Karl Haubenwallner, BSc.

ShapeGenetics

Using Genetic Algorithms for Procedural Modeling

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science

eingereicht an der
Graz University of Technology

Betreuer
Univ.-Prof. Dipl-Ing. Dr.techn. Dieter Schmalstieg

Mitbetreuer
Dr. Markus Steinberger

Institute für Computer Graphik und Vision

Fakultät für Informatik und Biomedizinische Technik

Graz, Februar 2017

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstrakt

In dieser Arbeit zeigen wir, dass Genetische Algorithmen (GA) dazu verwendet werden können, um die Ergebnisse von Prozeduralen Modellierungsalgorithmen zu kontrollieren. Wir zeigen einen effiziente Art die Entscheidungen, welche während einer prozeduralen Erzeugung getroffen werden, in einem hierachischem Genom zu codieren. Durch Verwendung von speziellen Mutations- und Reproduktionsoperatoren, welche auf die kontrollierte prozedurale Erzeugung abgestimmt wurden, kann unser GA eine Population von individuellen Modellen hin zu einem beliebigen abstraktem Ziel zu evolvieren. Mögliche Anwendungen sind u.a. ein Volumen, dass von einem prozeduralem Baum ausgefüllt werden soll, oder eine gemalte Shilouette, welche von der Skyline einer prozeduralen Stadt dargestellt werden soll. Diese Ziele sind für einen Künstler einfach aufzustellen, verglichen mit den zehntausen Variablen, welche ein erzeugtes Modell beschreiben, und vom GA ausgewählt werden. Frühere Ansätze für kontrollierte prozedurale Modellierung verwenden entweder Reversible Jump Markov Chain Monte Carlo (RJMCMC) oder Stochastisch geordnet Sequentielle Monte Carlo (SOSMC) als Optimizierungsalgorithmus. Während RJMCMC langsam konviergiert, und mitunter mehrere Stunden für die optimierung von größeren Modellen benötigt, erzeugt es doch qualitiative Modelle. SOSMC zeigt ein schnelleres Konvergierungsverhalten under strengen Zeitvorgaben, kann aber aufgrund von Entscheidungen in den Frühstadien der Optimierung stecken bleiben. Unser GA zeigt schnelleres Konvergierungsverhalten als SOSMC und erzeugt bessere Modelle als RJMCMC mit langer Laufzeit.

Abstract

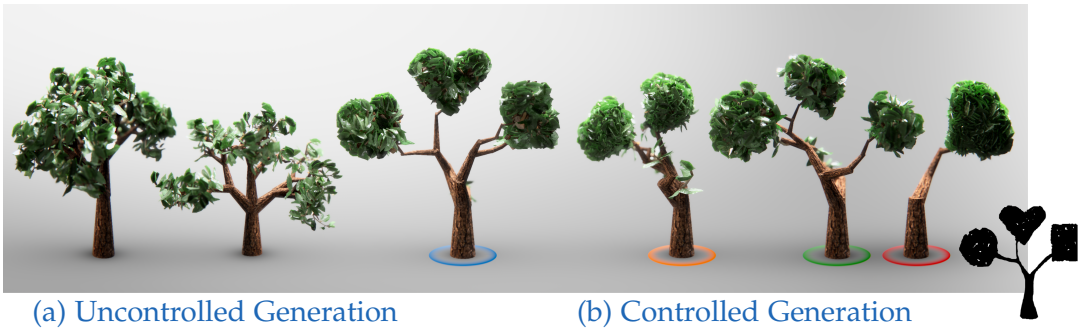
In this work, we show that genetic algorithms (GA) can be used to control the output of procedural modeling algorithms. We propose an efficient way to encode the choices that have to be made during a procedural generation as a hierarchical genome representation. In combination with mutation and reproduction operations specifically designed for controlled procedural modeling, our GA can evolve a population of individual models close to any high-level goal. Possible scenarios include a volume that should be filled by a procedurally grown tree or a painted silhouette that should be followed by the skyline of a procedurally generated city. These goals are easy to set up for an artist compared to the tens of thousands of variables that describe the generated model and are chosen by the GA. Previous approaches for controlled procedural modeling either use Reversible Jump Markov Chain Monte Carlo (RJMCMC) or Stochastically-Ordered Sequential Monte Carlo (SOSMC) as workhorse for the optimization. While RJMCMC converges slowly, requiring multiple hours for the optimization of larger models, it produces high quality models. SOSMC shows faster convergence under tight time constraints for many models, but can get stuck due to choices made in the early stages of optimization. Our GA shows faster convergence than SOSMC and generates better models than RJMCMC in the long run.

Contents

1 Introduction

Procedural modeling not only has a long history in computer graphics, but also sees increasing interest in recent years as the demand for detailed models and large virtual worlds is growing rapidly. Using a procedural approach, a wide variety of detailed variants of a model or family of models can be generated by altering the parameters that control the procedural generation. Examples include vegetation models, which can generate everything from small bushes to full leaf and needle trees (Pirk et al., 2012), and building grammars, which can generate everything from dog sheds to skyscrapers (Schwarz and Müller, 2015). While such generative models potentially offer technical artists the ability to create a complex object within seconds, achieving a desired result is often still a complicated and time-consuming process. Because the model's parameters can control recursive generation processes and small parameter adjustments can be amplified throughout the generation, the influence of a single parameter on the final model is often unpredictable and artists have to rely on excessive trial and error. Unsurprisingly, this cumbersome process turns many artists away from procedural modeling back to manual model generation.

As a remedy to this problem, the processes of procedural generation have been cast as a probabilistic inference problem (Talton et al., 2011). Given a procedural generator that has the expressive power to generate a suitable model and a scoring function that tells how close a model is to the desired goal, these approaches view the generation of a model as drawing a sample from a probability distribution. Naively speaking, the scoring function allows for an optimization over the entire space of possible models to find a model that fits the scoring function as closely as possible. The approach is complicated by the fact that the space of possible models is trans-dimensional, theoretically unbounded, and lies in a mixed domain space; the scoring function is in general non-linear and non-convex; and



(a) Uncontrolled Generation (b) Controlled Generation

Figure 1.1: (a) Procedural approaches can generate models with large variety. (b) Given a high level target as a sketch (drawing at the far right), the generation can be directed towards this goal. Previous approaches are either tuned for speed (SMC, red circle and SOSMC, orange circle) and thus might not reach the desired result, or require a long time to achieve a good result (RJMCMC, green circle); our proposed solution using a genetic algorithm converges fast and achieves high quality results (blue circle), clearly matching the target the best.

generating a model and evaluating the scoring function can take significant amounts of time. Clearly, the problem of finding a set of parameters that generates a model that matches a given target is a non-trivial problem, *e.g.*, consider the results shown in Figure 1.1.

Most previous approaches on probabilistic inference for procedural generation rely on Markov Chain Monte Carlo (MCMC) methods to solve the problem (Talton et al., 2011; Stava et al., 2014; Yeh et al., 2012). As alternative to the relatively slow converging MCMC, stochastically-ordered sequential Monte Carlo (SOSMC) has been proposed (Ritchie et al., 2015). While SOSMC, in general, improves on the convergence of MCMC, there is a possibility that it gets stuck in bad initial conditions, especially if the procedural models become more complex. Also, SOSMC requires the scoring function to yield reasonable results for unfinished models, which cannot be guaranteed in general. Given the mere existence of only two classes of approaches for the probabilistic inference problem in targeted procedural modeling and the increasing importance of (semi)-automated content creation, we see high potential benefits in exploring further alternatives.

In this work, we show that genetic algorithms (GA) can be used for controlled procedural modeling. While the principles of GAs are not new,

applying GA to a new domain is never without new challenges. We tackle these challenges and make the following contributions:

- We present a simple and yet efficient way to encode a variety of procedural generation approaches as *genomes* for the use in a GA for controlled procedural modeling.
- We show how the core operations of a GA—reproduction and mutation—can be implemented for these genomes such that only valid modeling operations are created.
- We show that our genome representation can not only be used to implement GA, but also MCMC variants and SOSMC using alternative mutation operations.
- We show that our GA yields significantly better convergence rates than the previous state-of-the-art implementations, especially when more complex models are generated.

In the following, we give an overview of related work (Section 2); introduce GA for controlled procedural modeling (Section 3); present our genome representation for procedural modeling (Section 3.2); explain the mutation and reproduction operations (Section 3.3); provide details on the GA setup (Section 3.4); show how MCMC and SOSMC can be modeled within the same genetic algorithm framework (Section 3.5); evaluate the influence of the parameters of the GA and compare our GA implementation against previous state-of-the-art for different kinds of scoring functions and generation methods (Section 4); and summarize the findings (Section 5).

2 Related Work

2.1 Procedural Modeling

Procedural modeling has been a part of computer graphics for decades. Among the first procedural modeling approaches were Stiny's original shape grammars (Stiny, 1975) and Lindenmayer's L-systems (Lindenmayer, 1968; Prusinkiewicz and Lindenmayer, 1991). Starting with these early works, a variety of approaches have been proposed, including realistic trees (Weber and Penn, 1995), plants (Lintermann and Deussen, 1998), trees interacting with their environment (Měch and Prusinkiewicz, 1996), split grammars for facades (Wonka et al., 2003), botanic trees (Okabe, Owada, and Igarash, 2005), a language for complex objects (Havemann, 2005), a grammar for buildings (Müller et al., 2006), self-organizing tree models (Palubicki et al., 2009), interconnected structures (Krecklau and Kobbelt, 2011), plastic trees (Pirk et al., 2012), and complex buildings (Schwarz and Müller, 2015). At the same time, speeding up procedural generation on modern hardware has received increasing interest (Magdics, 2009; Marvie et al., 2012; Steinberger et al., 2014). Our approach can be used for any of the previously mentioned procedural approaches and will benefit directly from faster generations.

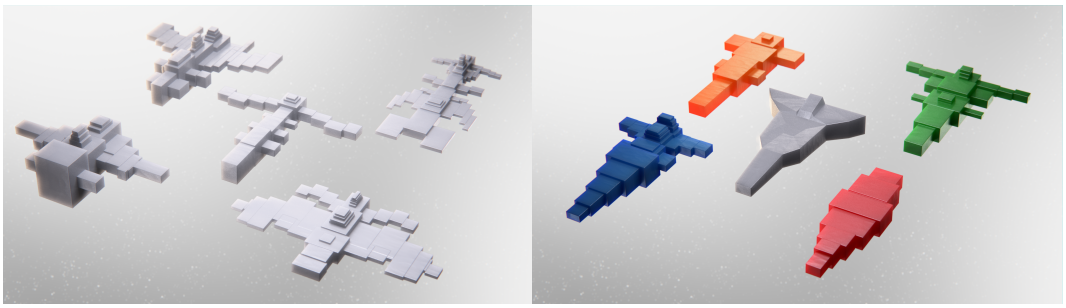
2.2 Controlled Procedural Generation

Controlled procedural generation has received increasing interest in recent years. Viewing procedural generation as an inference problem, high-level goals can be considered during generation. Talton et al. used the reversible jump MCMC (RJMCMC) algorithm, a variant of the Metropolis Hastings (MH) algorithm for controlled grammar-based procedural modeling (Talton

et al., 2011). While their implementation achieved impressive results choosing thousands of parameters for tree and building models to match target volumes and silhouettes, their MH algorithm also requires tens of thousands of iterations to converge, leading to running times of up to multiple hours. When using MCMC to only choose a few parameters, *e.g.*, to position a small number of pieces of furniture, good results can be obtained within a second (Merrell et al., 2011). However, more complex furniture layout generation, which also require RJMCMC methods, increases the running time again to minutes or hours (Yeh et al., 2012). In case one is not interested in a specific model, but rather a sub-family of models that are similar to a target, finding a suitable similarity measure is a non-trivial task (Stava et al., 2014). With such a similarity measure in place, MH can be used to choose a small number of meta-parameters for the sub-family. However, running times might still be long due to costly similarity evaluations. Tackling the rather long convergence time of the MH algorithm and its variants for controlling procedural modeling, Ritchie et al. proposed to use SOSMC (Ritchie et al., 2015). Their approach can be seen as a combination of guided procedural modeling (Beneš et al., 2011) with probabilistic inference. Models are not only scored when they are fully generated, but the score of early stages of the generation determines how likely it is that the generation of a model continues. For simple models, good results can be obtained within seconds or minutes, outperforming MH approaches when only little time is available. Our approach can be seen as an alternative to MH and SOSMC, showing even better convergence rates than SOSMC under tight time constraints and outperforming MH in the long run.

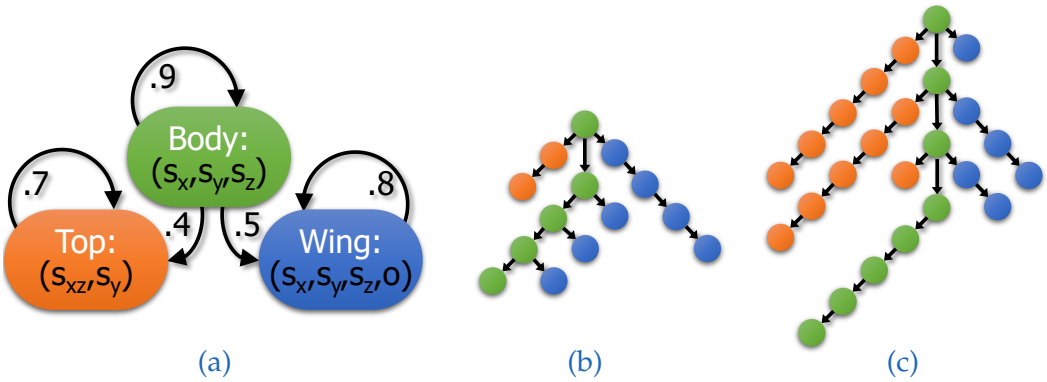
2.3 Evolutionary Approach

Evolutionary approaches have been used in computer graphics before. Most noteworthy is the work by Sims. He showed that evolutionary algorithms with simple mutation and random crossover operations can be used to generate 3D plant structures, images, textures, and animations (Sims, 1991). In his follow-up work, he described the genotypes and phenotypes of virtual creatures including their animation using evolutionary algorithms, allowing them to walk, jump, and swim (Sims, 1994). Sims' approach sparked a



(a) Uncontrolled Generation (b) Controlled Generation

Figure 2.1: (a) Although quite simple, the spaceship generator can produce a variety of ships. (b) Given a volumetric target (dark shape), the generation can be optimized to fill the target. Note that the best solutions of different algorithms (GA: blue, SMC:red, SOSMC: orange, MCMC:green) differs not only in the size of the parts but the structure of the entire ship.



(a) (b) (c)

Figure 2.2: (a) The generator graph for the simple spaceship models (see Figure 2.1) consists of three nodes: Each part can generate an instance of itself (probabilities next to edges). The body part may generate a top and a wing part. All parts are parameterized by scaling factors; the wing has an additional offset. (b,c) The derivation tree fully describes a generated model. (b) corresponds to the center model of Figure 2.1(a), (c) to the blue ship in Figure 2.1(b). Note that parameter choices are not shown here due to the size of the graphs.

variety of approaches following the same direction, like, *e.g.*, Creature Academy (Pilat and Jacob, 2008), which uses Sims' encoding schemes to evolve creatures that are capable of a variety of motions. Evolutionary algorithms have also been used to control L-systems to optimize parameters of 2D-plants Ochoa, 1998, or fit basic 2D shape grammars to simple targets O'Neill et al., 2009. They have also been used for real world objects (Funes and Pollack, 1998), enabling the design of robots that can be assembled using Lego bricks. Genetic algorithms can also be applied for shapes within shape collections, extending the variety of models in design galleries (Xu et al., 2012). In this "fit and diverse" gallery, evolution is not applied on an abstract genome representation, but directly on the models by exchanging their parts using fuzzy crossover operations. While the aforementioned evolutionary approaches have been used to represent shapes and models, they either work directly on model parts or a high-level, meta-graph which allows edges between arbitrary nodes. Applying GA to parameter selection for procedural generation has different requirements and a direct adoption of the previous representations is not possible. With our work we provide such a representation and underline the special requirements of controlled procedural generation with GAs.

3 Genetic Algorithms and Controlled Procedural Modeling

Virtually all approaches that are classified as procedural modeling, can be described in a unified way: they apply a sequence of modeling operations on objects. Each operation either alters an object or generates new objects, on which modeling operations can again be applied. These objects are in turn translated into geometry that constitutes the generated model. The operations are usually parameterized, *e.g.*, translate by vector $\mathbf{t}$, rotate by angles (ϕ_x, ϕ_y, ϕ_z), and so on. These parameters are generally chosen at random to add variations to the generated models. Also, the number and type of generated objects may be chosen randomly. Based on these observations it seems natural to capture procedural generation as directed graphs (with cycles). This fact has been explicitly noted for L-systems (Boers, 1995), shape grammars (Patow, 2012), and stack-based modeling languages (Havemann, 2005) before. Also languages like the one used by Ritchie et al. (Ritchie et al., 2015) can be translated into a directed graph by applying the principles of data flow programming (Wadge and Ashcroft, 1985). Thus, we do not limit our approach to any specific procedural approach, but only work with a directed graph representation, where nodes correspond to parameterized operations and the edges correspond to objects. We simply call the description of the procedural generation in their respective language a *procedural generator* and the associated graph the *generator graph*, see Figure 2.2(a) for a simple example.

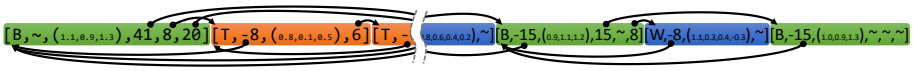
The generator graph describes the procedural generation itself and thus all models that can be generated by that generator. The generation of a specific model can also be described by a graph or, more specifically, a tree. This tree captures all intermediate objects as they move through the generator graph, with the operations applied to them including the

randomly chosen parameters. In grammar-based modeling this tree is called *derivation tree* (Sipser, 2006). Although we do not limit our approach to grammar-based modeling, we do adopt this name. As the operations can generate a different number of objects at random, the derivation trees for different models can vary in structure and number of parameters, as shown in Figure 2.2(b,c). In any case, a model is fully described by its derivation tree.

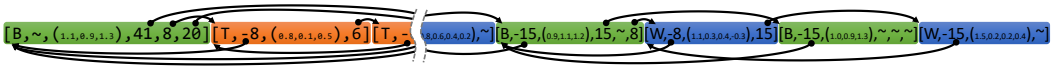
For complex models, derivation trees can become very large, containing tens of thousands of nodes with as many or more parameters. This underlines that expecting a technical artist to choose these parameters by hand is infeasible. As the derivation tree describes the generation, we can view controlled procedural modeling as an optimization problem or Bayesian inference problem of derivation trees. Each possible derivation tree corresponds to a sample from the procedural generator, leading to Bayesian inference. At the same time, finding those derivation trees that maximize the scoring function creates a classical optimization problem. The problem is obviously in a mixed domain: deciding whether a node should be added to the tree is a boolean decision; the parameters describing operations in Euclidean space are in the continuous domain. Also, derivation trees of different models have different structures, which emphasizes the trans-dimensional nature of the problem.

3.1 Genetic Algorithms

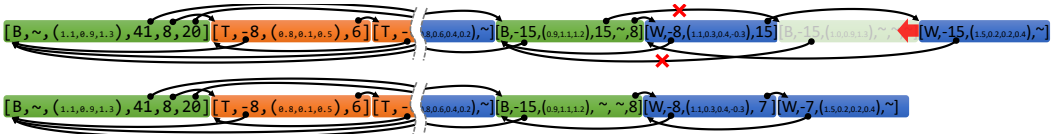
Genetic algorithms are well suited for this kind of problem. GA is inspired by evolution and natural selection, where traits and characteristics of individuals of a species are encoded as genes. Due to the selection process individuals with successful traits get a higher chance to pass on their genes to future generations, while less successful traits tend to disappear, overall leading to a better adapted population. The basic steps of GA are outlined in Algorithm 1. For controlled procedural modeling we want each individual to represent a derivation tree. During evolution individuals can be altered (mutated) and information from different individuals can be combined (reproduction). To use GA for controlled procedural modeling



(a) Our packed genome representation encodes a derivation tree with parameters flattened out in memory. Every gene (separate bar) has a symbolic link to its generation tree node (first entry), identifying the meaning of the stored parameters and making sure a valid generation is represented. Each gene stores the relative offset to its parent (second entry), as well as the relative offset to each child; if a child is not present the entry is empty.



(b) The grow mutation adds another gene to the chromosome. It can simply be added to the back of the chromosome and only the offsets of the parent need to be adjusted. Thus the operation is very efficient.



(c) The cut mutation (top to bottom) not only removes the cut gene, but the entire sub-tree of dependent genes (grayed out). Thus, arbitrary amounts of memory can be removed. However, with a single sweep starting at the gene that is removed, we can remove all dependent genes and at the same time compact the representation by copying the remaining genes to the front and adjusting the offsets.

Figure 3.1: Example of our genome representation and mutations applied to the example spaceship from Figure 2.1(a) and 2.2(b)

we require mutation and reproduction operations that work on derivation trees while making sure the resulting derivation trees conform with the generator graph. The complexity of the optimization—working in a mixed domain and the trans-dimensionality—is hidden in these two operations. To determine the fitness of individuals, as required by a GA, we directly use the scoring function, which can also include additional constraints like the size of the derivation tree or how balanced the tree should be. However, two challenges need to be solved: (1) an encoding of the derivation tree as genome representation is required, and (2) the genetic operations for mutation and reproduction need to be described.

```
Function GeneticAlgorithm
    population ← { }
    for i ∈ {1 ... PopulationSize } do
        append(population, newIndividual())
    end
    for j ∈ {1 ... MaxGenerations } do
        fitness ← evaluateFitness(population)
        sort(population, fitness)
        newPopulation ← { }
        for k ∈ {1 ... Elitism } do
            append(newPopulation, population [k])
        end
        for k ∈ {1 ... (PopulationSize- Elitism )/2} do
            par1 ← select(population,fitness)
            par2 ← select(population,fitness)
            if rand() ≤ MutationProbability then
                chld1 ← mutate(par1)
                chld2 ← mutate(par2)
            else
                chld1,chld2 ← reproduce(par1, par2)
            append(newPopulation, chld1, chld2)
        end
        population ← newPopulation
    end
```

Algorithm 1: The basic structure of genetic algorithms is straightforward. A population of individuals are managed for a given number of generations. The best individuals are allowed to influence the next generation, by either copying them directly, mutating them, or allowing them to reproduce.

3.2 Genome Representation for Procedural Models

The genome representation must be able to encode all possible solutions to the problem, *i.e.*, all possible derivation trees. To describe the genome representation we use the following terms: a *gene* is the smallest element of the genome; a *chromosome* is the set of genes that make up an individual. Ideally, the representation of a chromosome and hence a gene should be simple and allow for efficient genetic operations. Furthermore, the fitness evaluation requires a translation of the chromosomes into their expression, *i.e.*, into a geometric model. Thus, given the chromosome, we want to run the procedural generator replacing the random decisions with the parameters decoded from the chromosome in an efficient manner.

Because each chromosome needs to encode the derivation tree, the traditional GA approach of using bit-strings of fixed size as genome representation is not feasible. We propose to store the derivation tree directly as nodes and edges, similar to Sims' graph representation (Sims, 1994). Every node is described by one gene. Each gene keeps a reference to its corresponding node in the generator graph, the parameters chosen for its associated operation, and a pointer for each possible output object, *i.e.*, child node, which we set to empty if no output is generated. If an output is present, it points to the gene within the same chromosome that encodes the generated object. Furthermore, every gene stores a pointer to the gene that describes its parent node. We call the pair of pointers between a parent and child gene a *connection*. Each connection is associated with an edge in the generator graph. Note that genes are of different size, depending on which node and operation they describe. We pack all genes of one chromosome compactly in memory. The pointers are stored as local relative memory offsets, which allows for efficient copying of subtrees; see Figure 3.1(a) for an example.

Our genome representation allows for variable length chromosomes and easy insertion of new genes into the chromosome without changing the already existing entries. Furthermore, copying genes from one individual to another can be carried out efficiently due to the use of local offsets. The reference to the generator graph is required to ensure that operations on the

chromosomes do not yield individuals that cannot be created by the procedural generator, *i.e.*, lie outside of the sampling space. Within the generator graph we additionally store the distribution of each random variable and probability for each output object. Thus, the GA has all required information available in the generator graph and can construct any chromosome using the genome representation.

3.3 Genetic Operations

At the core of each GA there are two genetic operations that evolve the population: mutation and reproduction. Mutation allows the GA to explore the problem space outside the already existing population by introducing random modifications to the genes of an individual, *i.e.*, generate a sample that is outside the space spanned by the individuals of the current population. Reproduction combines features from two individuals, creating individuals that contain parts of both parents and possibly yield better results than either of them. Combining traits from different high-scoring derivation trees, the GA may be able to converge faster.

3.3.1 Mutation.

As the chromosomes of an individual not only encode parameters, but also the structure of the generation, we propose to use three different mutation operations: *grow*, *cut*, and *alter*.

Grow, outlined in Figure 3.1(b), appends a gene to a gene that does not generate the maximum number of children in its current form. We choose a gene i and its non-expressed child j with the probability

$$p_{grow,i,j} = \frac{p_{i,j}}{\sum_k \sum_l p_{k,l}},$$

with $p_{i,j}$ being the probability of object j being generated by a node of type i; k runs over all genes that do not express their maximum number of children; l runs over those non-expressed children. Thus, the probability of adding

a gene is proportional to the likelihood of the object represented by the gene being generated in an uncontrolled generation. After a new gene has been added we choose its parameters randomly based on the probability distributions stored in the generator graph. The gene can simply be added to the back of the chromosome and grow is thus very efficient.

Cut, outlined in Figure 3.1(c), removes a random gene and the sub-tree that follows it. The probability of choosing a gene i is

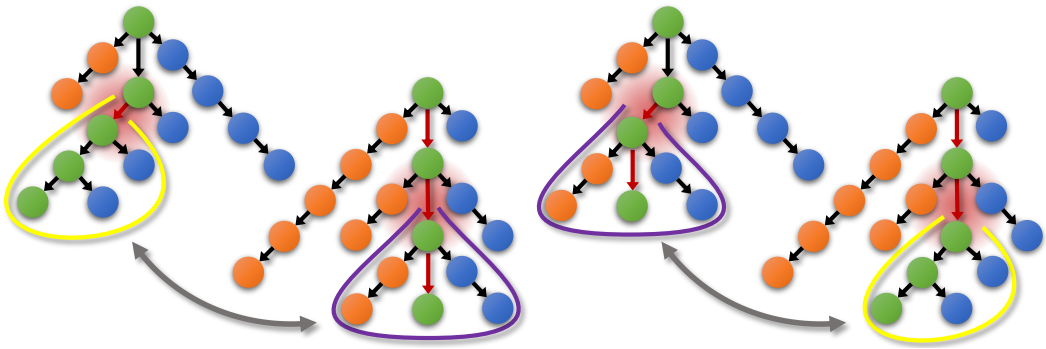
$$p_{cut,i} = \frac{1 - p_{j,i}}{\sum_k \sum_l (1 - p_{k,l})},$$

where k runs over all genes that have children and l over those expressed children. Thus, genes are removed with a probability that is inverse proportional to the likelihood that the object represented by the gene is generated in the uncontrolled production. Note that objects which are generated with a probability of one are never cut and thus no invalid derivation trees are generated. If there is no gene with a probability $p_{cut,i} \neq 0$, no cut is applied. Although cut may involve the removal of multiple genes which are spread across the chromosome, the operation can be completed efficiently, sweeping over the chromosome, moving non-removed genes further to the front essentially replacing removed ones and adjusting the local offsets. The operation can therefore be completed in $\mathcal{O}(G)$, where G is the chromosome length. This guarantees that the operation remains efficient even for more complex derivation trees.

Alter changes the parameter values of a random gene. We choose a gene with parameters from the chromosome with equal probability and replace all its parameters by drawing a new random sample from the parameter's distribution as described in the generator graph. Alter is obviously the simplest and most efficient mutation.

3.3.2 Reproduction.

Our reproduction operation is an adapted version of the single-point-crossover operation, which creates a valid pair of children given a pair



(a) (b)

Figure 3.2: (a) Our crossover operation chooses one random gene connection in the left parent (red highlight), identifies all matching connections from the right parent by consulting the generator graph (red edges) and chooses one at random (red highlight). (b) Exchanging the subtrees (yellow and purple) yields two offspring.

of parents. In its basic form the single-point-crossover operator selects a random crossover point in both parent chromosomes and creates the offspring by swapping the genes at the crossover points. The number of possible different children given the same set of parents is limited by the number of crossover points. This strategy cannot be directly applied to chromosomes describing a derivation tree since the results may be incompatible with the generator graph.

Our proposed crossover operation selects a random gene connection from the first parent, chooses another random connection from all *compatible* connections in the second parent, and exchanges the genes including all successors at these connections. The set of compatible connections contains those that describe the same edge in the generator graph. In this way, we can guarantee that the offspring are compatible with the generator. The crossover operation is outlined in Algorithm 2 and Figure 3.2. With our genome representation the crossover operation is efficient, since compatible connections can easily be selected based on the reference to the generator graph and copying genes from a parent can be viewed as combination of cut and grow operations, which are both efficient as well.

```
Function reproduce(parent1, parent2)
    for i ∈ {1 ... MaxRetries } do
        C1 ← connections(parent1)
        e1 ← selectUniformly(C1)
        type ← generatorGraphEdge(C1)
        C2 ← findMatching(parent2, type)
        if C2 is empty then
            continue
        end
        e2 ← selectUniformly(C2)
        child1 ← cloneUntil(parent_1, e1)
        child1 ← cloneFrom(parent_2, e2)
        child2 ← cloneUntil(parent_2, e2)
        child2 ← cloneFrom(parent_1, e1)
        return child1, child2
    end
    return parent1, parent2
```

Algorithm 2: The proposed crossover operation is computationally very efficient, as the explicit links to the generator graph make sure every offspring describes a valid chromosome. Thus, crossover usually only requires drawing two random numbers and copying genes from the parents to the offspring.

3.4 Genetic Algorithm Setup

3.4.1 Selection Method

The selection method is one of the central parts of a GA. It determines which individuals are allowed to reproduce and influence the next generation. The selection is usually done by selecting individuals according to their fitness, with some margin for stochasticity. A purely deterministic selection method would select the best individuals only, limiting the exploration of the problem space to the proximity of the fittest individual, while a purely random method would lead to an entirely undirected exploration decreasing the probability of finding a good solution considerably. There exist several selection methods that take the fitness into account while allowing for random deviation. The most widely used ones are *roulette wheel* and *k-tournament* selection (Blickle and Thiele, 1996).

Roulette wheel selection chooses an individual I with a probability p_I proportional to the individual's fitness f_I:

$$p_I = \frac{f_I}{\sum_j f_j},$$

where j runs over all individuals of the current generation. The name stems from the informal description of the method as a roulette wheel, where the size of each possible spot relates to the fitness of the individual occupying it.

k-tournament selection starts by selecting k individuals at random with equal probability. From that group, the individual with the highest fitness is selected, *i.e.*, letting the individuals fight in a tournament. k-tournament selection can be implemented efficiently, while the selection pressure can be controlled by adjusting k. In our approach we use roulette wheel selection to choose one parent and k-tournament to choose the other.

3.4.2 Initial Population

A good initial population is important to introduce enough variety into the evolution. We create the initial population starting with individuals that contain an empty chromosome. For each individual we repeatedly apply the grow mutation until their chromosomes reach a randomly chosen size. The GA converges fastest if the average chromosome length of the initial population is comparable to the final solution. However, as crossover and mutation change the chromosome length, even far off initial populations converge. As the grow mutation is very efficient, generating the chromosomes of the initial population has a low computational cost.

3.4.3 Elitism

Elitism is a technique that ensures that future generations also contain the best individuals of previous generations. Using elitism, the individuals with the highest fitness are copied unchanged to the new generation. However, these individuals can still also be selected as parents for reproduction. This ensures that the quality of the solution never decreases and increases convergence rates as good solutions are available for reproduction more often. However, if the population size is too small, elitism can lead to stagnation as the GA cannot explore different solutions. We use elitism in all our experiments.

3.5 MCMC and SOSMC as GA

As SMC, SOSMC (Ritchie et al., 2015), and reversible jump MCMC (Talton et al., 2011) for controlled procedural modeling work on individual particles or chains that can be viewed as individual (partially finished) procedural generations, we argue that they can be implemented in a GA framework. The following description should be considered an outline of how to implement these approaches, details can be found in the original papers.

3.5.1 SMC and SOCMC

*SMC*is arguably the simplest of the three approaches. The particles used in SMC can be seen as individuals, each starting with an empty chromosome. SMC does not allow for interaction between particles and thus can be described with mutations only. Ritchie et al. sample particles based on their fitness, which we can provide in terms of a selection method. As mutation operations, we only require an adjusted grow mutation: It is not allowed to choose any non-expressed child, but has to follow the order they are found in the chromosome. Additionally, the next child is only expressed with a probability equal to the original generation (as found in the generator graph). If it is not expressed, it is marked as such in the chromosome and will never be expressed. The mutation continues until a new geometric output object is generated, showing the same behavior as the original SMC (Ritchie et al., 2015).

SOSMC can be implemented the same way as SMC with the difference that the mutation choses a random non-expressed child and does not have to follow the order in the chromosome. This step again continues until a new geometric output object is generated. Note that the state of the chromosome can be seen as the structure storing the stochastic futures (Ritchie et al., 2015).

3.5.2 Reversible jump MCMC

Reversible jump MCMC is arguably more complicated to implement. Each production chain corresponds to an individual. In the default setting, no reproduction is supported either. The selection method picks each individual once, moving mutated versions of each individual to the next generation. The diffusion operations correspond to a sequence of alter mutations followed by a scoring function evaluation, which determines if the mutation is accepted or rejected. The jump moves can be viewed as mutations that combine a cut move with grow mutations on top of the cut gene. Furthermore, the initialization of the newly found subtree does not draw new random variables, but copies the ones from the cut subtree (new values are drawn if there are not enough for copying). The mutation is accepted based

on the fitness ratio between the mutated and original individual, and the probability of the jump move.

While the described setup models the basic reversible jump MCMC algorithm, further optimizations are required to achieve good results (Talton et al., 2011). These optimizations can also be modeled in a GA. *Non-terminal selection* is added by altering the probability with which the mutations choose genes. *Parallel tempering* adds a temperature to each individual. The temperature changes the acceptance probability and—implementing a reproduction operation—exchanges the temperature of two individuals depending on their fitness. *Delayed rejection* extends the jump mutation with additional alter mutations, which allows a mutated individual to adjust before being rejected. Finally, *annealing* can be added on top of all rejection tests, slowly reducing the chance of accepting a worse individual.

4 Evaluation

To evaluate the performance of GA for controlled procedural modeling, we compare it to the previous state-of-the-art for a variety of scenes, complexity levels, and types of scoring functions. As evaluation platform we used an Intel Core i5-4570 @ 3.2GHz with 8GB of memory and an NVIDIA Geforce GTX 970. For running the procedural generation, we used a custom, multi-threaded, template-meta programming C++ generator that works on abstract shapes. This generator can be used for L-systems, shape grammars, and custom modeling languages. To obtain fast generation speeds, the generator graph descriptions is input to the C++ compile step, specializing the generator for the approach at hand.

4.1 Volumetric targets

Volumes as targets are a common way to specify high-level goals for a procedural generator. To this aim, we take any number of volume-describing target models $\mathbf{T}_i$ and weights w_i and $\widetilde{w}_i$ as input, voxelize them and compute the scoring function s_{vol} for a generated model $\mathbf{M}$:

$$s_{vol}(\mathbf{V}_M) = \sum_i \left(\left(\sum_{v \in \mathbf{V}_i} w_i \cdot \mathbf{V}_M(v) \right) + \left(\sum_{v \in \mathbf{V}_M} \widetilde{w}_i \cdot (1 - \mathbf{V}_i(v)) \right) \right),$$

where $\mathbf{V}_M$ is the voxelized representation of $\mathbf{M}$ and $\mathbf{V}_i$ is the voxelized representation of $\mathbf{T}_i$. In this way, volumes that should be filled or avoided can be specified setting positive or negative w_i and $\widetilde{w}_i$.

For efficient implementation, we compute the maximum bounds among all $\mathbf{T}_i$ and combine the voxels of all targets into a regular voxel grid $\mathbf{V}_{comb}$,

where each voxel corresponds to the sum over the weights of all targets. During evaluation, we voxelize the generated model, compute the overlap with $\mathbf{V}_{comb}$ and sum up all weights. For all voxels that fall outside the bounds of the voxel grid, we add $\sum \widetilde{w}_i$ to the score. To perform the voxelization of each $\mathbf{M}$ efficiently, we use a GPU voxelizer implemented in CUDA.

During voxelization it is easy to detect self intersections of the model components. Thus, we take another optional weight w_{self} which is subtracted from the overall model score every time a voxel is hit multiple times. In this way, self intersections of models can be avoided completely by setting w_{self} to a high weight, or punished slightly in case a few self intersections are acceptable.

4.2 Image targets

Images as targets are also a common way to specify high-level goals. We take any number of floating point images I_i and camera parameters C_i as input and compute the scoring function s_{img} for a generated model $\mathbf{M}$:

$$s_{img}(\mathbf{M}) = \sum_i \left(\sum_{p \in I_i} project_{C_i}(\mathbf{M})(p) \cdot I_i(p) \right),$$

where $project_{C_i}(\mathbf{M})(p)$ corresponds to projecting $\mathbf{M}$ using the camera parameters C_i and sampling the image at position p. Regions that are very important to be hit by the generated model should have high positive pixel values, areas that should be avoided high negative pixel values, and areas that do not matter zero values.

For efficient evaluation of s_{img}, we render the generated model into a black and white texture using OpenGL. Then, we run a compute shader to multiply the rendering result with I_i and perform a parallel reduction of the obtained values on the GPU.

4.3 Parameter Selection

The selection of parameters for a GA is a complicated task, since the parameters are interdependent, e.g. a high mutation rate can produce good results, but only if the population size is large enough. There have been various attempts to optimize this selection, such as using statistical models (François and Lavergne, 2001), or even using other optimization algorithms to find the best set of parameters, which introduces the problem of finding parameters for that algorithm. Since the execution time of our implementation is manageable, we were able to find a good set of parameters by changing one parameter at a time and comparing the results. The basic parameters during our tests are fixed to the following baseline, unless stated otherwise:

Population size:	50 individuals
Initial population:	random with 10 symbols
Max. generations:	50
Elitism:	1 individual
First selection:	roulette wheel
Second selection:	k-tournament, size 10
Mutation prob:	30%
Mutation operator:	cut/grow/perm. uniform distr.

All values were averaged over three discrete runs using the spaceship generator defined in Figure 2.2 and the volumetric target seen in the center of Figure 2.1.

4.3.1 Selection method

The available selection methods are a roulette wheel selection, a k-tournament selection of size 10, and random selection. As shown in Figure 4.1, using a semi-stochastic method for at least one parent produces better results than purely random selection, with tournament selection performing better. The best results were produces by a combination of tournament and roulette wheel selection, although tournament selection for both parents increases the fitness values initially slightly faster.

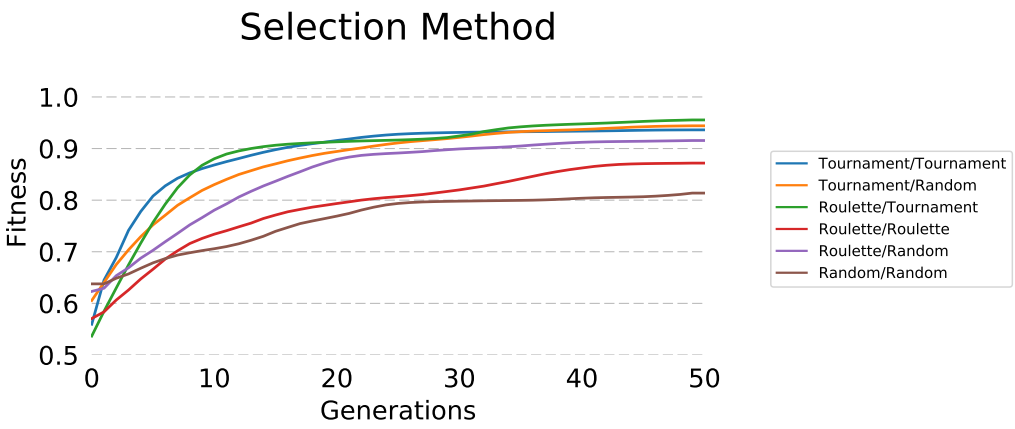


Figure 4.1: The fitness values with different selection methods.

4.3.2 Population size

Since the GA recombines parts of already existing solutions, a bigger population increases the chance to combine two good parts to create a better solution, and it also increases the probability that a individual already has a good fitness value from the beginning, therefore increasing the fitness of the solution immediately, as shown in Figure 4.2. Unfortunately, a increase in the population size also increases the execution time significantly, which requires finding a trade-off between speed and fitness. When increasing the population from 10 to 500 individuals, the fitness increases as well, but after a size of 200 individuals the increase is negligible compared to the increased execution time.

4.3.3 Mutation probability

If the mutation rate is too low, the probability to produce beneficial changes is low as well, while a high mutation rate can introduce disadvantageous changes to already good solutions. This can be mitigated to some extend with elitism, but this can introduce it's own set of problems. When we increase the mutation rate from 0% (only crossover) to 100% (only mutation), as shown in Figure 4.3, the fitness increases as well, although the difference

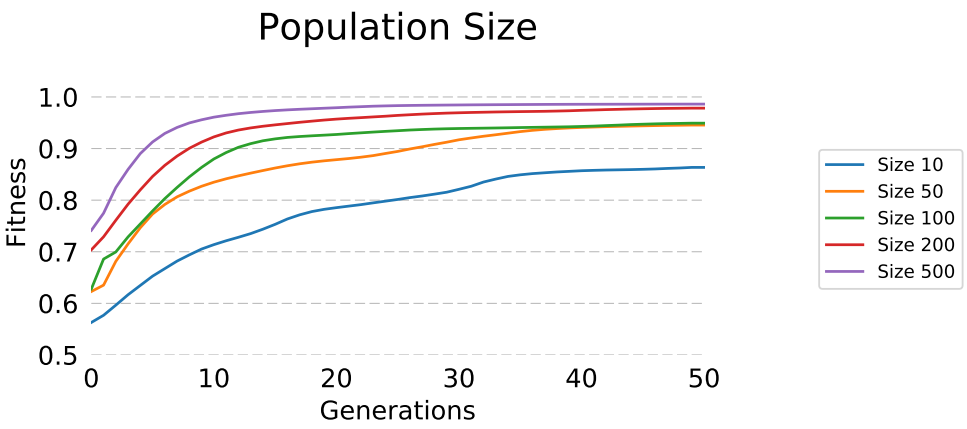


Figure 4.2: The fitness values with increasing population size.

is only significant in later generations. Using only mutation produces good results, but we found in further evaluation that the effect diminishes with a higher population size.

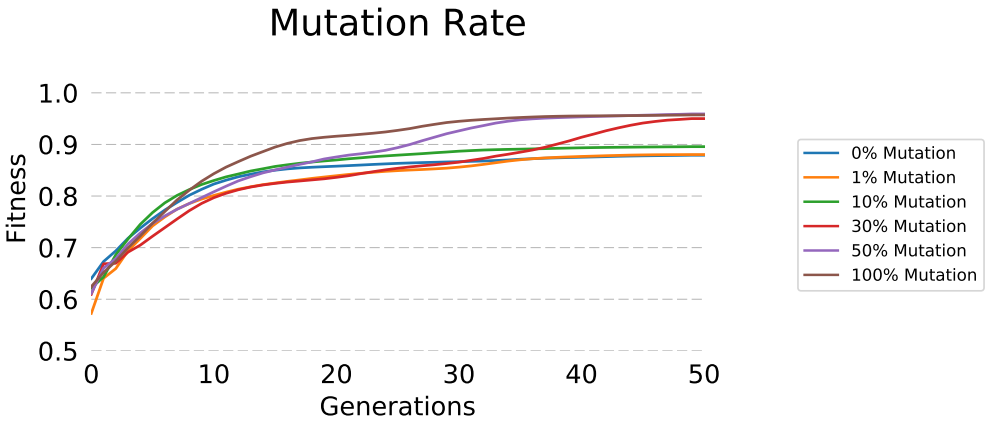


Figure 4.3: The fitness values with increasing mutation rate.

4.3.4 Elitism

Elitism allows the GA to explore the problem space surrounding good solutions by keeping them unchanged from one generation to the next, while still using them as parents. A small elitism rate in a large population can lead to a replacement of the elite in every turn, thus having no impact at all, while a large elitism rate can lead to stagnation. By changing the elitism rate from 0 to 45 individuals (90%), as shown in Figure 4.4, we find that the use of 10 to 15 individuals (20 - 30%) produces the best results, although the impact is not very significant. It does, however, ensure a steadily increasing fitness value. We also found in further evaluations that the impact of elitism is highest with small populations, and diminishes with increasing population sizes. And, as expected, keeping a large part of the population as elite does decrease the quality of the result significantly, and also leads to periods of stagnation.

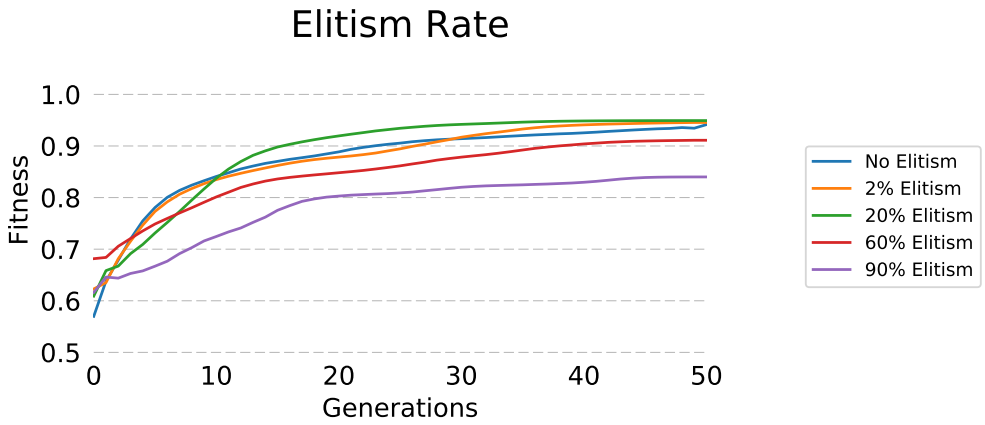


Figure 4.4: The fitness values with increasing elitism levels.

4.3.5 Initial Population

The length of the randomly initialized population was increased from 10 symbols up to 100 symbols (Figure 4.5). Since the target model can be modeled better with a certain number of symbols, the closer the length of

the initial population is to this value, the better the initial fitness value will be. This of course influences the initial performance of the algorithm. But we found that the GA is able to recover from a bad initial population within a few generations.

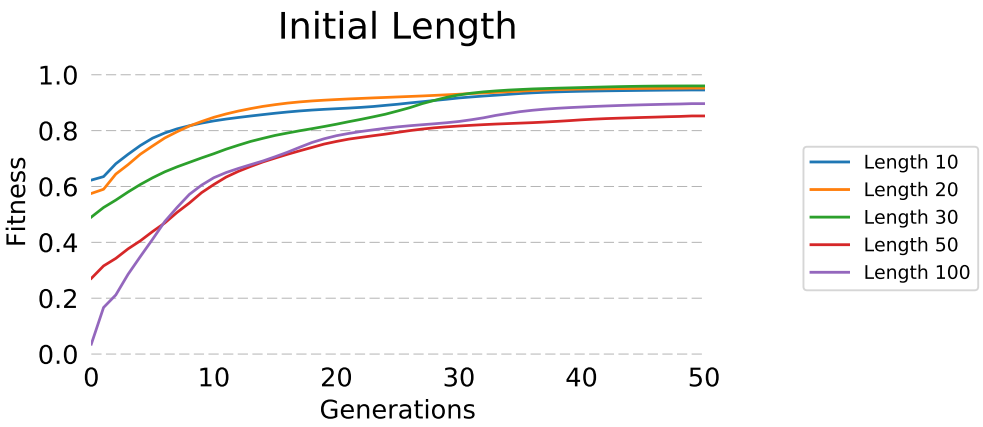
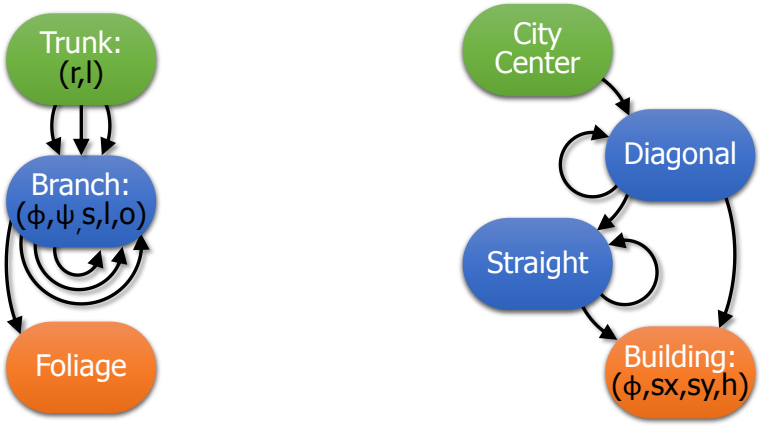


Figure 4.5: The fitness values with an increasingly complicated and less fit initial population.

4.4 Test scenes

The test scenes used recreate scenes from previous work (Talton et al., 2011; Ritchie et al., 2015): a spaceship generator with models shown in Figure 2.1 and the generator graph in Figure 2.2(a); a tree generator with models shown in Figure 1.1 and 4.11(a) (using a different foliage geometry), and generator graph in Figure 4.6(a), and a city generator with models shown in Figure 4.11(b-d) and generator graph in Figure 4.6(b). The spaceship generator is rather straight forward, as each new object occupies its own space and every node generates geometry. The tree generator is still relatively simple, however, there are many possible ways for branches reaching the same spot. Thus a variety of solutions and scenarios with self intersection are possible. Still, every node generates geometric output. The city generator has two recursive nodes which generate empty building lots and essentially iterate over the scene. Only the building node is responsible for creating output



(a) Tree (b) City

Figure 4.6: (a) The generator graph for the tree scene can output a trunk with a radius and length; the trunk and branches can have up to three sub-branches, which are each controlled by normal and inclination angles, radial scaling, length and offset; the foliage is always added as extension of a final branch. (b) While the city generator is simple too, only the building node directly generates visible objects; the blue nodes recursively iterate over the space of possible building lots; the building can be rotated, and its size and height is controllable.

geometry. Hence, decisions made during city generation might not directly lead to visible differences and a possibly large number of empty lots are required to extend the city towards areas where buildings should be set up. The targets and scoring functions for the scenes are: a sketch for the silhouette of the generated model from a fixed perspective in Figures 1.1 and 4.11(b,c), a volumetric target to fill for Figures 2.1 and 4.11(d), and a volumetric target to avoid for Figure 4.11(a).

4.5 Results

The Results obtained during optimization for the models presented in Figure 1.1 and Figure 2.1 are shown in Figure 4.7. The results obtained for the remaining test cases are shown in Figure 4.11. MH corresponds to the reversible jump MCMC for Metropolis procedural modeling including the

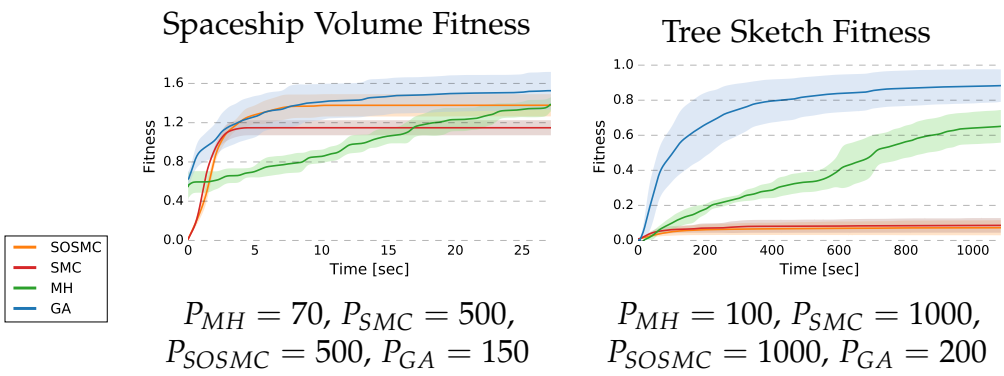


Figure 4.7: Development of the mean fitness values and standard deviations over time for the scenes shown in Figure 1.1 and 2.1. For a simple scene and short optimizations times as in the spaceship scenes, SOSMC and even SMC show good performance, while MH takes considerably longer to achieve good results. In complex scenarios, like the tree sketch, SMC and SOSMC only show good performance in the beginning, while MH slowly converges. GA in both cases shows very good early convergence rates and continues to improve over time.

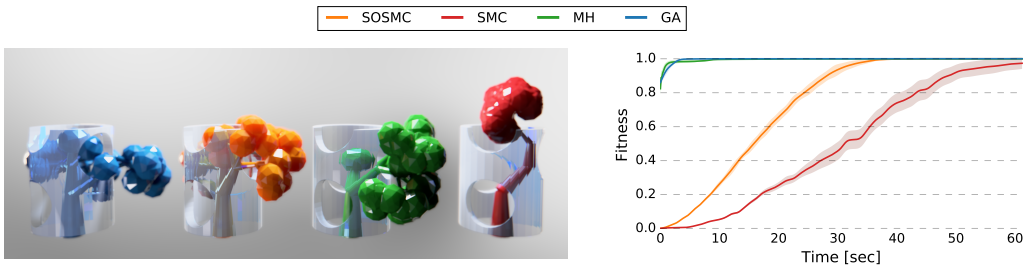
parallel tempering, nonterminal selection and annealing optimization described by Talton et al. (Talton et al., 2011). We use their proposed parallel tempering factor that assigns a 1% acceptance probability in the coldest chain to mutations that have a 70% chance in the hottest. As annealing factor we use 1.1. SMC and SOSMC correspond to the approaches described by Ritchie et al. (Ritchie et al., 2015). GA corresponds to the genetic algorithm proposed by us. In all cases, we run them in our framework, as described in Section 3.5. GA uses a mutation probability of 30%, divided into 13.5%, 13.5%, and 3.0% for grow, cut, and alter, respectively for tree and spaceship scenes and a probability of 40%, divided into 12.5%, 10% and 18.0% respectively for the city scenes. For tournament selection we use $k = 10$ and an elitism of one fifth of the population size. We tried to tune the population size for all approaches to achieve the best possible results and report them as P_{MH}, P_{SMC}, P_{SOSMC}, and P_{GA} alongside the test results. Note that SMC and SOSMC work better with larger population size while MH works more efficiently with smaller populations; GA usually works best with a value in between.

The spaceship and tree sketch scenes (Figure 1.1, 2.1 and 4.7) outline the performance of the approaches well. For simple models like the spaceships

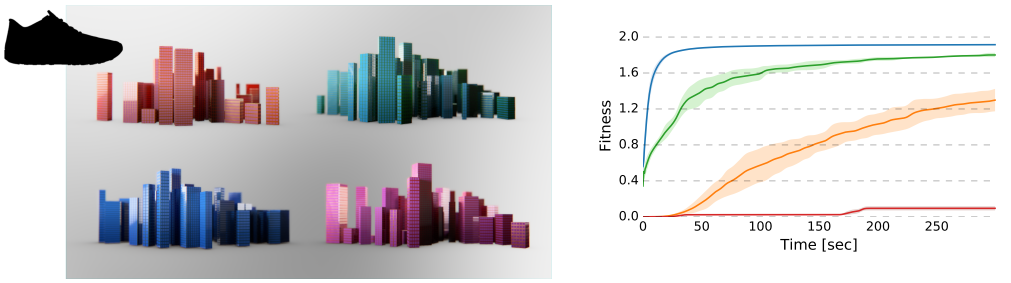
with volume evaluation, where every node of the generator graph leads to a geometric object, SMC and SOSMC perform quite well. As they require the evaluation of the scoring function for every step it takes some time until a first model of sufficient size is generated (about 4s). While SMC slightly outperforms SOSMC during this first seconds, SOSMC outperforms SMC in the longer run. However, both of them plateau pretty soon, as they cannot "undo" their initial choices. MH behavior is the opposite, since the initial state contains already full models, which score reasonably well. However, improving them takes a long time, falling below SMC and SOSMC. Then again, the improvement is steady and in the long run MH eventually outperforms SMC and SOSMC. GA also starts with a full initial population, from which it quickly increases the score, matching SOSMC during its best phase. However, in the long run GA keeps improving, similarly to MH. Note that the generation of the initial population hardly costs any time, as repeatedly calling the grow mutation has virtually no cost. For more complex targets, like the tree sketch image evaluation, SMC and SOSMC only show a good performance over the first seconds. Afterwards, MH and GA significantly improve the result, while SOSMC and SMC are stuck with the early parameter choices. GA in this case significantly outperforms MH. The GA solution after 3min achieves the same score as MH after 20min; SMC and SOSMC never reach this score. In this case GA profits strongly from the fact that it can copy partial solution from one of the three branches to the other.

The SMC and SOSMC methods have problems with complex generators, since choices made in the early iterations reflect in the performance during later stages. While this can be mitigated with a larger sample size at the cost of longer calculation time for each iteration, these complex generators usually also require a large number of geometric objects for good solutions. Which, due to the nature of SMC and SOSMC, require many iterations, therefore increasing the required calculation time even further. This makes choosing the right number of samples highly problem dependent, and especially cumbersome when aiming to achieve the best result within a limited computational budget.

We observe comparable behaviors in the other test cases shown in Figure 4.7. The large number of volumetric scoring function evaluations for uncompleted models reduce the early performance of SMC and SOSMC (a). More

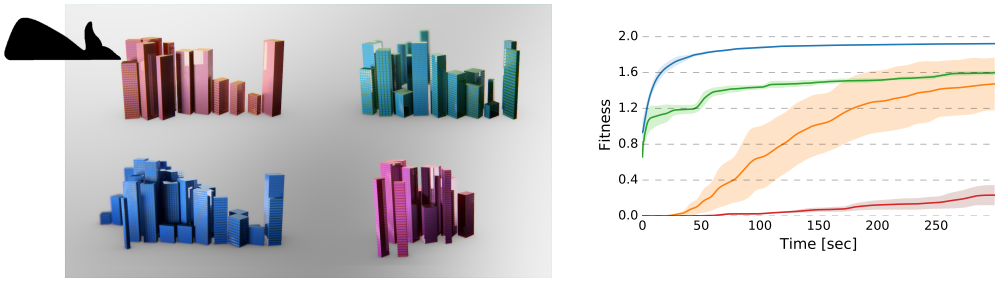


(a) Trees avoiding a cylinder with holes: $P_{MH} = 50$, $P_{SMC} = 250$, $P_{SOSMC} = 250$, $P_{GA} = 200$

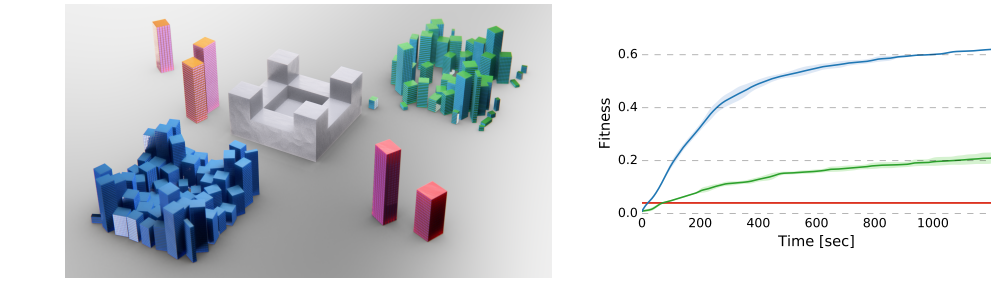


(a) Matching the skyline of a city: $P_{MH} = 60$, $P_{SMC} = 170$, $P_{SOSMC} = 170$, $P_{GA} = 180$

complex generators, like the city (b,c), which contain nodes that do not generate any visible objects are troublesome for the convergence of SMC and SOSMC. However, after these issues have been overcome, SOSMC at least can catch up with MH, before it plateaus. MH on the other hand always slowly and steadily converges. For the volumetric city scene (d), the weights were chosen such that wrongly placed buildings are punished severely. This removes the advantage of the initial population for MH and GA. But even so they manage to improve the score, with GA significantly outperforming MH, while SMC and SOSMC do not succeed to improve upon the solution after the first couple of generations. GA in almost all cases seems to combine the best of all worlds. Starting with a random population there is enough information to recombine good initial solutions which quickly increase the score. Furthermore, in contrast to SOSMC, GA continues to improve similarly to MH. Overall GA achieved the best results in all test cases during initial convergence and in the long run.



(a) Matching the skyline of a city: $P_{MH} = 80$, $P_{SMC} = 200$, $P_{SOSMC} = 200$, $P_{GA} = 200$



(a) Matching the volume of a city: $P_{MH} = 100$, $P_{SMC} = 800$, $P_{SOSMC} = 800$ $P_{GA} = 300$

Figure 4.11: Mean fitness values and standard deviation obtained for different test scenes plotted over time. SMC and SOSMC struggle during the beginning of the volumetric tree scene (a), as they run the relatively costly voxelization on many non-finished models. The image city scene (b,c) is also difficult for them as the generator graph is more complex. They fail to improve after the first few generations for the volumetric city scene (d). MH works reasonably well in all cases but shows slow convergence rates. GA in all cases shows the best convergence behavior among all tested approaches. Note that the fitness values are not normalized and depend on the chosen weights.

5 Discussion and Future Work

We have shown that genetic algorithms can be used for controlled procedural modeling in a variety of procedural approaches. Using our compact tree representation to encode genomes and linking this tree to the generator graph allows for efficient mutation and reproduction operations while making sure that all resulting chromosomes are valid. Comparing our GA to the state-of-the-art, we found that using a GA combines the best characteristics of previous approaches. The initial convergence is better than SOSMC and in the long run it significantly outperforms MCMC, yielding the best convergence in all stages of optimization. We attribute that to GA being able to combine the best features from an entire population. We also found that GA shows good performance independently of the complexity of the used generators and scoring function. Also, in our testcases it was rarely necessary to tune the parameters to achieve good results, which hints at good stability of the approach. A limitation we experienced is that by copying parts of the genome, similar features can be duplicated in the model, *e.g.*, the remains of a heart shape are visible in the left branch of the GA result in Figure 1.1. Furthermore, our implementation of elitism is somewhat susceptible to "sample impoverishment", leading to almost identical copies of the same individual in the elite population, but we did not notice a diminished performance caused by this.

In the future we will increase the efficiency of our approach further, by using a GPU-based generator. This will also increase the efficiency of the proposed scoring function evaluations which already use the GPU. Furthermore, we believe that combining SOSMC, MCMC, and GA and automatically choosing the best approach might even increase convergence further, *i.e.*, using SOSMC to create an initial population, switching to GA after a few iterations and then switching to MCMC in the long run. Considering our description of SOSMC and MCMC within our GA framework this seems

to be possible. Finally, the major challenge we see is tackling full inverse procedural modeling, where the generator consists of the entire set of modeling operations and the goal is given by a fully detailed object or a scan of a real world object. We believe genetic algorithms have the potential to help make full inverse procedural modeling possible.

Bibliography

Beneš, B. et al. (2011). "Guided Procedural Modeling." In: *Computer Graphics Forum* 30.2, pp. 325–334. ISSN: 1467-8659 (cit. on p. 6).

Blickle, T. and L. Thiele (1996). "A Comparison of Selection Schemes Used in Evolutionary Algorithms." In: *Evolutionary Computation* 4.4, pp. 361–394. ISSN: 1063-6560 (cit. on p. 18).

Boers, Egbert J. W. (1995). *Using L-Systems as Graph Grammar: G2L-Systems* (cit. on p. 9).

François, O. and C. Lavergne (2001). "Design of evolutionary algorithms - A statistical perspective." In: *IEEE Transactions on Evolutionary Computation* 5.2, pp. 129–148. ISSN: 1089778X. DOI: 10.1109/4235.918434 (cit. on p. 25).

Funes, Pablo and Jordan Pollack (1998). "Evolutionary Body Building: Adaptive Physical Designs for Robots." In: *Artif. Life* 4.4, pp. 337–357. ISSN: 1064-5462 (cit. on p. 8).

Havemann, Sven (2005). "Generative mesh modeling." PhD thesis. University of Braunschweig-Institute of Technology (cit. on pp. 5, 9).

Krecklau, Lars and Leif Kobbelt (2011). "Procedural Modeling of Interconnected Structures." In: *Computer Graphics Forum* 30.2, pp. 335–344. ISSN: 1467-8659 (cit. on p. 5).

Lindenmayer, Aristid (1968). "Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs." In: *Journal of theoretical biology* 18.3, pp. 300–315 (cit. on p. 5).

Lintermann, Bernd and Oliver Deussen (1998). "A Modelling Method and User Interface for Creating Plants." In: *Computer Graphics Forum* 17.1, pp. 73–82 (cit. on p. 5).

Magdics, Milán (2009). "Real-time Generation of L-system Scene Models for Rendering and Interaction." In: *Proceedings of the 25th Spring Conference on Computer Graphics*. SCCG '09. Budmerice, Slovakia: ACM, pp. 67–74. ISBN: 978-1-4503-0769-7 (cit. on p. 5).

Marvie, Jean-Eudes et al. (2012). "GPU Shape Grammars." In: *Computer Graphics Forum*. ISSN: 1467-8659 (cit. on p. 5).

Měch, Radomır and Przemyslaw Prusinkiewicz (1996). "Visual Models of Plants Interacting with Their Environment." In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, pp. 397–410. ISBN: 0-89791-746-4 (cit. on p. 5).

Merrell, Paul et al. (2011). "Interactive Furniture Layout Using Interior Design Guidelines." In: *ACM Trans. Graph.* 30.4, 87:1–87:10. ISSN: 0730-0301 (cit. on p. 6).

Müller, Pascal et al. (2006). "Procedural Modeling of Buildings." In: *ACM Trans. Graph.* 25.3, pp. 614–623. ISSN: 0730-0301 (cit. on p. 5).

Ochoa, Gabriela (1998). "On genetic algorithms and lindenmayer systems." In: *Parallel Problem Solving from Nature — PPSN V: 5th International Conference Amsterdam, The Netherlands September 27–30, 1998 Proceedings*. Ed. by Agoston E. Eiben et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 335–344. ISBN: 978-3-540-49672-4. URL: http://dx.doi.org/10.1007/BFb0056876 (cit. on p. 8).

Okabe, Makoto, Shigeru Owada, and Takeo Igarash (2005). "Interactive Design of Botanical Trees using Freehand Sketches and Example-based Editing." In: *Computer Graphics Forum* 24.3, pp. 487–496. ISSN: 1467-8659 (cit. on p. 5).

O'Neill, Michael et al. (2009). "Shape Grammars and Grammatical Evolution for Evolutionary Design." In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. GECCO '09. Montreal, Québec, Canada: ACM, pp. 1035–1042. ISBN: 978-1-60558-325-9 (cit. on p. 8).

Palubicki, Wojciech et al. (2009). "Self-organizing Tree Models for Image Synthesis." In: *ACM SIGGRAPH 2009 Papers*. SIGGRAPH '09. New Orleans, Louisiana: ACM, 58:1–58:10. ISBN: 978-1-60558-726-4 (cit. on p. 5).

Patow, G. (2012). "User-Friendly Graph Editing for Procedural Modeling of Buildings." In: *IEEE Computer Graphics and Applications* 32.2, pp. 66–75. ISSN: 0272-1716 (cit. on p. 9).

Pilat, M. L. and C. Jacob (2008). "Creature Academy: A system for virtual creature evolution." In: *2008 IEEE Congress on Evolutionary Computation*

(IEEE World Congress on Computational Intelligence), pp. 3289–3297 (cit. on p. 8).

Pirk, Sören et al. (2012). "Plastic Trees: Interactive Self-adapting Botanical Tree Models." In: *ACM Trans. Graph.* 31.4, 50:1–50:10. ISSN: 0730-0301 (cit. on pp. 1, 5).

Prusinkiewicz, Przemyslaw and Aristid Lindenmayer (1991). "The algorithmic beauty of plants (the virtual laboratory)." In: (cit. on p. 5).

Ritchie, Daniel et al. (2015). "Controlling Procedural Modeling Programs with Stochastically-ordered Sequential Monte Carlo." In: *ACM Trans. Graph.* 34.4, 105:1–105:11. ISSN: 0730-0301 (cit. on pp. 2, 6, 9, 19, 20, 29, 31).

Schwarz, Michael and Pascal Müller (2015). "Advanced Procedural Modeling of Architecture." In: *ACM Trans. Graph.* 34.4, 107:1–107:12. ISSN: 0730-0301 (cit. on pp. 1, 5).

Sims, Karl (1991). "Artificial Evolution for Computer Graphics." In: *SIGGRAPH Comput. Graph.* 25.4, pp. 319–328. ISSN: 0097-8930 (cit. on p. 6).

Sims, Karl (1994). "Evolving Virtual Creatures." In: *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '94. New York, NY, USA: ACM, pp. 15–22. ISBN: 0-89791-667-0 (cit. on pp. 6, 13).

Sipser, Michael (2006). *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston (cit. on p. 10).

Stava, O. et al. (2014). "Inverse Procedural Modelling of Trees." In: *Computer Graphics Forum* 33.6, pp. 118–131. ISSN: 1467-8659 (cit. on pp. 2, 6).

Steinberger, Markus et al. (2014). "Parallel generation of architecture on the GPU." In: *Computer Graphics Forum* 33.2, pp. 73–82. ISSN: 1467-8659 (cit. on p. 5).

Stiny, George Nicholas (1975). "Pictorial and Formal Aspects of Shape and Shape Grammars and Aesthetic Systems." PhD thesis (cit. on p. 5).

Talton, Jerry O. et al. (2011). "Metropolis Procedural Modeling." In: *ACM Trans. Graph.* 30.2, 11:1–11:14. ISSN: 0730-0301 (cit. on pp. 1, 2, 5, 19, 21, 29, 31).

Wadge, William W. and Edward A. Ashcroft (1985). *LUCID, the Dataflow Programming Language*. San Diego, CA, USA: Academic Press Professional, Inc. ISBN: 0-12-729650-6 (cit. on p. 9).

Weber, Jason and Joseph Penn (1995). "Creation and Rendering of Realistic Trees." In: *Proceedings of the 22Nd Annual Conference on Computer Graphics*

and Interactive Techniques. SIGGRAPH '95. New York, NY, USA: ACM, pp. 119–128. ISBN: 0-89791-701-4 (cit. on p. 5).

Wonka, Peter et al. (2003). "Instant Architecture." In: *ACM Trans. Graph.* 22.3, pp. 669–677. ISSN: 0730-0301 (cit. on p. 5).

Xu, Kai et al. (2012). "Fit and Diverse: Set Evolution for Inspiring 3D Shape Galleries." In: *ACM Trans. Graph.* 31.4, 57:1–57:10. ISSN: 0730-0301 (cit. on p. 8).

Yeh, Yi-Ting et al. (2012). "Synthesizing Open Worlds with Constraints Using Locally Annealed Reversible Jump MCMC." In: *ACM Trans. Graph.* 31.4, 56:1–56:11. ISSN: 0730-0301 (cit. on pp. 2, 6).