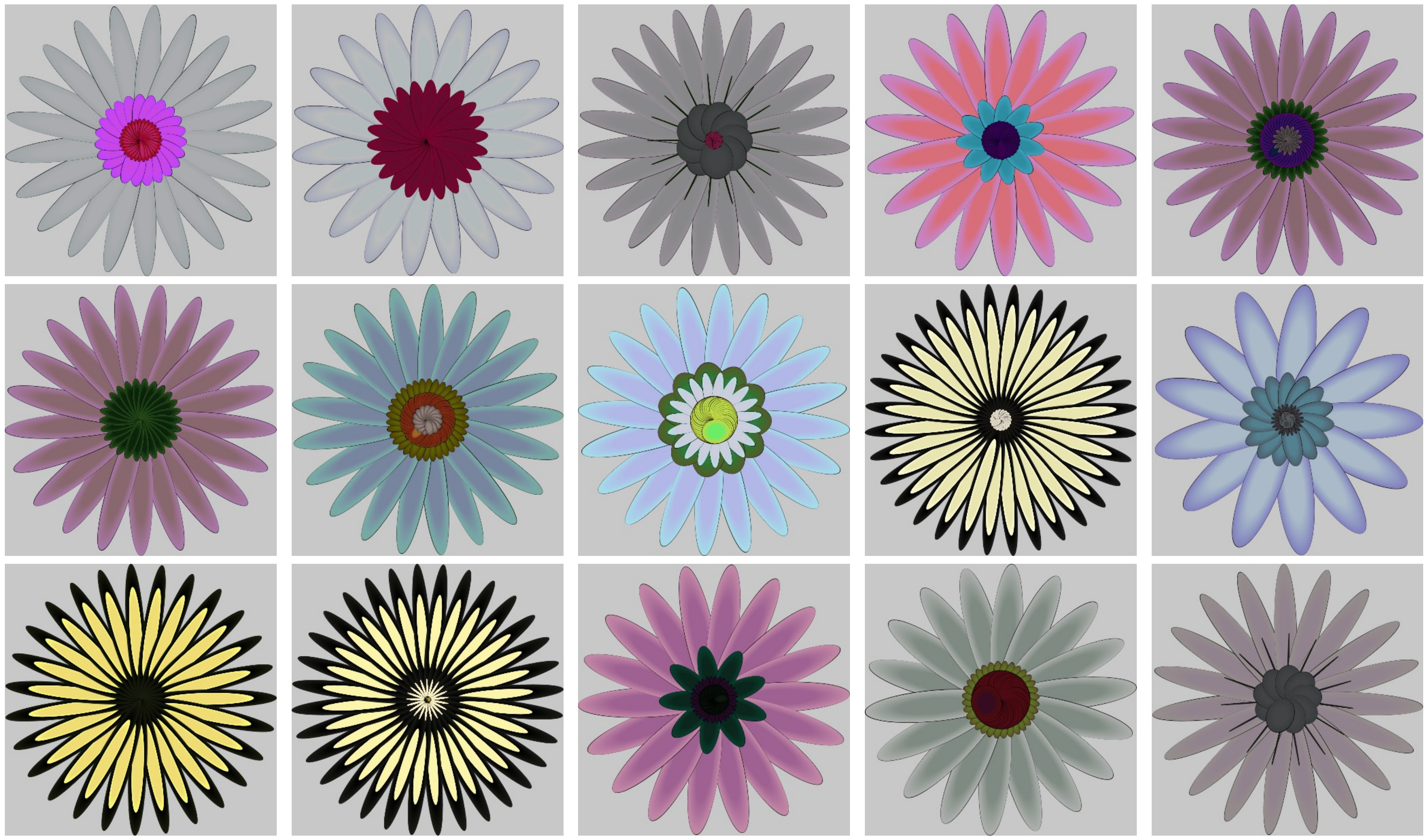


Table B.4: Some of the subjectively best 2D flowers in the CVT-MAP-Elites DNN-dhash score experiments (all five trials) - Top row: daisy, daisy, daisy, daisy, daisy - Middle row: daisy, daisy, daisy, sunflower, daisy - Bottom row: sunflower, sunflower, daisy, daisy, daisy

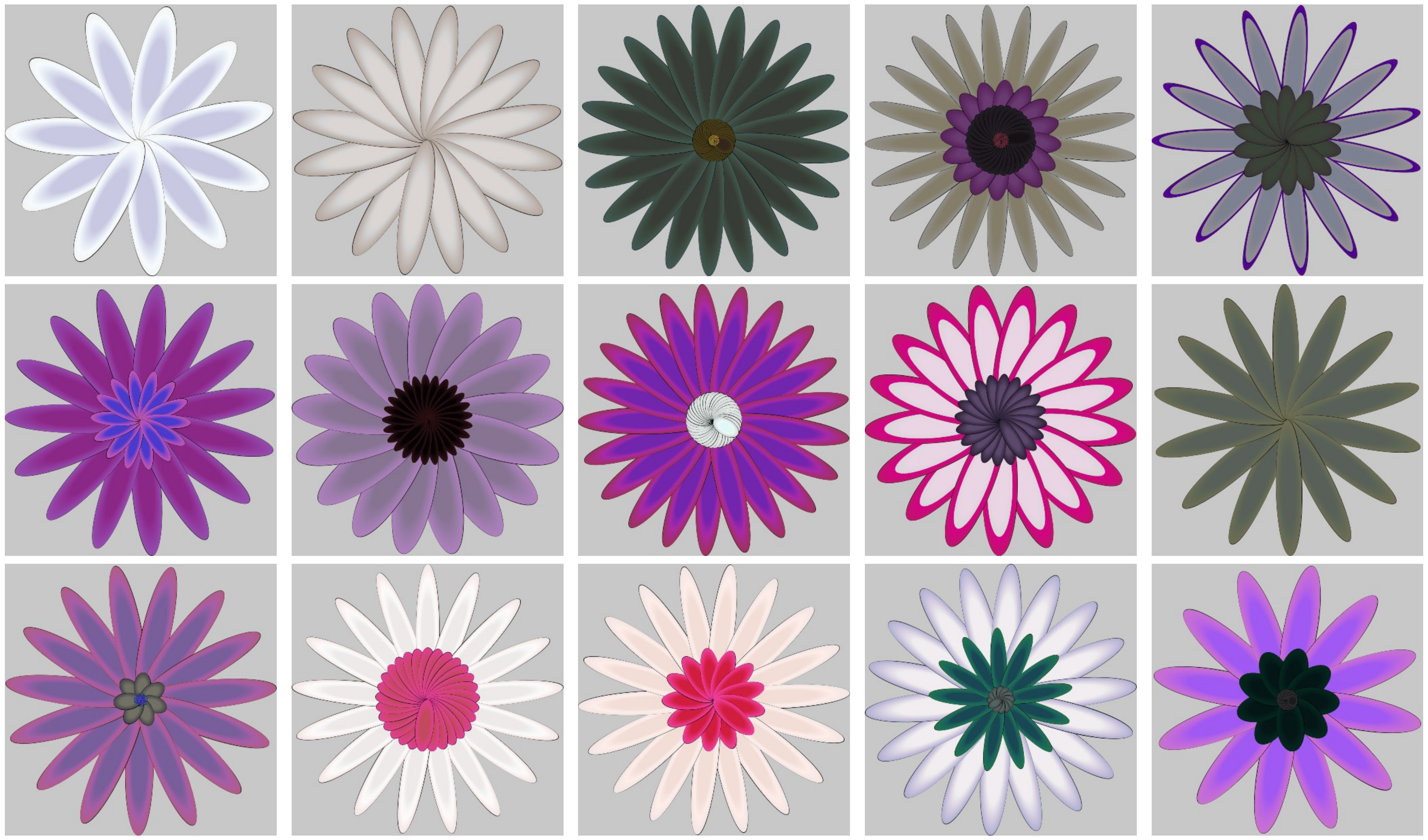


Table B.5: Some of the subjectively best 2D flowers in the CVT-MAP-Elites DNN-phash score experiments (all five trials) - all are daisies

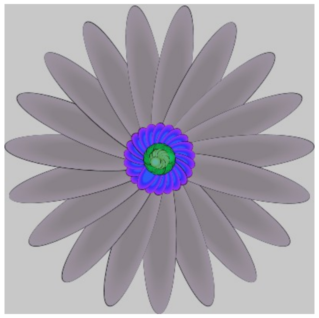
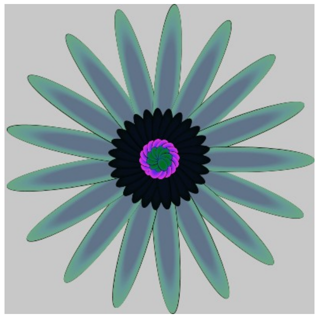
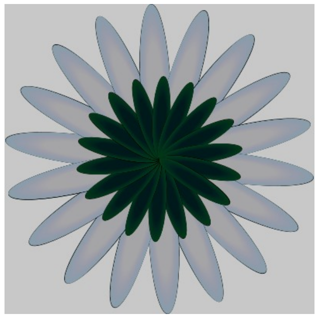
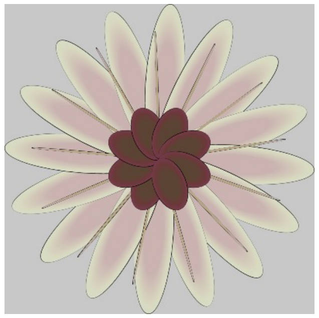
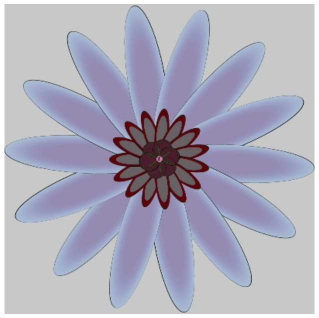
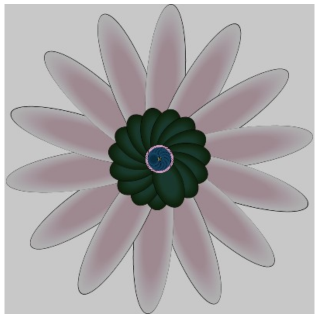
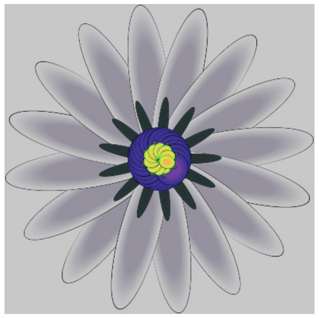
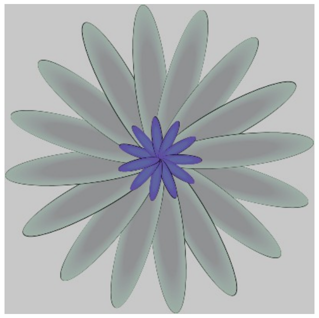
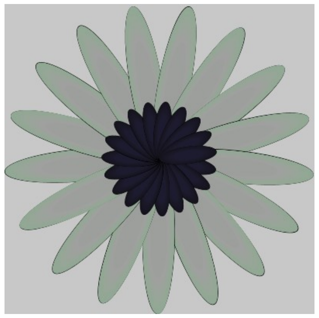
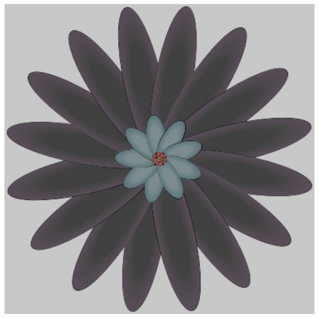
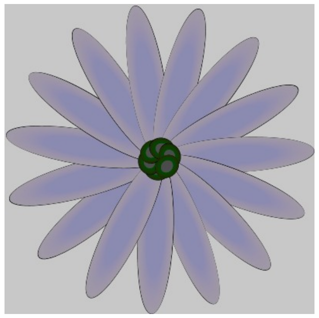
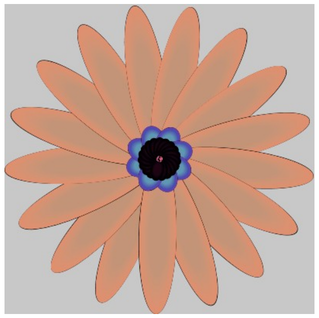
				
				
				

Table B.6: Some of the subjectively best 2D flowers in the CVT-MAP-Elites DNN-whash score experiments (all five trials) - all are daisies

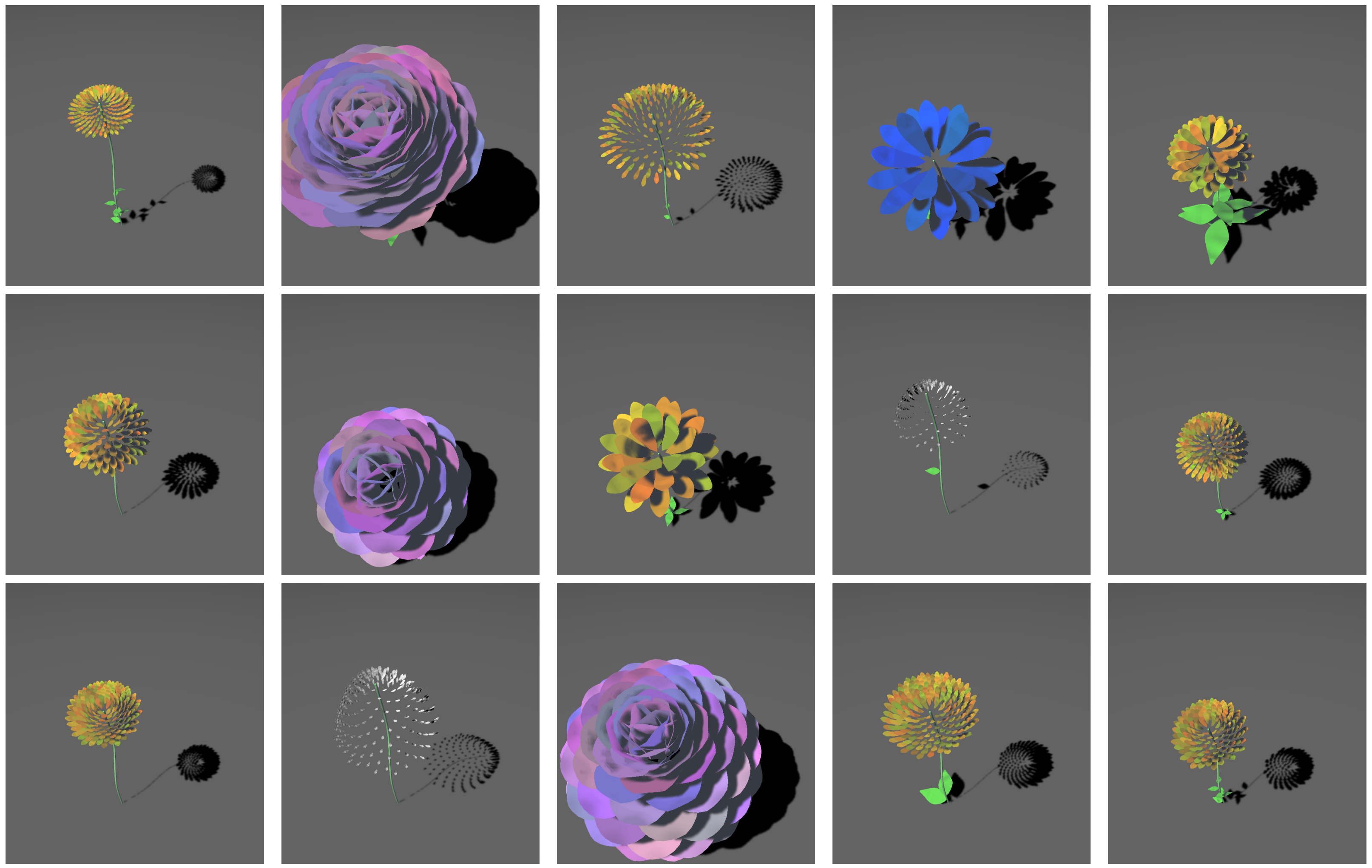


Table B.7: Some of the subjectively best 3D flowers in the CVT-MAP-Elites DNN-ahash composite score experiments (all two trials) - Top row: dandelion, rose, dandelion, daisy, sunflower - Middle row: dandelion, rose, sunflower, dandelion, dandelion - Bottom row: dandelion, dandelion, rose, dandelion, dandelion

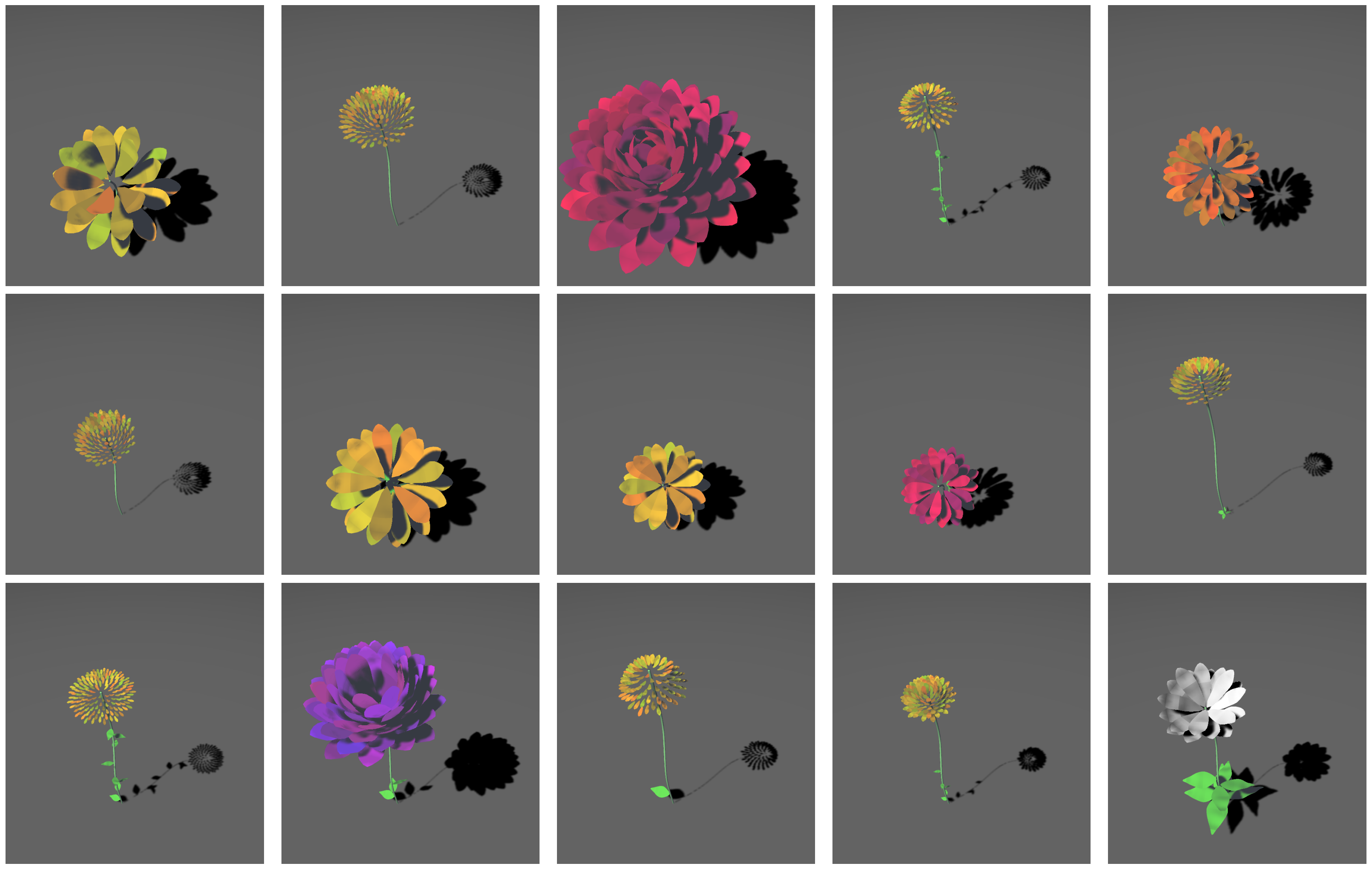


Table B.8: Some of the subjectively best 3D flowers in the CVT-MAP-Elites DNN-ahash composite score experiments (all two trials) - Top row: sunflower, dandelion, rose, dandelion, sunflower - Middle row: dandelion, sunflower, sunflower, daisy, dandelion - Bottom row: dandelion, rose, dandelion, dandelion, daisy

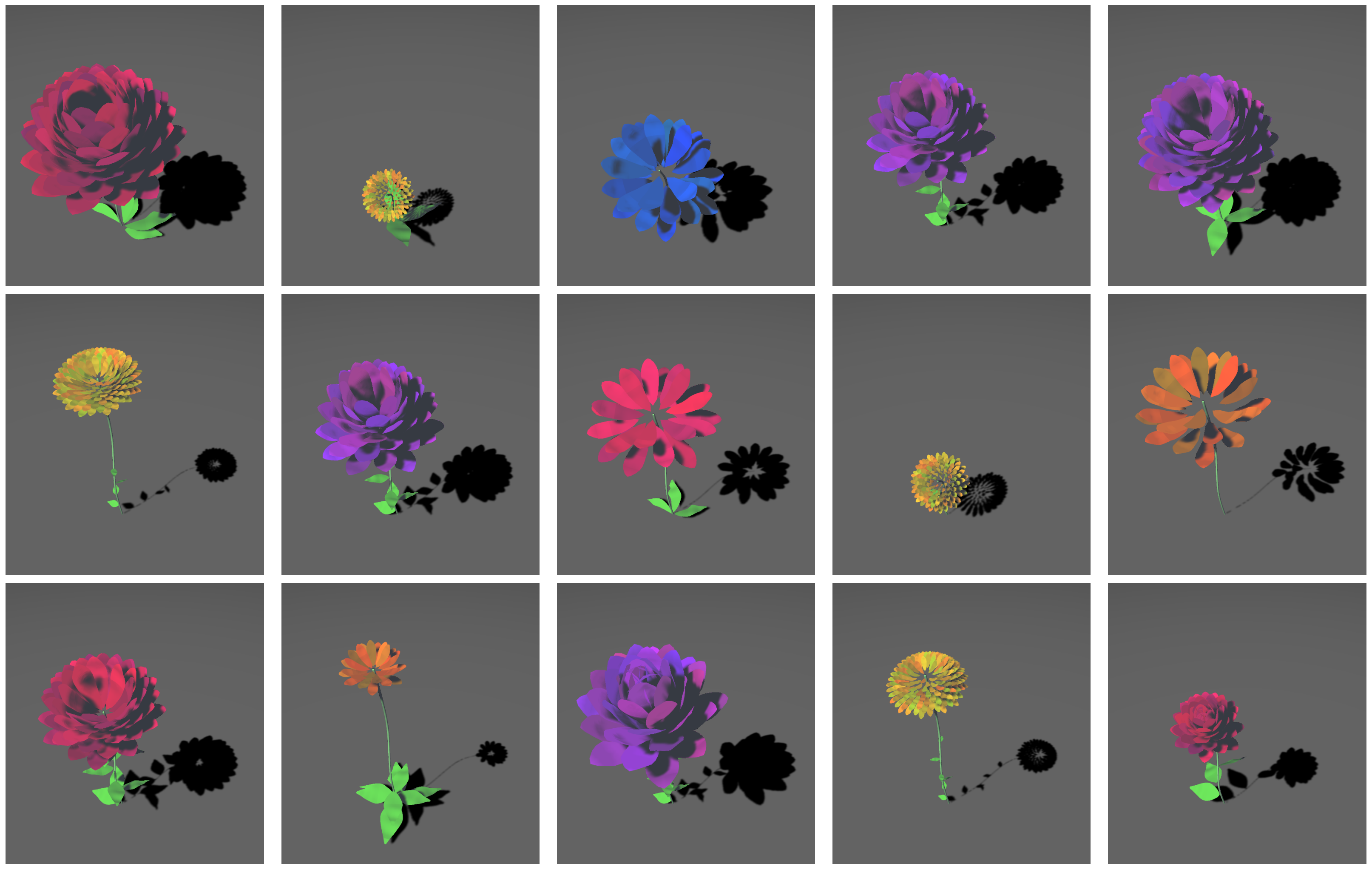


Table B.9: Some of the subjectively best 3D flowers in the CVT-MAP-Elites DNN-phash composite score experiments (all two trials) - Top row: rose, dandelion, daisy, rose, rose - Middle row: dandelion, rose, daisy, dandelion, daisy - Bottom row: rose, daisy, rose, dandelion, rose

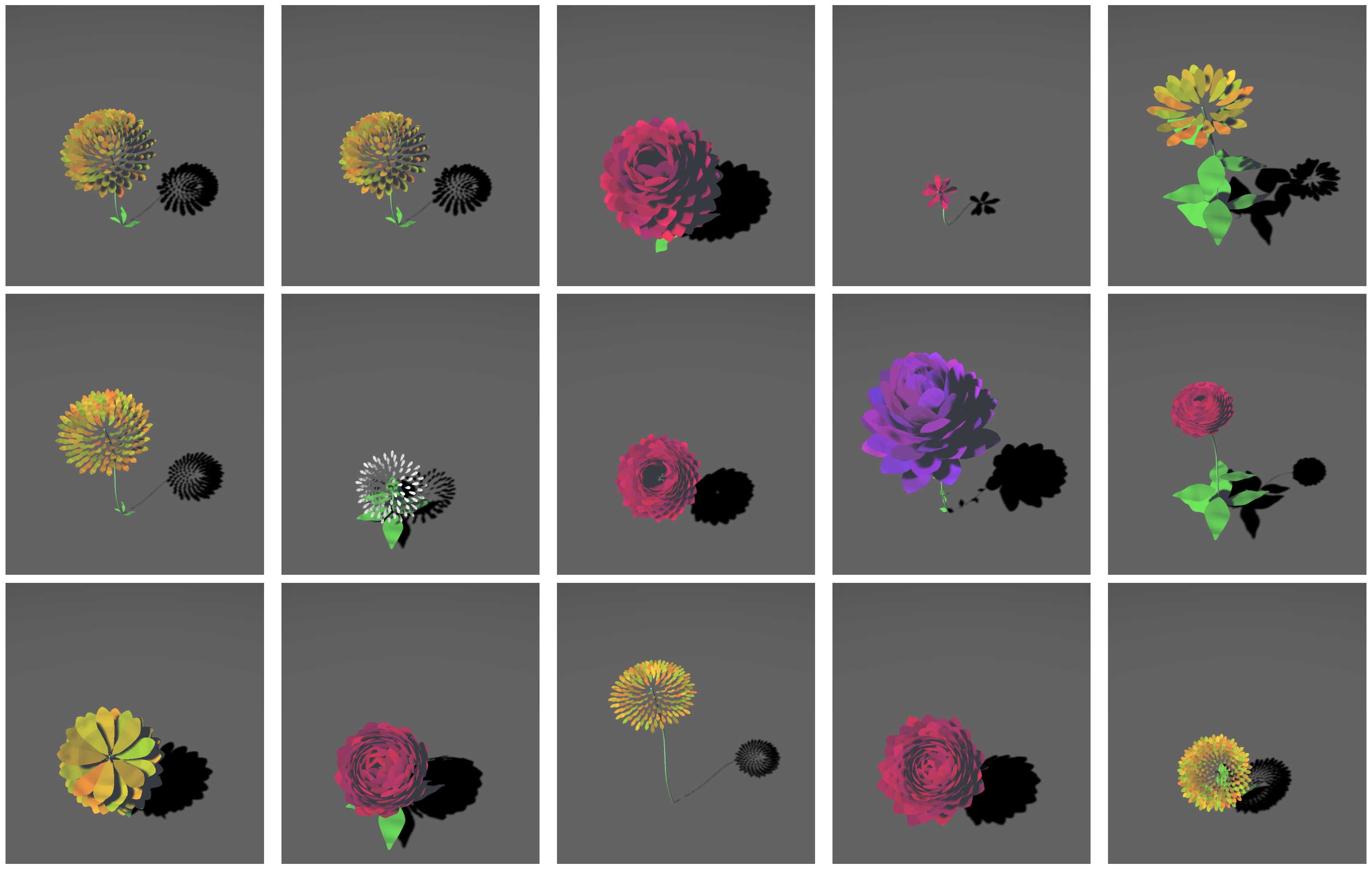


Table B.10: Some of the subjectively best 3D flowers in the CVT-MAP-Elites DNN-whash-haar composite score experiments (all two trials) - Top row: dandelion, dandelion, rose, daisy, sunflower - Middle row: dandelion, dandelion, rose, rose, rose - Bottom row: sunflower, rose, dandelion, rose, dandelion

Appendix C: Subjectively Bad CVT-MAP-Elites Generated Images

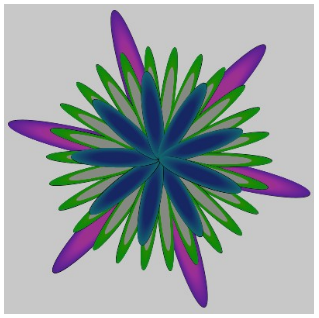
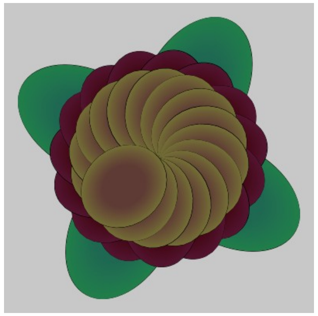
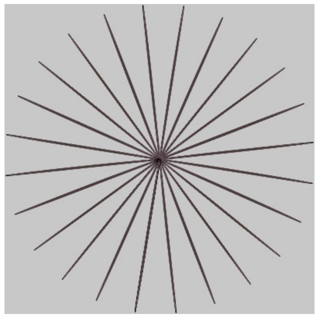
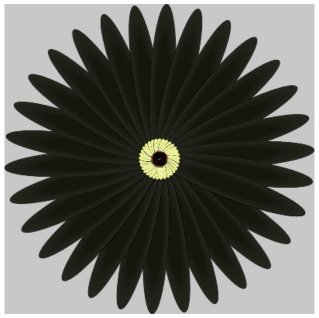
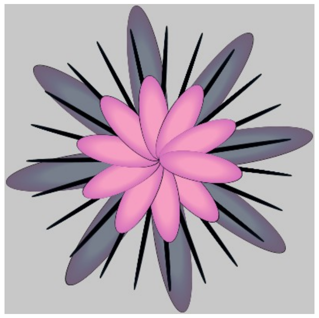
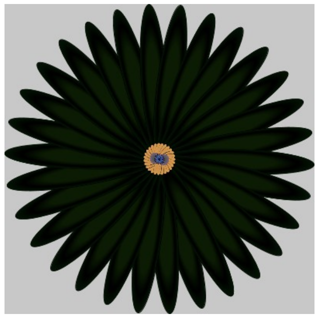
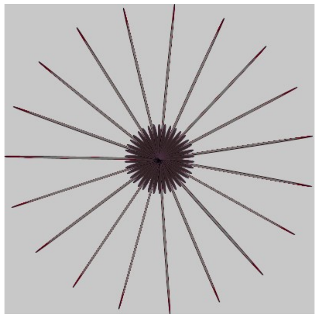
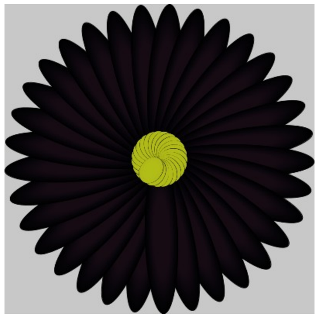
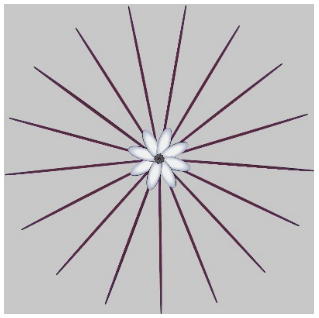
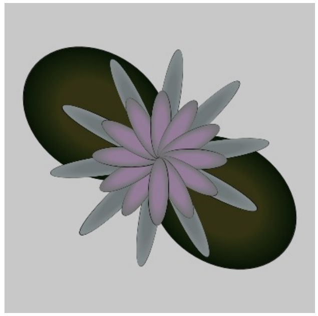
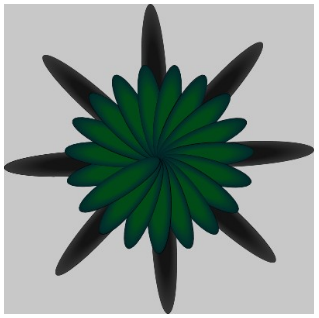
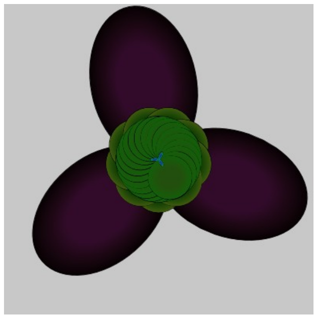
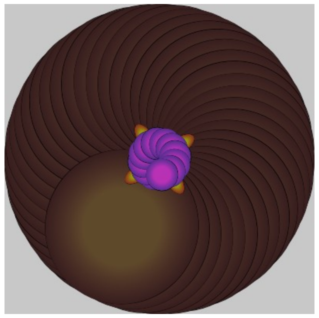
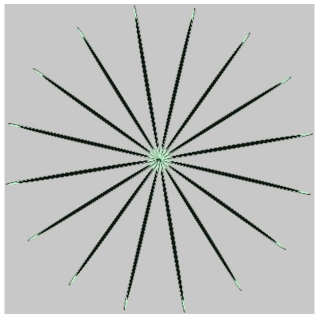
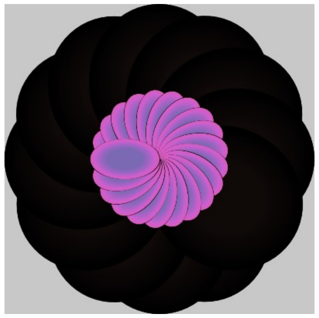
				
				
				

Table C.1: Some subjectively bad 2D flowers from the top 5% of solutions in the CVT-MAP-Elites no-hash score experiments (all five trials) - Top row: daisy, rose, dandelion, sunflower, daisy - Middle Row - sunflower, dandelion, sunflower, dandelion, daisy - Bottom row: daisy, rose, rose, dandelion, rose

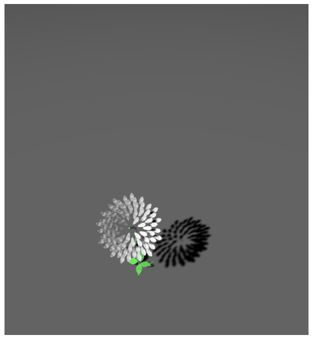
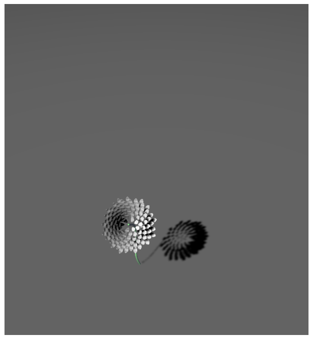
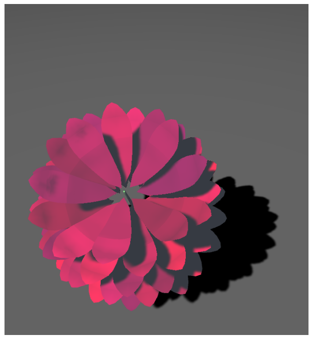
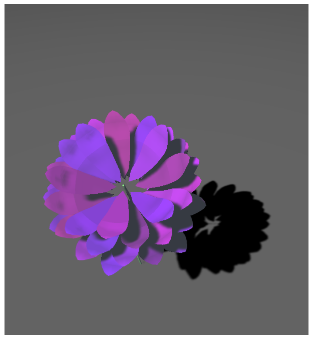
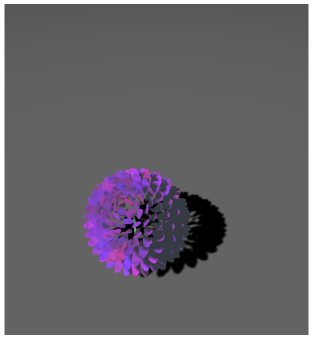
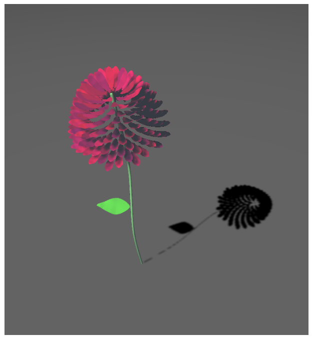
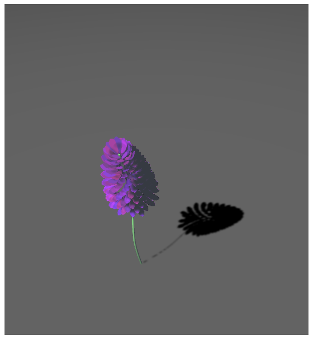
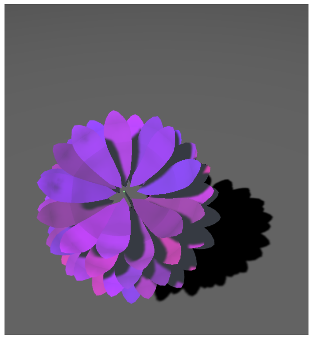
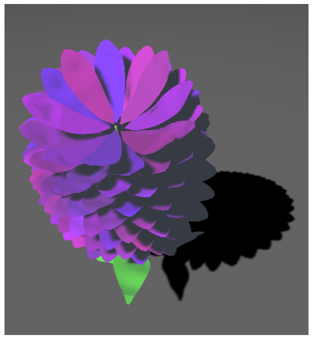
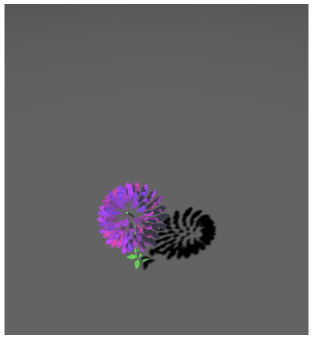
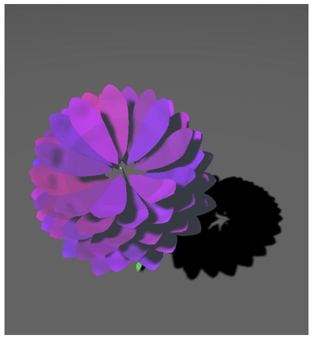
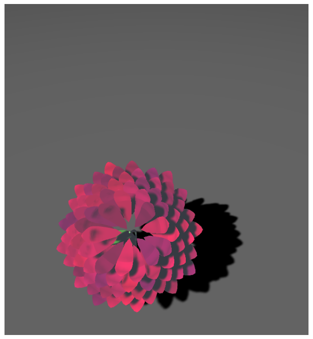
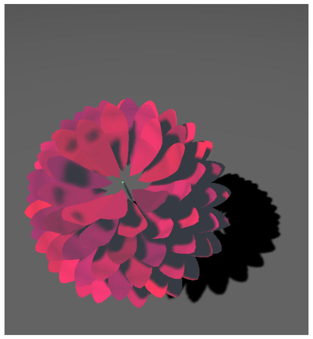
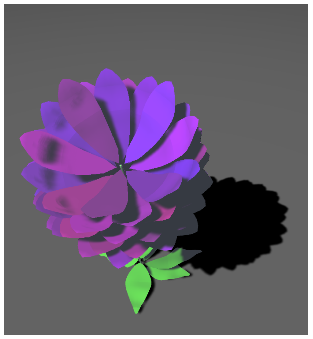
				
				
				

Table C.2: Some subjectively bad 3D flowers from the top 5% of solutions in the CVT-MAP-Elites no-hash score experiments (all five trials) - Top row: dandelion, dandelion, rose, rose, dandelion - Middle Row - dandelion, dandelion, rose, rose, rose - Bottom row: dandelion, rose, rose, rose, rose

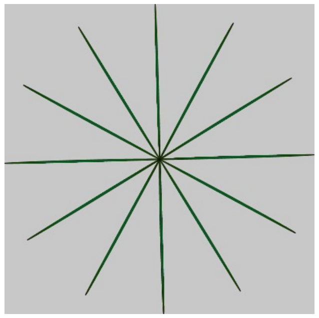
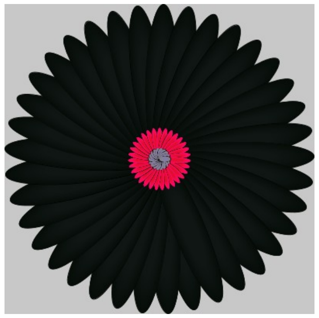
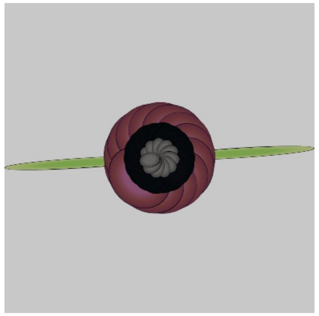
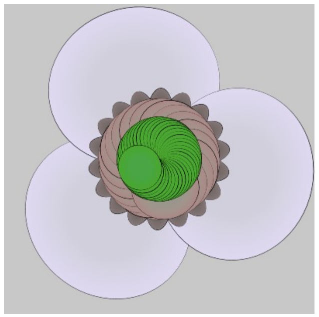
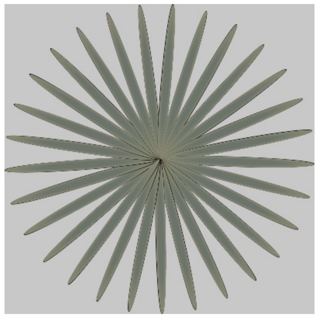
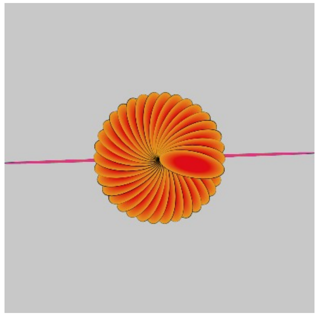
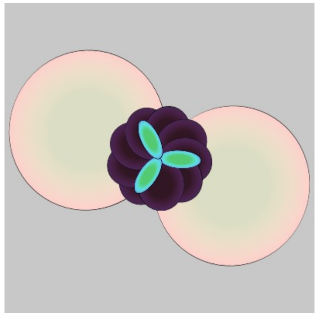
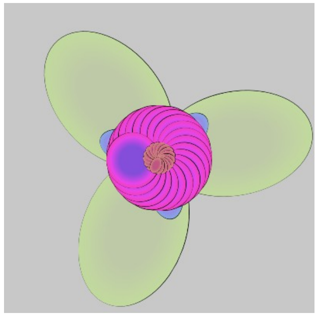
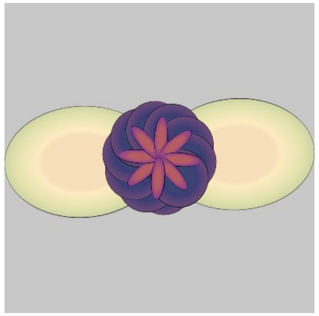
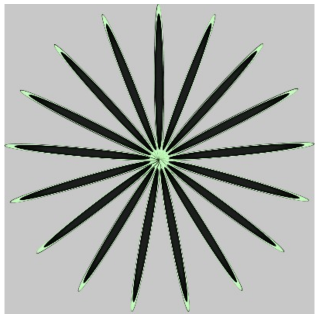
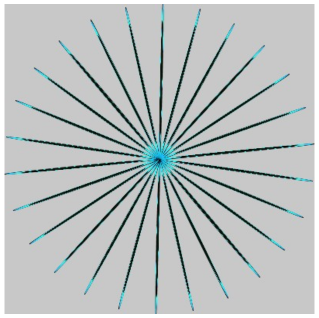
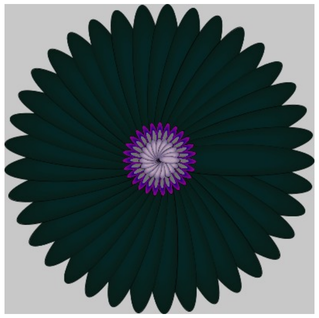
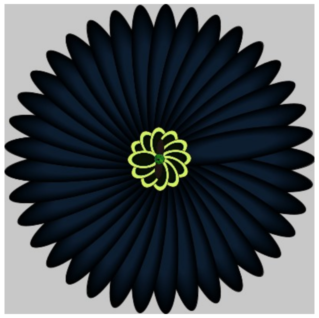
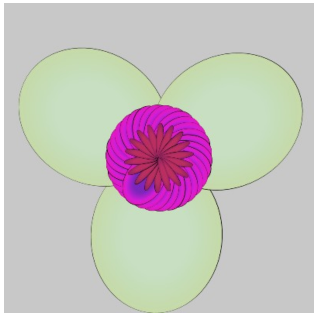
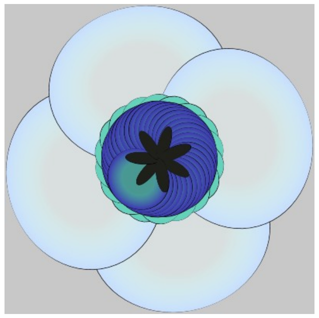
				
				
				

Table C.3: Some subjectively bad 2D flowers from the top 5% of solutions in the CVT-MAP-Elites DNN-ahash composite score experiments (all five trials) - Top row: dandelion, sunflower, rose, rose, dandelion - Middle Row - rose, rose, rose, rose, dandelion - Bottom row: dandelion, sunflower, sunflower, rose, rose

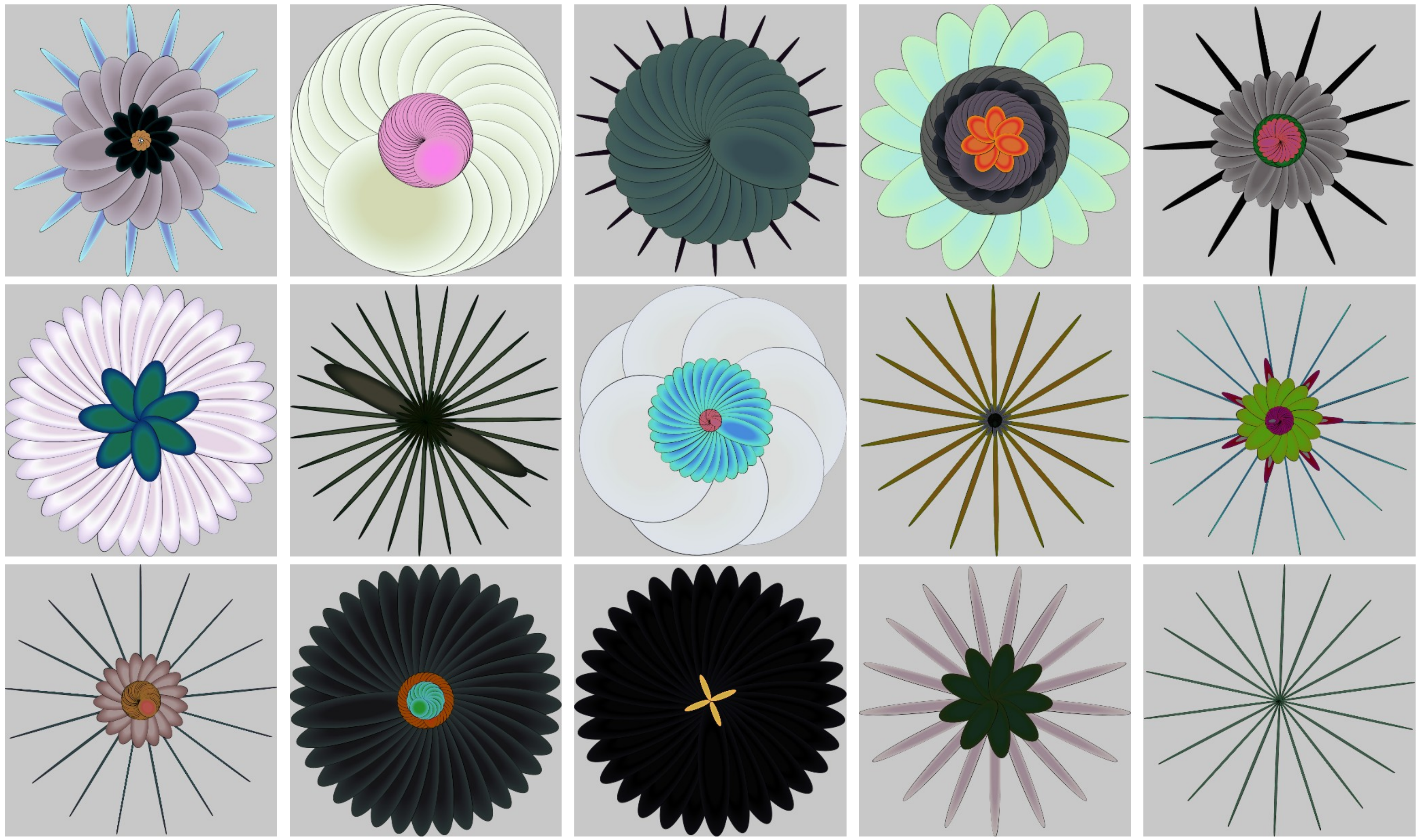


Table C.4: Some subjectively bad 2D flowers from the top 5% of solutions in the CVT-MAP-Elites DNN-dhash composite score experiments (all five trials) - Top row: daisy, rose, dandelion, rose, daisy - Middle Row - rose, dandelion, rose, dandelion, dandelion - Bottom row: dandelion, sunflower, sunflower, daisy, dandelion

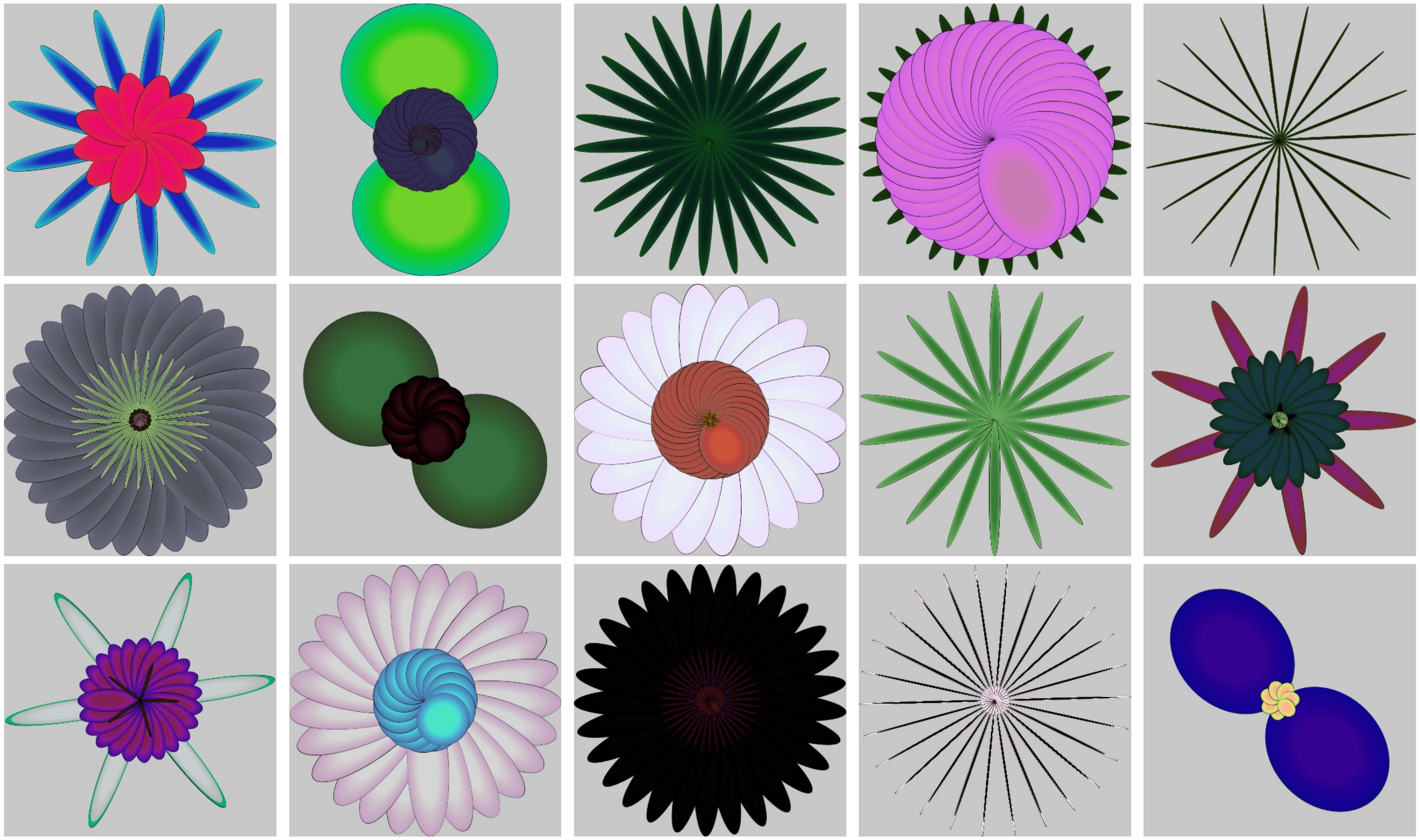


Table C.5: Some subjectively bad 2D flowers from the top 5% of solutions in the CVT-MAP-Elites DNN-phash composite score experiments (all five trials) - Top row: daisy, rose, dandelion, rose, dandelion - Middle Row - sunflower, rose, rose, dandelion, daisy - Bottom row: daisy, rose, sunflower, dandelion, rose

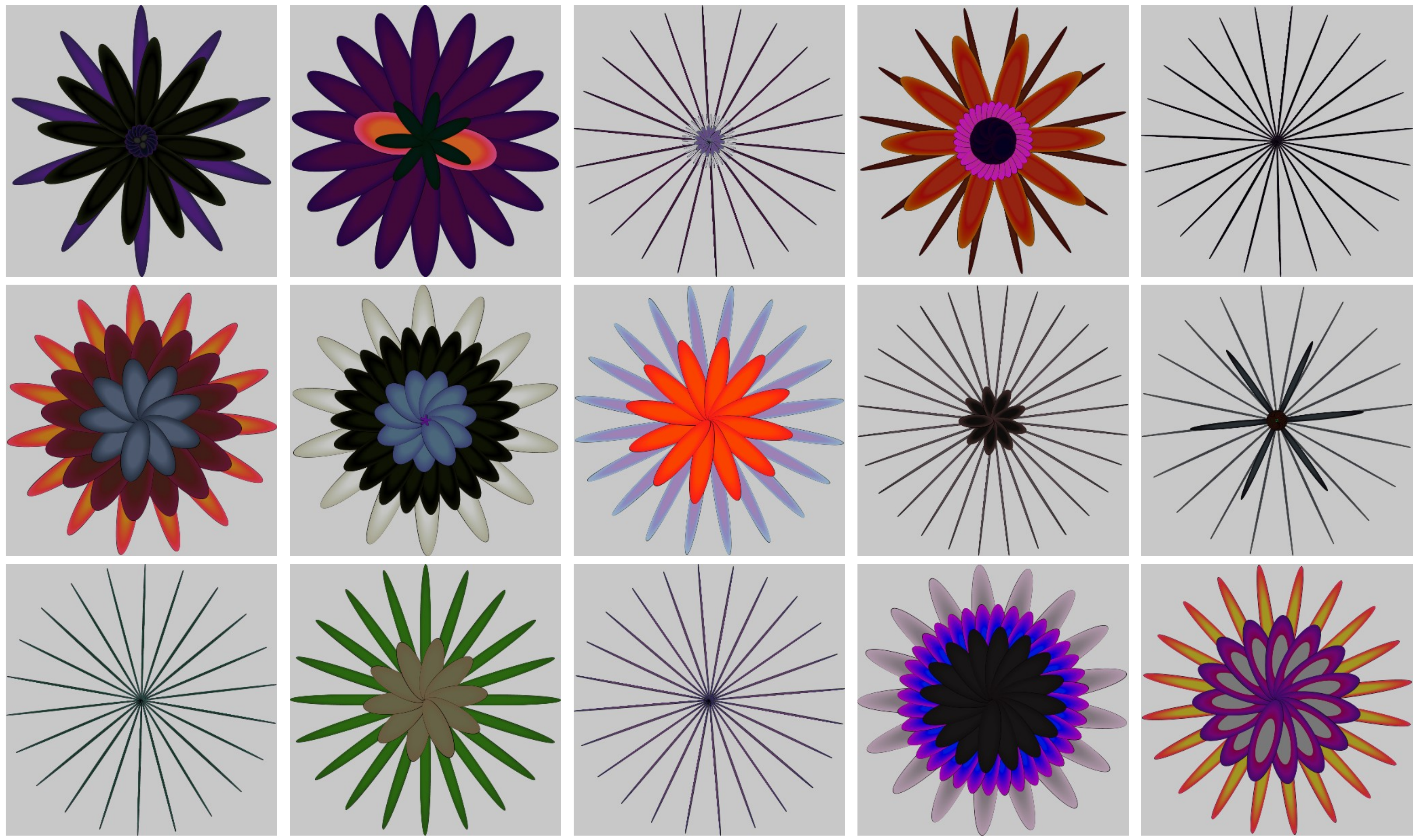


Table C.6: Some subjectively bad 2D flowers from the top 5% of solutions in the CVT-MAP-Elites DNN-whash-haar composite score experiments (all five trials) - Top row: daisy, daisy, dandelion, daisy, dandelion - Middle Row - daisy, daisy, daisy, dandelion, dandelion - Bottom row: dandelion, daisy, dandelion, daisy, daisy

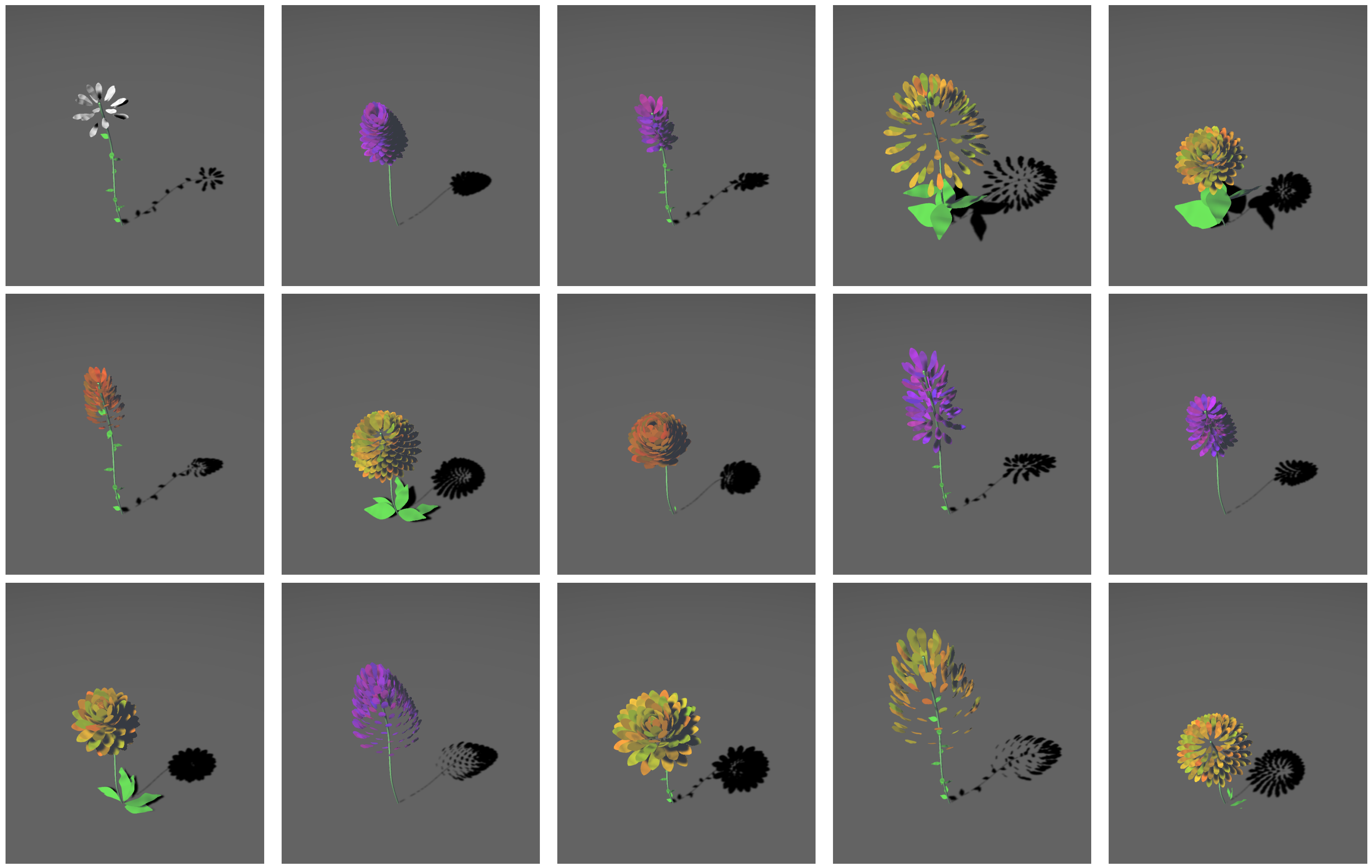


Table C.7: Some subjectively bad 3D flowers from the top 5% of solutions in the CVT-MAP-Elites DNN-ahash composite score experiments (all two trials) - Top row: dandelion, dandelion, dandelion, sunflower, sunflower - Middle Row - dandelion, sunflower, sunflower, dandelion, dandelion - Bottom row: sunflower, dandelion, sunflower, dandelion, sunflower

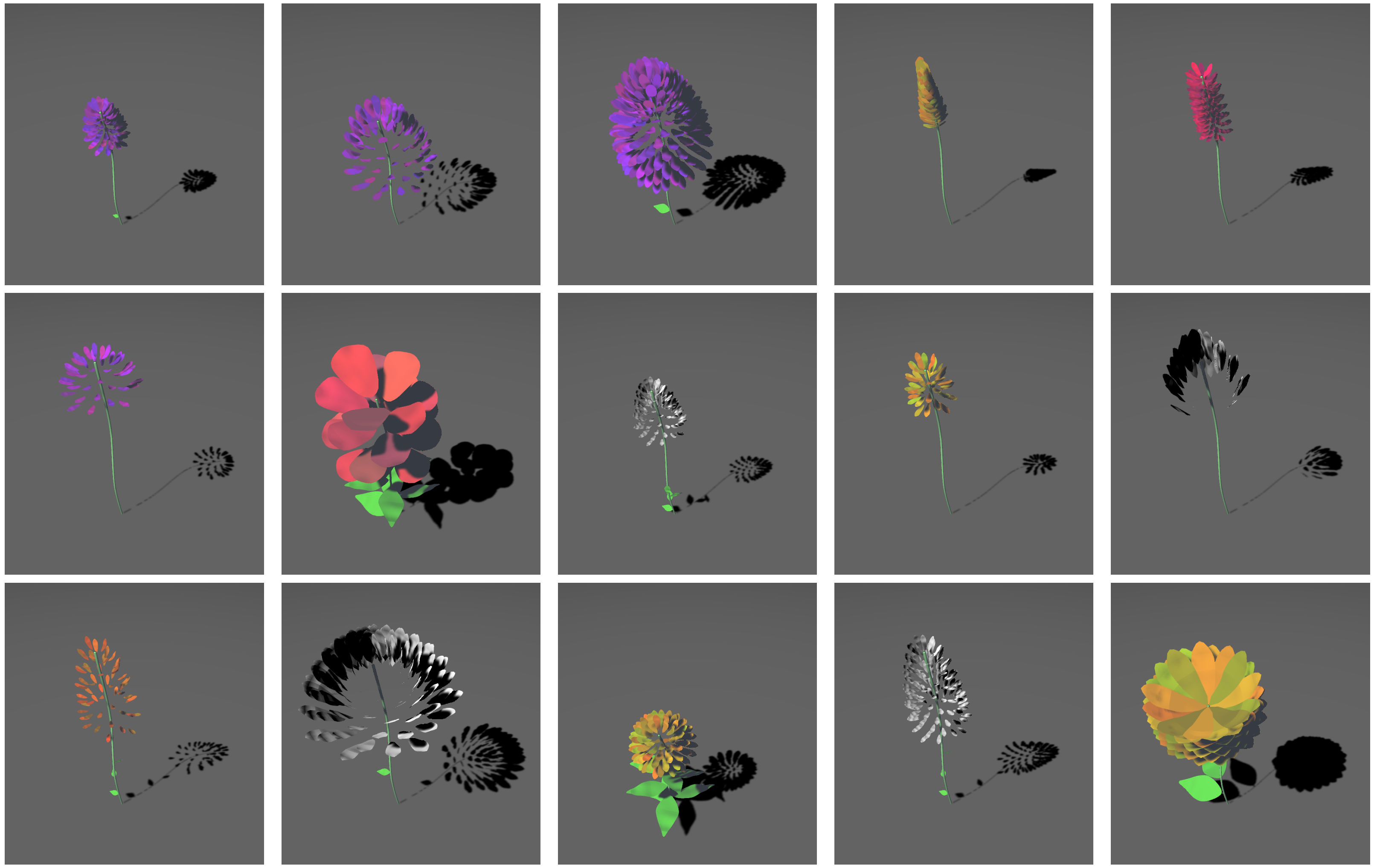


Table C.8: Some subjectively bad 3D flowers from the top 5% of solutions in the CVT-MAP-Elites DNN-dhash composite score experiments (all two trials) - Top row: dandelion, dandelion, dandelion, dandelion, dandelion - Middle Row - dandelion, rose, dandelion, dandelion, dandelion - Bottom row: dandelion, dandelion, sunflower, dandelion, rose

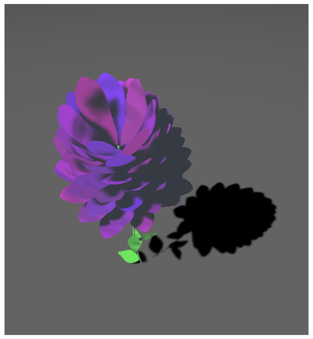
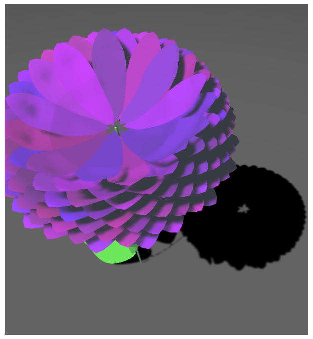
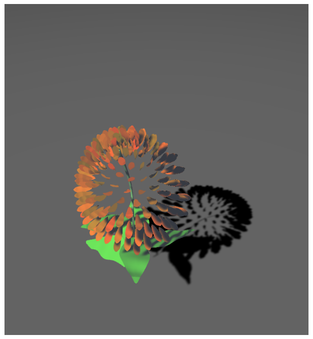
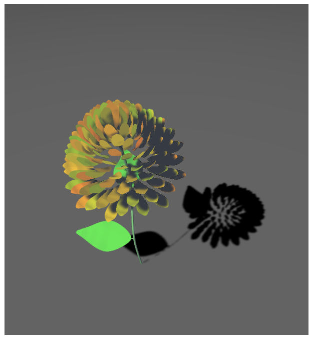
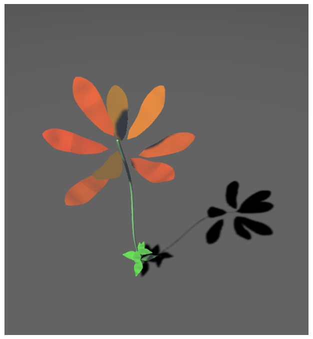
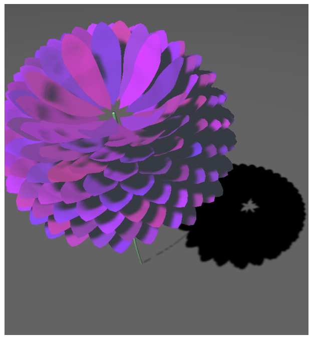
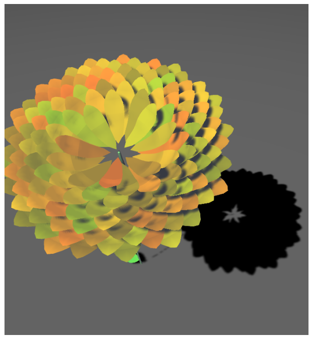
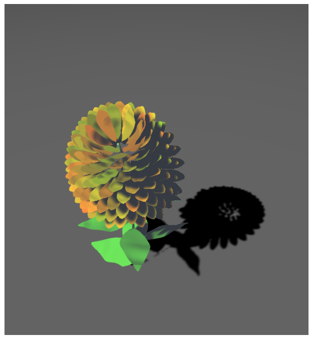
				
				
				

Table C.9: Some subjectively bad 3D flowers from the top 5% of solutions in the CVT-MAP-Elites DNN-phash composite score experiments (all two trials) - Top row: rose, rose, rose, rose, sunflower - Middle Row - rose, sunflower, rose, rose, rose - Bottom row: sunflower, rose, sunflower, rose, sunflower

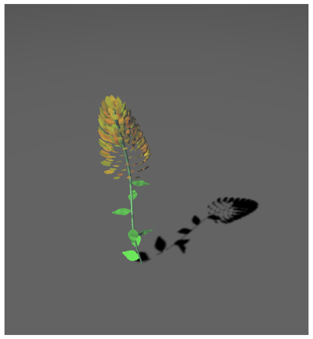
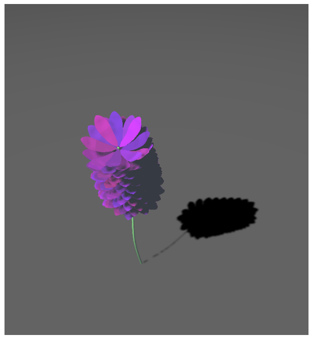
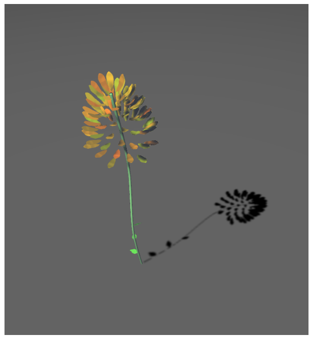
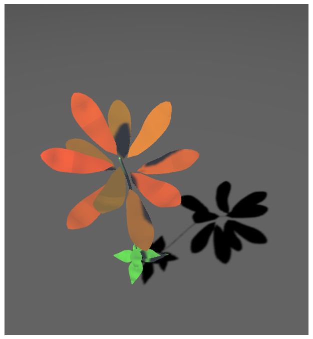
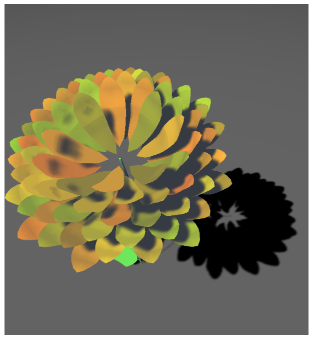
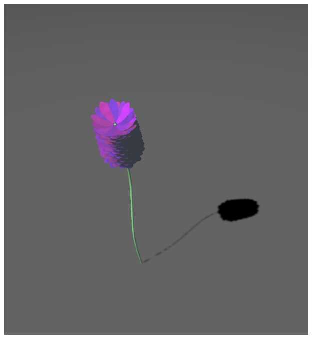
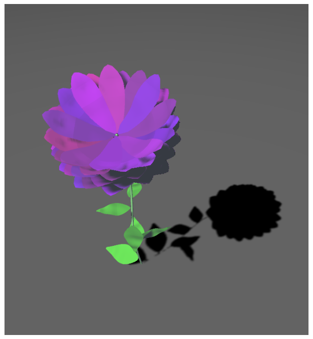
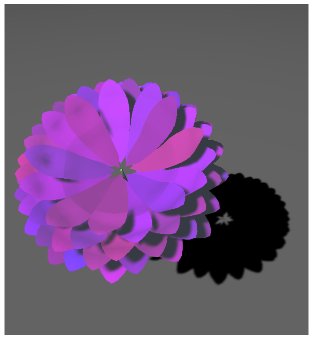
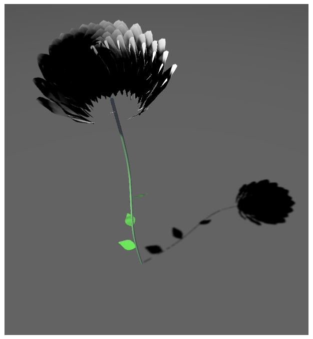
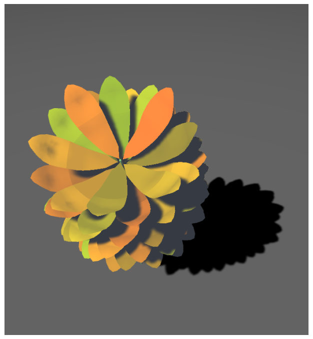
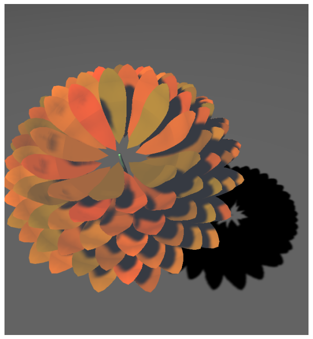
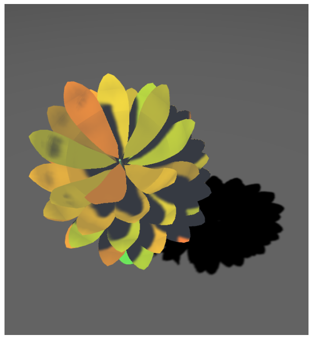
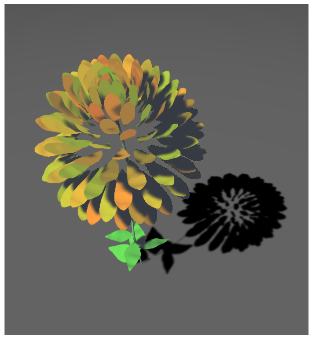
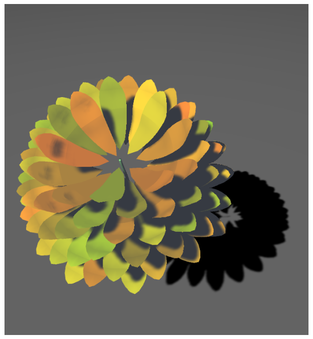
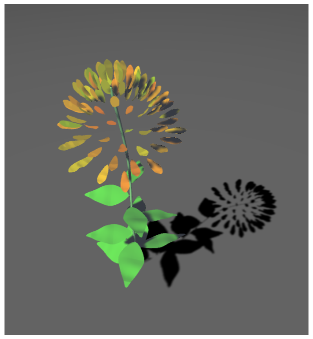
				
				
				

Table C.10: Some subjectively bad 3D flowers from the top 5% of solutions in the CVT-MAP-Elites DNN-whash-haar composite score experiments (all two trials) - Top row: dandelion, dandelion, dandelion, rose, rose - Middle Row - dandelion, rose, rose, dandelion, rose - Bottom row: rose, rose, sunflower, rose, sunflower