Example Based Procedural Distribution Tool

Anders Nord

A thesis presented for the degree of
Master of Science (M.Sc.)
Media Technology and Computer Engineering

Supervisor: Stefan Gustavson
Examiner: Patric Ljung

Deparment of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden
2014 - 09 - 21

Thanks

I would like to thank EA Frostbite for this opportunity and the Frostbite team for all the help provided. I would also like to thank my supervisor at Frostbite, Björn Ottosson, for good advice and guidance. A lot of the ideas in this thesis arose during our discussions.

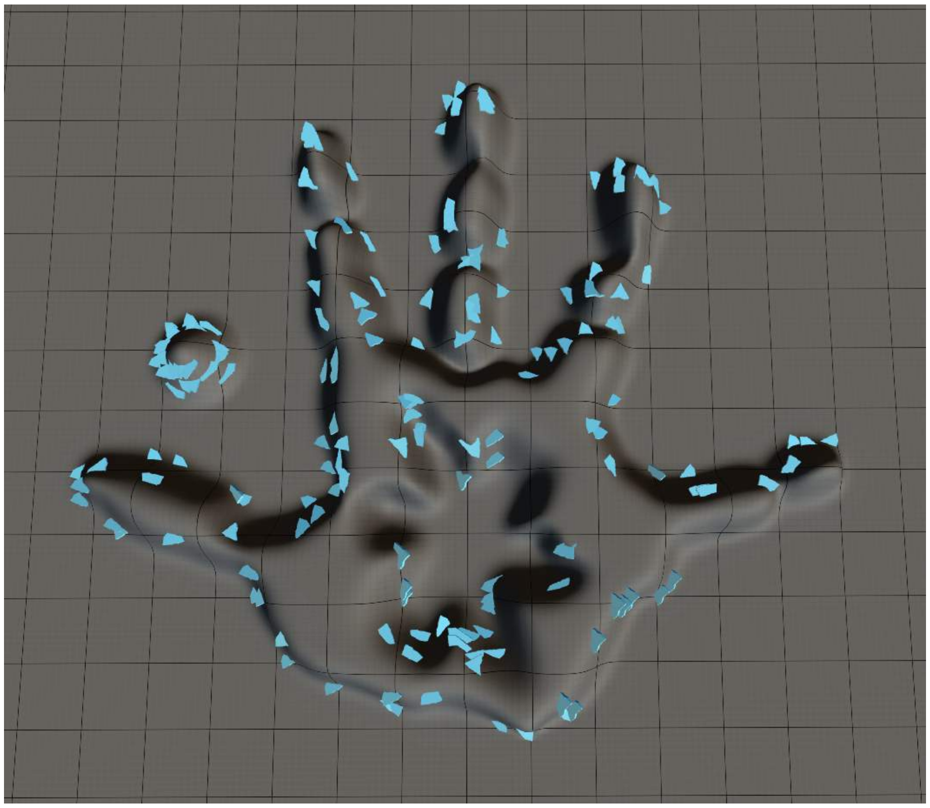


Figure 1: Frostbite logo inside FrostEd. Shards placed with the implemented tool.

Abstract

This report will deal with the process of creating an example based procedural distribution tool. This is accomplished within the Frostbite game engine editor, FrostEd. The design of the tool is based on artist's desires and wishes. By using actual placements of objects in the editor as in-data, the tool provides the artist with an unmatched visual feel for calibrating its properties and settings. Note that this is a unique technique and was invented during the creation of this tool. The tool is based on a machine learning approach. It creates a feature vector from the example placements for each type of object. These vectors are then used to create statistical models which in turn are used to generate new object placements. The process of determining the position and rotation when generating an object is divided into two parts. A new concept called Feature Function (FF) is utilized to provide each element in the population with a probability to obtain a certain position and rotation.

For position dart throwing makes a first weed out. Simulated annealing helps avoiding getting stuck in local maximas. And finally the Kolmogorov Smirnov Normality Test evaluates the probability for each sample that the simulated annealing has provided. For the second part multivariate distributions evaluate how the object is rotated according to certain predefined vectors, such as the surface normal and the world-up vector. The results were satisfying and proved that this new method works. The implemented tool had a distribution time that scaled badly, but there are explanations to why and how to solve this issue in the report.

Acronyms

2D	Two-Dimensional. 4, 14, 15
3D	Three-Dimensional. 9, 14
CDF	Cumulative Distribution Function. 13
CV	Coefficient of Variation. 31, 54
ECDF	Empirical Cumulative Distribution Function. 13
FF	Feature Function. i, 7, 8, 17–19, 22–24, 31, 33, 41, 42, 44, 51, 52
FPF	Feature Placing Function. 8, 9, 11, 13, 18, 20, 22, 23, 32, 40, 46, 50, 52
FRF	Feature Rotation Function. 9, 18, 20, 27–30, 41, 48
GUI	Graphical User Interface. 19
HD	High-Definition. 1
KS	Kolmogorov Smirnov. 13, 18, 32, 33, 46–48, 52
MPDF	Multivariate Probability Density Function. 9, 10, 52
SA	Simulated Annealing. 11–13, 18, 31, 32, 44, 45, 51, 52
SI	International System of Units. 13, 31
STD	Standard Deviation. 8, 12, 33, 42, 44, 46–48, 50, 51
UPDF	Univariate Probability Density Function. 8, 9, 12, 13, 21

Contents

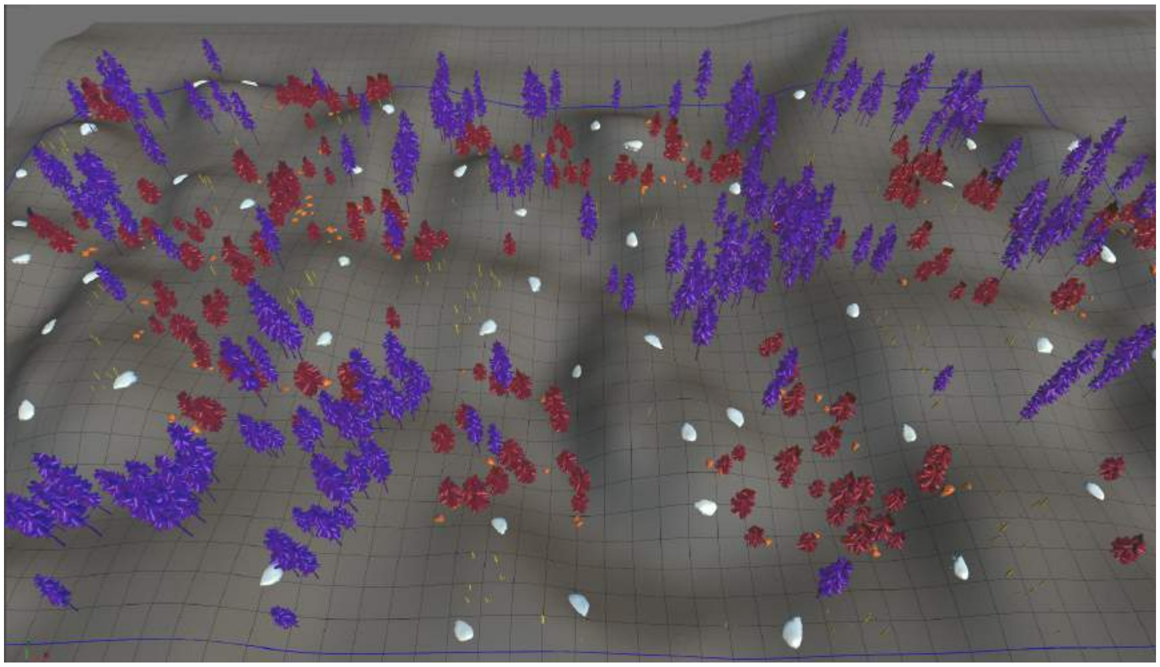


Figure 2: A forest generated by the final implementation of the tool. The feeling and composition created by the different objects are based on the example area seen in figure 19, which is created by an artist.

Part I

Introduction

Game worlds are increasingly getting bigger, which means that more High-Definition (HD) content needs to be created. Hiring more artists is not really an option since there is today already a couple of hundred people working on a AAA-game (pronounced "triple A"). AAA is the classification term used for describing high quality games with a large budget, the equivalent in the film industry would be a blockbuster. By implementing artist friendly procedural tools, the game worlds can be created in a more efficient and quicker way without losing quality.

The goal of this thesis was to create an artist friendly example based procedural distribution tool. By using sample placements, there will be an increase in the amount of productivity and control that the artist can achieve. The artist will be able to focus on the things that are important, rather than creating similar areas over and over. To quote Ruben M. Smelik *et al.*[1]: "A simple set of input parameters or a few generation rules of the procedural model yield a wide variety of models". This leads to the questions:

- How can procedural methods be applied when creating a useful procedural distribution tool which has the desired level of artistic control?

- How should a tool like that be designed?

The thesis has been delimited by criteria based on interviews carried out with artists. The main focus is environments that can somehow be described by certain abilities. Like trees getting clustered in nature, or trash getting collected close to the sidewalk. The artist should be able to have good control over the objects getting created. They should also be able to remove the things they do not like and get direct feedback to make iteration easy. So a lot of earlier research in procedural content generation does not work very well for this type of tool and workflow.

Part II

Background

There are a lot of different methods for creating game worlds procedurally. This thesis started out with a very general direction. Many different areas were examined and evaluated accordingly to what tools already existed inside FrostEd and to what seemed exploratory. Three artists were interviewed about the chosen area, procedural object distribution. The questions dealt with how they felt about the distribution tools already existing inside FrostEd today and what was requested from the new tool. These interviews concluded that:

- Tools for placing smaller non colliding objects like ferns and gravel-rocks works well and does what is desired.
- The object distribution tools that exists does not create realistic looking distributions or areas when placing objects. They do not follow any specific pattern and that makes them hard to use and predict.

And important aspects in general:

- Be able to iterate quickly to create a good workflow.
- Rotations often has to be altered later when procedurally distributed.
- Be able to align objects against the surface.
- In environments things are often clustered naturally.
- Key features of the environment are: Corners and slopes, material on the ground and topography in general.
- A function for "freezing" certain areas and redistribute the non-frozen areas.
- A high-quality foundation since there is often not enough time to place as many objects as desired.
- Have presets since it sometimes can be hard to understand a tool with many input parameters.

Part III

Theory

As an introduction to get a general overview of procedural creation see [1]. For an introduction to the different types of procedural methods and to the importance of artistic control, see [2]. There were no actual papers about example based procedural tools found that seemed appropriate for this task. But many of the methods involved to achieve the final result has been researched individually and have papers dedicated to their solutions.

1 Sampling

To be able to compare different element placements, their positions are sampled in a discrete Two-Dimensional (2D) space. Together these elements create a population. There are many different ways of sampling an arbitrary amount of elements. The methods researched during this thesis were random sampling, jitter grid sampling and Poisson disk sampling.

1.1 Random sampling

Random sampling, also known as stochastic sampling, samples elements with random positions. It is easy to implement but creates some unnatural clumping due to the nature of choosing random numbers. This can be observed in figure 3. For further reading see [4] and appendix A.1 for source code.

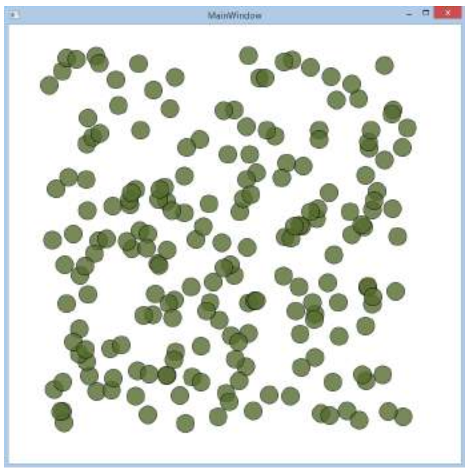


Figure 3: 200 elements uniformly sampled randomly. This method is computationally cheap but ends up clumping the elements together if no bounding conditions are set.

1.2 Poisson disk sampling

Poisson disk sampling samples evenly distributed elements with a random looking feel. This is achieved by sampling new elements around already added elements with a minimum distance condition. This method is however computationally expensive and therefore time consuming. Depending on the application, this might be more or less justifiable. The elements in figure 4 have been sampled using Poisson disk sampling. See [5] for further reading and code implementation.

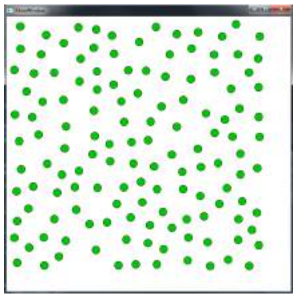


Figure 4: Elements sampled with Poisson disk sampling. This method solves the clumping problem seen in figure 3 but is computationally more expensive.

1.3 Jitter grid sampling

The grid will distribute the elements evenly, as shown in figure 5.a. To get a random looking feel, the elements positions are displaced from the grid, as shown in figure 5.b. They are both easy to implement and provide good performance. For further reading containing a more detailed explanation, see [4]. See appendix A.2 for source code.

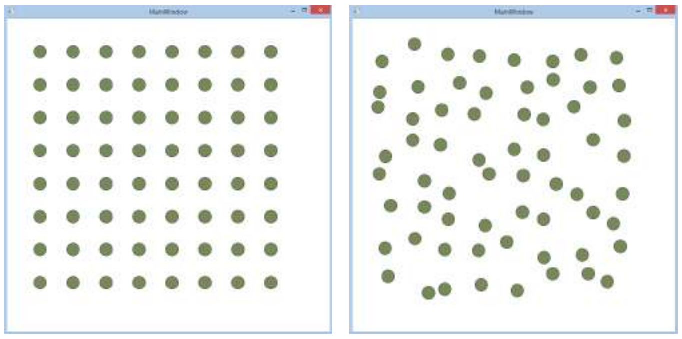


Figure 5.a Figure 5.b

Figure 5: 5.a: Elements sampled using grid sampling. 5.b: Elements sampled using jitter grid sampling. Just by moving the elements a small distance a random feeling is created at a much lower cost than by using poisson disk sampling.

2 Final sample

From the sampled elements representing the total population, only a few will be in the final sample. This is achieved by using teleological or ontogenetic methods. A teleological method is an accurate physical model of the environment, e.g. an eco-system. An ontogenetic method is an attempt to reproduce natural results with arbitrary algorithms.

The distance between sampled elements can be utilized to avoid overlapping by re-sample them until a shortest distance condition is met. Compare figure 6 and figure 3 and visit [4] for a similar approach. There are other ways to achieve the same result by using 3D-meshes. In that case bounding boxes, bounding spheres, ray-tracing or vertex-intersection are a few of the methods that can be used for collision detection.

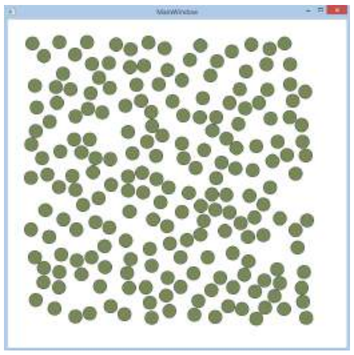


Figure 6: 200 elements uniformly sampled randomly with a radius-distance collision condition. Avoiding elements intersecting is often desired to avoid 3D-meshes from getting too close to each other.

2.1 ECO - Teleological

When sampling different species, every sampled element has a circle radius. Biologically motivated rules make use of the interaction between the intersecting circles. In [8] the rules are defined as: If the circles representing two plants intersect, the smaller plant dies and its corresponding circle is removed from the scene. Plants that have reached a limit size are considered old and die as well. For further reading visit [8] and [9].

2.2 Density function - Ontogenetic

By letting a continuous function affect the density, irregularity or regularity can be obtained. These type of functions are however hard to control. See figure 7 and figure 8 for visual results using density functions.

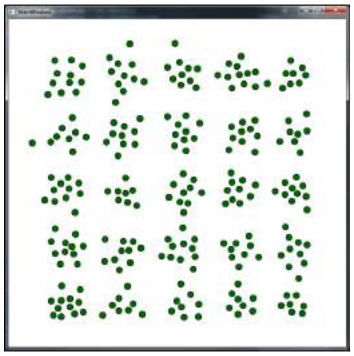


Figure 7: A density function consisting of sin and cos elements. This shows that it is possible to control the density of the elements but the feeling of randomly generated positions is lost.

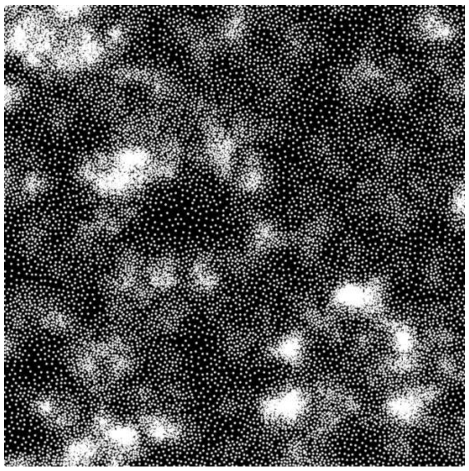


Figure 8: Poisson disk sampling, where the minimum distance is driven by Perlin noise. The clumping of elements is great when placing grass and many other type of objects. Picture taken from [5].

2.3 Probability sampling - Ontogenetic

Each element is assigned an individual probability value. This is achieved by letting them get evaluated by a FF. Briefly the idea is to let each element in the current sample get evaluated by a FF that returns a value according to a specified task. In figure 9 a FF that returns the distance to the closest element in the current sample has been used. See 3 for a more detailed explanation.

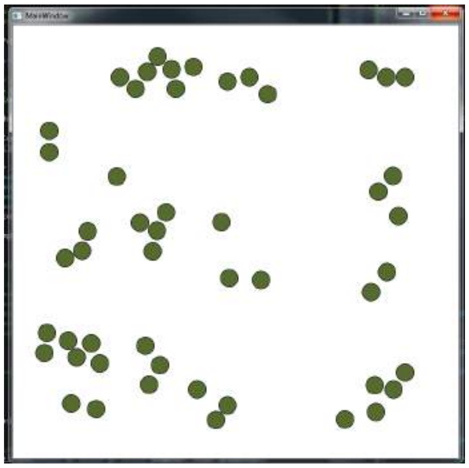


Figure 9: The final sample of elements that has used a FF that returns the distance to the closest element in the current sample. This FF replaces the need for having a radius-distance collision condition.

3 Feature function

A Feature Function (FF) is a short function that evaluates each element in a sample separately and returns an individual value accordingly. This individual value corresponds to what the FF takes into account. Note that this is a unique technique created for this tool. In practice it is possible to let the FF calculate a value based on whatever is desired.

3.1 Feature placing function

The Feature Placing Function (FPF) uses the position of the element in various ways to determine a return value. In figure 10 the stone-elements (grey circles) have used a FPF that returns the distance to the closest tree-element (green circles). This is the individual value used to evaluate how likely the element is to end up in the final sample. The evaluation uses a Univariate Probability Density Function (UPDF). The mean and Standard Deviation (STD) are set by the artist or according to example element placements. E.g. the mean distance to trees and the estimated differential in the earlier example.

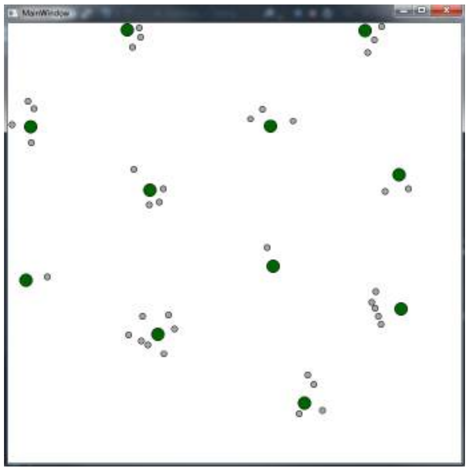


Figure 10: A final sample of stones (grey circles) that has used a FPF which returns the distance to the closest tree (green circles). The visualization creates a clear understanding of how the positions of the stones are related to the positions of the trees.

3.2 Feature rotation function

The Feature Rotation Function (FRF) utilizes arbitrary Three-Dimensional (3D)-vectors represented in world space to describe the rotation of an object. The vectors are transformed into the local space of the examined object. This local representation can contribute to a general perception of how the object is rotated compared to its X, Y and Z axes.

In figure 11 the red arrow is represented in two different coordinate systems. In world space, figure 11.a, it is parallel with the objects green Y-axis-arrow. In 11.b it is represented in the objects local space with different X, Y and Z values. So the basic idea of a FRF is similar to the one of a FPF. But instead of returning a single value to evaluate, it returns a 3D-vector. To evaluate this vector, a Multivariate Probability Density Function (MPDF) is used instead of a UPDF.

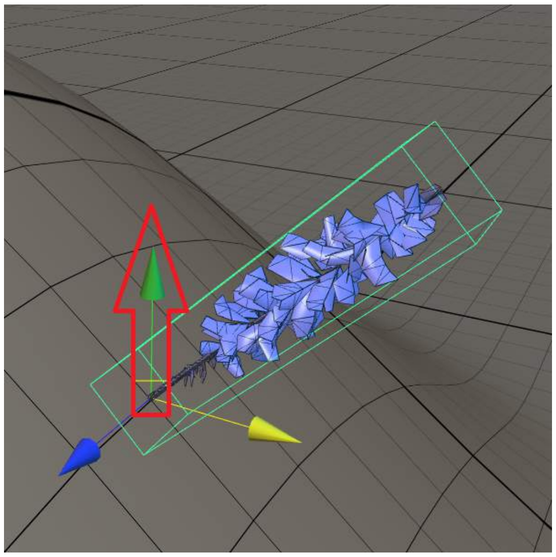


Figure 11.a

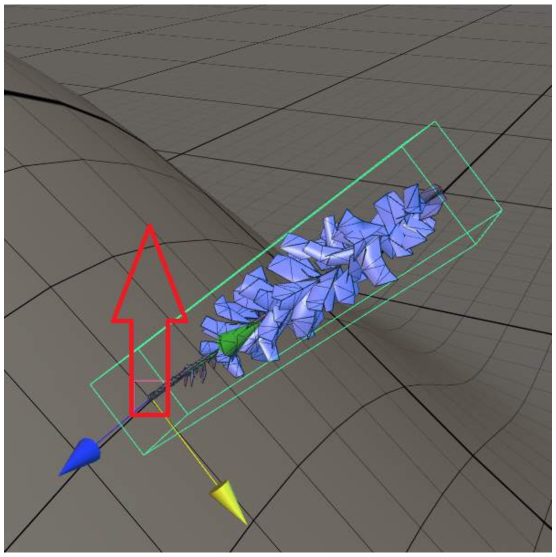


Figure 11.b

Figure 11: 11.a: World space coordinate system. 11.b: Local space coordinate system. The red arrow represents the world space up vector.

3.2.1 Quaternions

To be able to find rotations for the elements in the final sample that corresponds to the earlier apprehended local vectors, random unit quaternions are utilized. The identified world space vectors are transformed into the local space of the quaternion. Then the MPDF, created from the mean of the examined objects local vectors, can return individual probability values for each quaternion. This value represents how well it compared with the vectors represented by the MPDF. See appendix A.5 for source code.

4 Search heuristics

When examining the different samples and their element distributions, the goal is to find the optimal global distribution. It is important to realize that there will be a large search space if thousands of elements are going to be evaluated, whilst maybe only a few of them will be in the final sample.

4.1 Hill climbing

The hill climbing algorithm starts out with an arbitrary solution to the problem. It then tries to find a better solution by iteratively stepping forward. If a better solution is found, it will be the new best solution. This is repeated until no better solution can be found. The problem here is that the hill climbing algorithm is a local search algorithm, meaning it will get stuck in local optimums. Depending on the initial state it may or may not find the best solution. In figure 12, the hill climber will get stuck in the local maximum.

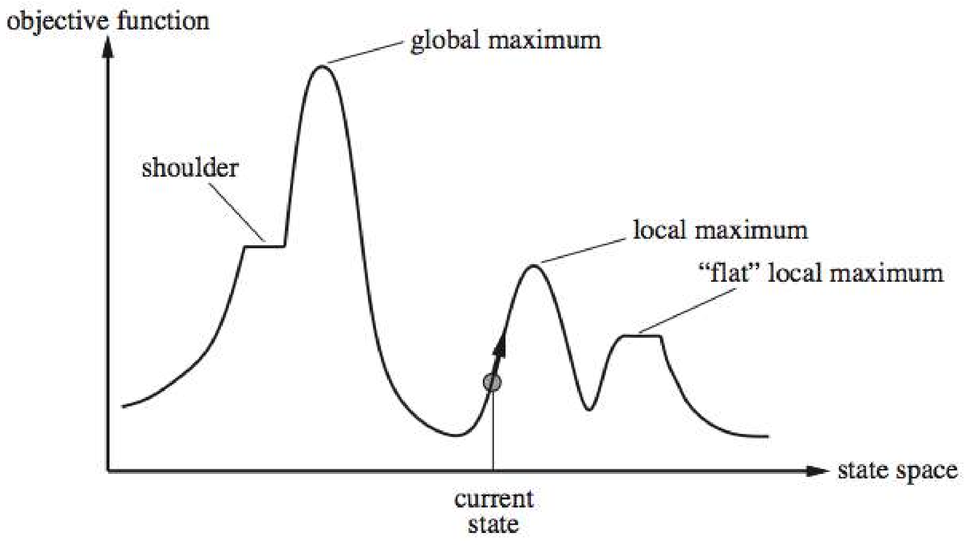


Figure 12: Hill climber algorithm getting stuck in a local maximum. Each FPF used would increase the probability of getting stuck in a local maximum.

4.2 Simulated annealing

Simulated Annealing (SA) uses annealing in metallurgy as a metaphor to describe how it works. When annealing metal, the metal is exposed to heat above its critical temperature which makes it possible to alter its physical properties. The metal solidifies as it cools and adapts to its new form. This cooling process is simulated by a temperature variable. As the algorithm runs, the temperature is slowly decreasing. The effect is that the higher the temperature, the bigger the jumps in the search space. The algorithm is also likelier to accept a worse solution than the current one. The gist from this is that local maxima's and minima's can be avoided. This can be observed in figure 13, where the SA jumps from 2 to 3 and from 4 to 5, even though they are worse solutions than the current ones. SA can contribute to find an acceptably good solution within a shorter amount of time than an exhaustive enumeration would. But it might not find the best possible solution.

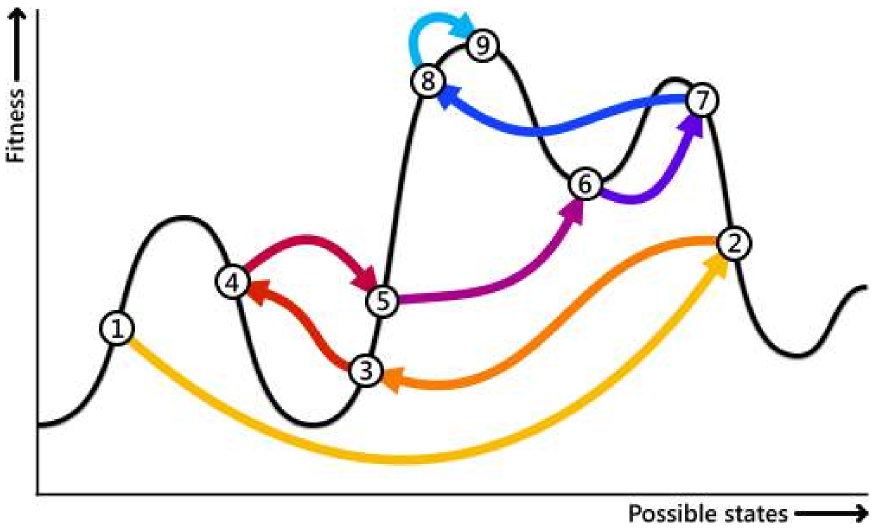


Figure 13: SA jump behavior. This is a computationally expensive way to find a good solution, but badly needed for all the fall pits in the global space.

SA overview pseudocode:

1. Set the initial temperature and solution
2. Start the loop that iterates on the algorithm. This loop will stop when either the temperature is too low or when a good enough solution has been found.
3. Make a change to the current solution, small or big depending on the temperature.
4. Decide whether or not to move to the new state by using an acceptance function.
5. Decrease the temperature and continue looping.

For implementation details see [10] and appendix A.4.2.

5 Test of normality

To be able to know how well the current element distribution fits the UPDF a test is performed. This is called a test of normality.

5.1 Arbitrary test

To compare the observed values against the UPDF in a simple manner, the mean and STD values are utilized as in equation 1.

$$f(x) = (pdf_{Mean} - observed_{Mean})^2 + (pdf_{Std} - observed_{Std})^2 \tag{1}$$

This method does not work well if different International System of Units (SI) are used by the different FPFs. It also fails to know whether or not the elements are evenly distributed accordingly to the UPDF. Also only works well with a normal UPDF.

5.2 Kolmogorov smirnov

The Kolmogorov Smirnov (KS)-test calculates the biggest difference between a distribution evaluated by the Empirical Cumulative Distribution Function (ECDF), seen in equation 2 where I is the indicator function, and the Cumulative Distribution Function (CDF), seen in equation 3.

$$\hat{F_n}(t) = \frac{\text{number of elements} \leq t}{n} = \frac{1}{n}\sum_{i=1}^{n} I(x_i \leq t) \tag{2}$$

$$F(x) = \int_{-\infty}^{x} f(u)du. \tag{3}$$

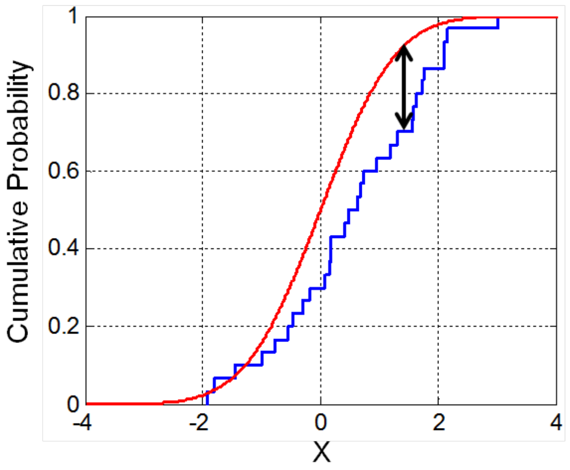


Figure 14: The red line is the CDF, the blue line is the ECDF, and the black arrow is where they differ the most, the KS value. By looking at how similar they are, it is easy to see how well the SA performed.

Figure 14 visualizes the ECDF, the CDF and where they differ the most. The difference is the KS value. The KS-test is independent of what type of SI-unit and magnitude the evaluated distribution is using since the CDF always returns a value between 0.0 and 1.0. It also tries to make sure that the elements are distributed according to the UPDF. For more specific details see [11].

6 Artistic control

One of the most important things when creating tools is the artist's workflow and experience when using it. This is a complicated problem when using procedural content creation. To be able to control and apprehend where the final element distribution will end up is not an easy task. The result is often shaped by many stages.

6.1 Sampling area

The sampling area determines where the elements are sampled. This area is used as a high level description controlled by the user.

To find the objects height positions in a 3D space, each position is raytraced from above. This is due to that the position of the element is originally defined in the 2D plane. The density affects the result in a very visual way. It is either defined separately, as in equation 4, or combined with a high level description.

$$Density = \frac{\text{number of objects}}{\text{sampling area}} \tag{4}$$

See [13] for more high level description techniques and example based workflows.

6.1.1 Square

A square that represents where the elements will get sampled. See figure 15.

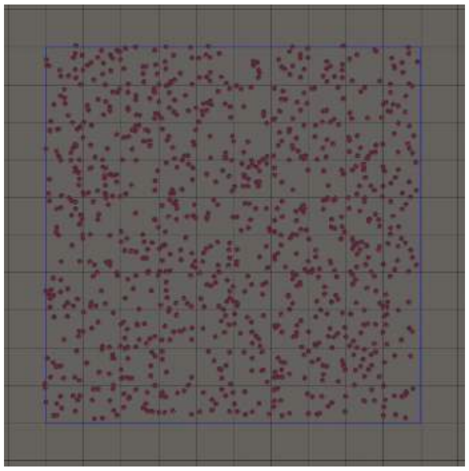


Figure 15: The blue square indicates the sampling area. The red objects are generated from elements.

6.1.2 A volume

When the sampling area is represented by a volume, there are some neat benefits. The raytracing mentioned in 6.1 can start out from the "roof" of the volume. This way objects can get generated inside caves and such. The shape of the volume can also be altered. To know how you are inside the volume visit [12]. They implement a check to see if a point is inside a polygon in 2D.

6.1.3 Masks

The mask is painted on top of a terrain or object where the artist wants to generate new objects. Masks are good at giving an indication about density and placing. See figure 16.

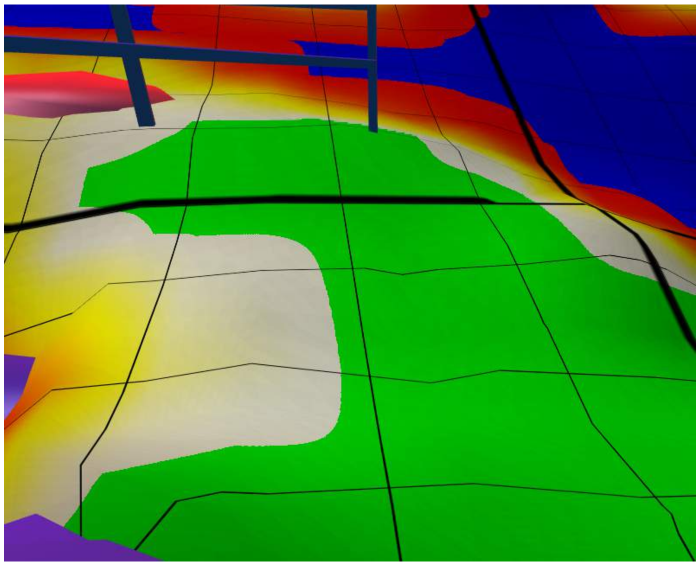


Figure 16: The color of the mask indicates the magnitude of density. Green indicates high density, red indicates low density and blue indicates zero density. This is a very intuitive and visual way of describing where the elements will end up.

6.2 Iteration and workflow

To be able to iterate when procedurally creating big areas, it is desirable to be able to control what stays and what should be re-generated.

6.2.1 Seed

Normally the pseudorandom number generator uses a random seed. But the seed can be predetermined and this would have the algorithm generate numbers in the same sequence every time. This would ensure the elements to turn up on the same location each time, even on different machines.

6.2.2 Freeze

This effect can be created by freezing the objects that you want to keep and then re-generate the ones that you don't want to keep. See 8.5 in the method chapter for pictures and workflow.

Part IV

Method

7 Preliminary study

For a background read part II. The main idea at this point was to focus on the usability and the workflow of the tool. This turned out to be difficult because of a plurality of factors that are mentioned in section 9.

An early decision was to ignore elements overlapping due to the observation studies performed on maps that shipped with battlefield 4. A lot of objects were tucked into each other, which reads into the need for elements to be able to be placed on top of each other with mesh intersection as a result. Another observation from the maps was that objects tended to be located near corners and slopes. Also certain rotations like stones rotated away from bigger stones and a strong dependency of the normal and world up vector were noticed.

8 Implementation

8.1 Overview

The final tool is based on a stratified sampling technique [3]. It divides the different types of objects into separate groups referred to as strata's. The strata's are then generated in a predefined order set by the user. This causes the strata's to have different dependency-possibilities between each other. For example: If trees and stones are two strata's and the trees are sampled first, they can only depend on themselves and the environment. The stones are sampled second and can depend on themselves, the environment and the trees. This can be observed in figure 17 where the trees cannot depend on the positions of the stones. The sampling-order-dependency has a great impact on how the FFs are structured.

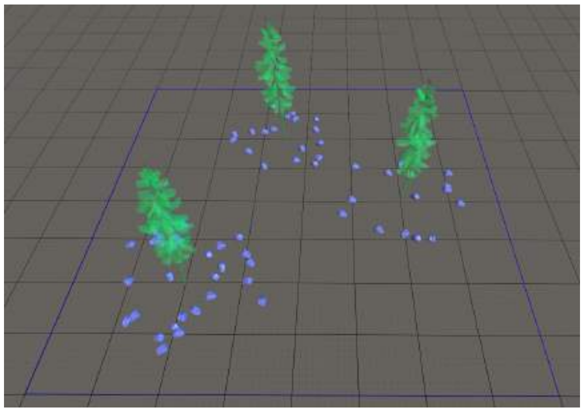


Figure 17: The trees were sampled first, only depending on the distance to each other. Secondly the stones were sampled, only depending on the distance to the trees.

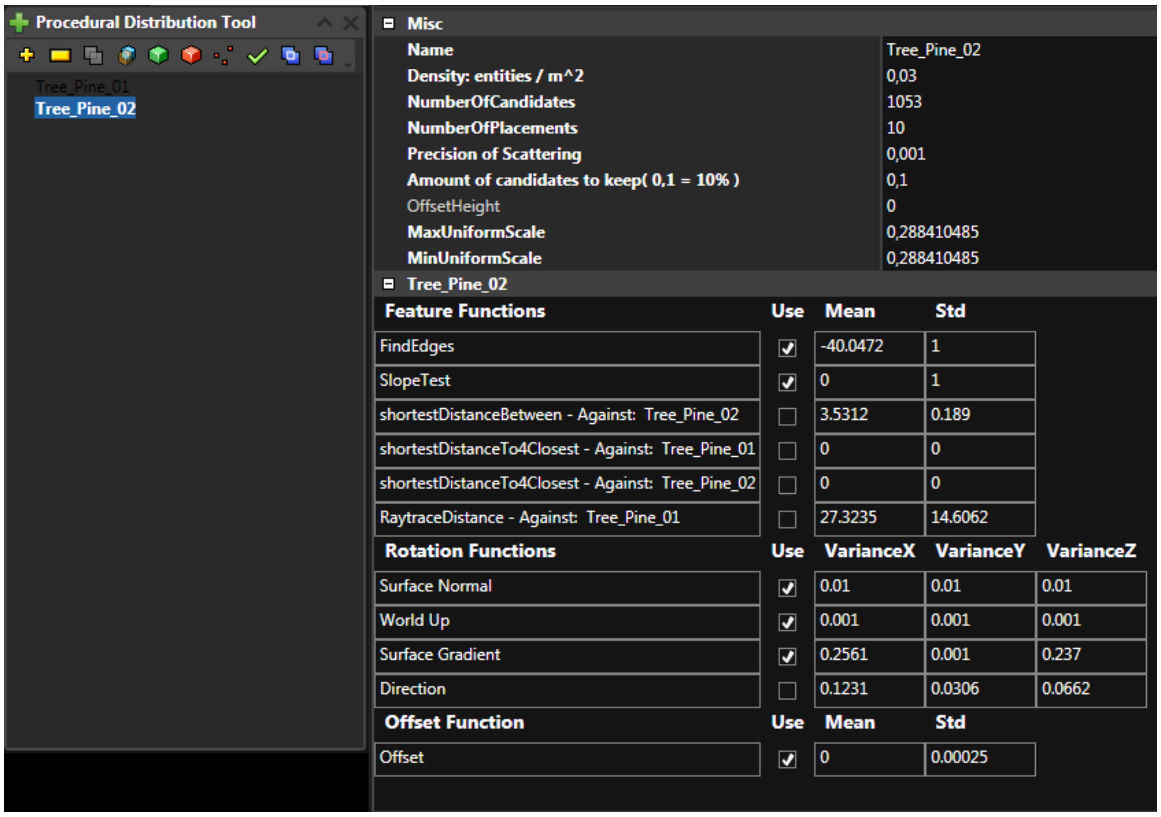
FFs provides the possibility to analyze example placements in a natural way. A clear advantage over the other final-sample-deciding methods. Normal univariate 5 and normal multivariate distributions are the only distributions implemented. The normal distribution was used because of two reasons: It is a distribution pattern occurring in many natural phenomena's but also because of limited development time. See the 11.2 section for a discussion regarding this subject.

$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{5}$$

8.2 Execution order

The elements are randomly sampled and sent through the FPFs where they obtain their initial individual probability values. When using more than one FPF, their probability values are multiplied together into a total probability value. These total probability values are then being normalized. Before entering the SA loop there is a weed out of the elements. This weed out is performed by a dart throwing algorithm explained in section 8.4.1. This decrease in the element population affects SA and KS performance and results. SA combined with KS is utilized to find a suitable final distribution of the elements. The FRF will then decide the rotation of the final elements.

8.3 Graphical User Interface (GUI)



Misc	
Name	Tree_Pine_02
Density: entities / m^2	0,03
NumberOfCandidates	1053
NumberOfPlacements	10
Precision of Scattering	0,001
Amount of candidates to keep(0,1 = 10%)	0,1
OffsetHeight	0
MaxUniformScale	0,288410485
MinUniformScale	0,288410485

Tree_Pine_02

Feature Functions	Use	Mean	Std
FindEdges	✓	-40.0472	1
SlopeTest	✓	0	1
shortestDistanceBetween - Against: Tree_Pine_02	☐	3.5312	0.189
shortestDistanceTo4Closest - Against: Tree_Pine_01	☐	0	0
shortestDistanceTo4Closest - Against: Tree_Pine_02	☐	0	0
RaytraceDistance - Against: Tree_Pine_01	☐	27.3235	14.6062

Rotation Functions	Use	VarianceX	VarianceY	VarianceZ
Surface Normal	✓	0.01	0.01	0.01
World Up	✓	0.001	0.001	0.001
Surface Gradient	✓	0.2561	0.001	0.237
Direction	☐	0.1231	0.0306	0.0662

Offset Function	Use	Mean	Std
Offset	✓	0	0.00025

Figure 18: The context pad on the left shows the different object-strata's in the order they will be sampled. The property grid on the right shows the different settings available for the selected strata. The goal with the tool is that the artist should never have to worry about or change these values.

The implemented GUI can be seen in figure 18. Notice how the second object contains FFs that depends on the object that gets sampled first. E.g. RaytraceDistance - Against: Tree_Pine_01.

The workflow:

- 19: Example object placements
- 20.a: Select the objects and analyze them
- 21.a: Mark the sampling area and start sampling
- 21.b: The result

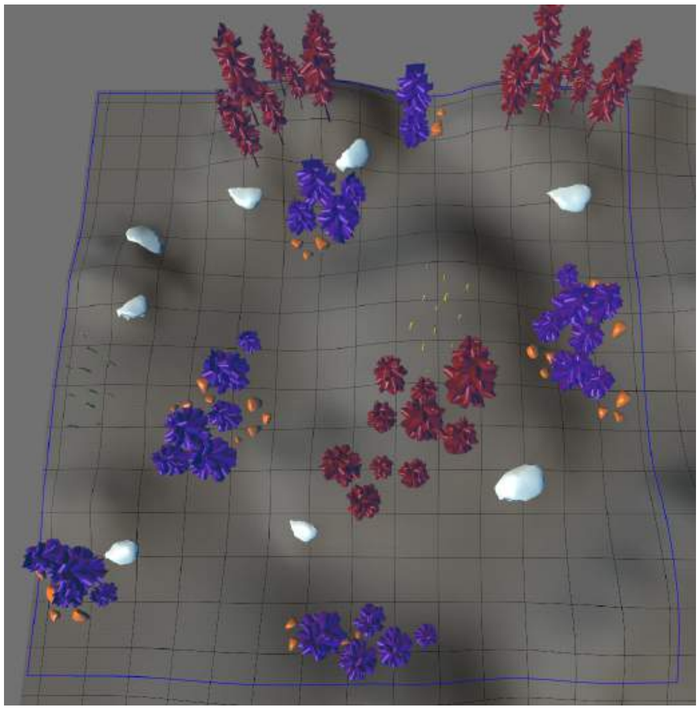
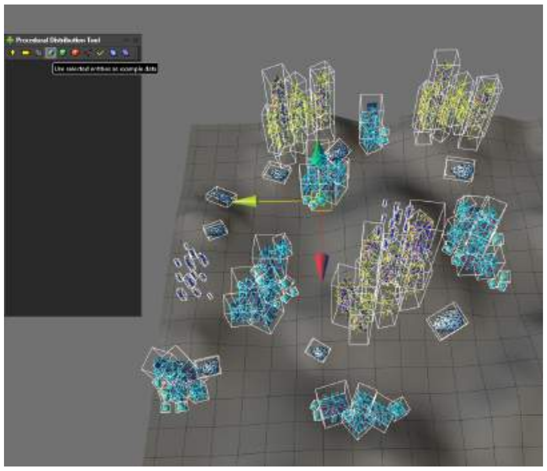
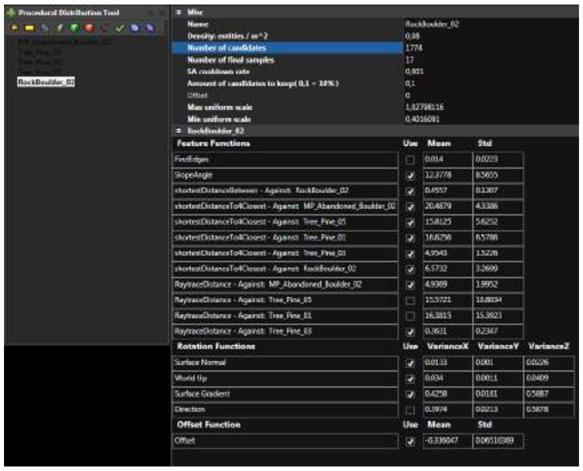


Figure 19: The example placements. This is all that the artist have to create to control how the resulting distribution ends up looking.


Procedural Distribution Tool
Use selected entities as example data



Misc

Name	RockBoulder_02
Density: entities / m^2	0,08
Number of candidates	1774
Number of final samples	17
SA cooldown rate	0,001
Amount of candidates to kept(0.1 = 10%)	0,1
Offset	0
Max uniform scale	1,82798116
Min uniform scale	0,4016091

RockBoulder_02

Feature Functions	Use	Mean	Std
FindEdges	☐	0.014	0.0223
SlopeAngle	☑	12.3778	8.5655
shortestDistanceBetween - Against: RockBoulder_02	☑	0.4557	0.1307
shortestDistanceTo4Closest - Against: MP_Abandoned_Boulder_02	☑	20.4879	4.3386
shortestDistanceTo4Closest - Against: Tree_Pine_05	☑	15.8125	5.6252
shortestDistanceTo4Closest - Against: Tree_Pine_01	☑	16.6256	6.5786
shortestDistanceTo4Closest - Against: Tree_Pine_03	☑	4.9543	1.5226
shortestDistanceTo4Closest - Against: RockBoulder_02	☑	6.5732	3.2699
RaytraceDistance - Against: MP_Abandoned_Boulder_02	☑	4.9369	1.9952
RaytraceDistance - Against: Tree_Pine_05	☐	15.5721	10.8034
RaytraceDistance - Against: Tree_Pine_01	☐	16.3815	15.3923
RaytraceDistance - Against: Tree_Pine_03	☑	0.3631	0.2347

Rotation Functions	Use	VarianceX	VarianceY	VarianceZ
Surface Normal	☑	0.0133	0.001	0.0226
World Up	☑	0.034	0.0011	0.0409
Surface Gradient	☑	0.4258	0.0181	0.5887
Direction	☐	0.3974	0.0213	0.5878

Offset Function	Use	Mean	Std
Offset	☑	-0.336047	0.06510369

Figure 20.a

Figure 20.b

Figure 20: 20.a: Analyze the selected example placements. 20.b: The resulting context pad and property grid. By selecting the objects in the marked area and press analyze, the tool will calibrate for all the FPFs and FRFs needed to recreate a similar area.

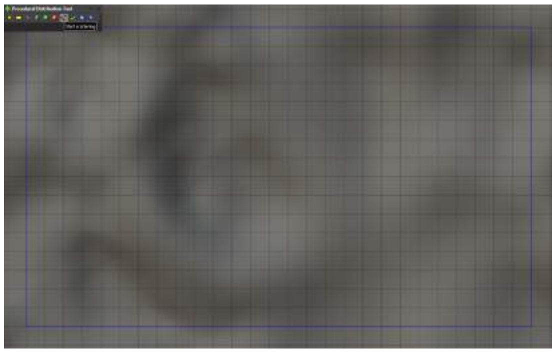


Figure 21.a

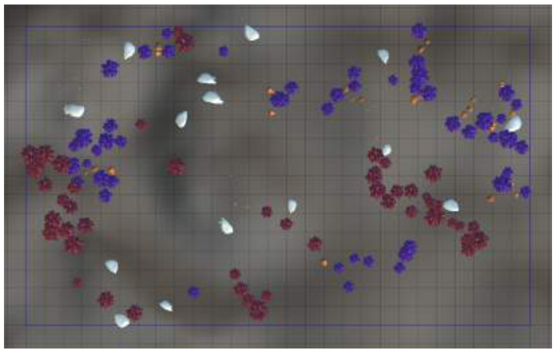


Figure 21.b

Figure 21: 21.a: Mark the distribution area and start scattering. 21.b: The resulting distribution.

8.4 Technical implementation

8.4.1 Dart throwing

This dart throwing algorithm was invented for this tool and is utilized to weed out elements from the population. The algorithm is most likely to pick elements with a high probability, but can also happen to pick elements with a low probability. This is essential so that later, the population will be able satisfy the UPDF.

The idea is to generate a random number between zero and the sum of all the individual probability values. These values are added together into a total value, one at a time. When the total value is larger or equal to the random value, the current element-index is returned. The result ends up being that indices for high probability elements are most often returned, but not always. See figure 22 for an example and appendix A: A.3 for source code.

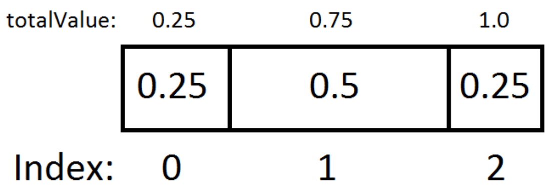


Figure 22: Illustrates an array containing 3 individual values. For random value = 0.7, index 1 would have been returned.

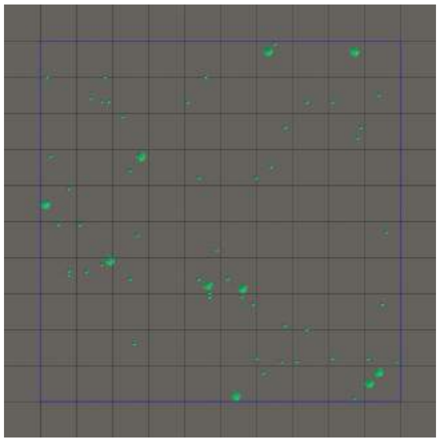


Figure 23.a

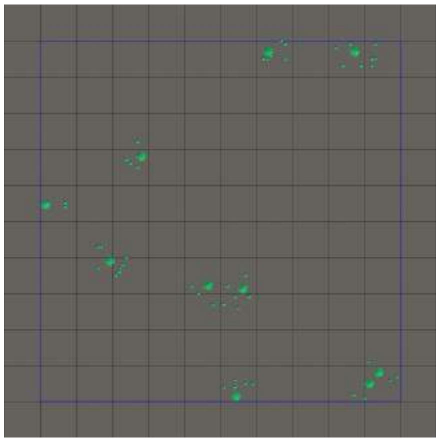


Figure 23.b

Figure 23: 23.a: No dart throwing used. 23.b: Dart throwing used only. The smallest green dots obtained probability values from a "distance to the closest tree" FF and were then evaluated by dart throwing only.

8.4.2 Feature Function

Since this technique was invented for this tool, so was all of the implemented FFs. They reflect the specific desires of this tool, which is good to bear in mind.

Placing functions
There are two types of placing functions. The ones that can take the current strata and other strata's into account, these are called dependency functions. And there are the ones that only take the environment into account these are called solo functions. When the dependency functions perform calculations against its own strata, it is called a selfie-dependency-function. This will affect how and when they are calculated. What needs to be avoided in a selfie-function-case are calculations being performed against the element currently being evaluated. E.g. if an element wants to find the shortest distance to any other element in the current sample, calculations against itself would end up being devastating for the result.

The FFs are created using something called closures in the C# language. Closures make it possible to have a list for each stratum with the FPFs that they are going to use. The closure also lets the function keep track of against which strata it is supposed to perform calculations. The FPFs will then be named "*FPF name* - Against: *name of strata*".

It was a big challenge to get these FPFs to work well together, but it is the key to make a method like this produce good results. The individual probability values from each FF is multiplied together as a total probability for each element. The tool settings in the property grid can be set using example data or by hand. Adjusting the settings by hand can be hard when some FFs do not use easily understandable units. This justifies example data analysis and why it is something worth putting more effort into. These are the main implemented FPFs:

ShortestDistance

This is a dependency function. It calculates and returns the distance to the closest element in the current sample. Only the positions of the elements are utilized. This function is used when the distance between elements needs to be controlled. E.g. elements overlapping can be avoided and it measures in meters.

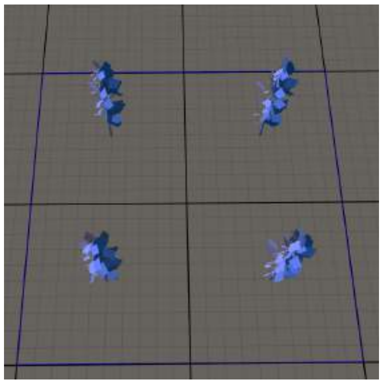


Figure 24.a

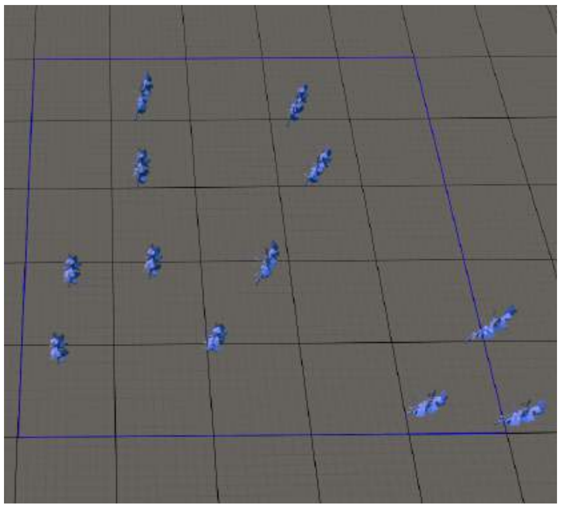


Figure 24.b

Figure 24: 36: Example placings. 24.b: Result.

ShortestDistanceTo4Closest

This is a dependency function. It calculates and returns the total distance to the four closest elements in the current sample. Only the positions of the elements are utilized. This function is used when clumping of elements is desired and it measures in meters.

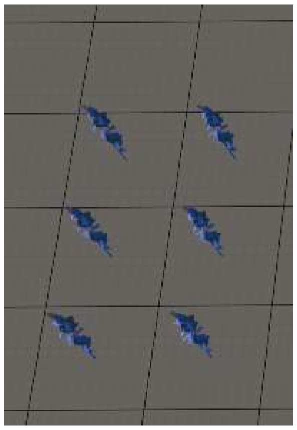


Figure 25.a

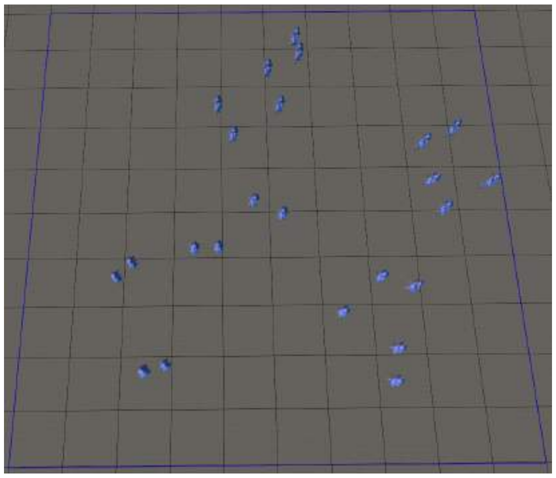


Figure 25.b

Figure 25: 25.a: Example placings. 25.b: Result.

RaytraceDistance
This is a dependency function. It calculates and returns the distance to the closest earlier placed strata-object-mesh. The difference between the RaytraceDistance and the Shortest-Distance FF is that RaytraceDistance will cast a ray towards the strata-objects-mesh. If the element is located inside the mesh, RaytraceDistance will return a negative distance value. This ends up working similar to a signed distance field. The RaytraceDistance function is computationally more expensive than ShortestDistance but will find the actual edge of the earlier placed mesh. It measures in meters.

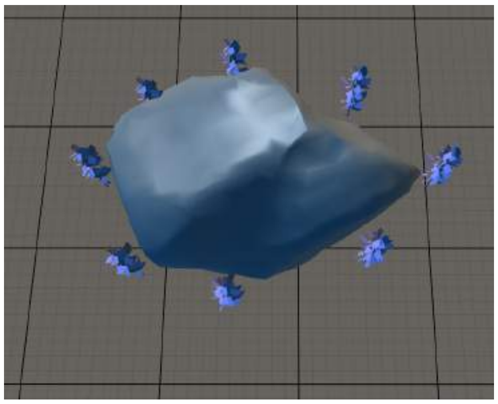


Figure 26.a

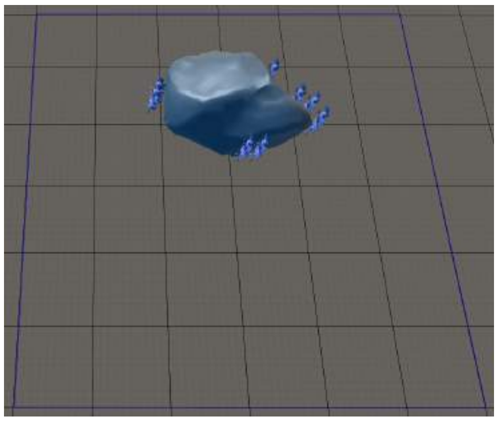


Figure 26.b

Figure 26: 26.a: Example placings. 26.b: Result.

FindEdges

This is a solo function. It uses the discrete Laplace operator seen in equation 6 to calculate if the element is located near an edge or slope. It can separate between if a position is below or above an edge.

$$\begin{aligned}\Delta f(x,y) \approx (&f(x-h,y)+f(x+h,y)+f(x,y-h)\\ &+f(x,y+h)-4f(x,y))/h^2\end{aligned} \tag{6}$$

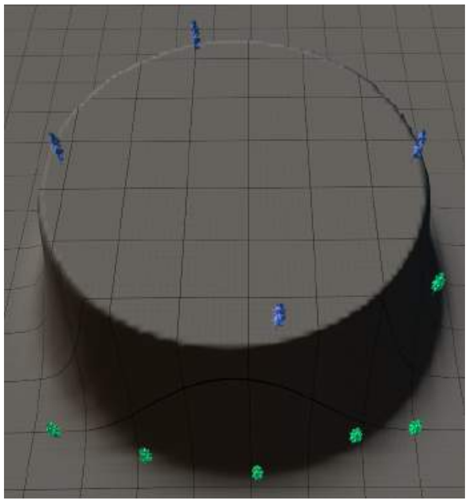


Figure 27: Example placings.

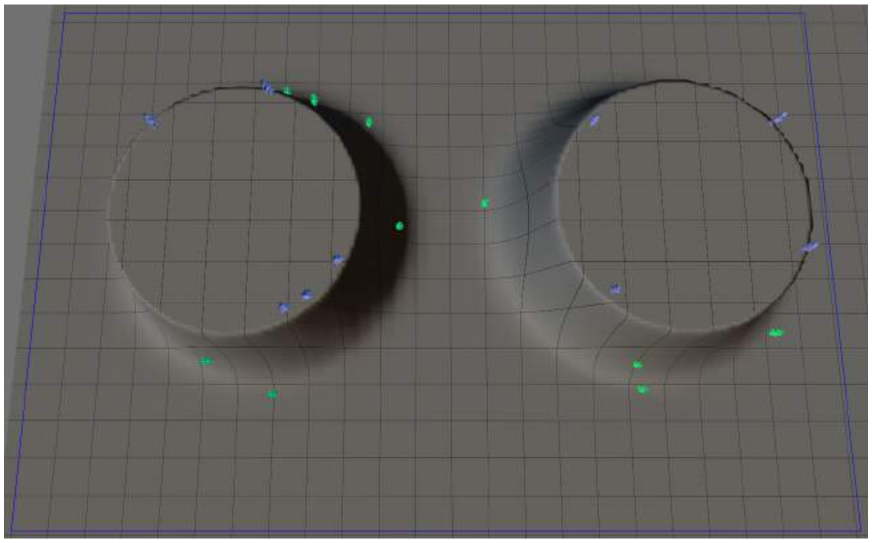


Figure 28: Result.

SlopeAngle

This is a solo function. It calculates the angle between the normal and world up vector, seen in equation 7, where the element is located. It measures in degrees.

$$f(x, y) = acos(worldup \cdot normal) * \frac{180}{\pi} \tag{7}$$

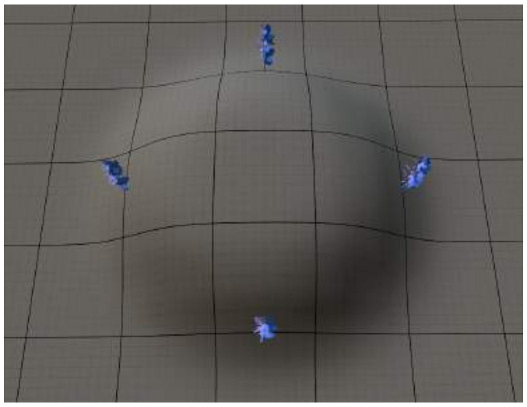


Figure 29.a

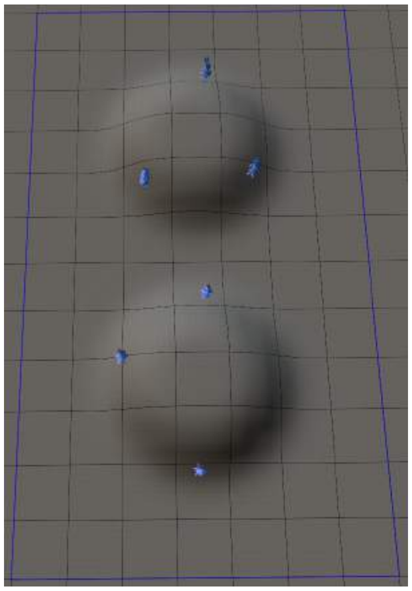


Figure 29.b

Figure 29: 29.a: Example placings. 29.b: Result.

Rotation functions Arbitrary landmark vectors were used to identify the rotations. These were apprehended by talking to the artists and by looking at levels from Battlefield 4. The conclusions drawn:

- The surface normal is important when aligning objects with the terrain.
- The World Up vector is important in cases when vertical objects (e.g. trees) are located in slopes. Even though they are in a slope, they point almost straight up.
- It is important to be able to know how the object is rotated in general, not just how aligned it is with the surface. To solve this, the gradient of the slope and direction towards other objects was utilized.

The general happening for each object when analyzing example placements rotations, ended up looking like:

```
foreach object
  Extract landmark vector in world space

  Transform landmark vector into local space
end

Create a MPDF from observed vectors
```

The general happening for each element in the final sample, when applying rotations, looks like:

```
foreach quaternion
   Extract landmark vector in world space

   Transform landmark vector into quaternion local space

   Let the MPDF evaluate the transformed vector
end

Use dart throwing to choose quaternion
```

It was made sure that there were enough quaternions that satisfied a rotation for the world up vector. This was achieved by generating extra quaternions with a small random deviation from the desired rotation. The more quaternions generated, the likelier it is to find a good fit for the distribution. In the following FRF-descriptions, only the current function is taken into account when choosing which quaternion to use.

Surface Normal

This FRF transforms the surface normal into the local coordinate space of the object. In figure 31, the trees local up vector is aligned with the surface normal direction.

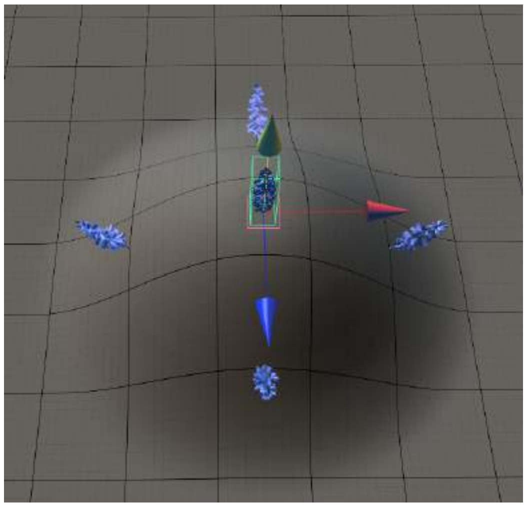


Figure 30: Example placings.

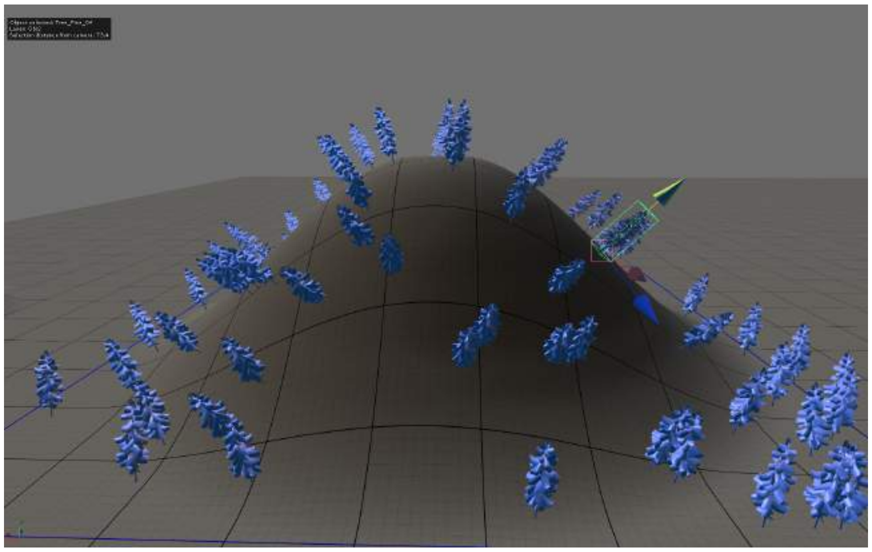


Figure 31: Result.

WorldUp

This FRF transforms the World up vector into the local coordinate space of the object. In figure 33, the trees local up vector is aligned with the world up vector.

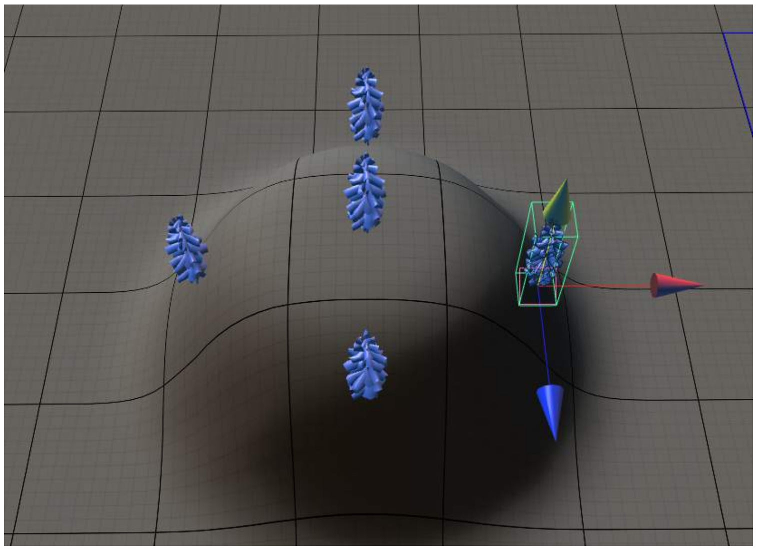


Figure 32: Example placings.

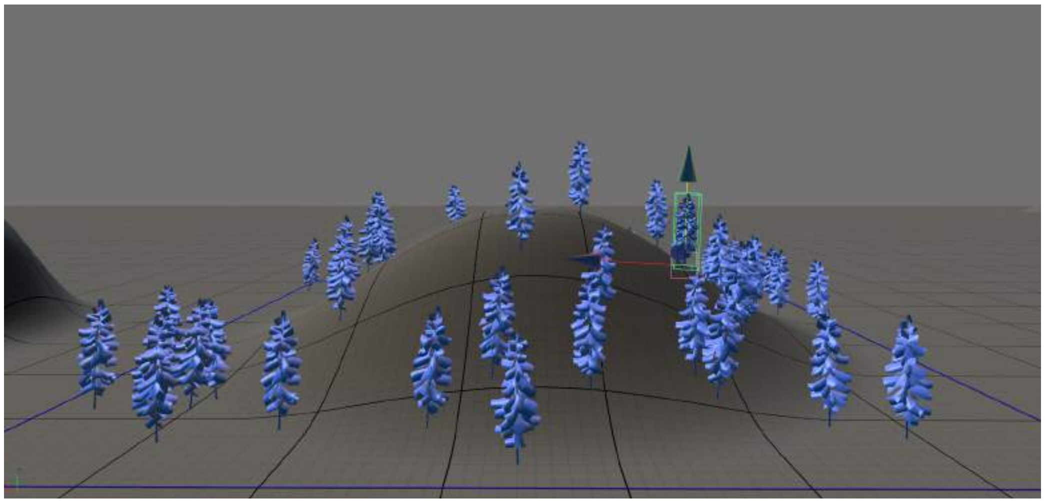


Figure 33: Result.

Surface Gradient

This FRF transforms the surface gradient into the local coordinate space of the object. In figure 35, the shards rotations are depending on the surface gradient. They are perpendicular to the world up vector and rotated away from the hill.

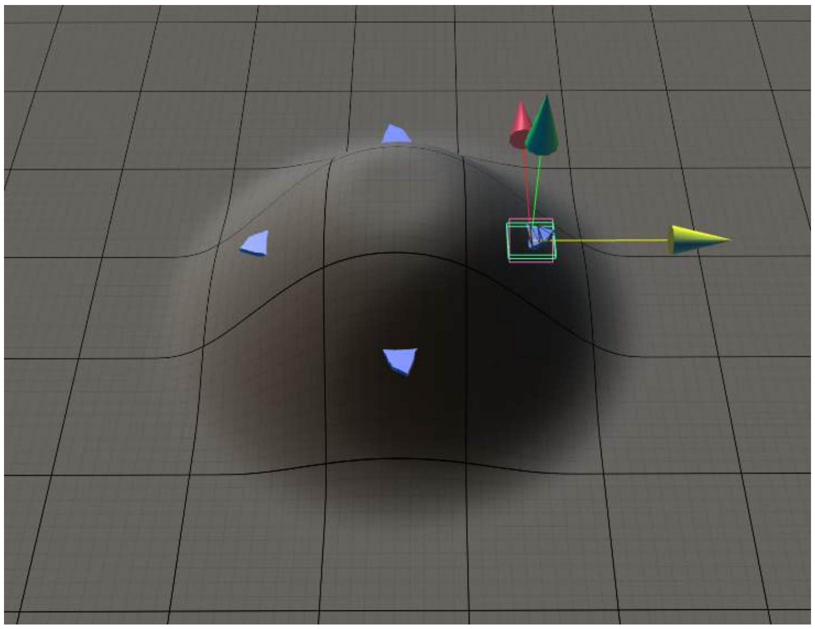


Figure 34: Example placings.

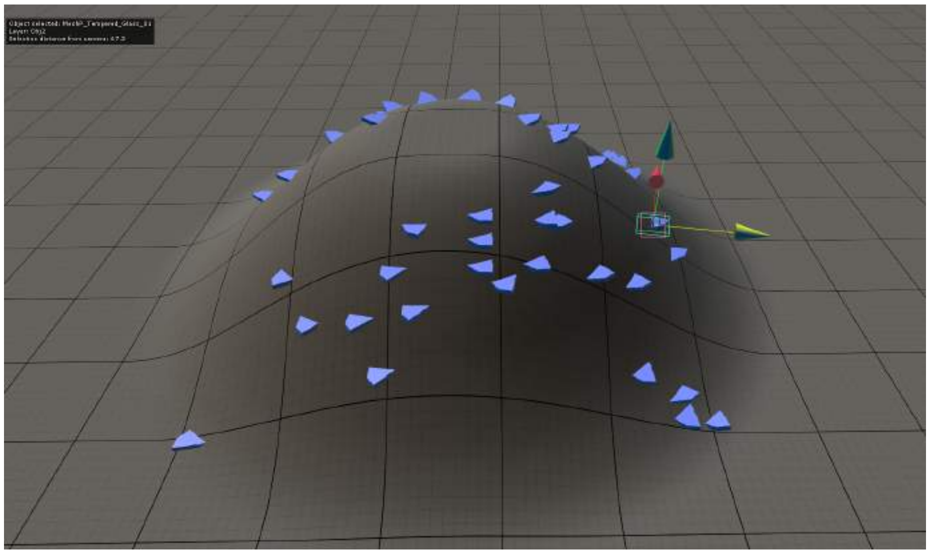


Figure 35: Result.

Direction

This FRF transforms the direction against the closest object, from another stratum, into the local coordinate space of the object. In figure 37, the shards rotations are depending on the direction towards the rock.

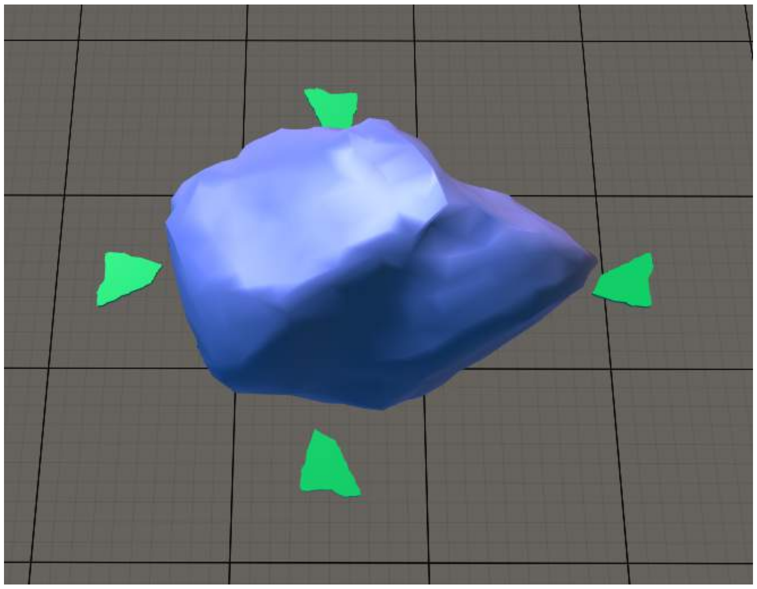


Figure 36: Example placings.

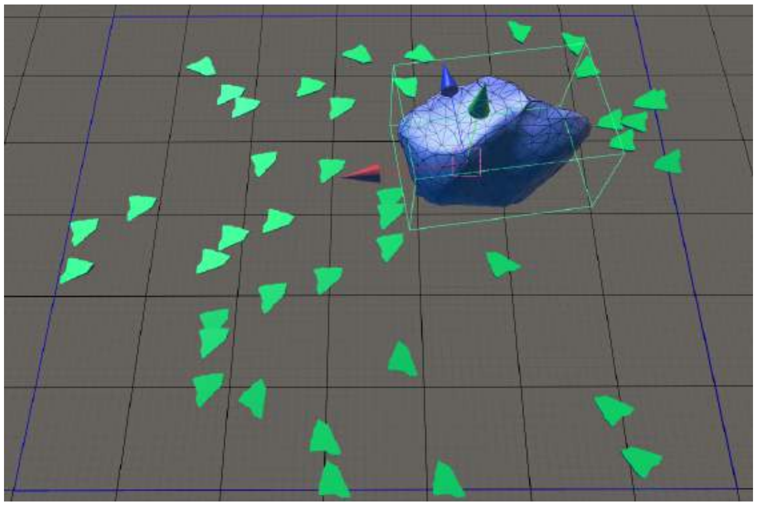


Figure 37: Result.

8.4.3 Deciding which FF to use

When using example placing's there is a need for knowing if a FF is relevant. In this case, a relevant FF would describe something that the artist have taken into account when placing the objects. This is important mainly due to performance but also for avoiding biased results. The most common technique to decide if a distribution is relevant is called Coefficient of Variation (CV). The CV is defined as the ratio of the standard deviation σ to the mean μ as seen in equation 8.

$$c_v = \frac{\sigma}{\mu} \tag{8}$$

The CV is a dimensionless number, meaning it does not matter in which unit it has been measured. And a lot of different SI-units are used in the FFs. The CV however, does not work well when dealing with distributions that can have mean values close to zero. This did not get solved during the thesis but propositions for solving this can be found in section 11.2.

Simulated annealing

The SA is utilized as a global optimization algorithm to test out different combinations of elements. Functionality for deciding if an element is in the current sample or not is implemented. The temperature will affect how many elements that are switched each time and randomly chosen elements are set as the initial current sample. When the SA starts, a new sample is generated by a function that switches the elements. This switch-function uses the current sample as a foundation for the new sample. The new sample is then first re-evaluated by the selfie-FPFs, then evaluated by the KS-test and finally by an acceptance function. If the new sample gets accepted by the acceptance function, it overwrites the current sample. The SA also keeps track of the best sample that has occurred so far, according to the KS-test-value. In figure 38 the SA is running. The current sample is red, the new sample is green and the best sample is blue. The new sample does not differ from the current sample. This indicates that the new sample was accepted.

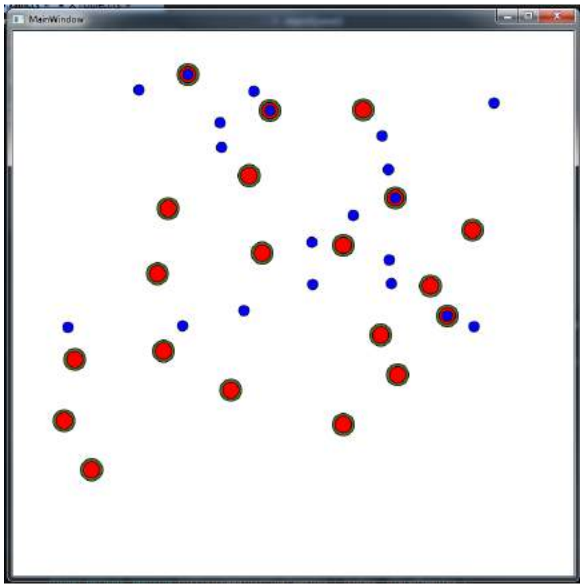


Figure 38: SA while testing different samples. Red are current sample, blue is the best sample and green is the evaluated new sample.

Pseudocode for the SA-algorithm:

```
Set an initial current sample
while temperature > arbitrary value
  Switch-function generates a new sample

  The new sample is re-evaluated by the selfie-feature placing functions

  KS-test evaluates new sample

  if the new sample is accepted
    set new sample as current sample;
    if new sample > best sample
      set new sample as best sample;
  Decrease temperature;
  end
end
```

Source code can be found in appendix A at: A.4.

8.4.4 Kolmogorov Smirnov

The KS-test uses the total probability from each element as indata. The following figures has used the FF shortestDistance 8.4.2 with a mean $\mu = 1{,}0$ and STD $\sigma = 0{,}1$. These are measurements between visual result and the corresponding KS-test values:

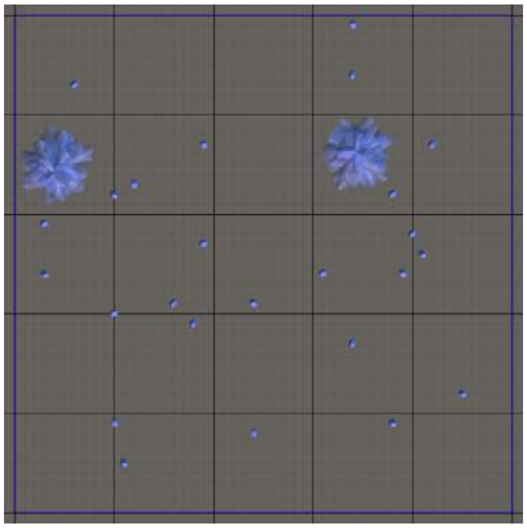


Figure 39.a

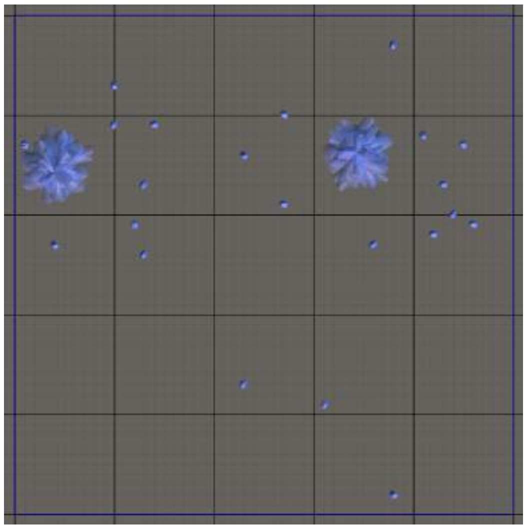


Figure 39.b

Figure 39: 39.a: Visual result - SA cooling constant = 0,1 - Final ks value = 0,632. 39.b: Visual result - SA cooling constant = 0,01 - final ks value = 0,217.

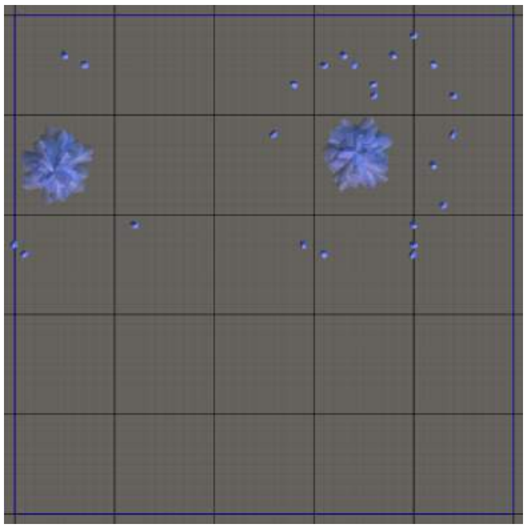


Figure 40: Visual result - SA cooling constant = 0,0001 - final ks value = 0,039.

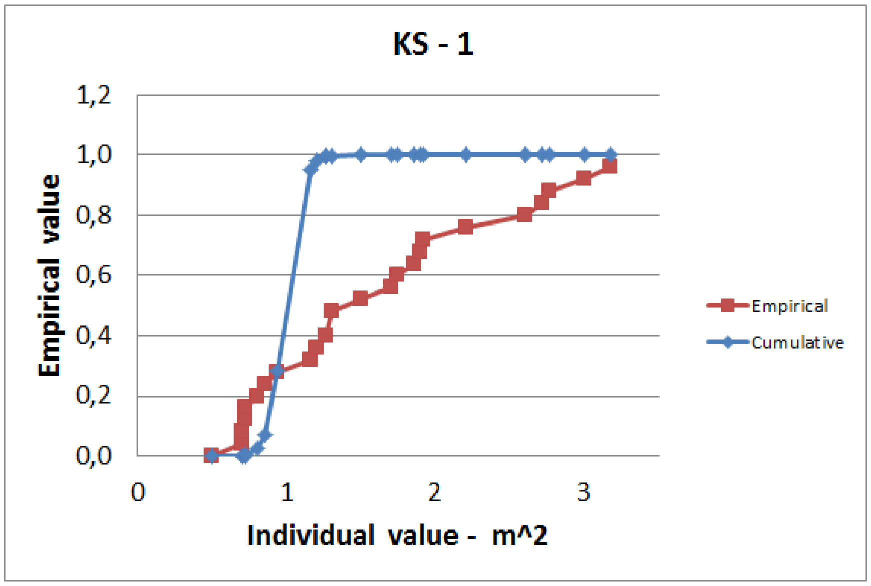


Figure 41: The cumulative and empirical values from figure 39.a.

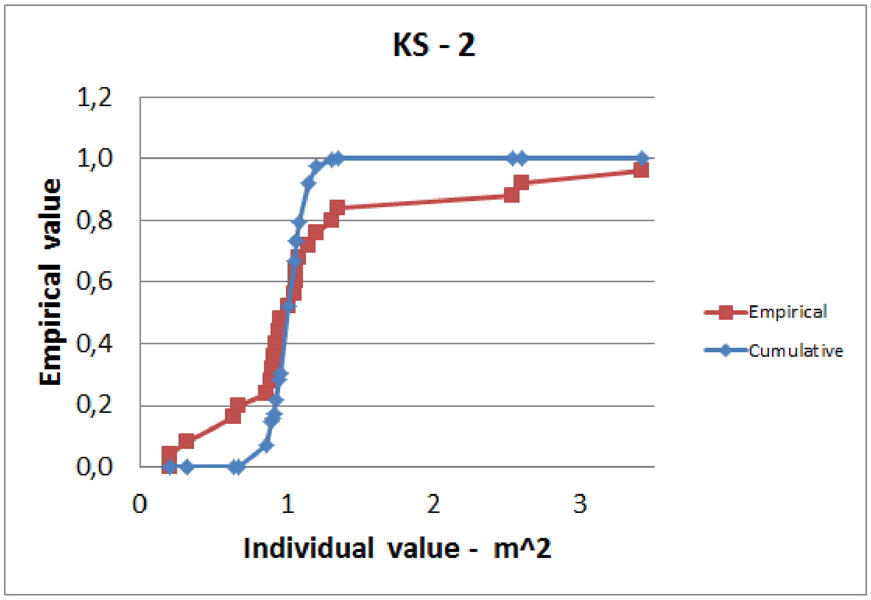


Figure 42: The cumulative and empirical values from figure 39.b.

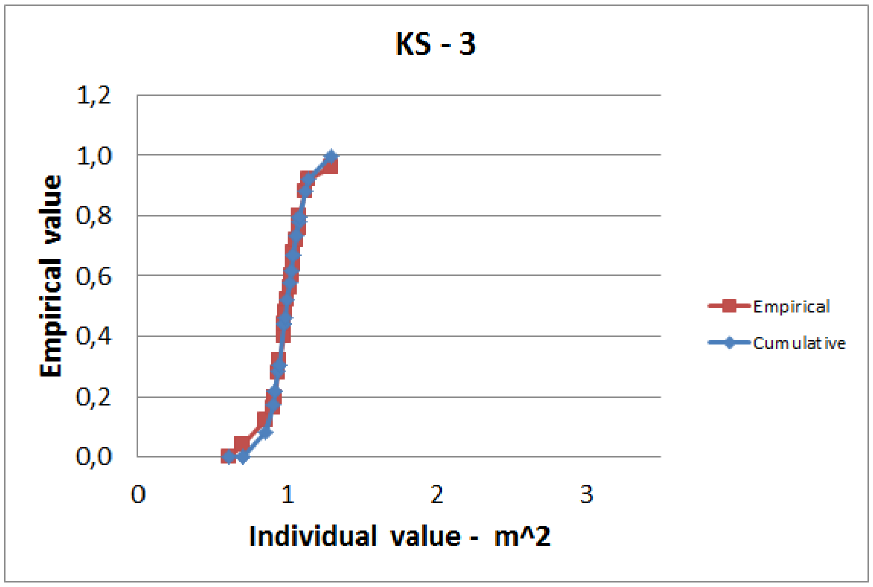


Figure 43: The cumulative and empirical values from figure 40.

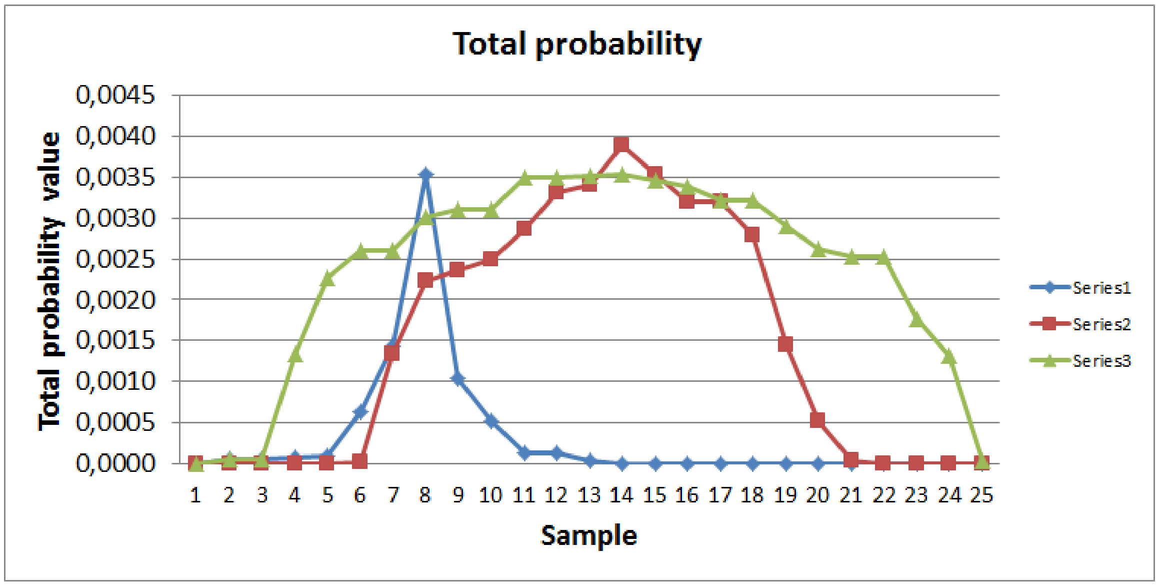


Figure 44: The elements total probability values from each stage. Notice how series3 is shaped like the normal distribution.

8.5 Freeze objects

The ability to freeze objects between re-sampling was implemented. After the initial sampling, the objects that are desired for keeping can be frozen. Then when the sampling-button is pressed again, all of the objects except the ones that are frozen gets re-sampled.

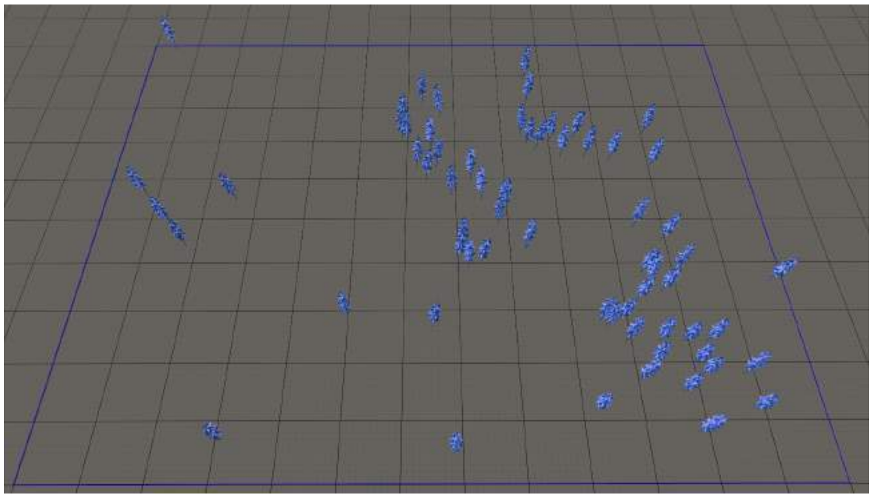


Figure 45: Initial sampling.

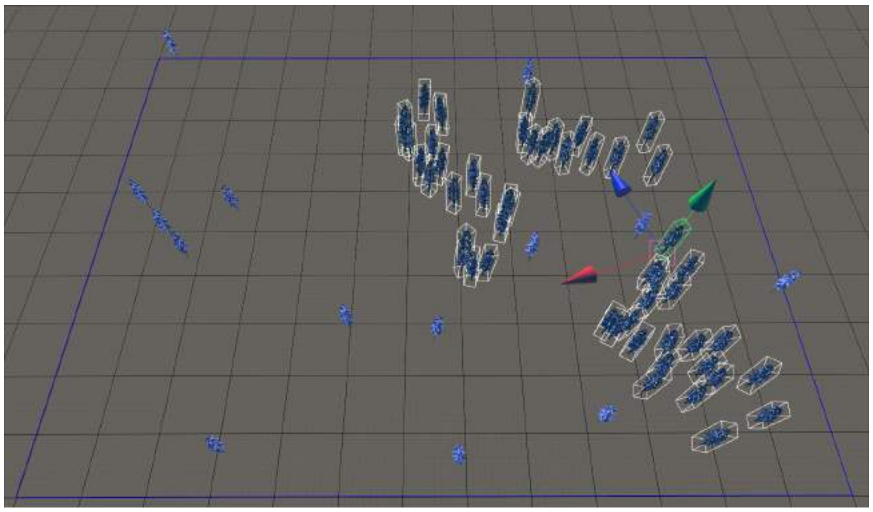


Figure 46: Select objects to freeze.

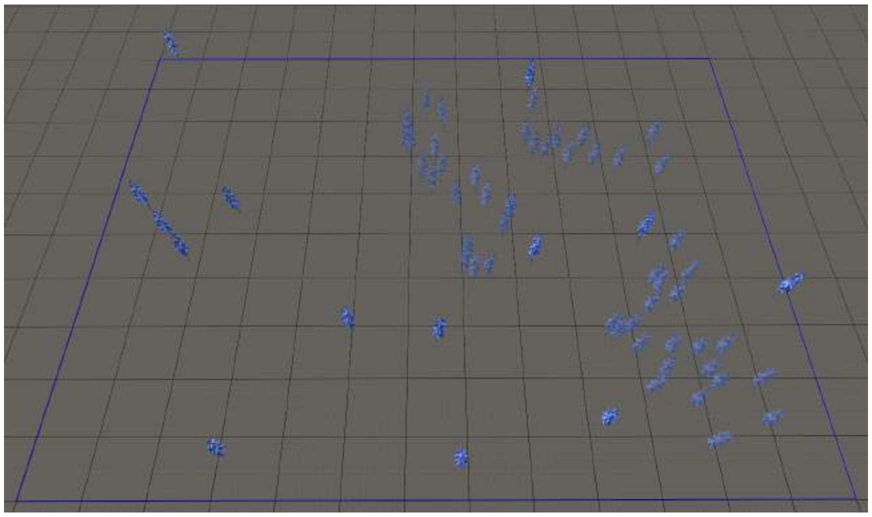


Figure 47: Frozen objects.

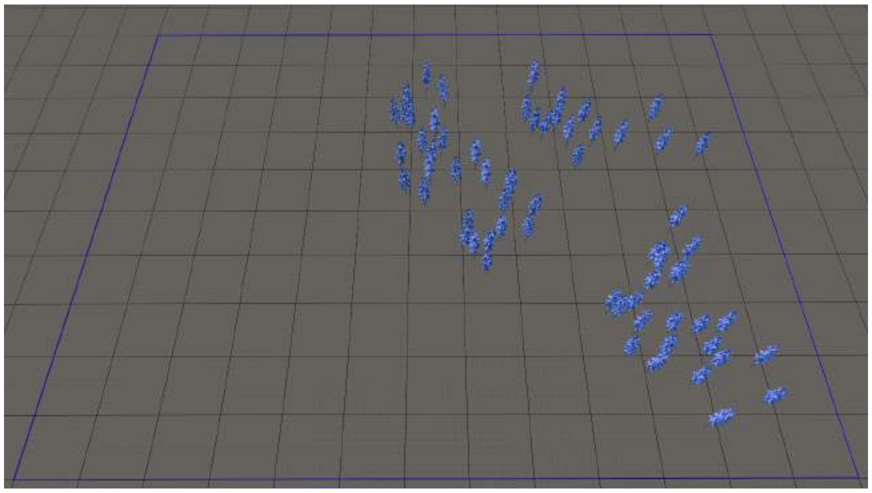


Figure 48: Result after re-scattering.

Part V

Result

8.6 Preliminary study

The obtained information from the preliminary study contributed to apprehend a deeper knowledge and insight into procedural methods. Based on this study, the tool and vision for the thesis could be justified.

8.7 Implementation

8.7.1 Sampling area

The example placings used can be seen in figure 19. Here are the resulting final sample for different sampling areas:

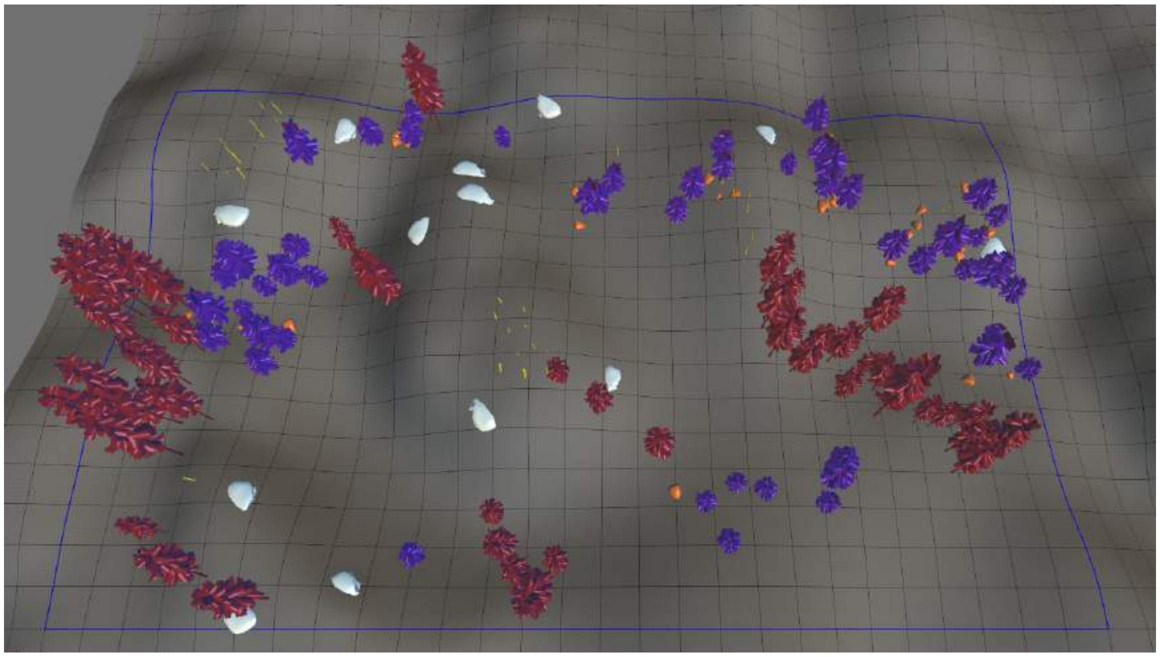


Figure 49: Total time: 2:44 minutes - Area: 432 m^2.

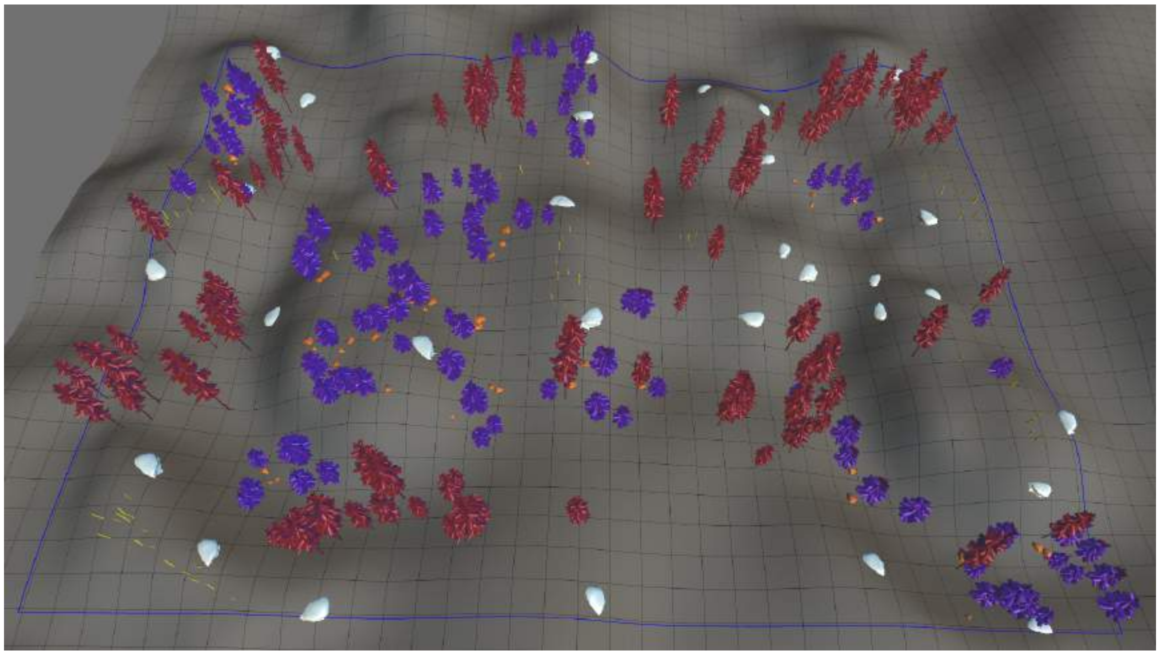


Figure 50: Total time: 10:48 minutes - Area: 864 m^2.

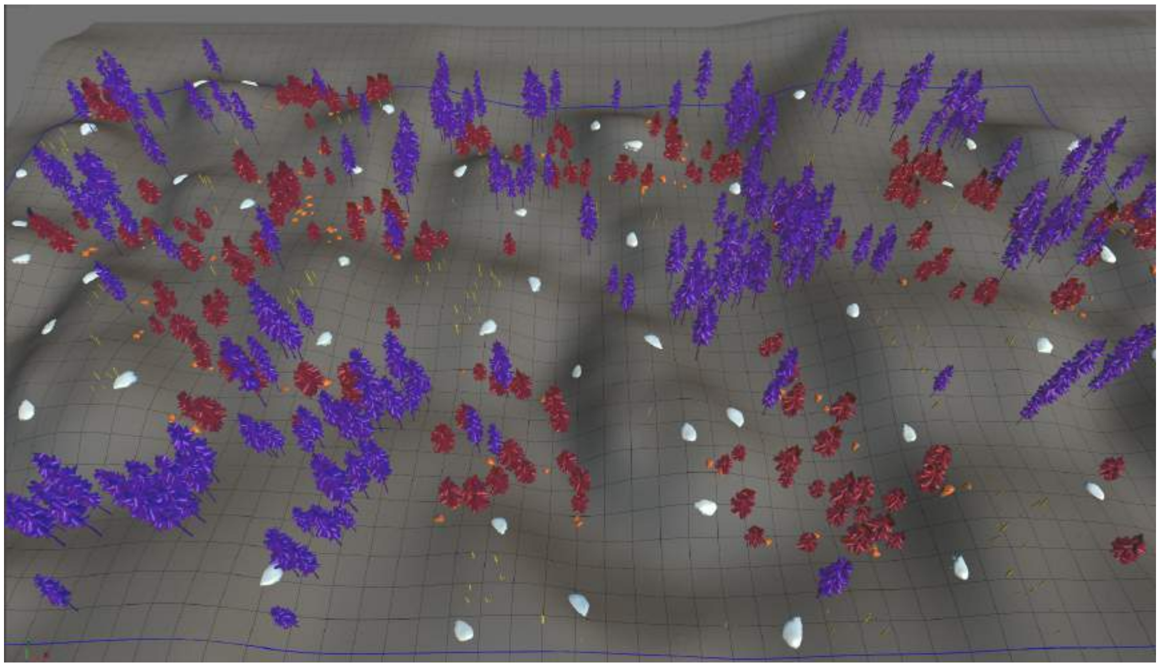


Figure 51: Total time: 50:30 minutes - Area: 1728 m^2.

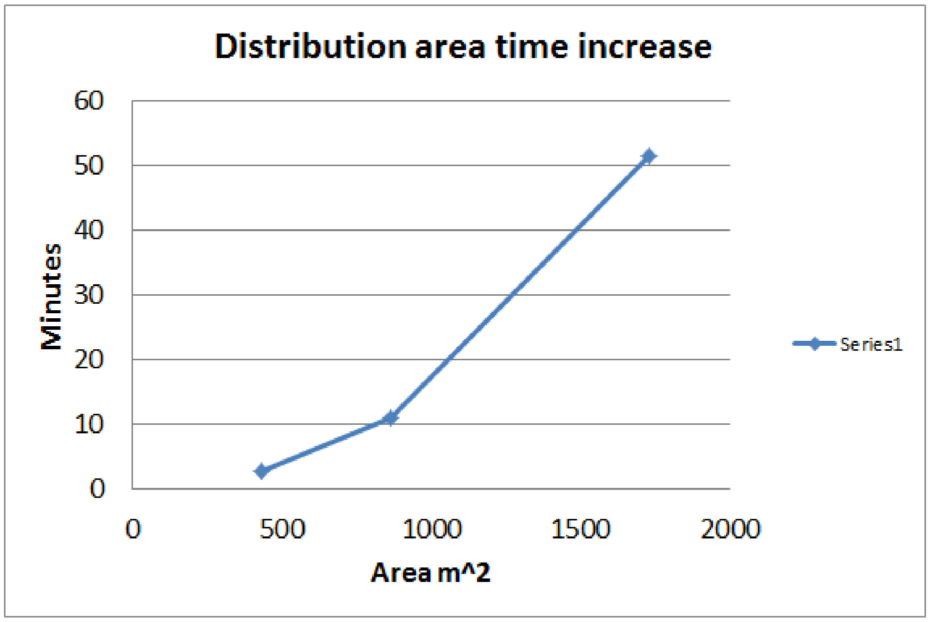


Figure 52: Measured from section 8.7.1.

8.7.2 Showcase

Edge finding

This example demonstrates how the FPF FindEdges can distinguish between positions above and below edges.

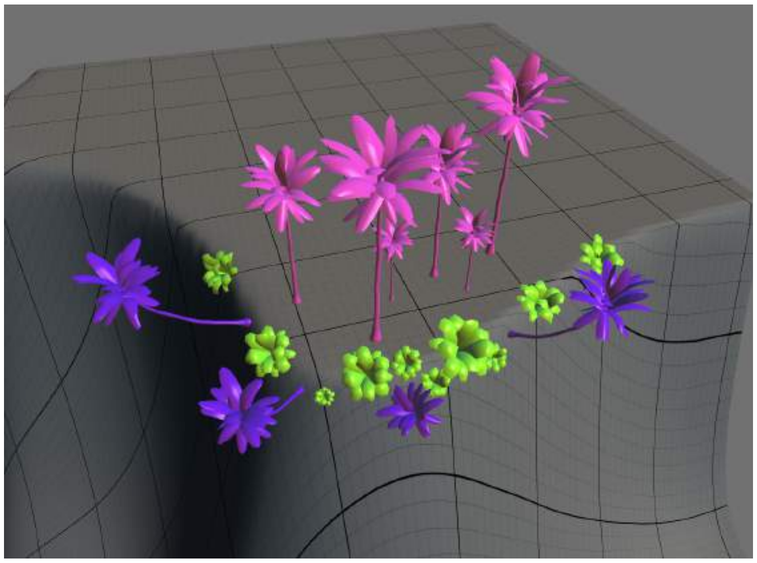


Figure 53: Example placements.

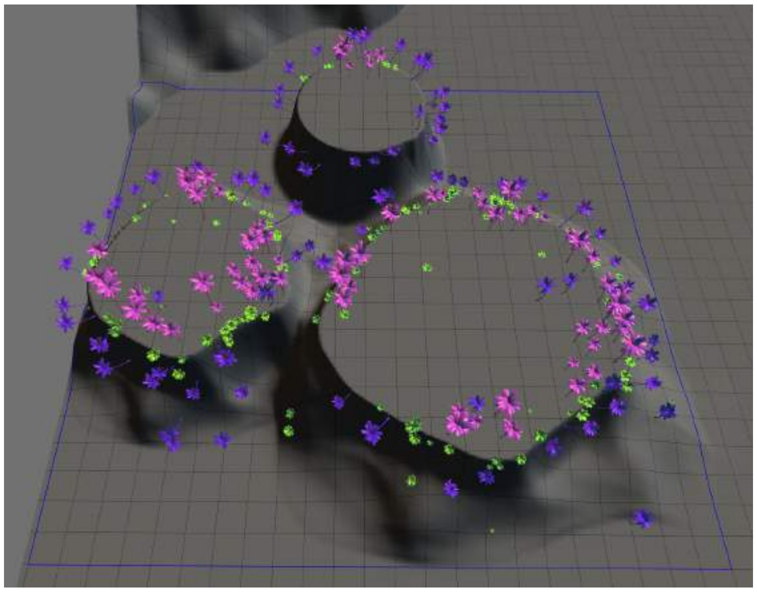


Figure 54: Result using FFs FindEdges and SlopeAngle.

Rotation

This example demonstrates how the FRF Direction can identify and apply a rotation depending on where another object is located.

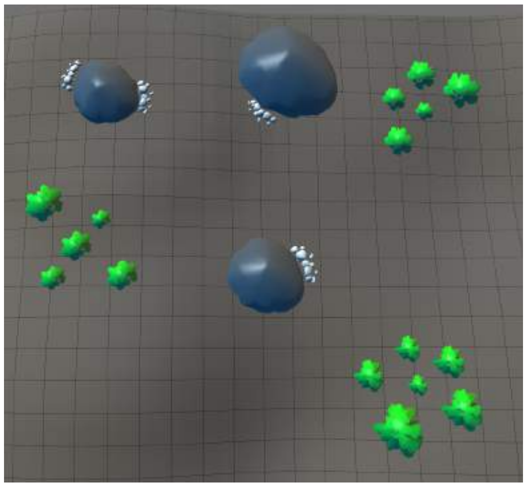


Figure 55: Example placements.

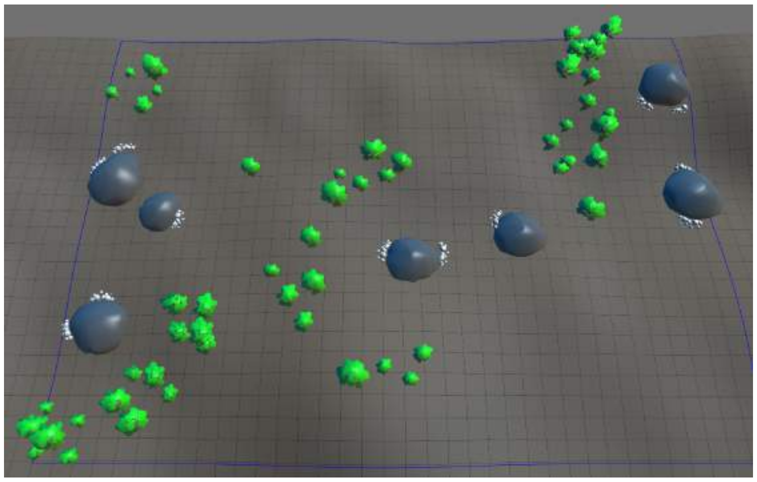


Figure 56: Result using FF ShortestDistanceTo4Closest, RaytraceDistance and Direction.

Pre-placed objects
The tool is able to use pre-placed objects and take them into account when sampling new objects.

8.7.3 Observations and measurements

Dart throwing
In this example the tool was set to sample 5000 candidate elements and have 50 elements in the final sample. The RaytraceDistance FF was used with a mean μ = 0,2 and STD σ = 0,1.

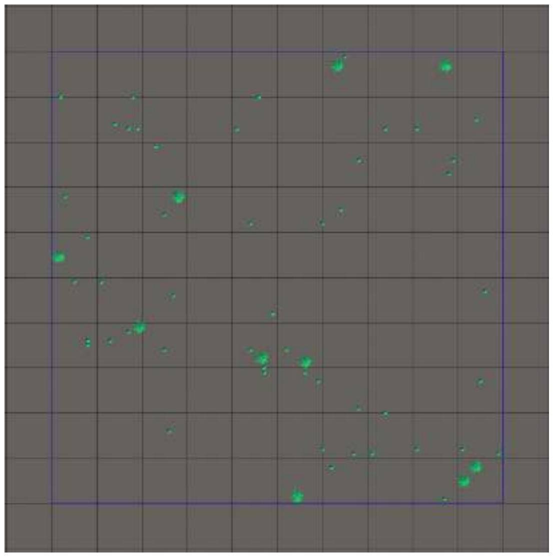


Figure 57.a

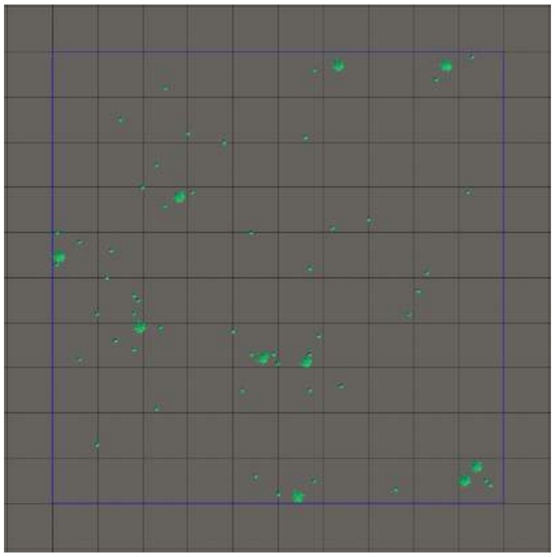


Figure 57.b

Figure 57: 57.a: Result when dart throwing set to keep 100% of the candidate elements. 57.b: Result when dart throwing set to keep 50% of the candidate elements.

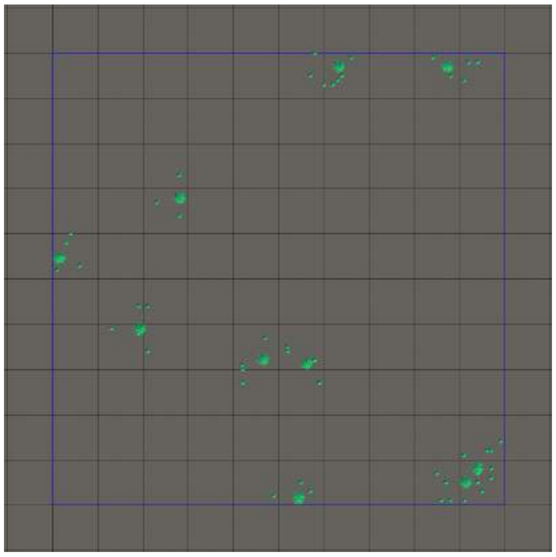


Figure 58.a

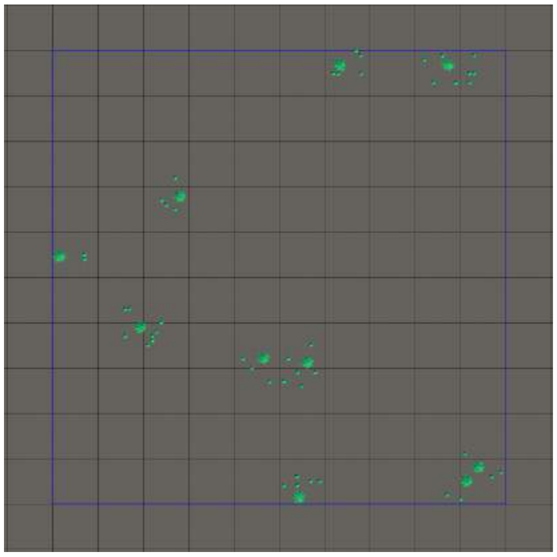


Figure 58.b

Figure 58: 58.a: Result when dart throwing set to keep 10% of the candidate elements. 58.b: Result when dart throwing set to keep 2% of the candidate elements.

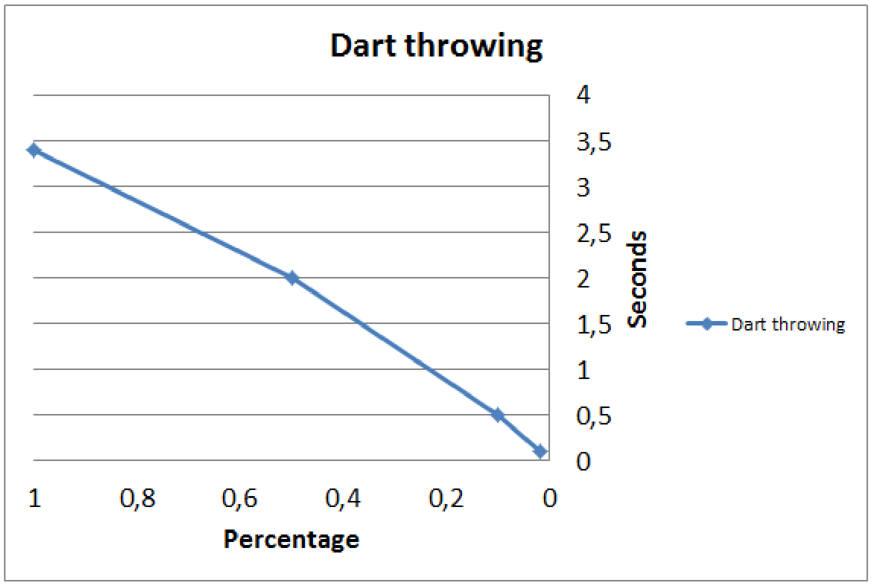


Figure 59: Sampling time visualized for different dart throwing values.

Simulated annealing

In this example the tool was set to sample 5000 candidate elements and have 50 elements in the final sample. The RaytraceDistance FF was used with a mean μ = 0,2 and STD σ = 0,1.

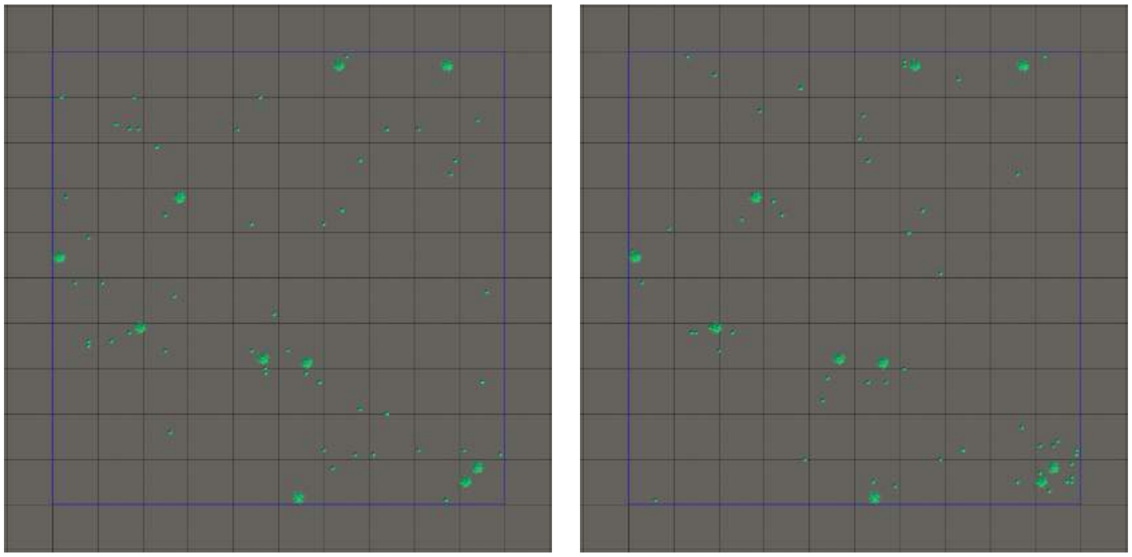


Figure 60.a

Figure 60.b

Figure 60: 60.a: Result when SA cooldown rate set to 0,1. 60.b: Result when SA cooldown rate set to 0,01.

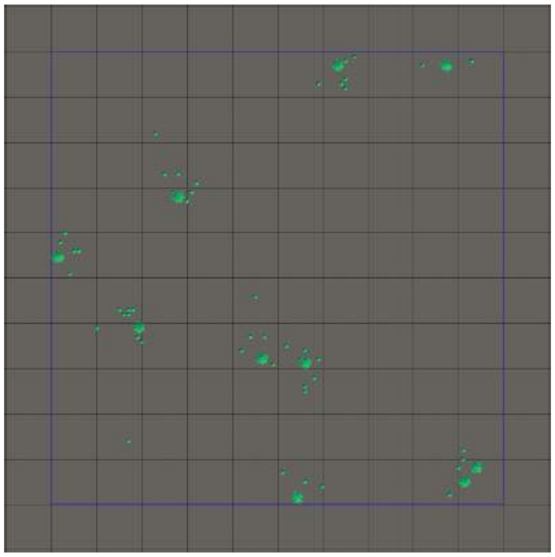


Figure 61.a

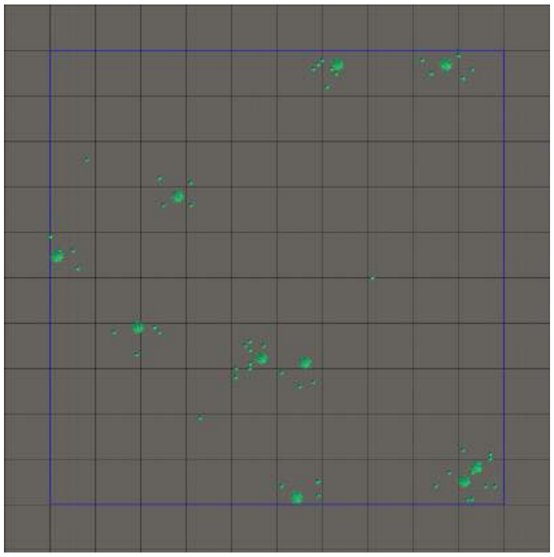


Figure 61.b

Figure 61: 61.a: Result when SA cooldown rate set to 0,001. 61.b: Result when SA cooldown rate set to 0,0001.

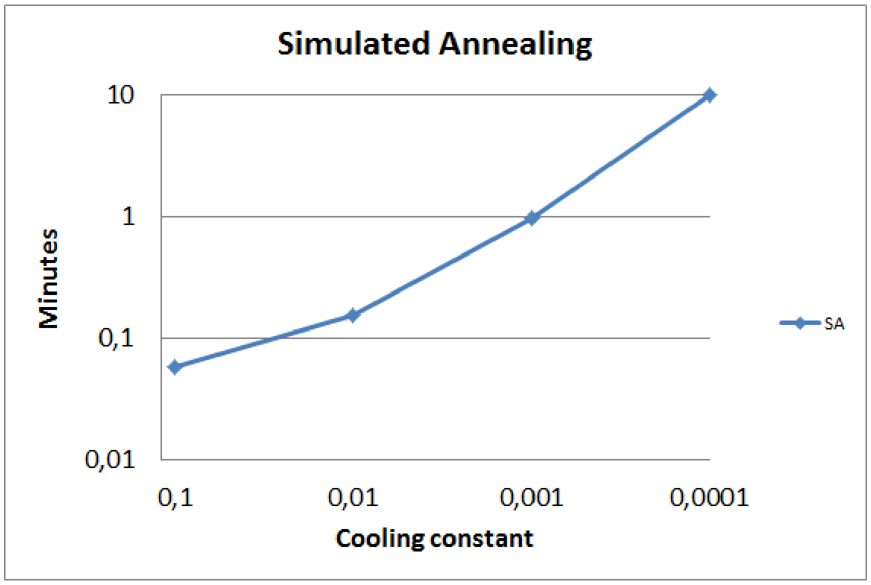


Figure 62: Sampling time visualised for different SA cooldown rates.

Number of candidate elements

The chart in figure 63 visualize how the amount of candidate elements affects the sampling time:

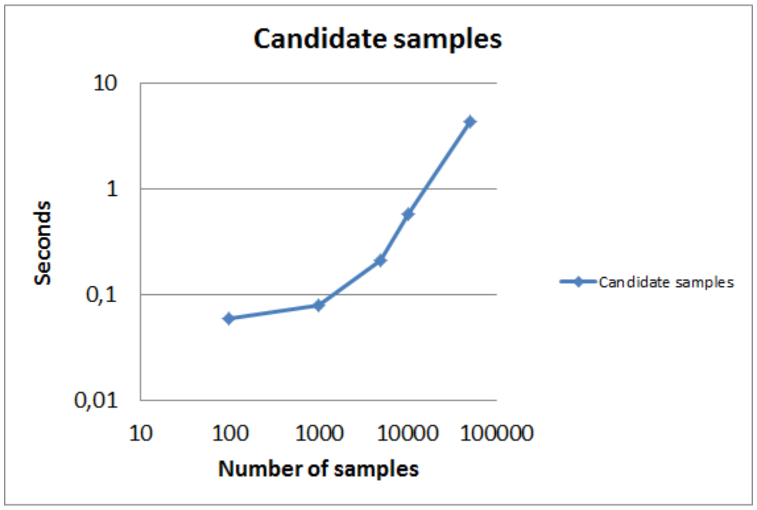


Figure 63: Sampling time visualized for different amount of candidate elements.

Different Standard Deviation values for Feature Placing Functions

The STD values have a great impact on the results and do not always act as predicted. A lower STD value generates less deviation from the mean value. But if the STD value is too low; a lot of elements will obtain a zero probability value. This phenomenon can be seen in figure 64.a where the STD value is very low. Here are a few pictures that show how STD affects the KS value and the visual result:

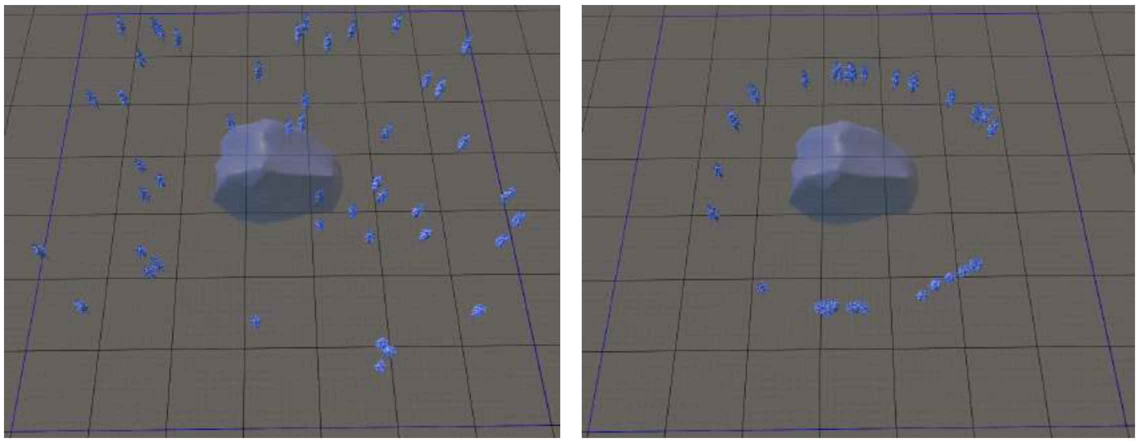


Figure 64.a Figure 64.b

Figure 64: 64.a: KS = 0,604 and STD = 0,001. 64.b: KS = 0,26 and STD = 0,01.

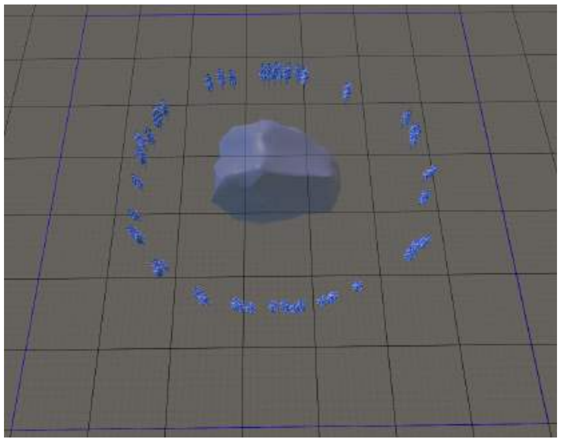


Figure 65.a

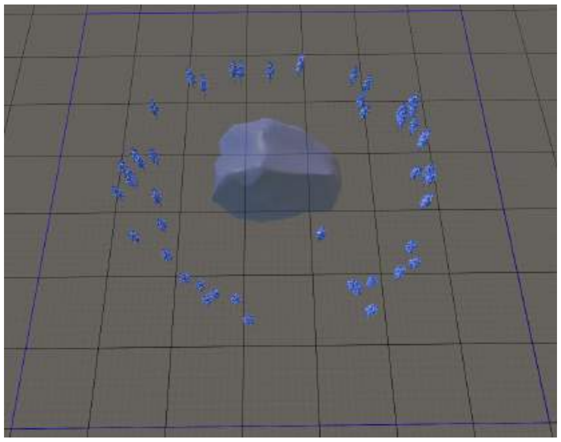


Figure 65.b

Figure 65: 65.a: KS = 0,12 and STD = 0,1. 65.b: KS = 0,14 and STD = 0,5.

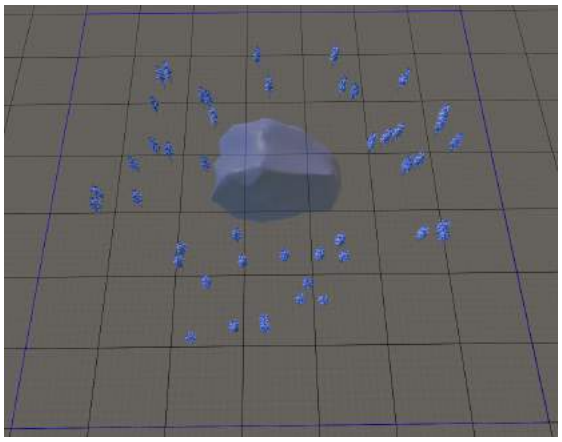


Figure 66.a

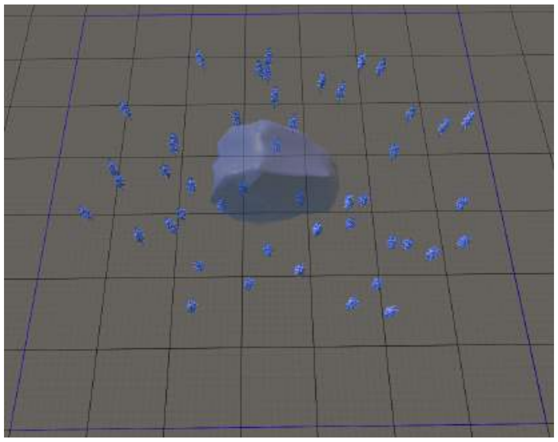


Figure 66.b

Figure 66: 66.a: KS = 0,127 and STD = 1,0. 66.b: KS = 0,15 and STD = 2,0.

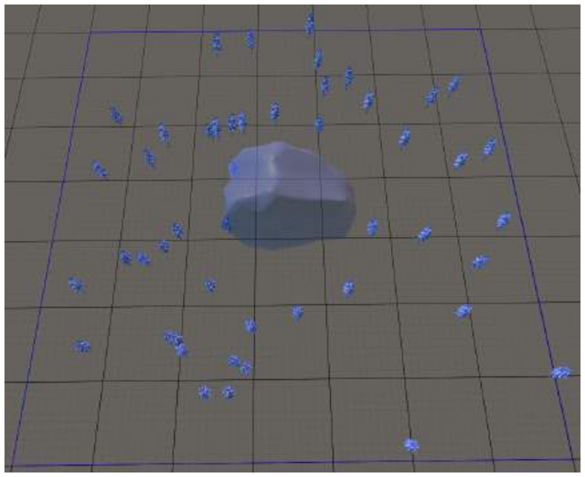


Figure 67: KS = 0,14 and STD = 5,0.

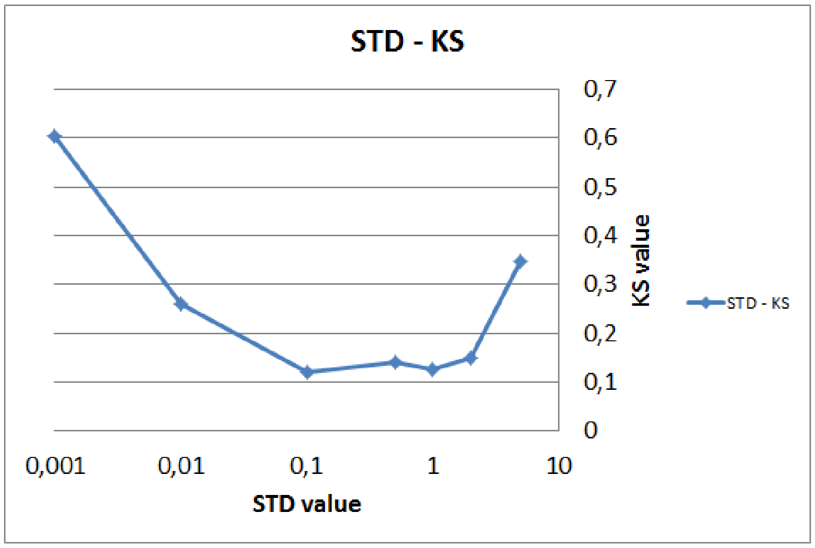


Figure 68: Chart visualizing the STD results.

Different variance values for Feature Rotation Functions

Demonstrates how much the variance can affect the ability to find a good quaternion. The only FRF used is the Normal:

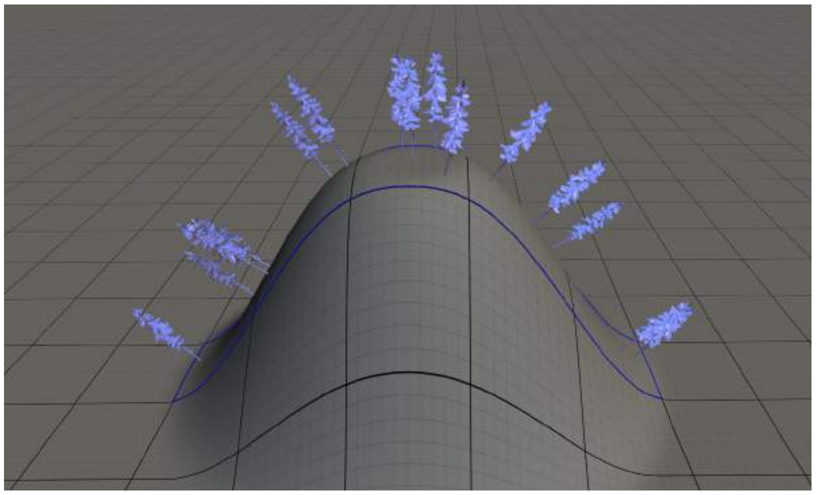


Figure 69: XYZ Variance = 0,001.

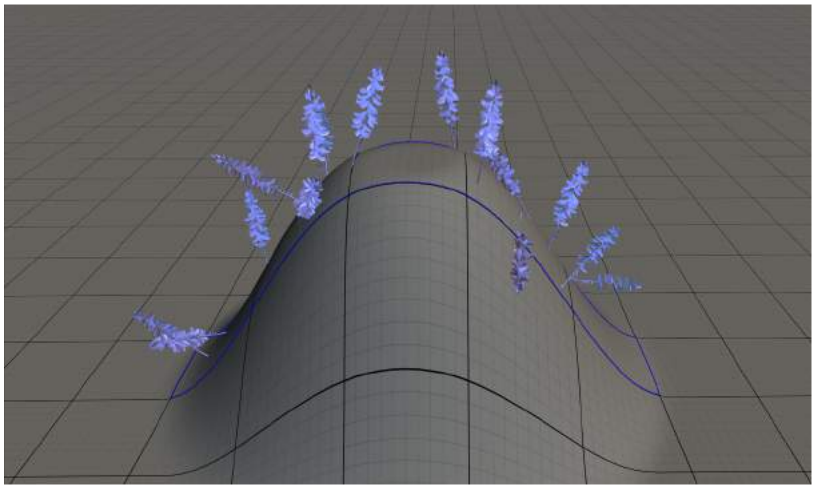


Figure 70: XYZ Variance = 0,5.

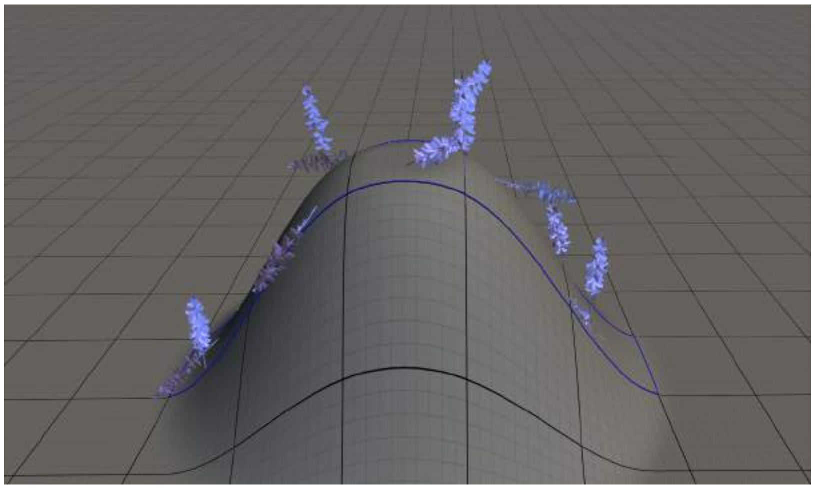


Figure 71: XYZ Variance = 1,0.

Normal distribution close to edge

Figure 72.a and figure 72.b show the effects of using a normal distribution. A potential problem is that the new objects gets placed inside the rock when the STD value increases. With a normal distribution it is not possible to create a gradient of trees fading away from the rock. This issue is mentioned in the future work section 11.2.

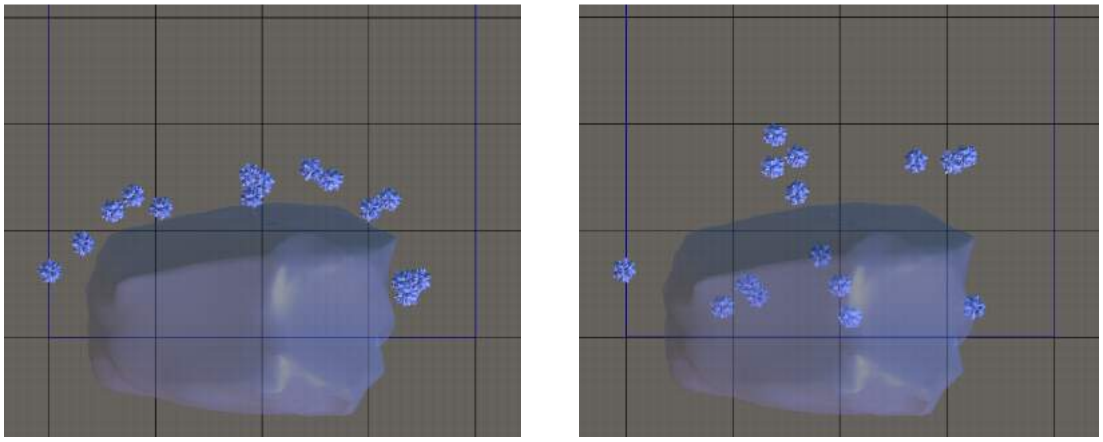


Figure 72.a

Figure 72.b

Figure 72: Trees using the RaytraceDistance 8.4.2 FPF against the rock. 72.a: STD = 0,1 . 72.b: STD = 0,5.

8.8 Evaluation

It is hard to evaluate a method that can't be compared with any equivalent existing method. This evaluation will be based upon opinions and measured data that has been discussed with various different people.

When using example placements, the artist does not need to know about all the properties that each stratum has. This is a big advantage even though it is good to know about them if small changes has to be made. From a workflow and artistic point of view, the tool is a bit unintuitive. The FFs contributes to a versatile tool and workflow. If an artist needs the tool to take something new into account in the environment, the programmer can easily add such a FF. The different parts of the algorithm can each individually have a great impact on the outcome, hence it is important not to neglect any part of it. The STD value, the variance, the dart throwing, the SA cooling rate and the amount of candidate elements generated will all have great impact on the final sample.

9 Discussion

9.1 Method

The final method is producing good results and proves that these new methods for controlling how elements get generated works well. There are however some drawbacks in performance visualized in figure 52.

As stated earlier, the implementation of FFs is a new solution for distributing objects procedurally. Both the placing and rotation methods were invented for this tool. Because of this a lot of effort was put into developing and testing these theories as how to implement and find solutions for them. This meant that other parts had to suffer, which should have had a greater impact on the problem statement. It has been difficult to produce an intuitive tool with good workflow when so much was unknown about the method and solution from the beginning.

The solutions for the encountered problems were though up along the way and there were problems that were hard to foresee. An example would be when the SA and KS took too long to produce good results, which lead to an implementation of dart throwing. But there were also problems that could have been though about earlier. An example of this would be that the rotations was not though about until after the FPF was proven working. The amount of different implementations and the versatile approaches have contributed to time being consumed which in turn has led to little time for optimizations and fine tuning of the tool.

There were some problems with the covariance matrices in regard to the MPDF. If the user changes the variance of a MPDF, the covariance matrix easily becomes non positive definite. This could however probably be solved by scaling the whole covariance matrix.

The idea from the beginning was to create a good foundation for the artists to hone. A lower SA cooling rate might produce better results, but the time increase is huge as seen in figure 62. Since the artist will still want to do changes afterwards, it might be better to increase the cooling rate and still get an acceptable result.

Combinations of different FFs can produce results that differ from the individual FFs. FindEdges combined with SlopeAngle can for example help each other to find edges with flat tops, as seen in figure 54.

It is great with a tool that can cluster objects from a performance perspective since environment artists today cannot create a thick forest if they want great detail.

Finally the tool and functions have been implemented in $C\sharp$ which could affect the performance as well.

9.2 Result

The increase in time when adding strata's and when enlarging the distribution area are the biggest issues. Seen in 8.7.1 there is an time increase in the order of 5 times per m^2 between figure 49 and 51.

10 Conclusions

Developing tools that are going to be used by someone else is tricky and the fact that they probably have a totally different technical background does not help. This is why it is critical to get input and opinions from the end user.

The problem statement was as following: How can procedural methods be applied when creating a useful procedural distribution tool which has the desired level of artistic control? How should a tool like that be designed?

The effort regarding workflow and getting artistic input was hard to maintain when so much time and energy had to be put into developing the new technique. Also since no reports were followed, the entire method was though up along the way. This led to a lot of testing and research.

The level of artistic control is good. But for the artist to be able to do desired changes in the tool; they need to understand quite well what is happening.

Because of the fact that the tool settings can be set and controlled in such a visual way and without the artist having to calibrate anything, example based procedural tools should be regarded as the future of procedural tools.

11 Future work

11.1 Optimization

The huge time increase when enlarging the area was mainly due to the global optimization. Instead by dividing the sampling area into smaller individual areas, a local optimization can be applied. Then by running the scattering algorithm locally instead of globally will save a lot of computing power. The individual square areas should probably not be smaller than the original example placements area.

11.2 Distribution fitting

By implementing distribution fitting instead of just using normal distributions, a few advantages could be gained. When fitting a distribution the exact distribution would be obtained. The tool would be able to mimic the example placements better. This would be able to solve the edge-problem with normal distributions mentioned and seen in 8.7.3. It would also lead to the possibility that every distribution could be taken into account, hence the CV would not be needed. A downside from a performance point of view is that all distributions might still not be needed to create a good final sample.

References

[1] Ruben M. Smelik, Tim Tutenel, Rafael Bidarra and Bedrich Benes. *A Survey on Procedural Modeling for VirtualWorlds.* 2014: COMPUTER GRAPHICS Forum `http://graphics.tudelft.nl/~rafa/myPapers/bidarra.CGF.2014.pdf`.

[2] Markus Lipp *Direct Artist Control for Procedural Content Generation of Urban Environments.* Oktober 2010: Dissertation `http://www.cg.tuwien.ac.at/research/publications/2010/lipp_markus-2010-DAC/lipp_markus-2010-DAC-Thesis.pdf`.

[3] SAGE *Choosing the Type of Probability Sampling.* - `http://www.sagepub.com/upm-data/40803_5.pdf`.

[4] Mick West *Random Scattering: Creating Realistic Landscapes.* 2008: Gamasutra `http://www.gamasutra.com/view/feature/1648/random_scattering_creating_.php?print=1`.

[5] Herman Tulleken *Poisson Disk Sampling.* 2008: devmag `http://devmag.org.za/2009/05/03/poisson-disk-sampling/`.

[6] Robert Bidson *Fast Poisson Disk Sampling in Arbitrary Dimensions.* 2007: ACM SIGGRAPH `https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph07-poissondisk.pdf`.

[7] Ares Lagae and Philip Dutré *A Comparison of Methods for Generating Poisson Disk Distributions* 2008: COMPUTER GRAPHICS forum `http://people.cs.kuleuven.be/~ares.lagae/publications/LD08CMGPD/LD08CMGPD.pdf`.

[8] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomir Mech, Matt Pharr and Przemyslaw Prusinkiewicz *Realistic modeling and rendering of plant ecosystems* 1998: SIGGRAPH `http://www.graphics.stanford.edu/papers/ecosys/ecosys.pdf`.

[9] Bedrich Benes, Michel Abdul Massih, Philip Jarvis, Daniel G. Aliaga and Carlos A. Vanegas *Urban Ecosystem Design* 2010: Purdue University `https://www.cs.purdue.edu/cgvlab/papers/aliaga/i3d11.pdf`.

[10] Lee Jacobson *Simulated Annealing* 2013 `http://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6`.

[11] NIST/SEMATECH *e-Handbook of Statistical Methods* 2012 `http://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm`.

[12] Tasos Giannakopoulos *VoidInSpace* 2013 `http://www.voidinspace.com/2013/06/procedural-generation-a-vegetation-scattering-tool-for-unity3d-part-i/`.

[13] T Ijiri, R Mech, T Igarashi & G Miller *An Example-based Procedural System for Element Arrangement.* 2008: EUROGRAPHICS `http://www-ui.is.s.u-tokyo.ac.jp/~ijiri/publications/ijiri_EG08_ExampleBasedProceduralModeling.pdf`.

[14] Jay L. Devore & Kenneth N. Berk *Modern Mathematical Statistics with Applications.* 2007 `http://books.google.se/books/about/Modern_Mathematical_Statistics_with_Appl.html?id=3X7Qca6CcfkC&redir_esc=y`.

List of Figures

A Source code

A.1 Random sampling

```
//w = width
//h = height
public void Random ( int w, int h, int numOfSamples )
{
  Random rnd = new Random();
  for(int i = 0; i < numOfSamples; i++)
  {
        var x = rnd.Next( 0, w );
        var y = rnd.Next( 0, h );
        AddSample( x, y );
  }
}
```

Listing 1: C♯ code for generating random samples

A.2 Jitter grid sampling

```
//w = width
//h = height
//d = displacement
public void Random ( int w, int h, int d, int rows, int cols )
{
  Random rnd = new Random();
  var xd = w/rows;
  var yd = h/cols;

  for(int y = 0; y < rows; y++)
  {
    for(int x = 0; x < cols; x++)
    {
      var dx = rnd.Next( -d, d );
      var dy = rnd.Next( -d, d );
      AddSample( x * xd + dx, y * yd + dy );
    }
  }
}
```

Listing 2: C♯ code for generating samples with a jitter grid

A.3 Dart throwing

```
public int GeneralDartThrowing ( List<float> individualValues )
{
  Random rnd = new Random();
  var sum = individualValues.Sum ();
  var randValue = rnd.NextFloat ( sum );

  float totValue = 0.0f;
  int index = 0;
  while (totValue < randValue && index < individualValues.Count - 1)
  {
        totValue += individualValues[index];

    if (totValue < randValue && index < individualValues.Count - 1)
    {
      index++;
    }
  }

  return index;
}
```

Listing 3: C♯ code for dart throwing algorithm

A.4 Simulated annealing

A.4.1 Flip function

```
public void SimulatedAnnealingFlipFunction ( List<SampleExtension>
    pSamples, int pNumOfPlacements, float currTemp, float startTemp )
{
  float tempValue = currTemp / startTemp;

  int countTotalFlips = 0;

  while (countTotalFlips == 0) // Will always flip at least once.
  {
    foreach (SampleExtension s in pSamples)
    {
      float rndFlipValue = FastRandom.Instance.NextFloat ();
      if ( rndFlipValue < tempValue && s.m_inScene )
        {
        s.m_inScene = false;
        int rndIndex = FastRandom.Instance.Next ( 0, pSamples.Count
            );
        while (pSamples[rndIndex].m_inScene)
        {
         rndIndex = FastRandom.Instance.Next ( 0, pSamples.Count );
        }
        pSamples[rndIndex].m_inScene = true;
        countTotalFlips++;
      }
    }
  }
}
```

Listing 4: C♯ code for the sample flip function

A.4.2 Acceptance funtion

```
public double AcceptanceProbability ( double currDistPoints, double
   newDistPoints, double currTemp, double startTemp )
{
  double tempValue = currTemp / startTemp; //Between 0.0 and 1.0
  tempValue = Math.Pow ( tempValue, 2.0 );

  // If the new solution is better, accept it
  if (newDistPoints < currDistPoints)
  {
        return 1.0;
  }

  // If the new solution is worse, calculate an acceptance
     probability
  if (!(currDistPoints - newDistPoints < 0.0))
  {
        return 0.0;
  }

  return Math.Exp ( (currDistPoints - newDistPoints) / tempValue );
}
```

Listing 5: C♯ code for the acceptance function

A.5 Random unit quaternion

```
public static List<Quaternion> GenerateRandomUnitQuaternions (int
   numOfGeneratedQuaternions)
{
  Random rnd = new Random();

  List<Quaternion> quaternionList = new List<Quaternion> ();
  var sphereRadius = 1.0;
  int numOfQuaternions = numOfGeneratedQuaternions;
  for (int i = 0; i < numOfQuaternions; i++)
  {
    var theta0 = 2.0 * Math.PI * rnd.NextDouble ();
    var theta1 = Math.Acos ( 1.0 - 2.0 * rnd.NextDouble () );
    var theta2 = 0.5 * (Math.PI * rnd.NextDouble () + Math.Acos (
       rnd.NextDouble () );

    var x0 = sphereRadius * Math.Sin ( theta0 ) * Math.Sin ( theta1
       ) * Math.Sin ( theta2 );
    var x1 = sphereRadius * Math.Cos ( theta0 ) * Math.Sin ( theta1
       ) * Math.Sin ( theta2 );
    var x2 = sphereRadius * Math.Cos ( theta1 ) * Math.Sin ( theta2
       );
    var x3 = sphereRadius * Math.Cos ( theta2 );

    Quaternion randomQuaternion = new Quaternion ( (float)x0,
       (float)x1, (float)x2, (float)x3 );
    randomQuaternion.Normalize ();

    quaternionList.Add ( randomQuaternion );
  }

return quaternionList;
}
```

Listing 6: C$\sharp$ code for generating random quaternions. Formula from: http://mathproofs.blogspot.se/2005/05/uniformly-distributed-random-unit.html