Procedural generation of populations for storytelling

September 22, 2015

ACKNOWLEDGEMENTS

I would like to thank Rafael Bidarra and Ben Kybartas for allowing me to do my thesis with them in Delft, and their patience in helping me with the project. Ben has helped a great deal coming up with solutions for problems that arose, brainstorming together to get possible solutions. Rafael and Ben pushed me toward writing a paper about my thesis and helped me write it, resulting in a publication[8]. I would like to thank John-Jules for being my supervisor-from-afar in Utrecht, letting me do my own thing in Delft, but there for me once I needed to conclude things in Utrecht. Finally, I would like to thank my parents for supporting me throughout my studies.

Abstract

Procedural world generation is often limited to creating worlds devoid of people and any background. Because of this, creating a vibrant, living world is still a problem that requires a skilled designer. In this thesis, we present a method that generates a socially connected population in any virtual terrain, using a mixed-initiative simulation of settlements that adapt to the world and to a designer's input. Using this simulation, we develop a number of sample worlds that convey the expressive potential of the approach. We further evaluate ease of use with a user study. As a proof-of-concept, we implement the system to bridge the output of a terrain generation tool to the input of a narrative generation tool.

Contents

Chapter 1

Motivation

Whenever someone asks me what artificial intelligence will allow us to do in the future, first I ask them what their job is. Then, I proceed in telling them that their job will almost certainly be taken over by robots... and that I will be making those robots. To me, the question is not *if* robots take over every single one of humankind's jobs, it's *when* - preferably, as soon as possible! We are not there yet though, and my thesis is meant to contribute to this arguably noble goal. Of all the possible jobs robots and computers could take over, my thesis focuses on taking the jobs of story writers.

As I have always been interested in procedural content generation, it was clear to me I wanted to do something in that field. To me, procedural generation is very interesting because in a way it allows a computer to be creative - and creativity is a typical trait that gets attributed solely to humans, and perhaps animals. I do not believe humans are all that special - we have no secret hidden ingredient that allows us to be creative, an ingredient that computers supposedly cannot have. Still, jobs involving creativity will probably be among the last to be stolen by computers, and with my thesis I hope to contribute in making this process faster.

When I first heard the term 'Procedural Storytelling' I was immediately intrigued by the idea of having computers create stories by the press of a button. Just imagine feeling like reading a book and being able to create one simply by stating what it should be about, or playing a computer game that has infinite story content because it just generated its plot!

Utrecht University did not really have a department dealing with procedural population, so instead I visited Rafael Bidarra at TU Delft and got introduced to Ben Kybartas. Ben was already working on procedural

storytelling and together we made a plan for my own thesis. Procedural generation of stories is not a trivial problem, and of a much greater scope than a master's thesis project. Also, Ben had already done research and created ReGEN, a system that can automatically generate narratives based on an existing world.

Since procedural terrain generation tools already exist, and Ben's system needed a populated world for input, the missing link was clear, and with it our research question:

How can we procedurally generate a population fit for storytelling using an existing terrain?

Chapter 2

Introduction

Procedural generation techniques are often necessary for creating the diverse, rich worlds found in many exploration and role-playing games. Games like Skyrim [2] make use of procedural landscapes and, uniquely, procedural story generation (in the form of the Radiant Quest system). However, the landscapes by themselves are boring and lifeless, and fail to provide the most important elements for a story, the people. We argue that to better integrate story generation into existing procedural generation techniques, we need to explore methods for creating populations that contain the necessary data to be used for story generation. Dwarf Fortress [1] notably creates worlds filled with an overwhelmingly diverse population, by simulating the *history* of the world, creating towns, populations and social relations out of the evolution of this world over time. This leads to very diverse results, however the methods of Dwarf Fortress do not allow for designer intent and are therefore inescapably tied to one context.

This thesis describes a method that takes a landscape as input, and uses a designer-driven historical simulation to generate an entire population, complete with settlements, individual people and social relations. The designers can customize their worlds in a number of ways, and work at the level of population building rather than individual character design. Using the definitions provided by the designer, a world is simulated, using an optimization algorithm based upon evolutionary algorithms to accurately determine whether populations migrate, collapse, or develop new relations with each other. In the case of *offline* generation, the designer is granted further interactions during simulation, to be able to fine-tune the positions and layouts of their populations as seen fit. A massive set of characters are created as the result of this simulation, complete with relations and properties

relating to their corresponding settlement. Since the population is created after the simulation, the user has full control over its size. For experimentation, the algorithm has been tested with a variety of input landscapes and population designs, and an evaluation was performed to determine the ease of use and openness of the algorithm to iterative design. Furthermore, as a proof-of-concept, the algorithm has been successfully used to connect a sketch-based landscape generation tool [15] to a narrative generation tool specifically geared towards creating quests for RPG games [9].

2.1 Related Work

Storytelling systems which work with large populations are quite rare. In interactive fiction, the worlds are typically populated with a small number of well-defined, complex, hand-authored characters. Even larger scale emergent storytelling games such as McCoy et al.'s PROM WEEK [13] contain relatively few characters.

Typical approaches to combining procedural content with game worlds have focused on tying the story directly into the world generation, by creating dungeons [4], maps [16] or even entire game worlds [7]. In the latter case characters are generated, but only to serve particular events in a given plot. Our approach, instead, targets *emergent* game environments, in that we do not care about the representation of a single story, but in creating what Mateas [11] describes as a *narratively pregnant* world, one rich with potential for many stories.

Lebowitz [10] explored methods for creating characters to be used for storytelling in his UNIVERSE system. His focus was on creating characters with a personality that was consistent and coherent with an existing world. However the goal of that study was to create complex new characters that integrate properly into a hand-authored set of characters. Our method does not focus on creating complex, completed characters, but instead on large-scale populations with plausible and consistent interrelations.

The history generation of DWARF FORTRESS [1] served as inspiration for our approach to population generation. While the details of the generation are unknown, the game creates relatively templated characters, but situates them in a world in which several hundred years of history are simulated. Likewise, the large scale approach to having relations between settlements is inspired by the complex social relations present in CRUSADER KINGS II [14], where European countries develop alliances and conflicts based on events during game play.

Even though population generation is never mentioned, Emilien et al. [5] use an algorithm based on lichen growth [3] to determine the placement of villages in an arbitrary terrain. Inspired by the results of this method, we expand upon the original method to integrate social interactions and support population generation.

2.2 Overview

Our method can produce a world populated by virtual characters that can be used for storytelling, using an empty landscape as input. The resulting population is not just a large set of randomly generated characters - each character has a little background and relations to other characters and places in the world. Characters generated by our method are simple, but still contain enough information to allow them to become a believable and interesting part of a story, while all of this information is still consistent with other characters from the population.

To create our population, we simulate a number of settlements in the landscape, much like the method described by Emilien et al. [5]. These settlements form the basis of the creation of the population, as each settlement represents a sort of blueprint for the characters that are generated from them.

We use an algorithm that optimizes the fitness score of each settlement, which is determined by the landscape and the designer-defined way the settlement interacts with this landscape and other settlements. Our method lets these settlements optimize, and also allows relations between them. We allow a designer to create populations by looking at the landscape and specifying how characters would live there. They can design the types of settlements they'd like to see as well as the relations that may exist, and simulate the world until the results are satisfactory.

Chapter 3

Method description

In this section, we take an in-depth look at the proposed method, how to design a population, and how the simulation and character generation take place.

3.1 Definitions

The input of our method, **the landscape**, is defined as

$$\langle H_m, T_m, (F) \rangle$$

where H_m is a height map, T_m is a terrain type map, and F is an optional set of terrain features such as forests and rivers. The height map represents the height of the terrain in meters as a float value for each pixel. The terrain type map simply indicates the terrain type for each pixel on the terrain. Examples of these terrain types are grasslands, mountains, hills, etc. Both the terrain types and terrain features can be defined by the designer and are used later to determine where settlements, and their inhabitants, prefer to be. In the implementation, we used input directly from SketchaWorld [15], which contains both this height data, terrain type data and terrain feature data. It was very easy to convert this to our own format, allowing us to use SketchaWorld to quickly create a landscape with.

The output of our method is a population composed of virtual characters. A **character** is defined as

$$\langle S, D, R \rangle$$

where S is the character's settlement, D is the character's district, and R is the set of relations this character has to other characters and settlements.

The characters generated by our method are relatively simple: they do not have a strong personality, life goals or a specific history. What they do have, however, is a general but consistent background and social network, which should make it fairly easy for a storyteller to adapt them to be more specific. Essentially, our characters are nodes in a population's social network. We define no id, but instead leave it up to the storytelling system to define that.

A **settlement** is defined as

$$\langle P, D_l, R, P_r \rangle$$

where P is a position in the virtual world, D_l is a list of one or more districts, R is the set of relations this settlement has with other settlements and P_r is this settlement's prototype. Settlements are used as the main tool for creation of the population, as they serve as a blueprint for characters generated from them.

A **district** is defined as

$$\langle N, P_d, R_a \rangle$$

where N is a list of needs, P_d is a list of products and R_a is a list of relations this district allows its parent settlement to use. A settlement adopts needs, products and allowed relations from its districts, making districts an important building block of settlements.

A **product** is a resource function, much like those used by [5] and [3]. These functions can be seen in figure 3.1. Examples for use of these functions are distance from water (Close distance function), terrain slope (Balance) and terrain types that provide resources - woods, mountains, water (for fishing) and fertile land (Open distance function). These functions are especially well suited for our optimizing algorithm, as their gain diminishes as they approach the optimum. To illustrate the difference between an open and a close distance function, consider a village that wants to be as close to the water as possible because fishing will be easier (open distance function). If the village gets too close however, it might be victim of a flood. Similarly, a hunting village might not want to live too close to a wood because of wild animals attacking villagers.

A **relation** is defined as a

$$\langle T, S_s, D_s, R_e, (A) \rangle$$

where T is the relation's type, S_s is the set of settlements the relation applies to, D_s is the pair of distances relevant for this relation (a preferred distance and a maximum distance), R_e is the set of resources that can be

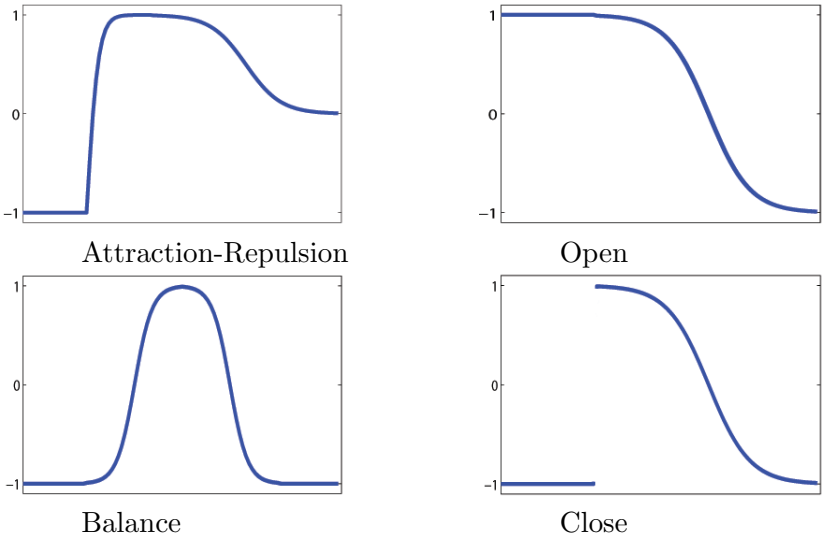


Figure 3.1: The resource functions[5] used by a district to determine its resource production efficiency. The x-axis is the relative distance from the terrain feature that allows resource income, the y-axis is the resource fitness - the amount of resource produced.

exchanged for this relation, and A is the relation's optional attitude, which might restrict other relations from forming once an attitude is established. Relations form an important way for settlements to exchange resources, and also a strong basis for the relationships between the characters in the population.

3.2 Designing the virtual world

To create a world, a designer needs to define three of the method's main ingredients: Districts, Relations and Prototypes. Also, designers must tell the system how many settlements they want of each prototype, and from there they can just watch the world unfold, interfering if desired.

The relations and districts are strongly tied to the population that is later generated: Any relation that is between two settlements will cause such relations to exist between members of their respective populations, while districts can say a lot about a character's background. **Prototypes** have an indirect effect, as they force settlements to adopt a certain stereotype. The effect is that rather than adapting to the world, the settlement

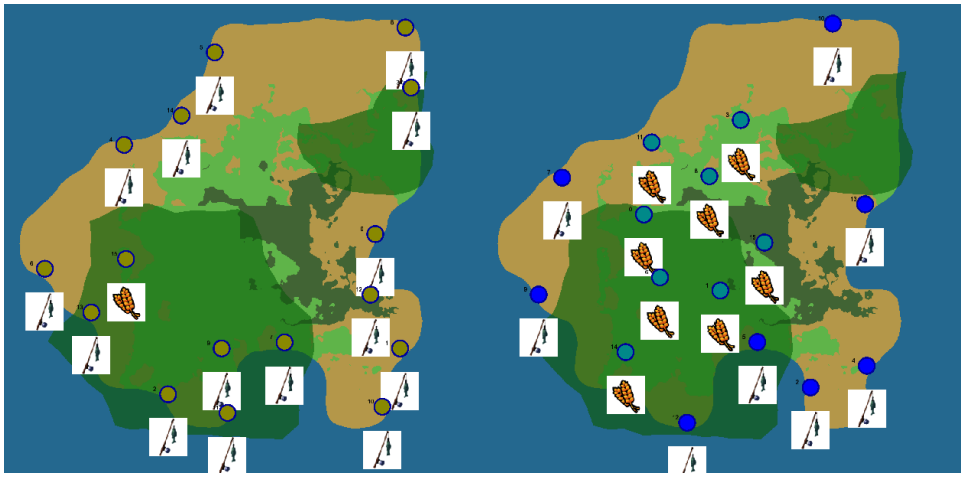


Figure 3.2: Left: A simple island with only fishers and farmers, no relations and no defined prototypes. Right: A simple island with only fishers, farmers, no relations, but with two prototypes that cause a designer-defined amount of fishers (8) and farmers (8) to exist.

should instead find its optimal place in the world. Essentially, prototypes are an additional modifier for the evolutionary algorithm's objective function, causing those settlement that conform to their prototype to score higher. The effect on a settlement's fitness is strongly determined by the user's design of the prototype, making it a efficient tool for guiding the population's generation. To illustrate the strong effect of prototypes on designing a world, consider Figure 3.2. In the first image, we did not design any prototypes, and designed fishers to be slightly more efficient at producing food when close to the sea. As a result, most settlements decide to be fishing settlements. In the second image, we introduce two prototypes: Fishers, which demands the settlement has a fishing district and no agricultural district, and Farmers, which works the other way around. Both examples were allowed 15 iterations of optimization.

Designing a district is very simple. Throughout the thesis, please refer to Table 3.1 for some definitions. In these examples, the resource functions should be read as:

$$\delta(R, R_m, T, D_{min}, D_{max})$$

where δ is the type of distance function, for example distance_open, R is the resource that is being produced, R_m is the multiplier, signifying how much of the resource is being generated, T is the type of terrain feature, D_{min}

is the minimum distance to this terrain feature and D_{max} is the maximum distance to this terrain feature.

By adding the option of a *resource multiplier*, one can make certain districts more efficient at producing them. In effect, a multiplier of 1.0 means that under optimal conditions (for example, a fishing village at its minimum distance to water), will produce exactly enough food to sustain itself.

Using only the definitions from Table 3.1, we can create military settlements raiding fishing settlements, while other settlements are optimized to have agricultural, mining and military districts and to be self-sufficient. In this example, we are using only three resource types and two types of relations. Recall that the open distance function has optimal results for close distances, but declines as it gets closer to the maximum. In the Fishing district, the production of food is an open distance function using water as terrain feature, a minimum distance of 30 and a maximum distance of 150. This means that such a settlement's fitness will be optimized by moving closer to water, until its distance is 30, or closer. The Military district, described in Section 6, behaves similarly, but our domination map used values in a range [0.0 1.0], hence the smaller range. This domination map was generated from the height map, and is useful to determine relative height for each terrain pixel. The military district has two resource production functions, one that relies on the terrain's domination, and one that produces its resource manpower at a constant rate. All resources and types of districts are simply examples we came up with, so a designer can define any named resource by defining it as need or product for a district.

Designing relations is quite easy, as they are modeled as restrictions on a resource exchange. In Table 3.2 we can see two basic relations we used during testing. In this example, the Trade relation has no restrictions on its import or its export resources, it has a preferred distance of 550, a maximum distance of 650 and it requires the existing attitude not to be Negative. The attitude of a relation prevents another relation that has a different attitude to also establish between two settlements. For example, if settlement A raids settlement B, we would not expect B to establish a trading relation with A. To do this, we allow relations to have an attitude *effect*, and an attitude *requirement*, which can take the value 'Positive', 'Negative' or 'Neutral'. As soon as two settlements have a relation between them that has such an effect, the attitude between those takes on this value. For example, a raid relation's effect is 'Negative', while a trade relation's requirement is 'not Negative'.

Designing prototypes involves determining what the restrictions are for a settlement. The main restrictions the designer can apply are:

Table 3.1: Example district definitions

Fishing
needs: food produces: distance_open(food, 2.0, water, 30, 150) relations: Trade
Agricultural
needs: food produces: distance_open(food, 3.0, fertility, 0, 5) relations: Trade
Military
needs: food, metals produces: distance_open(manpower, 4.0, domination, 0, 0.3) constant(1.0) relations: Raid
Mining
needs: food produces: distance_open(metals, 1.0, mountains, 0, 200) relations: Trade

Table 3.2: Example relation definitions

Trade
in(*) out(*) distance_open(550, 650) attitude_required(not Negative)
Raid
in(*) out(manpower) distance_open(250, 400) attitude_effect(Negative)

- Which district and relation types are (not) allowed?
- How many districts/relations are allowed?

- How does the settlement respond to proximity of other settlements?

With these simple properties, prototypes become a powerful tool for designers to steer their settlements in their desired direction. For example, by limiting the maximum number of relations and districts, we can force settlements to specialize. We can also create a simple raiding prototype by using Military as the only allowed district, and having Raid as preferred relation.

Finally, we allow our designer to determine a prototype's *social* type, which helps determine how the settlement deals with other nearby settlements. The social type can have two values, or it can be undefined: Master or Slave. A master settlement has a territory and benefits from having slave settlements in that territory, and is penalized when its territory overlaps other masters' territory. A slave settlement benefits from being in any master's territory, and only has slight penalties for being close to other slave settlements. If no social type is defined, a settlement simply benefits from keeping its own preferred distance from other settlements. A master's territory and regular settlements' preferred distances can be defined for each prototype, making it very easy to introduce a degree of size to each prototype's settlement. We experimented with around 5 master settlements and around 20 slaves, and simulating the world quickly caused the masters to take their own space in the world, while the (smaller) slave settlements quickly distributed in these territories. In the resulting population, there was a clear social network between masters and their slaves, while settlements without a social type tended to be solitary. Table 3.3 shows a few prototype definitions. Note how definitions can be used as 'parent' definition, copying all properties from it; in this way, for example, Small_Food has all properties from the Small prototype.

3.3 Simulating the virtual world

In this section, we discuss the simulation of the virtual world, and the mixed-initiative interaction between the simulation and designer. It is possible to simply simulate a number of generations, then generate a population with the press of one button. However, if designers have a certain world in mind, we wish to enable them to create it just as they want it. For this reason,

Table 3.3: Example prototype definitions

Large_Military
districts: Military social_type: Master master_territory_size: 350

Small
max_districts: 2 max_relations: 3

Small_Food : Small
districts: not Military max_districts: 1 social_type: Slave

Extra_Large
social_type: Master master_territory_size: 450

Small_Raider : Small
districts:Military relations: Raid districts: not Fishing not Agricultural

Small_Mining
social_type: Slave districts: Mining max_districts:1

we designed the method to be mixed-initiative, allowing the designer to proceed to the next generation of settlements, make customizations, and continue. We designed our interface to allow designers to simply drag and drop settlements, and change a settlement's properties, immediately seeing the effect that it has on their fitness. At any time, designers can allow the system to simulate the next generation. Once the designers are done, they can generate the population from the current generation, and still keep going after that too.

Before the initial generation can be created, the designers specify the types of settlements they would like to see in his world, by supplying a list of prototypes. Each prototype is assigned to a settlement and placed in the world by randomly picking, or 'polling', a number of locations (we polled 15 times) in the landscape, and determining the fitness of the settlement, should it be placed there. For each settlement, the best scoring poll is selected for the first generation. The advantage of using a polling method is that the resulting locations are nondeterministic and cause the settlements to be in local optima rather than global optima after simulating. After all, we are not looking for the optimal situation of the simulated world - rather, we want to generate a possible way a population may exist in that world. The fitness of a location for a settlement is determined by the following factors:

Landscape : is the slope suitable, is the spot legal (e.g. no water)?

Resource : based on the settlement's districts and relations, how well will it do resource-wise?

Spacing : are we too close or too far from other settlements. This depends

on the social type of the settlement.
Prototype : is the settlement true to its prototype?

Because we are working with an optimization algorithm, we scored all these factors on a scale from 1.0 to potentially negative infinity (but usually -1.0). For example, if the landscape is perfect - no slope and the position is legal - the score for that will be 1.0. For resources, this was trickier, since more should always be better. For this reason, meeting all needs gives a score of 0.5, and as the amount of resources approaches infinity, the score approaches 1.0. We use similar strategies for the other fitness factors and compute the final fitness by taking their equally weighted sum, causing settlements to optimize on all fronts.

The spacing fitness is determined by distance to all nearby settlements. It depends on the social type of the settlement, but each settlement always has a preferred distance to each other settlement. A settlement with no other settlements in its preferred distance radius has a spacing fitness of 1.0, while each nearby settlement diminishes this value. Master and slave settlements are an exception: Master settlements do not lose fitness when slaves are in their territory, while slaves lose fitness if they are not in a master's territory, proportional to the distance to the nearest master (causing them to move toward that master).

When a new generation is created, each settlement makes copies of itself and mutates them slightly. Mutation involves a change in one or more properties including position, relations and districts. The 'child' that has the highest scoring fitness is chosen and used in the next generation. To prevent a settlement from becoming less optimal by mutation, we always include the original settlement in the selection process.

3.4 Population generation

When the designer is satisfied with the world, the population generation can take place. For each settlement (prototype), the designers can determine how many characters they want to have generated by the method. In this phase, the settlements act as character generators: each settlement now has districts, which help determine the background for a character. For example, a settlement with a Military district might produce soldiers or raiders (depending on relations), while fishing villages should produce mainly fishermen. Furthermore, settlements have relations to other settlements, and often they already have attitudes to other settlements too. So, if we have settlement A raiding settlement B, the attitude between these two

settlements is Negative (see Table 3.2). If we now generate a character for settlement B, it can automatically inherit this from its settlement: it has a negative attitude towards A, and anyone from A. To give the character more flavour, we can even give this attitude flavour: the attitude is negative because A raids B.

However, if we generate all characters like this we end up with a whole group of characters from settlement B that hates every single person from settlement A. For a number of reasons, this is undesirable. For example, a story such as Romeo and Juliet would not be possible. It also makes sense that many characters from settlement B do not even know anyone from settlement A. This is why we allow the designer to set a few simple values for character generation:

inter_settlement_relation_chance: the chance a character knows another character from a different settlement.
inter_district_relation_chance: the chance a character knows another character from a different district (in the same settlement).
intra_district_relation_chance: the chance a character knows another character from the same district.
attitude_change_chance: the chance a random attitude change occurs between any two characters.

These values can be used to add some variation to the initially generated characters. We found that if these values are not used, an enormous amount of relations is being generated: every character will have a relation to every other character in the settlements their own settlement has a relation to. However, since we expect the output of our method to be used by a procedural story generator, it can also be considered the responsibility of that system to tune these possibilities.

Chapter 4

Method design

Throughout the development of the method, we have had many brainstorming sessions on how to tackle certain problems and because of this, a number of solutions that didn't work out were redesigned. In this section, we will consider the most important ones, some of which were not used in the eventual method.

4.1 Biases

When we started testing the evolutionary algorithm, it quickly became clear that it was very good at optimizing the individual settlements. While this was good, it also made it very hard to make interesting worlds - many of the settlements were the same because this just happened to be the optimal solution for them! To counter this problem we eventually came up with prototypes. Before the prototypes we first tried a similar idea: *Biases*. Like prototypes, they are a property of a settlement and influence the settlement's fitness score in the selection process. With a fantasy world setting in mind, a bias such as *Race* would be possible, ie Human, Elf or Dwarf. Furthermore, we wanted to see biases such as *Wealth* (Rich versus Poor) and even *Alignment* (Good versus Evil). However, even though these concepts can be very strong aspects in a story, we found it impossible to relate them to the evolutionary algorithm because it was very hard to link these often abstract concepts to actual heuristics for the algorithm. Since these biases were part of the user design, this would have made that process much more complex.

4.2 3D visualization

Our application now uses a simple two dimensional user interface and visualizer. Originally, we modeled the terrain in a three dimensional environment, using *Ogre3D* as rendering engine. Because of the complexity of modifying aspects of the world rendering, and the overhead in compilation time we decided to use a two dimensional engine instead. The implementation still allows for easy modification of its rendering engine, and can be easily adapted to work with another.

4.3 Procedural terrain

Since the method is designed to work with arbitrary terrains, originally we wanted to include a simple terrain generator. Even though the terrain generator worked, we very much preferred working with pre-designed terrains because they allowed us more control to include or exclude certain terrain features. Since we hardly used this generator halfway through the project, it was removed so it would not be a distraction later.

4.4 Two stage algorithm

When we were designing the evolutionary algorithm, the original plan was to have it simulate two stages: a *nomad/settling* stage and a *survival/optimization* stage. In the nomad stage, settlements would start out in a random position and attempt to find the perfect spot to 'settle down' in. Then, in the survival stage, these settlements would actually compete for resources and be able to initialize relations between them. During testing we found that modifications to a settlement such as addition of districts or relations can drastically influence their optimal position. Because of this, being able to change locations after the nomad stage would be beneficial and improved the results of the simulation. Furthermore, some user defined districts and prototypes might require relations in order to work at all, for example - a raider prototype that permits its settlement to gain resources solely by stealing them will be very inefficient if no relations are allowed (yet). Because of this, these kinds of settlements fall back to just being fishers instead of raiders, and this gives them a bias that's unfit for the raiders they might become in the survival stage. The two stage approach is no longer used, and now the evolutionary algorithm simply uses a single approach where settlements can both optimize their relations and districts, as well as their

position.

4.5 Evolutionary Algorithm

For the optimization process, we chose for an evolutionary algorithm, as it has some desirable features. Our problem is an optimization problem with multiple objective functions, which is especially suitable for evolutionary algorithms. The inspiration for this is the growth of lichen, detailed in [3] and [5]. Another less obvious advantage is the algorithm's tendency to get stuck in local minima - in our method, this actually corresponds to the settlements' lack of knowledge about the world. One missing element is crossover. In an attempt to include crossover we tried having multiple generations of settlements evolve at the same time in the same landscape, and using crossover between those generations for future generations. However, the generations were too dependent on their own distribution of settlements, and these crossovers did not improve the algorithm, while the overhead of having so many generations did have a negative impact on the performance of the algorithm. The most challenging aspect of designing the evolutionary algorithm was the fitness functions. We are optimizing a number of different and often conflicting criteria, and weighting these objective functions took much experimentation and planning. For example, how do you compare the score of a settlement's location to its resource amount score? And does this apply to every user design or input world? Because these questions are not answerable to an acceptable degree, the weights of the objective functions are made user-definable as well.

4.6 Prototypes

For a great part, our method is a creative tool, which should allow designers to create a world as they envision it, assisted by the tool as efficiently as possible. Using just an algorithm to populate a world and simulate it does nothing to actually assist the designers in reaching their goals. To help with this, we came up with *prototypes*, which serve as settlement blueprints that can be defined by the designer. Introducing prototypes increased the amount of tasks for the designers, but should decrease their overall amount of work.

A very important aspect of the prototypes is that they are not part of the algorithms optimization because they cannot change during the simulation. This also means that the user will have to define the number of times each

prototypes should exist in the world, since every settlement now needs to have a prototype assigned to it. With prototypes, settlements still attempt to reach a optimum, but their prototype helps define what that optimum should be. This way, designing a world becomes much more intuitive since the designer can now determine what certain settlements should look like when they are optimized.

4.7 Mixed initiative approach

Since we want our method to be both flexible and autonomous, we designed it with a mixed initiative approach in mind. Once the designer finished defining the districts, relations and prototypes, the method can automatically take over and finish. However, the designer might have planned certain things like settlements in certain places or relations between certain settlement types. If these are not found in the world, we still want the designer to be able to modify such things, for example by dragging around settlements or simply modifying properties of them or their relations. Thanks to this approach, we can now easily create a world without any intervention, but at the same time we can fine tune a world to any extent, making the method very flexible.

4.8 Resource-based heuristic

Even though we are using a resource-based heuristic for the evolutionary algorithm, we are explicitly not interested in resource management. Tracking actual resources, resource sources and trades of resources would make the simulation very complex, not to mention the amount of extra research required in such fields. Instead, we take a very abstract approach concerning resources: Settlements can produce resources, they can need them and they can trade their resources using relations. Even though settlements keep track of an arbitrary value representing an amount for each of their resources, we leave it to the designer to define what a certain amount means. During testing, we found out that this requires some degree of trial-and-error - for example in the amounts of food produced by certain districts. However, the effects of changing these values was usually quite clearly visible in the simulation and therefore intuitive.

A particularly hard problem has been the design of the evolutionary algorithm's objective functions for the resources of a settlement. Since resource quantities are not strongly defined, it is very hard to determine how fit a

candidate solution is based on its resources. What we do have is a needed quantity to compare the 'have' quantity to, so we are able to measure the shortage or excess compared to the needed amount. However, this brings another problem: If the objective function is based on percentages instead of absolute values, more can be less - If a settlement gains a need by gaining a district, its resource fitness may suddenly become lower because of it! This has been an unsolved problem for a while, but it was partly solved by the addition of prototypes in the method.

The formula used for determining a single resource's fitness score is:

$$f(x) = \begin{cases} 0.5 & \text{if } have = need \\ 0.5 * \frac{have}{need} & \text{if } have < need \\ 1.0 - \frac{0.5}{have/need} & \text{otherwise} \end{cases}$$

A settlement's final resource fitness score is the average of all its resources' values. A settlement that has exactly sufficient resources (or no needs) has a resource score of 0.5, while settlements with shortage have linearly lower scores, and those with excess have exponentially declining higher scores, up to 1.0. This way, settlement can keep improving their resource score even when 'sufficient' - selection pressure still is a factor at that point.

4.9 Districts

When designing the user interaction with the method, we considered the question 'what gives a character personality?'. A very defining property of a character and his background is his profession - what he does all day. For example, when asking 'who is Jack Sparrow?', his profession - a pirate - is the first thing that comes to mind. An agricultural village would not make a whole lot of sense as background for Jack Sparrow, and it is clear that it might not make sense to try to fit a character into a background after creating it. However, it should be noted that even though it's unlikely, such a background is still possible! Instead of creating a background for an existing character, we decided to focus on creating characters the other way around, by starting out with the background. Districts are the very core of the method, as they are the components in settlements that are most alike the characters that are being created from them. When a district is being designed, the designer's main thought should be what the characters in it should be like.

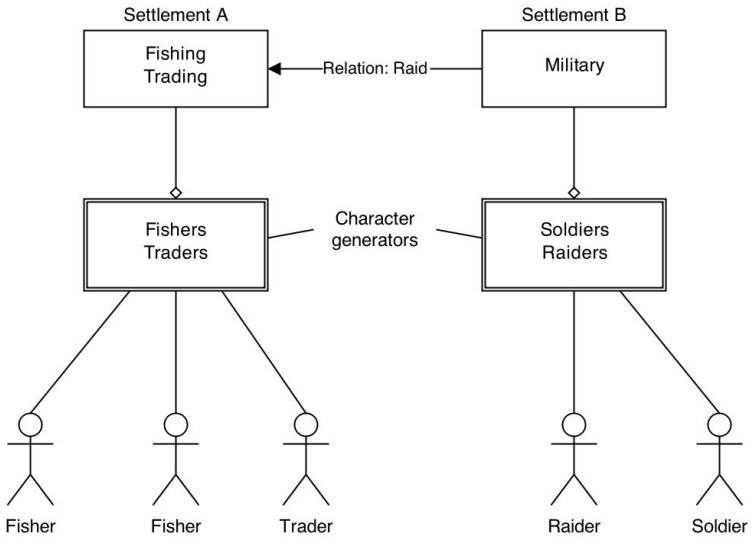


Figure 4.1: An early image showing the design of the character generators, and their place in our system.

4.10 Hierarchical design

Even though the characters created by our method are relatively simple, they still have a strong background because of how the method is set up: A character is generated from a district, which is part of a settlement, which is somewhere in the world. This hierarchical design is meant to not only add a sense of uniqueness to characters, it should also allow the designer to fine tune characters to a great degree, since their background isn't very specific yet. Image 4.1 shows an early image of this design. The most important aspect of the hierarchical design is that all generated characters inherit properties from the layers above them. In this design, all characters generated from the same district are identical. This may not seem desirable, but we chose to keep it that way so instead, the storytelling system can make these types of adaptations to the characters.

4.11 Relations

Relations are one of the most important components of any story. During the design of our relations we have had to make some very important decisions about relations and the way they work. We encountered a few problems,

and during development we have had to re-design the relations to cope with them.

One-way versus two-way

Originally, we designed relations to work in a two-way fashion: When a relation exists from settlement A to settlement B, this would imply that there is also a relation from B to A. Intuitively, it makes sense to look at relations in such a way, however we found a few problems with this approach. First of all, having relations be a property of two settlements is not desirable in a evolutionary algorithm, because it affects both at the same time. A relation that is beneficial for one settlement might be detrimental for another, immediately introducing a problem for the algorithm: Do we allow settlements to terminate any relation as part of their optimization? If we would, we can expect to no longer see any relations like these, so this was not an option.
One possible solution we considered was to isolate the relation optimization from the settlement optimization. Instead of having relations be a settlement's property, they would have their own evolutionary optimization algorithm and objective function. However, modeling different relations was still hard: How do we design an objective function that allows both relations that benefit both settlements and relations that benefit only one, while being disadvantageous to the other? For this, we tried to come up with a system that allowed 'altruistic' and 'egoistic' relations, but we quickly found out that this made the system more complex than it ought to be. Finally, we decided relations should be part of a settlement after all, so that they could be part of their evolution. This meant relations had to be one way, instead of two-way.
Having one way relations still had its problems, but we have been able to solve them. The one way relations were initialized from a settlement, without directly affecting the other settlement. This means that when settlement A raids settlement B, settlement B has no negative effects, and will not attempt to, for example, move away. Even though this feels unintuitive, we are not modeling a real progressive history for our world, so it is very much acceptable. An important choice has been how to model resource exchanges for relations. Essentially, all relations are exchanges of resources between settlements, but modeling all resource quantities in the world is far outside the scope of this project, nor is it of any significant consequence for the story.

Attitude

When designing relations we kept some important 'base' relations in mind: a mutually beneficial trade relation, and a relation that only benefits its initiator, for example a raid relation. A variant of the latter would be a relation that would only benefit the target, for example a protect relation where military assistance is given to the target. Since we're working with an optimalization algorithm, having relations that are bad for one side can pose a problem, since that side will try to optimize the relation away. Using one way relations partly solved this, but still left a problem. Intuitively, when a settlement raids another settlement, we would (normally) not expect any other relations between them. However, during testing we found out that a settlement that is being raided would casually initialize a trade relation with that settlement, too. Throughout the project, we've tested a few different approaches to tackle this problem (including Biases), but none of them worked out in a satisfactory way. For example, the addition of Biases worked well with relations, but it made the design too complex and too abstract. We want to have a general system that does not force users to use specific and potentially complex concepts like 'good' and 'evil'. However, we also needed to be able to distinguish harmful and helpful relations, and for this we introduced attitudes. Starting with 'good' versus 'evil', we moved to 'altruistic' versus 'egoistic' and eventually ended up using 'positive', 'neutral' and 'negative' (and none). Even though we have been quite skeptical about the attitude system, during testing it has not been a limiting factor during relation design.

Chapter 5

Implementation

In this chapter, we look in detail at the implementation of the method, and discuss some of the choices made in it. We look at the main method's implementation, and also discuss the GUI implementation briefly.

5.1 Overview

We implemented our method in C++, using Simple and Fast Multimedia Library (SFML) for simple 2D rendering. Most tests were done on a computer with a Intel Core i7 @2.20 GHz. Our implementation is capable of automatically iterating through 50 generations of settlements in under 2 s on average, then generate around 1450 characters in under 1 s on average.

To create our landscape model, we used the procedural terrain sketching tool SketchaWorld [15]. SketchaWorld allowed us to make landscapes in minutes, which made the whole progress much more streamlined.

Other libraries we used include:

- libnoise, a C++ library that allows the use of many different procedural generation techniques such as perlin noise.
- A number of boost libraries, including boost::regex for regular expressions (parsing of user files), boost::thread for multithreading and boost::geometry for working with polygons.
- pugixml, a C++ library that allowed us to read xml files fast and easily.
- visual leak detector, a library that keeps track of allocated memory and notifies when a memory leak is detected.

Our main goal is to make characters fit for storytelling, and to test this goal, we successfully integrated our system with ReGEN [9], a graph-rewriting tool that can be used for narrative generation. We were successfully able to generate narratives within our populations, and further have the population be updated by changes made within each narrative.

Our method has shown to be an effective tool for designing a population for a world. It is true that one cannot fully control or predict how exactly certain ideas play out, since many factors have influence on settlements in the world. For example, when a designer wants a world with lots of farms and fisher villages, but fails to define prototypes, they might find all of them become fishers (if those are defined to be more efficient). However, even if the design is nontrivial, the mixed-initiative designer interaction during the simulation makes up for that.

5.2 Method implementation

5.2.1 Landscape loading

Since we want to be able to work with multiple types of landscape sources for the method, we designed a TerrainLoader base class that can be used as base for any new terrain loader. Currenly, we have 2: SketchAWorldTerrainLoader and LibNoiseTerrainLoader, which is used for generating terrains using the libnoise library. We will primarily consider the SketchAWorldTerrainLoader since that was used for all the examples seen in section 6. The LibNoiseTerrainLoader is a terrain loader that uses libnoise, a c++ noise generation library, to load random procedural terrains on the fly. However, working with more predictable pre-defined terrains was almost always preferable. This class makes use of a few optimization techniques which are discussed in section 5.2.3.

The SketchAWorldTerrainLoader loads the following data from any SketchAWorld project:

Ecotope definitions:
Ecotope data as provided by SketchAWorld. Contains information about all ecotopes, including name, id, (debug) colours, as well as nutrition, rockiness and wetness values which allowed us to determine properties like fertility for each ecotope.

Terrain features:
Terrain features are saved in a separate XML file in a SketchAWorld project, and imported by our method. Recall that terrain features are special zones in the terrain like forests, rivers and lakes. These zones are described in

```
float* height_data_;
unsigned char* type_data_;
float* slope_data_;
float* dominance_data_;
unsigned char* feature_data_;
float* water_distance_map_;
float* mountain_distance_map_;
float* forest_distance_map_;
float* fertility_distance_map_;
```

Figure 5.1: The data held by the chunk datatype. Most data is dynamically allocated after the chunk's size is determined.

the XML file by series of points that form polygons. However, because we actually needed the spacial data for these polygons, we converted them to boost::multi_polygon format, allowing us to perform fast operations like boost::multi_polygon::within() to find out whether a point is within the polygon.

Terrain height/type data:

Even though our SketchAWorld landscapes are of a fixed size, we designed the method in such a way that terrains of arbitrary sizes can be loaded fully or partially. To achieve this, our TerrainLoader classes work with a datatype called *ChunkGroup*, which loads in only a part of the terrain, a *Chunk*, on demand. This approach worked very well with SketchAWorld landscapes since they are already saved as chunks in a project. The *Chunk* data class holds all relevant data for a chunk of the loaded terrain. The data that is used can be seen in Figure 5.1. Because we are using large arrays for all the 'raw' data a chunk uses, the memory usage is quite large. However, accessing the data is also very fast since it requires no further computation once the chunk is initialized. We chose for speed over memory efficiency because the world data will be (randomly) accessed by every settlement and its candidate solutions during each generation of the simulation of the world.

Chunk data

The data saved in the Chunk class is partly derived data which is saved for faster performance. We will consider each one shown in Figure 5.1 briefly.

- Height data: The height of the terrain in meters, one float per pixel of the map.
- Type data: The type of the terrain, one byte per pixel of the map. The possible values are defined by SketchAWorld for the SketchA-

WorldLoader, and need to be defined by the user for any other terrain loaders.

- Slope data: Derived from height data, the slope data ranges from 0.0 at no slope to 1.0 at any slope steeper than 45 degrees.

- Dominance data: Derived from height data, the dominance value of a pixel represents how relatively high that part of the terrain is. We calculate this value by taking samples from a 9 x 9 grid around the source pixel, and taking the average of these values. The resulting value ranges from 0.0 to 1.0, where 0.0 means the surrounding terrain is higher on average, and 1.0 means the surrounding terrain is lower on average. The algorithm used is detailed in Appendix 1.

- Feature data: Even though SketchAWorld saves all terrain features in a single file, we wish to access relevant data for each seperate chunk. To achieve this, we load in the whole terrain feature and add the appropriate parts of it to each chunk, by taking the set intersection of the chunk's rectangle and the full terrain feature.

- Water/Mountain/Forest/Fertility distance maps: The pixels in each of these maps simply represent the euclidean distance from that pixel to the nearest occurrence of that terrain feature. These maps were make using *Distance Transforms of Sampled Functions*[6]. These distance maps are one of the most important components of the objective functions of the evolutionary algorithm, and adding more is simply a matter of linking a terrain feature or terrain type with a resource. Our method then automatically generates a distance map for this resource, which can be used to determine the distance from that resource.

5.2.2 Population Simulator

The core of our method is the PopulationSimulator class, which loads the user definitions, and then runs the population simulation. The simulator uses a Generation class that represents a single generation of settlements, and saves a copy of each generation in a vector for later reference. To simulate the world, the simulator calls the Generation class' NextGeneration method, causing it to use an evolutionary algorithm to attempt to improve each settlement's fitness. The algorithm detailed in Appendix 2 shows how a Generation object saves different fitness values for each settlement:
Location fitness, which returns 1.0 if the location is valid, and 0.0 otherwise.

Settlement distance fitness, which is a quite complex function and can be reviewed in Appendix 3. The complexity of the function is caused by the different ways the system deals with distances between settlements, and their effect. For example, a master-prototype settlement will want to be close to slave settlements, but not too close. On the other side, all slave settlements want to be close - but not too close- to their master, and also not too close to other slaves.
Next, *resource fitness* is determined by looking at all of a settlement's needs and produces, including those it can get from relations. Exchange of resources between settlements is done in a quite abstract manner - the only requirements for a trade to be possible are that the delivering party can provide one of the receiving party's needs, and that the receiving party has any type of spare resource to exchange it for. This means that the trades are not very realistic, but at the same time they are defendable because the resources do exist, and are produced. But most importantly, designers aren't required to understand the complex underlying mechanics of realistic trades and resource management, meaning they can focus on the design aspects that are actually important!
The *relation distance fitness* is measured by how well the distances between a settlement and the settlements it has a relation with conform to their relation's minimum and maximum distances. If all distances are within their bounds, the fitness is 1.0. For each distance that is too far away, a fixed penalty (we found out a value of 0.15 worked well) is applied linearly as the distance exceeds the difference between a relation's maximum distance and its preferred distance. For example, if the maximum distance is 10.0 and the preferred distance is 6.0, the difference is 4.0 - if the actual distance is 8.0, the penalty is $0.15 * (8.0/4.0) = 0.30$.
The *prototype fitness* works with a similar penalty system - for each property of a settlement that does not conform to its prototype, it's fitness is deduced by a fixed penalty, in this case a more severe, non-linear penalty of 0.35 was used. Because these penalties are subtracted from the fitness value directly, the fitness value can drop below 0.0. For the evolutionary algorithm, this is no problem, and it works very well because every improvement in the settlement is of equal value fitness wise, meaning there will always be a similar pressure for it to improve.

5.2.3 Optimization

We implemented a number of optimization techniques to speed up the terrain preprocessing and the population simulation. In the project, we often

worked with large landscapes, with a lot of image processing involved. When a landscape's height map was loaded, all other maps needed to be generated from it - the slope map, dominance map and all distance maps. To speed this up, we used binary files to save all the raw data to. Now, when a terrain was loaded for the second time, its derived data could be loaded from the hard drive directly - this change decreased the terrain loading time from 11 seconds to 2 seconds!
To further increase the program's speed, we used the OpenMP multithreading API in various placed throughout the implementation. For example, the creation of all the data maps for the terrain is highly optimized this way.

5.3 GUI Implementation

When we decided to try to get a paper published, we also decided the application needed a graphical user interface so it could be shown off. For this, we used Simple and Fast Multimedia Library (SFML) and Texus' Graphical User Interface (TGUI). The GUI for the method was really a separate project that managed all the simple text files used by the simulator, and made adapting the values in them more user-friendly. The design of the GUI can be seen in Figure 5.2. The files containing the user definitions for the districts, relations and prototypes are simple text files that are interpreted by a parser. An example of the districts file is shown in Appendix 4.

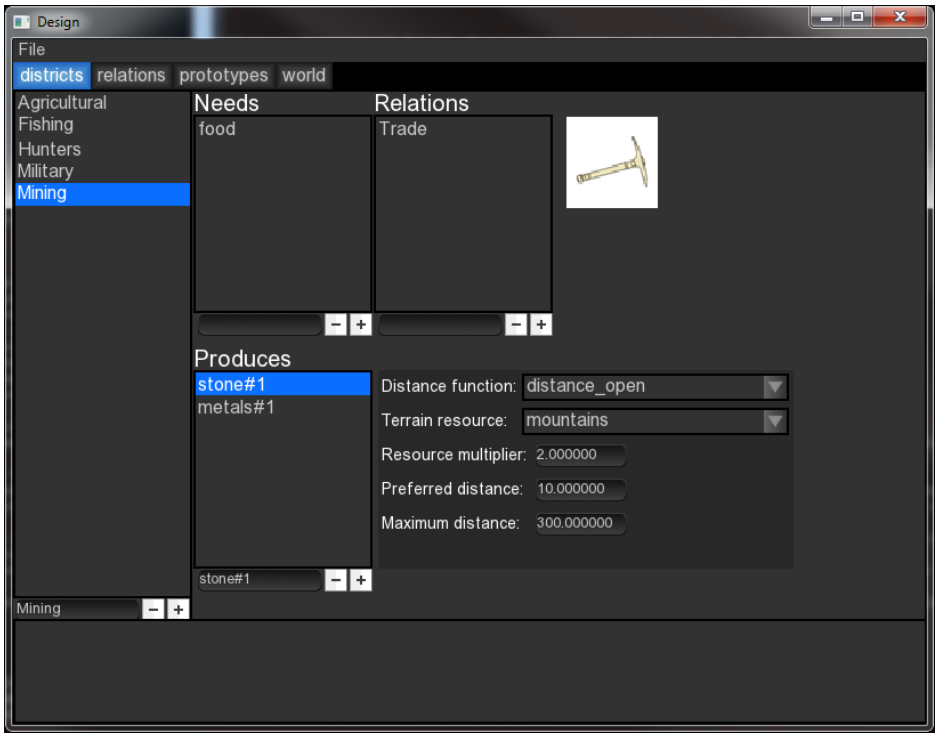


Figure 5.2: The Graphical User Interface, allowing users to modify definitions of districts, relations and prototypes, and run the simulation when they are done.

Chapter 6

Application

To test our method, we have built a number of different input landscapes, and designed varying sets of districts, relations and prototypes. In this section, we present a few designs we made, and show how our method deals with these inputs. We show a simple world in detail, and consider some others more broadly.

Our first world is a medieval-themed design, and has also been the running example throughout the thesis. We used exactly the districts as presented in Table 3.1, the relations in Table 3.2 and the prototypes in Table 3.3. The initial generation of the resulting world can be seen in Figure 6.1 (top). The images under a circle signify which districts a settlement has. Even though no optimization has yet taken place, the settlements often have locations that are quite suitable for their prototype. However, without simulation the social graph is quite limited. Figure 6.1 (bottom) shows the same world after 10 generations without user intervention. It is clear that most settlements remain true to their stereotype, but without user intervention, some others have not been able to escape their local maxima (yet). For example, the settlement in the far left bottom should be large and have a military district, but since there are absolutely no other settlements around to have relations with, it became a fishing settlement instead. Of course, this problem was already there in generation 0, and could have been fixed by the designer, e.g. by simply dragging it to a new position.

Expanding this first example to be more specific is quite simple. We added a hunter-gatherer district that generates food from forests, and to give our world a fantasy game-like feeling, we introduced a few prototypes: *Elven* : Using the new hunter-gatherer district only, making them drawn to forests. *Dwarven* : Using the mining district, making them drawn to

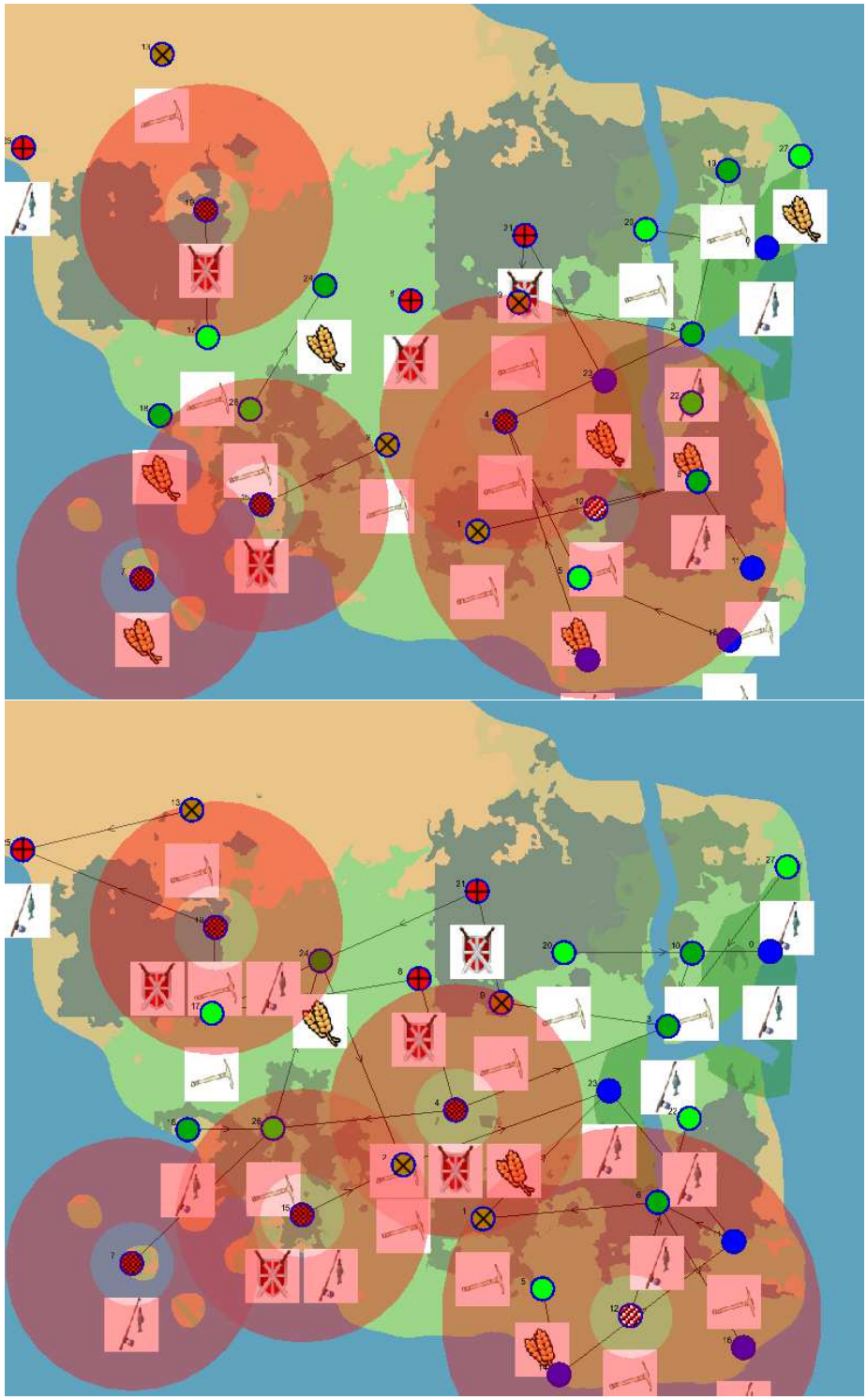


Figure 6.1: The initial generation of a medieval-themed world, defined with the values presented in the example tables. A plus sign is a Raider prototype, a cross is a Mining prototype, settlements with a territory are the Large (Military) prototypes, and all the others Small. Top: Initial generation. Bottom: world after 10 generations.

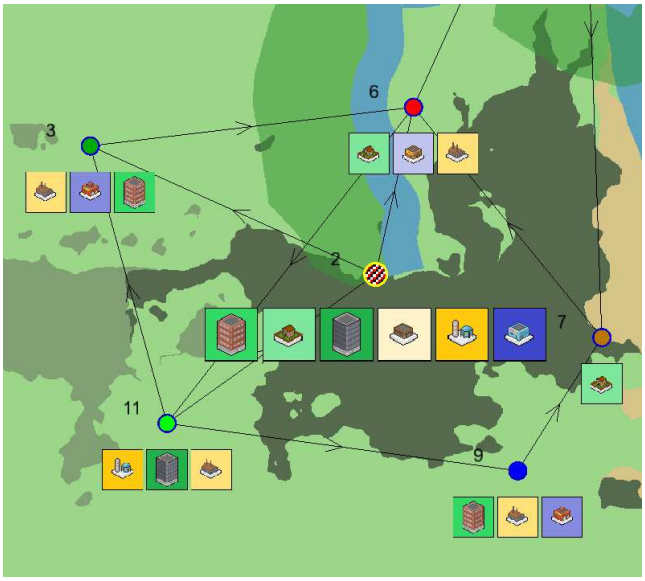


Figure 6.2: Simulation of a SimCity [12] like world. Commercial, industrial and residential districts of low, medium and high wealth form cities that exchange resources like workers, materials, products and even influence and money.

mountains. *Orcs* : Only allowed to use Military districts, and Raid relations. By making only these simple changes, the designer can expect elven settlements (usually) in forests, dwarven settlements in the mountains, and orcs raiding everyone, everywhere. Although interestingly, sometimes an orc settlement would be too far away to raid anyone and would instead become a self-sustaining fishing settlement.

Beyond this quite straightforward example, we developed some very different approaches. Figure 6.2 shows a region of the same landscape for our fantasy-themed example, but modern settlements are being simulated instead. Aiming to mimic the basic mechanics of Simcity [12], we defined residential, commercial and industrial districts of different social classes: low wealth, medium wealth and high wealth. We model unique relations between cities, allowing trade and freight shipments, but also allow cities the option to use influence and bribery to get resources from other cities. The result is a much darker, more corrupt vision of modern society that may be more befitting of storytelling.

Another popular setting for storytelling and games is the wild west, so

we simulated a world where the collection of gold is the goal for (most of) the population. We set up our landscape to contain just a few places where gold can be collected, and designed a specialized prototype that can collect gold while the others cannot. Other settlements are allowed to be towns that provide workers and earn gold by having them work in the mines, bandits that can steal gold, and 'lawbringers' that can arrest bandits. In Figure 6.3, we can see how all settlements converge toward each other, even though we did not define any master or slave prototypes in this scenario.

Exploring alternative forms of resources, our final population example aimed to mimic zombie apocalypse worlds. This setting uses people and zombies as the main resources, allowing zombies to infect human settlements and humans to cure zombie settlements. Relations are mapped to scavenging, zombie hunting, resource gathering, and acquiring medical supplies. To demonstrate the versatility of the system, we generated four different populations in very different landscapes, as shown in Figure 6.4. Likewise, we split the prototypes into one large city which contains the medical center and laboratory needed to create zombie cure, several cities largely infected with zombies, and then a number of small scavenger or survivor groups.

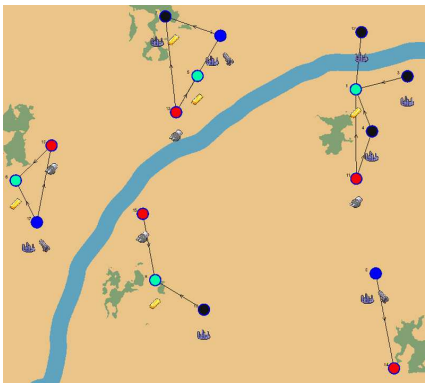
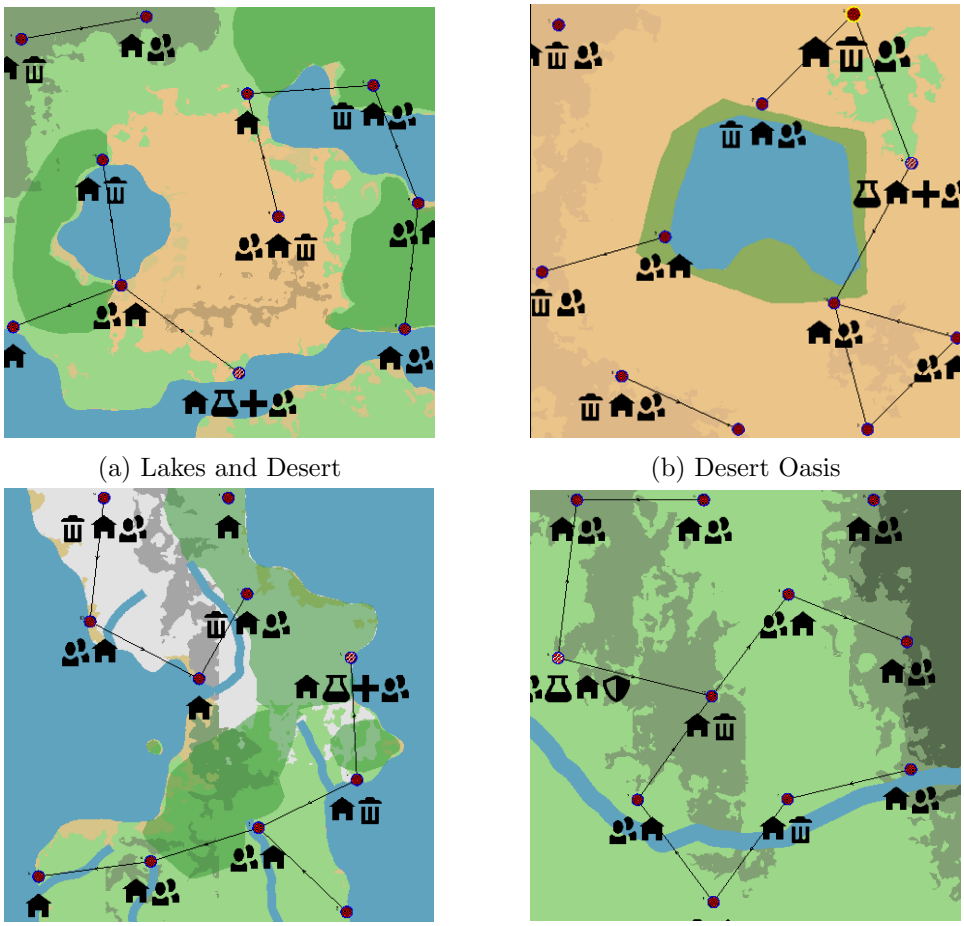


Figure 6.3: A wild west scenario, with only gold as terrain-based resource. Because of this, all settlements tend to converge to these locations.



(a) Lakes and Desert

(b) Desert Oasis

(c) Frozen Island

(d) Natural Valley

Figure 6.4: Four different populations of the zombie apocalypse scenario. Each contains radically different resources available, yet the system is still able to create logical populations for each landscape.

Chapter 7

Evaluation

With our method, designers can populate a virtual world by designing districts, relations and settlement prototypes. Since this approach is rather involved, we wanted to test how easily people unfamiliar with the system could use it to create a population.

For the experiment, we asked 9 persons to do a number of tasks with our method. Our participants were all between 20 and 26, 5 were female, 4 were male and none of them has a background in computer science, although two have experience playing computer games. We briefly explained what the method does, how the designer can interact with the system and introduced the medieval population example shown in Figure 6.1. Furthermore, the participants were given a 'cheat sheet' that displayed the main options and available terrain features for the landscape they worked in. After this introduction, the participants were given definitions of the Fishing district as shown in Table 3.1, the Trade relation as shown in Table 3.2 and the Small prototype as shown in Table 3.3 to serve as a start for their design. Using this, they were asked to do the following tasks to incrementally improve their world.

1. Add an agricultural district that produces a resource 'food' from the terrain feature 'fertile'.

2. Create a world that has around 8 settlements that have the Agricultural district only, and around 8 that have the Fishing district only.

3. Create a large settlement that has surrounding villages.

4. Create small settlements that cannot produce food, but rather have to steal it from other settlements (raiders).

5. Make the raider settlement rely on 'metals' as well, and create a district that can generate this resource from mountains.

We were primarily interested in seeing how many times participants had to make a change to their design after getting results that differed from their expectations. After working with the system ourselves, we already noticed that trial-and-error is a common way of fine-tuning your world, and this is visible in the retries of participants for each trial shown above:

Task	1	2	3	4	5
Average retries	1.6	5.7	2.3	4.0	2.9

Task 2 proved especially challenging, as our landscape had a great amount of fertile land. Because of this, the chance of a settlement becoming agricultural was greater, and participants needed to utilize prototypes to make fishers appear more often. We required the participants to have 7 to 9 of each settlement type in the first generation, without user intervention, in order to complete this task. Similarly, task 4 required participants to define a new relation, and this was especially interesting because that boils down to using the more abstract resource 'manpower'. Four of our participants used similarly abstract resources ('soldiers', 'barbarians' and 'manpower'), while the rest used resources like 'weapons'.

After the tasks, we allowed our participants to freely try adding districts, relations and prototypes to the world, and asked them if they felt how well the method could help them to create a population as they wanted it. The responses were rather similar: Most participants found the method to be quite complex, but also rather intuitive once they understood the basics. In particular, 8 out of 9 participants mentioned 'trial-and-error', meeting our expectation, especially considering none of the participants had a background in computer science.

Chapter 8

Conclusion

In this thesis, we presented a method for the generation of large populations of virtual characters, with basic but intuitive relations between them. A designer is required to do the creative work, while the method itself simulates a believable world based on the designer's ideas. Even when the designers do not get exactly what they want, they can still strongly influence the outcome of the program. The method is quite fast, making it easy for the designer to experiment and try different approaches. After the designers are satisfied with their world, they can proceed to the generation of meaningful characters who have a real place in the world they were created in, and personality traits that are derived from their origin.

We applied our method to fundamentally different scenarios, proving that it can deal with a wide variety of worlds and populations, from medieval/fantasy themed worlds to wild west and apocalyptic settings.

By letting a small number of people without background in computer science test our method, we have seen that technical knowledge is no prerequisite for using our method. However, the method is not trivial either, and requires designers to understand a few basic concepts before they can start. The experience of our participants has shown that effectively creating a world requires some degree of trial-and-error, since the interaction between settlements, the world and the designer can become quite complex.

For future work, we will improve the interaction the designer has with the method. Right now, designers can edit their designs, then reload the program to see their changes take effect. Instead, being able to make changes to districts, relations and prototypes while running the program would make the interaction even more fluent. Our method is particularly well suited for this kind of editing, because it will simply keep optimizing based on its new

inputs.

The paper on our method was published in the Proceedings of the FDG workshop on Procedural Content Generation in Games [8].

Bibliography

[1] T. Adams. Slaves to Armok: God of Blood Chapter II: Dwarf Fortress. Bay 12 Games, August 2006.

[2] Bethesda Game Studios. The Elder Scrolls V: Skyrim. Bethesda Softworks, 2013.

[3] B. Desbenoit, E. Galin, and S. Akkouche. Simulating and modeling lichen growth. *Computer Graphics Forum*, 23(3):341–350, 2004.

[4] J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):216–228, Sept 2011.

[5] A. Emilien, A. Bernhardt, A. Peytavie, M.-P. Cani, and E. Galin. Procedural generation of villages on arbitrary terrains. *Visual Computer*, 28(6-8):809–818, June 2012.

[6] P. Felzenszwalb and D. Huttenlocher. Distance transforms of sampled functions. *Theory of Computing*, 8(19), September 2012.

[7] K. Hartsook, A. Zook, S. Das, and M. O. Riedl. Toward supporting stories with procedurally generated game worlds. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 297–304. IEEE, 2011.

[8] B. in het Veld, B. Kybartas, R. Bidarra, and J.-J. C. Meyer. Procedural generation of populations for storytelling. 2015.

[9] B. Kybartas and C. Verbrugge. Analysis of ReGEN as a graph-rewriting system for quest generation. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2):228–242, 2014.

[10] M. Lebowitz. Creating characters in a story-telling universe. *Poetics*, 13(3):171–194, 1984.

[11] M. Mateas. *Interactive Drama, Art, and Artificial Intelligence*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 2002.

[12] Maxis. SimCity. Electronic Arts, March 2013.

[13] J. McCoy, M. Treanor, B. Samuel, A. Reed, M. Mateas, and N. Wardrip-Fruin. Social story worlds with Comme il Faut. *Computational Intelligence and AI in Games, IEEE Transactions on*, 6(2):97–112, June 2014.

[14] Paradox Development Studio. Crusader Kings II. Paradox Interactive, February 2012.

[15] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra. Interactive creation of virtual worlds using procedural sketching. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–8. ACM, June 2010.

[16] J. Valls-Vargas, S. Ontanon, and J. Zhu. Towards story-based content generation: From plot-points to maps. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8, Aug 2013.

Appendices

```
float GetDominance(const int& global_x, const int& global_z, ChunkGroup& chunk_group, const int&
    chunk_sizem1) {

#define GRID_SAMPLES_SQRT 9
#define GRID_SIZE 450
#define HEIGHT_SCALE 10.0f // The height scale to base the dominance map on. This is the presumed max
    height difference

  const int grid_point_distance = GRID_SIZE / GRID_SAMPLES_SQRT;

  float values[GRID_SAMPLES_SQRT][GRID_SAMPLES_SQRT];

  for (int i = 0; i < GRID_SAMPLES_SQRT * GRID_SAMPLES_SQRT; ++i) {
    int _x = ((i % GRID_SAMPLES_SQRT) - GRID_SAMPLES_SQRT/2) * grid_point_distance;
    int _y = ((i / GRID_SAMPLES_SQRT) - GRID_SAMPLES_SQRT/2) * grid_point_distance;
    Chunk* chunk;
    int local_x;
    int local_y;
    chunk_group.GlobalToLocal(global_x + _x - 1, global_z + _y - 1, &chunk, local_x, local_y);
    values[i % GRID_SAMPLES_SQRT][i / GRID_SAMPLES_SQRT] = chunk->height_data()[local_x + chunk_sizem1
        * local_y];
  }
  float mid = values[4][4];
  float total = 0.0f;
  for (int i = 0; i < GRID_SAMPLES_SQRT * GRID_SAMPLES_SQRT; ++i)
    total += values[i % GRID_SAMPLES_SQRT][i / GRID_SAMPLES_SQRT] - mid;

  return std::min(1.0f, std::max(0.0f, 1.0f - (1.0f + total / (HEIGHT_SCALE * GRID_SAMPLES_SQRT *
      GRID_SAMPLES_SQRT)) / 2.0f));
}
```

Figure 1: The algorithm used to determine dominance values for SketchAWorld terrains. The amount of samples from the terrain is GRID_SAMPLES_SQRT * GRID_SAMPLES_SQRT.

```
void Generation::NextGeneration() {
  ++generation_number_;
  for (std::map<unsigned int, SettlementData>::iterator it = settlements_.begin(); it != settlements_.
      end(); ++it) {
    // Recalculate this settlement's fitness:
    it->second.location_fitness = it->second.settlement.GetLocationFitness();
    it->second.settlement_distance_fitness = GetSettlementProximityFitness(it->second.settlement.
        GetPosition(), settlements_, it->second.settlement.GetPrototype(), it->second.settlement.GetID
        ());
    it->second.settlement.SetTradingQuantities(GetBestRelationResources(it->first));
    it->second.resource_fitness = it->second.settlement.GetResourceFitness();
    it->second.relation_distance_fitness = GetSettlementRelationDistanceFitness(it->first);
    it->second.prototype_fitness = it->second.settlement.GetPrototypeFitness();
    // Create this settlement's offspring:
    std::vector<SettlementData> offspring;
    unsigned int offspring_size = settings_.offspring_size;
    offspring.reserve(offspring_size);
    // Add the original settlement to the offspring:
    offspring.push_back(it->second);

    for (unsigned int i = 0; i < offspring_size; ++i) {
      SettlementData child(it->second);
      child.settlement.Mutate();
      // 1 to 3 times max:
      if (rand() % 6 == 0)
        for (unsigned int i = rand() % 3; i < 3; ++i)
          switch(rand() % 3) {
          case 0:
            AddRandomRelation(child.settlement); break;
          case 1:
            RemoveRandomRelation(child.settlement); break;
          case 2:
            ChangeRandomRelation(child.settlement); break;
          }
      UpdateRelations(child.settlement);

      child.location_fitness = child.settlement.GetLocationFitness();
      child.settlement_distance_fitness = GetSettlementProximityFitness(child.settlement.GetPosition(),
          settlements_, child.settlement.GetPrototype(), child.settlement.GetID());
      child.settlement.SetTradingQuantities(GetBestRelationResources(it->first, &child.settlement));
      child.resource_fitness = child.settlement.GetResourceFitness();
      child.relation_distance_fitness = GetSettlementRelationDistanceFitness(it->first, &child.
          settlement);
      child.prototype_fitness = child.settlement.GetPrototypeFitness();

      offspring.push_back(child);
    }
    SettlementData const* winner = NULL;

    winner = PickBest(offspring);

    it->second = *winner;

    // Update all relations, since they might have all changed (position)
    for (std::map<unsigned int, SettlementData>::iterator it = settlements_.begin(); it != settlements_
        .end(); ++it)
      UpdateRelations(it->second.settlement);
  }
}
```

Figure 2: The algorithm used to proceed from a generation to the next generation.

```
float GetSettlementProximityFitness(Point2Df const& position, std::map<unsigned int, SettlementData>
    const& settlements, SettlementPrototype const& prototype, unsigned int exclude_id = -1) {
  float result = 1.0f;
  float best_master_penalty = 9999999.9f;
  bool use_penalty = false;

  for (std::map<unsigned int, SettlementData>::const_iterator it = settlements.cbegin(); it !=
      settlements.cend(); ++it) {
    if (exclude_id != -1 && exclude_id == it->first) continue; // Don't compare with self
    Settlement const& other_settlement = it->second.settlement;
    Point2Df pos1 = other_settlement.GetPosition();
    SocialType other_social_type = other_settlement.GetPrototype().social_type;
    float dx = pos1.x - position.x;
    float dy = pos1.y - position.y;
    float distance = std::sqrt(dx * dx + dy * dy);
    float min_distance = prototype.settlement_distance_min;
    float max_distance = 999999.9f;
    bool use_max_distance = false;
    if (exclude_id != -1) { // Some special situations change the min distance:
      Settlement const& this_settlement = settlements.at(exclude_id).settlement;
      switch (prototype.social_type) {
      case ST_Master:
        if (other_social_type == ST_Slave) // Other is slave
          min_distance = 0.0f; // No penalty for slaves being near
        else if (other_social_type == ST_Master)
          min_distance = prototype.master_territorium_max + other_settlement.GetPrototype().
              master_territorium_max;
        break;
      case ST_Slave:
        {
          if (other_social_type == ST_Master) { // The other is a master..
            SettlementPrototype master_prototype = settlements.at(it->first).settlement.GetPrototype();
            min_distance = master_prototype.master_territorium_min;
            max_distance = master_prototype.master_territorium_max;
            use_max_distance = true;
          } else if (other_social_type == ST_Slave) { // Both slave, same master
            // Right now this can't work (since we can't determine the master of this settlement)

            //min_distance = master_prototype.master_slave_distance_min;
          }
        }
        break;
      }
    }
    if (distance < min_distance) {
      float fitness = distance / min_distance;
      if (fitness < result) result = fitness;
    }
    if (use_max_distance)
      if (distance > max_distance) {
        float penalty = (distance / max_distance) - 1.0f;
        if (penalty < best_master_penalty) {
          best_master_penalty = penalty;
          use_penalty = true;
        }
      } else {
        // distance <= max_distance
        best_master_penalty = 0.0f;
        use_penalty = true;
      }
  }
  if (use_penalty)
    result -= best_master_penalty;

  return result;
}
```

Figure 3: The algorithm used to determine a settlement's proximity fitness.

```
district Fishing {
  needs {
    food
  }
  produces {
    food {
      distance_open(3.0, water, 30.0, 150.0)
    }
  }
  relations {
    Trade
  }
  image(icons/fishing64.bmp)
}
district Military {
  needs {
    food
    metals
  }
  produces {
    military_supplies {
      distance_open(8.0, domination, 0.0, 0.3)
      constant(1.0)
    }
  }
  relations {
    Raid
    Protect
  }
  image(icons/military64.bmp)
}
district Mining {
  needs {
    food
  }
  produces {
    stone {
      distance_open(2.0, mountains, 0.0, 200.0)
    }
    metals {
      distance_open(1.0, mountains, 0.0, 200.0)
    }
  }
  relations {
    Trade
  }
  image(icons/mining64.bmp)
}
```

Figure 4: The districts.dat file, containing the user-defined district definitions.