Bachelor thesis

Jonas Seegers

Data-driven generation of artworks

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jonas Seegers

Data-driven generation of artworks

Bachelor thesis eingereicht im Rahmen der Bachelor

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Wendholt
Zweitgutachter: Prof. Dr. Jenke

Eingereicht am: 09. April 2018

Jonas Seegers

Thema der Arbeit
Data-driven generation of artworks

Stichworte
generative art, genetic programming, data-driven, artworks

Kurzzusammenfassung
In this thesis, it will be shown how data-driven artworks can be generated. The generated artworks aim at pleasing mass appeal. In order to do so, the artworks will be shown to users who can vote for their favourite artworks. A genetic algorithm will be used in order to process the voting results and create new artworks, that may please the users' tastes better.

Jonas Seegers

Title of the paper
Data-driven generation of artworks

Keywords
generative Kunst, genetic programming, data-driven

Abstract
In dieser Thesis dreht es sich um das Erzeugen datengetriebener Kunstwerke. Ziel der Kunstwerke ist es, den Massengeschmack zu treffen. Um dies zu bewerkstelligen, werden die generierten Kunstwerke paarweise einer Benutzerschaft präsentiert. Die Benutzer können dann für die Werke abstimmen, die ihnen am besten gefallen. Auf Basis dieser Ergebnisse wird ein genetischer Algorithmus neue Kunstwerke berechnen, die dem Massengeschmack noch besser treffen sollen.

Contents

Introduction **4**
1 Motivation 4
2 Goals 5
3 Structure 5

Basics and Related Work **6**
1 Definition of generative art 6
2 Techniques 6
2.1 More predictable methods 8
2.2 Less predictable methods 15
3 Related Work 24
3.1 "Design and Implementation of a Procedural Content Generation Web Application for Vertex Shaders" 24
3.2 Aesthetic-Oriented Generation of Architectonic Objects with the Use of Evolutionary Programming 25
4 Conclusion 26

Requirement analysis **28**
1 Roles 28
2 Functional requirements 28
2.1 Vote 29
2.2 Process vote 30
2.3 Compute new artworks 30
2.4 Render images 30
3 Nonfunctional requirements 31
3.1 Accessibility 31
3.2 User engagement 31
3.3 Variance 31
3.4 Extensibility 31
3.5 Loose coupling 31
4 Conclusion 32

Design and implementation **33**
1 Reasoning for a distributed system 33
2 Components 34
2.1 Web Application 34

2.2 Voting process 35
2.3 Rendering 36
2.4 Genetic Algorithm 44
2.5 Web services adapter 46
3 Infrastructure 47
3.1 Cloud Computing 48
3.2 Amazon Web Services 48
3.3 Chosen services 49
3.4 Putting components into service 50
4 Example run 51
5 Conclusion 52
6 Lessons learned 53
6.1 Amazon web services 53
6.2 Overhead of the distributed system 53

Conclusion **54**
1 Outlook 55
1.1 Web application 55
1.2 Rendering 55
1.3 Genetic algorithm 56
2 Monolithic approach 56

Bibliography **57**

Introduction

1 Motivation

Generative art is becoming more and more popular and its techniques are used for commercial purposes, live performances and fine art exhibitions. In the past, artists used rather simple algorithms to create two-dimensional images. Since modern day approaches lean towards complex and unique 3D renderings, the requirements for the software systems have risen enormously: Firstly 3D renderings demand way more performance than 2D renderings. Especially for live performances, it's crucial that an application runs fast enough to render artworks on the fly. Secondly, artists strive for possibilities to embody their own style into the generated artworks. Therefore long-serving techniques like using the Mandelbrot set don't qualify to fulfil these desires anymore. In contrast, it's more suitable to use custom computer code to define the system that generates the artworks, because it grants artists a lot of freedom. This, on the other hand, requires an artist to have programming skills. Under the assumption that not every artist who is interested in generative art originates from computer science, it is helpful to offer a beginner friendly starting point. Finally, one of the main reasons to use a generative art system is the possibility to create artworks with a very high level of detail, that would be impractical to achieve with human means only. In order to achieve this level of detail, artists delegate some artistic decisions to the system. Oftenly the system makes these decisions based on unpredictable factors, like randomization or gathered data that changes from time to time. This also enables an artist to generate a series of distinguishable artworks using the same system. The challenge using these techniques is to find a good balance between a high level of detail and variance while filtering illegitimate content. For an artwork, this could mean that the actual composition of the artwork can be random, but there is a constraint for it to show at least one object.

Right now there are many solutions for creating generative art that range from prebuilt software like fractal generators, frameworks like openFrameworks, OpenCV and Cinder to languages like Processing. All of these have their strengths and weaknesses when it comes to fulfilling the mentioned requirements. A prebuilt software is very beginner friendly, but all the outcomes look very similar. Therefore the possibility for an artist to create something unique is not given. The language Processing is also designed to be rather easy to learn. Considering that Processing allows the user to use computer code to build an art generating system, the required freedom to design something unique is given. Unfortunately, when it comes to performance, the high abstraction level and the fact that the language is built on a Java core makes it less suitable for 3D applications.

While frameworks like openFrameworks promise to offer a way better performance due to the fact, that they follow a very low-level approach, they require the user to have advanced C++ skills, which makes them way less beginner friendly.

2 Goals

For this bachelor thesis, I will create a system that generates data-driven 3D artworks. I will confront users via a graphical user interface with two images, that were randomly generated beforehand. The users can vote for the image that they liked better. Based on these votes the system will create new artworks that fit the preferences of the voters. In order to define how the new images should look like, a genetic algorithm shall be used. The new artworks again will be shown to the users and so forth. The goal is to make every new generation of artworks more compatible with mass appeal than the previous one. Everybody who is interested in art shall be able to take part. Therefore the user experience shall be accessible and fun for the participants.

For the artworks, there shall be a high variance. I want to omit that all the artworks look more or less the same after a few generations. The artworks shall have a new look and feel. This is opposed to an approach where, for example, the votes would decide on parameters of a Mandelbulb, because there have been a lot of works using Mandelbulbs.

In addition, the system shall be designed to make it easily extensible. Artists who have programming skills shall be able to implement their own effects. In order to do so, the artist shouldn't need to have advanced knowledge about the low-level implementation details of the system. It should be enough to have basic knowledge about the structure of 3D images. The system shall provide a clear structure that makes the creation of effects as beginner friendly as possible. In addition, the system shall be built in a way that fulfils the high performance requirements. Finally, the artwork creating component shall be loosely coupled in order to allow different data-driven art projects to use it.

3 Structure

In the following, an overview of generative art and it's techniques are given in chapter 2. In chapter 3 a requirement analysis for the desired system is set out. Chaper 4 shows which measures had been taken in order to implement that very system. Finally, in chapter 5 a retrospect of the thesis is given, an outlook is given and an alternative approach is discussed.

Basics and Related Work

This chapter conveys an overview of generative art. Therefore, at first, a definition of what can be considered generative art will be introduced in 1. Afterwards, the most commonly used techniques for generating artworks will be listed in 2. In addition, in 3, related works will be presented, that use these very techniques. Finally, in 4 it will be discussed, why in 2018 it is preferred to use the techniques from 2.2 over the ones shown in 2.1.

1 Definition of generative art

When looking for definitions on generative art, multiple definitions can be found. For this thesis, this definition by Philip Galanter will be used:

> "Generative art refers to any art practice where the artist uses a system, such as a set of natural language rules, a computer program, a machine, or other procedural invention, which is set into motion with some degree of autonomy contributing to or resulting in a completed work of art." [1, p. 4]

What is notable about this definition that it doesn't limit generative art to artworks created by computers. In the same paper, Galanter mentions generative artworks that didn't use any computer at all. For example Hans Haacke's condensation cube [2] utilizes condensing water inside a plexiglass cube in order to generate patterns. However, this thesis focuses on computer-based generative art, that apart from that limit fits the cited definition.

2 Techniques

In the following, the most commonly used techniques for generative art will be discussed. What most of these techniques have in common is that they are to some degree inspired by nature's laws and mechanics. For example, Robert Hodgin's "Written Images: Coronal Mass Ejection" uses a simulation of gravity in order to create a pattern for his artwork, as seen in figure 1. Another example are the objects created for Andy Lomas' "Hybrid Forms". For this project, Andy Lomas build a system that simulates cell growth, as seen in figure 2. Furthermore, most techniques that are used for generative art were originally developed for simulation purposes. Daniel Shiffman's "Nature of Code" [3] features many of the algorithms that are presented in the following. It is noteworthy that the book is not focused on generative art of any kind but on "[. . .] the

programming strategies and techniques behind computer simulations of natural systems [...]".

In addition to the definition discussed in 1, in the same paper, Galanter ranks techniques for generative art based on their randomness. For the presentation of the most used techniques in 2 a similar approach is taken by categorizing techniques as "more predictable" in 2.1 or "less predictable" in 2.2. For the same initial conditions, a more predictable technique will result in equal or very similar results. The outcomes of a less predictable method may seem more random and the algorithm has more freedom to create. Since the research showed, that great potential lays in the use of genetic programming, its most used techniques are listed in 2.2.3. Many of these algorithms are taken from the book "A field guide to genetic programming" by Riccardo Poli, William B. Langdon and Nicholas Freitag McPhee [4].

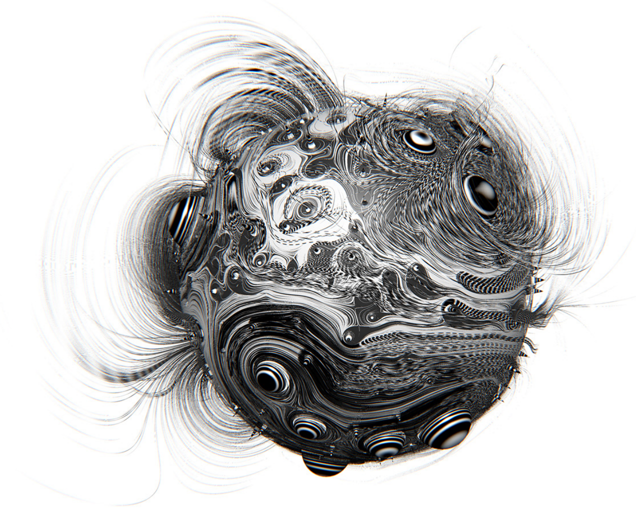


Figure 1: Rendering from Robert Hodgin's "Written Images: Coronal Mass Ejection"[1]

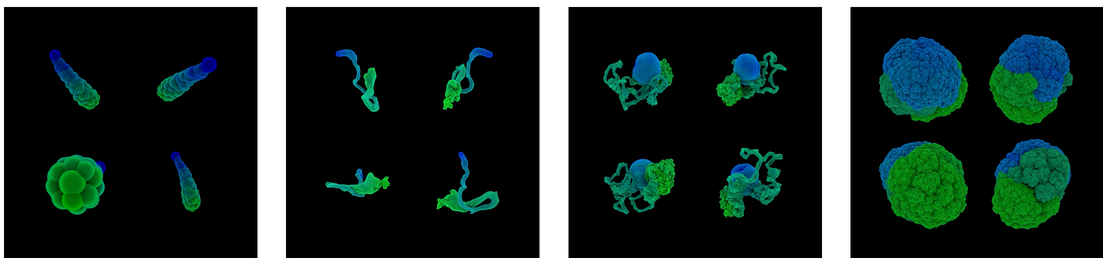


Figure 2: Stills from a video of Andy Lomas "Hybrid Forms" series[2]

[1]Source: http://roberthodgin.com/portfolio/work/written-images-cme/

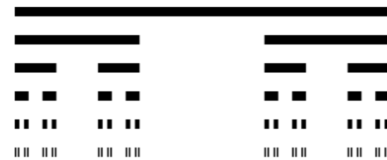
2.1 More predictable methods

At first, structures of complex strucutre, named fractals are presented in 2.1.1. Afterwards, the probably most known fractal, the Mandelbrot set will be introduced in 2.1.2, followed by it's 3D representation, the Mandelbulb in 2.1.3. In addition, it is shown in 2.1.4 how the Lindenmayer-System can be used in order to use grammars and string manipulations in order to create organic looking shapes. In 2.1.5 and 2.1.6 it is shown how cellular automata and Voronoi diagrams can be used to create interesting patterns. Finally, in 2.1.7 it will be shown how particle systems can be used to create effects like smoke and fire. In 2.1.8 the suitability of the techniques for the goal of this thesis will be discussed.

2.1.1 Fractals

Fractals are objects that have a very complex structure, that can't be described by Euclidean geometry. The term "fractal" was coined by Benoit Mandelbrot's book "The fractal geometry of nature" [5]. Furthermore, fractals are defined by having a self-similar structure. That means that fractals can be divided into parts that look the same or alike. In theory, they have an unlimited level of detail. When visualised on a computer screen there is a limit, because there is no smaller measurement than a pixel. Examples for well known fractals are the Sierpinski triangle, the Menger sponge, both sown in figure 4, and the Cantor set, as seen in figure 3a. Fractals are often created by recursive functions. For example, the algorithm for drawing the Cantor set in pseudo code is shown in figure 3.

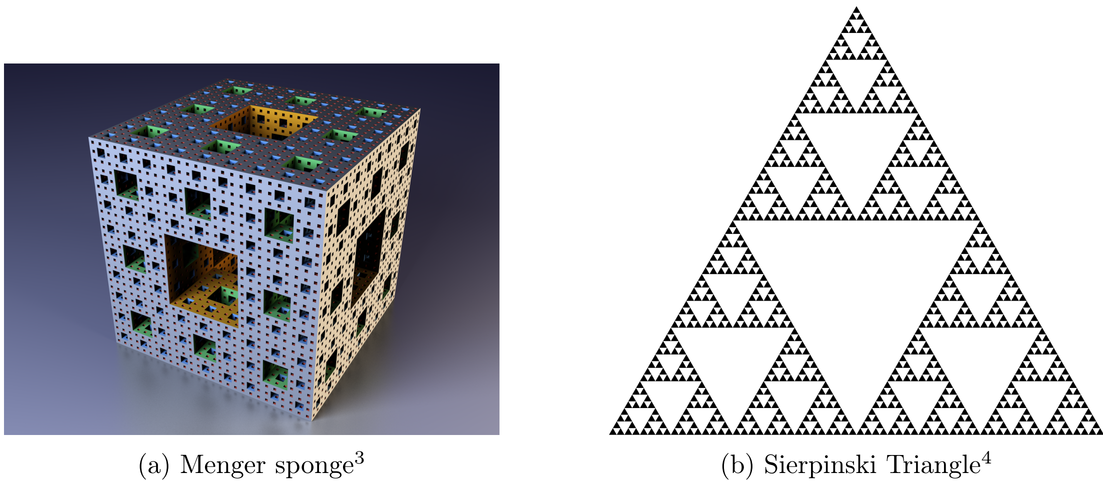
```
1: function CantorSet(line, y)
2:     if getLength(line) >= 1 then
3:         draw(line, y);
4:         parts ← divideInThreeParts(line)
5:         cantorSet(parts[0], y + 100)
6:         cantorSet(parts[2], y + 100)
7:     end if
8: end function
```



(a) The Cantor set

Figure 3: Pseudocode for drawing the Cantor set as seen on the right hand side

[2]Source: http://www.andylomas.com/hybridForms.html



(a) Menger sponge[3]   (b) Sierpinski Triangle[4]

Figure 4

2.1.2 The Mandelbrot set

> "The Mandelbrot set consists of all of those (complex) c-values for which the corresponding orbit of 0 under x2 + c does not escape to infinity." [6]

The mandelbrot is a very famous and heavily used fractal. In order to visualize the Mandelbrot set a complex plane[7] is used. When using a computer to calculate whether a complex number is part of the Mandelbrot set or not, it is advised to limit the number of iterations. The formula to compute the next iteration for a value is $x_{n+1} = x_n^2 + c$. If the resulting values are smaller than a defined threshold, they're considered as part of the Mandelbrot set. Many examples of the Mandelbrot set use a black pixel to mark a complex number that is part of the Mandelbrot set and a white pixel to mark a point that isn't. Assigning a specific colour to the number of iterations a complex number needed to surpass the threshold — as seen in figure 5 — leads to a very colourful visualisation that many people consider generative art.

The Mandelbrot set has an infinite level of detail. This can be explored by zooming into the visualisation of the set. Furthermore, the structures that emerge by zooming in look very similar to the fully zoomed out state. For the same zoom level, the Mandelbrot set will always look the same, because it won't change if a value is a member of the Mandelbrot set or not. This makes this approach fully predictable.

[3]Source: https://de.wikipedia.org/wiki/Datei:Sierpinski-Trigon-7.svg
[4]Source: https://commons.wikimedia.org/w/index.php?curid=7818920

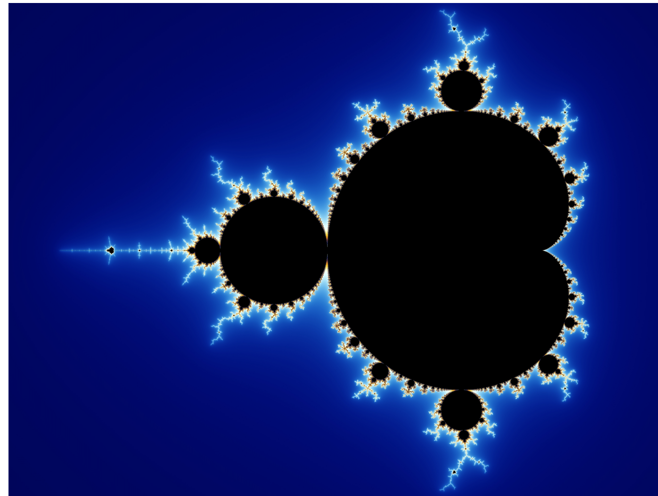


Figure 5: The Mandelbrot set. Black pixels are part of the Mandelbrot set. Other pixels are colored based on how many iterations it took to surpass a threshold[5]

2.1.3 Mandelbulb

Using the same method shown in 2.1.2, but mapping the complex numbers to spherical coordinates, instead of a complex plane can result in a 3D object called "Mandelbulb", as seen in figure 6. The Mandelbulb has the same attributes regarding predictability and detail as the Mandelbrot set.

Figure 6: Rendering of a mandelbulb[6]

[5]Source: https://en.wikipedia.org/wiki/File:Mandel_zoom_00_mandelbrot_set.jpg#/media/File:Mandel_zoom_00_mandelbrot_set.jpg

2.1.4 Lindenmayer System

Lindenmayer systems ("L-systems") are grammar-based systems, that create new words using substitution. L-systems were originally found by Aristid Lindenmayer in order to describe cell growth of plants. An L-system consists of a set of variables, a set of constants, a starting word ("axiom") and production rules on how to replace variables with different variables or constants. In a computer program, the words are usually represented by strings. Each character of the string represents a variable or constant. For each iteration, the characters get replaced according to the rules. In generative art it is common to use L-systems in combination with a set of instructions, like "draw", "stop drawing", "move x pixels to the right" or "rotate 30 degrees clockwise", which are often refered to as "turtle graphics" [7]. In order to use an L-system as an approach for generative art, each variable and constant is mapped to an instruction, for example, the constant R could tell a program to move a fixed amount to the right. This is heavily used for organic structures like plants, as seen in figure 7

There are several variations for L-systems. A context-free L-system takes only one variable at a time and replaces it according to one fixed rule. The chosen rule of a context sensitive L-System depends not only on one variable but on its possible predecessors and successors. Both of these are fully predictable. In addition, there are L-systems using a stochastic grammar, which consists of multiple replacement rules for the same variable and a probability for the application of each rule. This approach is a little less predictable and aims at creating more organic looking structures. Although the last method promises less predictability than the first two, most results still have a low level of variance, because usually there are only a few rules per variable.

[6] Source: `https://upload.wikimedia.org/wikipedia/commons/a/a0/Power_8_mandelbulb_fractal_overview.jpg`

[7] `https://en.wikipedia.org/wiki/Turtle_graphics`

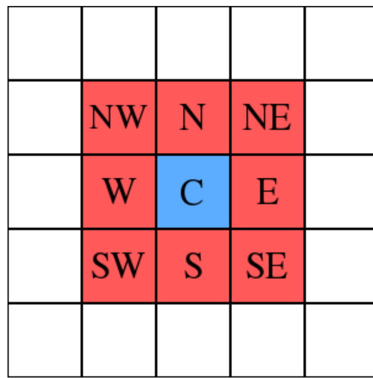
Figure 7: Plants drawn with a Lindenmayer system[8]

2.1.5 Cellular automata

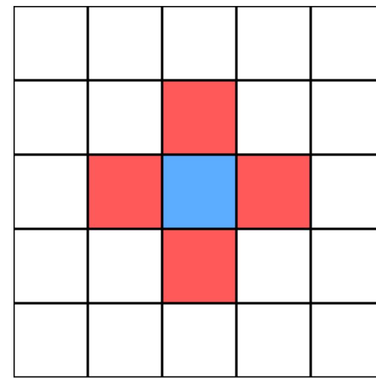
Cellular automata consist of a grid of cells. Each of these cells has a state, for example on or off. The overall state of the cellular automata is defined as the sum of the cell's states. In addition, there are rules on how to change a cell's state. These rules take the cell itself and it's neighbours into consideration. There are several definitions of neighbourhood for cellular automata. Most common are the Moore neighbourhood, as seen in figure 8a and the von Neumann neighbourhood, as seen in figure 8b. An example of a cellular can be seen in figure 9a. Figure 9b shows an example for the usage of cellular automata in architecture.

The predictability for cellular automata differs significantly for different rules. In his book "A new kind of science" [8] Stephen Wolfram introduced four classes for cellular automata that define their complexity, with the highest class consisting of cellular automata, that require a very large amount of steps to reach a stable state. Nevertheless, cellular automata are classified as a more predictable method, because if used as an approach for generative art most patterns look similar and the changes are visually not very significant.

[8]Source: `https://en.wikipedia.org/wiki/L-system#/media/File:Fractal-plant.svg`

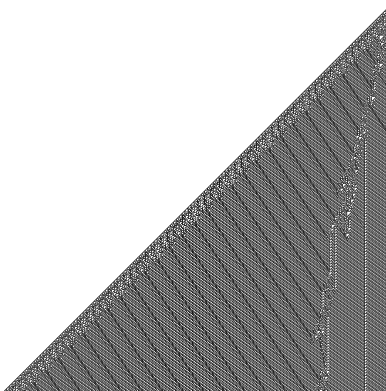


(a) Moore neighbourhood[9]



(b) Von Neumann neighborhood[10]

Figure 8



(a) Cellular automata created using the "Rule 110"[11]

(b) Cambridge north[12], a railway station in Chesterton. The window tile desig is inspired by "Rule 30"[13]

2.1.6 Voronoi diagram

Voronoi diagrams consist of a given set of points ("seeds") and according Voronoi cells, as seen in figure 10b. A Voronoi cell consists of all points that are closer to the encapsulated seed than to any other seed. Voronoi diagrams are widely used in many scientific

[9]Source: https://de.wikipedia.org/wiki/Moore-Nachbarschaft#/media/File:Moore_neighborhood_with_cardinal_directions.svg

[10]Source: https://de.wikipedia.org/wiki/Von-Neumann-Nachbarschaft#/media/File:Von_neumann_neighborhood.svg

[11]Source: https://en.wikipedia.org/wiki/Rule_110#/media/File:CA_rule110s.png

[12]Source: https://en.wikipedia.org/wiki/Cambridge_North_railway_station#/media/File:Cmglee_Cambridge_North_front_night.jpg

[13]https://en.wikipedia.org/wiki/Rule_30

and creative fields, such as procedural content generation and architecture, as seen in figure 10a. For the same set of seeds the Voronoi diagram will look exactly the same for each time it's rendered. Therefore, it's fully predictable. One could argue, that there is potential for unpredictability in the random distribution of the seeds, but that doesn't change the overall look and feel of a Voronoi diagram. This is why Voronoi diagrams are categorized as more predictable.

(a) Alibaba group Headquarters in Hangzhou[14].

(b) Multicolored Voronoi cells[15]

Figure 10

2.1.7 Particle systems

Systems that consist of many small objects ("particles"), that are emitted from a source are called particle systems. Apart from the source of emission, a particle system consists of a set of rules for each particle that affect lifespan, speed, colour and movement among many others. Particle systems are often used for effects that require are very fine granularity like fire, as seen in figure 11 or smoke. Most particle systems are tailor-made for a certain purpose like visualising fire or smoke. Therefore it's desired, that the particle system is predictable.

[14]Source: `https://de.wikipedia.org/wiki/Alibaba_Group#/media/File:Alibaba_group_Headquarters.jpg`

[15]Source: `https://en.wikipedia.org/wiki/Voronoi_diagram#/media/File:Euclidean_Voronoi_diagram.svg`

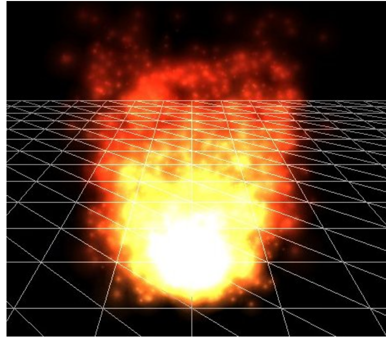


Figure 11: Fire effect that consists of particles[16]

2.1.8 Evaluation of more predictable methods

Until this point, it has become clear, that there are many techniques for creating generative artworks that may lead to interesting and aesthetically pleasing results. However, most of the techniques have already been used many times in generative art projects. Therefore it is not given, that using one of these techniques may result in the creation of something new and unique. In addition, most techniques still require a lot of work in order to reach a high level of detail. For example, for a very organic looking object created by an L-System, one would have to define many different rules.

2.2 Less predictable methods

The less predictable methods provide an approach to generative art, that is less predetermined than the more predictable methods. Noise functions, that are introduced in 2.2.1, provide an example of the combination of randomness and control, that leads to interesting results with comparatively low effort. In 2.2.2 it is shown, how processing data can lead to a high level of detail and variance. Finally, genetic programming is introduced in 2.2.3. Genetic programming shows great potential to create many artworks of high variance at the same time and let them increase in quality iteratively. In 2.2.4 it is discussed why the less predictable methods are favoured as techniques, that are used within this paper.

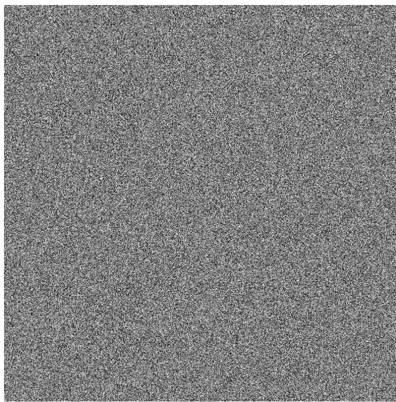
2.2.1 Noise functions

Noise functions are heavily used by generative artists when (pseudo-)random numbers come into play. A noise function takes in a given number of parameters. The exact number of parameters depends on the dimension, that the noise function is used for. For example, a two-dimensional noise function takes in two paramters. Instead of returning any random number, a noise function returns similar numbers for parameters, that are

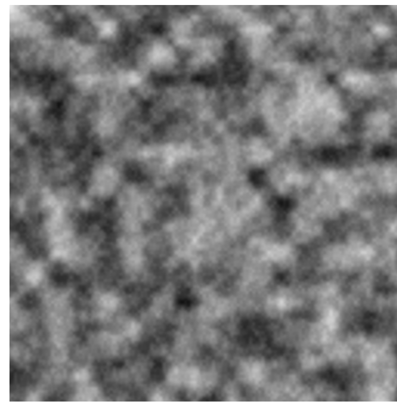
[16]Source: `https://upload.wikimedia.org/wikipedia/commons/f/f2/Particle_sys_fire.jpg`

close to each other. In the example of a two-dimensional Perlin noise, this is done by seeding a two-dimensional grid with pseudo-random gradient vectors. Given that the parameter values belong to a point anywhere in this grid, the algorithm determines a grid cell that encapsulates the point. The grid cell's corners are the four gradient vectors, that are closest to the point. At first, the vectors from the corner points to the given point are calculated. Then, the dot product between these vectors and the gradient vector at each corner is taken. The final value is an interpolation of these four dot products.

Noise functions provide a great example for a level of unpredictability that is desirable for a generative artist. Although it is not predictable which exact numbers are returned by a noise function, the artist can still control the function by passing parameters in, which is impossible for conventional random functions. As seen in figure 12 this leads to results, that feel more organic than conventional random functions. The most known and used noise functions are the Perlin noise[17], which was developed by Ken Perlin in order to create special effects for the movie "Tron" and it's successor Simplex noise[18].



(a) The brightness of each pixel is determined by conventional random function



(b) The brightness of each pixel is determined by a noise function, based on the pixel's coordinates

Figure 12

2.2.2 Data-driven approaches

As the name suggests, a data-driven approach turns a set of data into an artwork. Mainly, there are two paths of gathering the data. Firstly, there is the use of public data sources, like social media streams, data on flight information, as seen in figure 13a, or data that can be gathered from public APIs, as seen in figure 13b. Secondly, the data can be collected or created especially for the project. This may be the case for live

[17]`https://en.wikipedia.org/wiki/Perlin_noise`
[18]`https://en.wikipedia.org/wiki/Simplex_noise`

performances, where a sonic output may lead to visual representations, as seen in the project “Flume: The infinity prism tour”[19].

One of the reasons to use a data-driven approach is the connection between the artwork and its surroundings. Another reason is that huge amounts of data can be gathered in a comparatively short span of time. This allows an artist to create artworks with a large amount of detail and variance, without the need to handcraft everything that is used in his artwork. Furthermore in most cases the data an artwork relies on isn't static, but changes from time to time. This leads to artworks that change from time to time.

(a) “Flight Patterns” by Aaron Koblin “Flight Patterns” by Aaron Koblin[20]

(b) “Taxi, Taxi” by Robert Hodgin & Jonathan Kim. It uses New York City's data on taxis to create visuals[21]

Figure 13

2.2.3 Genetic Programming

The goal of genetic programming is to mimic Darwin's survival of the fittest [9]. This is desirable when there is not a clear solution to a problem but a way to measure the quality of a solution. In genetic programming possible solutions are called individuals that are grouped into generations. A genetic algorithm transforms these generations in order to make them more suitable as a solution to the given problem. The more suitable a solution is, the higher is it's fitness value. In order to determine the fitness value, genetic algorithms make use of a custom-made fitness function.

The fitness function usually returns a number for each individual. The number can either represent the amount of error and the algorithm's goal is to minimize it, or the

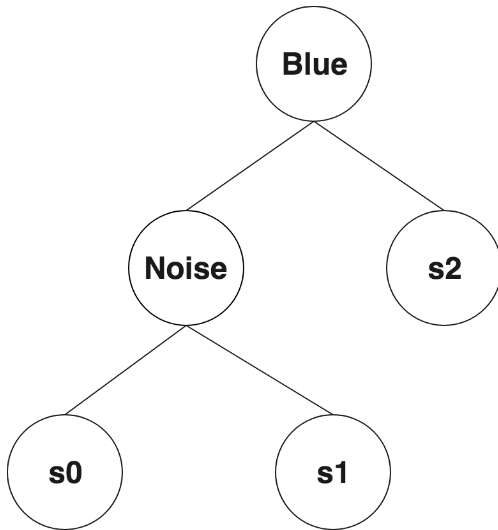
[19] `https://www.tobyandpete.com/work/#/flume-prism-tour/`
[20] Source: `http://roberthodgin.com/portfolio/work/taxi-taxi/`
[21] Source: `http://www.aaronkoblin.com/project/flight-patterns/`

amount of success. In this case, the algorithm will favour individuals with a higher number. In genetic programming, there are multi-objective problems and single-objective problems. The fitness value of a solution to a multi-objective problem is accumulated from multiple criteria. The fitness value of a single-objective problem only consists of one indicator. Since the fitness value for this project will be related to the voting results only, the focus of this thesis lays on on single-objective strategies.

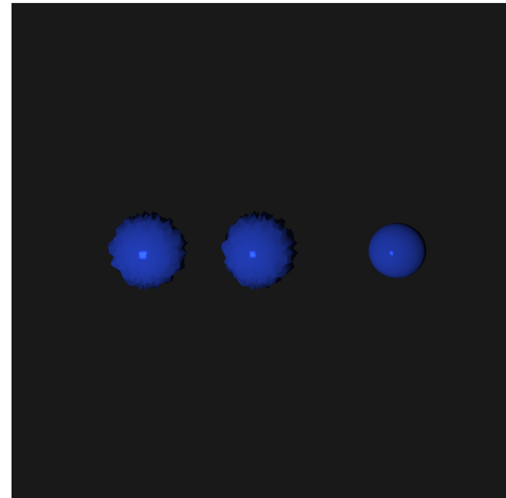
The outcome of a genetic algorithm is usually split into two parts: The genotype, as seen in figure 14a and the phenotype, as seen in figure 14b. The phenotype is the possible solution for a problem. For example, if the genetic algorithm is used for generative art, the artwork is the phenotype. The genotype is the representation of that very artwork. A common procedure for genetic algorithms is to use the genotype for initialization, selection, crossover and mutation, which will be discussed further below, and the phenotype for the fitness function. Although there are other ways to structure a genotype, most algorithms use trees. This thesis will also focus on trees as a data structure.

The nodes of a genotype's tree are either one of the following: Variables and constants, which are called terminals, or functions that are applied on the terminals. The sum of all available terminals and functions is called primitive set. Terminals or usually the leaves of a tree, while functions make up the rest of the tree.

A genetic algorithm usually goes through four phases: Initialization, Selection, Crossover and Mutation, as seen in figure 15. For each of these phases, there are many algorithms. In order to create a successful solution, the algorithms have to be picked wisely. In the following, strategies for each phase of the genetic algorithm will be presented.



(a) A tree representation of an artwork's genotype. “Blue” and “Noise” are functions that alter the appearance of the terminals. The terminals s0, s1 and s2 are spherical primitives. In a scene graph, one would expect nodes that determine the sphere's positions. For the sake of simplicity, these are left out in this example.



(b) The phenotype: The actual artwork, that belongs to the tree representation.

Figure 14

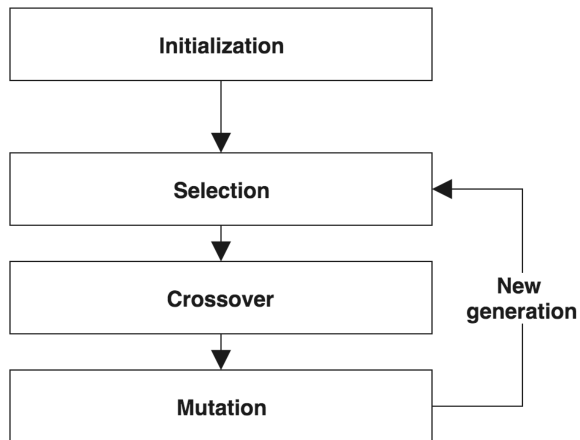


Figure 15: The stages of a genetic algorithm

Initialization During the initialization process, the first generation will be set up. In the following different algorithms will be presented that can be used to create an individual's genotype. In order to set up the whole generation, this will be repeated, until the generation is completed. For most use cases it is desired, that the initial generation doesn't dictate a direction for following generations. For example, if a high percentage of individuals within the initial generation have the same genotype, most selection algorithms may favour it over a solution with fewer members in that same generation, although it may have equal or even slightly higher fitness. Therefore, it is advisable to pick an algorithm that promises a high level of variance. The following algorithms will be presented in tables, that contain information on the algorithm's procedure, as well as advantages and disadvantages of the algorithm.

	Full method
Algorithm	In order to use the full method, one has to define a depth limit for the tree. Until the depth limit is reached, the algorithm will pick a random function from the primitive set. If the depth limit is reached, the algorithm picks a random terminal.
Advantages	The full method is advisable if all the trees need to have the same depth. If the arity of every function in the primitive set is the same, the trees will even have the same number of nodes. Furthermore, it's easy to implement and use.
Disadvantages	For many use cases, it's not needed that the depth of each tree is the same. If so, this technique may unnecessarily limit the variance of the initial population.

	Grow method
Algorithm	The grow method works very similarly to the full method. The only difference is that a terminal can be chosen before the defined depth is reached.
Advantages	The grow method, like the full method, is very easy to implement and use. Additionally, it adds more variance to the initial generation, because trees may have different depth levels.
Disadvantages	If there are significantly more terminals than functions, the trees tend to get very short, although a high depth level may have been defined.

Ramped half and half	
Algorithm	The ramped half and half approach uses the full method to generate the first half the initial generation and the grow method for the second half.
Advantages	A higher level of variance than the full method, while having at least the half of the generation reaching the depth limit.
Disadvantages	-

Seeding	
Algorithm	For the seeding method, the developer adds premade individuals to the initial generation. This can be combined with other algorithms. For example, 90% of the initial generation may be generated by a ramped half and half algorithm, while the other 10% are made up of premade solutions.
Advantages	This method is good in order to prove a solution against other solutions. For example, the winner of a previous run may be seeded into a new run.
Disadvantages	It is possible that the premade solutions are too dominant and prevent the genetic algorithm from finding new solutions that may have been found if the seeded solutions wouldn't have been part of the initial generation.

Selection The selection step is usually the first step, that is taken in order to compute a new generation. During the selection phase, individuals are chosen based on their fitness values. Two chosen individuals are meant to be the parents of a new individual, that will be part of the following generation. The actual recombination of the parent's genes in order to compute the new individual is usually done in a separate step, that will be discussed further below. Since two parents are needed in order to compute a new individual, the selection phase needs to select twice as many individuals as required in the following generation. A common pitfall regarding selection algorithms is to put too much weight on the fitness value. This may lead to generations that consist of only one solution after very few iterations. In the following the two most common algorithms are discussed in the same format as the initialization algorithms.

Tournament selection	
Algorithm	A predefined number of individuals is picked randomly from the current generation. Like in a tournament, individuals are paired up against each other. The higher rated individual reaches the next round until one individual wins the tournament. In order to create a new individual for the succeeding generation, it needs two tournaments — one per parent.
Advantages	Easy to implement, individuals with average quality have a chance to be picked.
Disadvantages	Since the individuals for the tournament are picked randomly there is a chance for a good solution not to be picked.

Roulette-wheel selection	
Algorithm	Each individual has the chance of $\frac{fitness_i}{\sum_{j=0}^{n} fitness_j}$ to be picked as a parent. It is comparable to a roulette table with unevenly sized fields.
Advantages	Since only one value has to be chosen randomly, the roulette selection is very fast and very easy to implement.
Disadvantages	A member with a very high fitness value may be chosen multiple times, which can lead to a small level of variance.

Crossover During the crossover process, new individuals will be created. This is by done using the pairs of parents, which were picked during the selection phase. The new individuals consist of a recombination of their parents' nodes. In the following, the two most common algorithms for the recombination are discussed.

Subtree crossover	
Algorithm	A randomly chosen subtree of the first parent is replaced by a randomly chosen subtree of the second parent.
Advantages	Very fast and easy to implement.
Disadvantages	No evaluation of the subtrees. The subtree that caused the high fitness value may be cut off.

Uniform crossover	
Algorithm	For each node in the child tree, a coin is flipped to decide if a node from the first or the second parent should be used.
Advantages	Uniform crossover tends to create more variance than subtree crossover
Disadvantages	Needs significantly more steps than a subtree crossover. Especially for big trees.

Mutation The mutation process is usually the last stage of a genetic algorithm. In order to increase the variance, mutation is applied on a small percentage of the individuals, that are inside the generation that has been computed during the crossover phase. This is the only phase where nodes may be inserted into the generation that weren't part of it before. In the following, the most common algorithms for mutation are discussed.

Subtree mutation	
Algorithm	A randomly picked subtree gets replaced by a new randomly computed one
Advantages	Mutating a subtree may add new nodes to the gene pool, which increases variance. Quick and easy to implement.
Disadvantages	Neither the replaced subtree nor the newly generated one are evaluated. This may lead to a subtree with a high fitness being replaced by a subtree that will receive a very low fitness rating.

Point mutation	
Algorithm	Random nodes are picked and replaced by another random node from the primitive set.
Advantages	Since this doesn't only affect one subtree but potentially every node in the tree, this method promises even more variance than the subtree mutation.
Disadvantages	The algorithm requires more steps, which can be problematic for very large trees.

2.2.4 Evaluation of less predictable methods

The huge advantage of the less predictable methods over the more predictable methods is, that there is a greater potential for detail without the need of handcrafting it. Furthermore, the less predictable techniques may be used to create a series of artworks with the very same algorithm. For example, a data-driven artwork that uses social media streams may never process the same stream twice. This can be very exciting for an artist and for the user because for most static artform like painting and photography, it's impossible for the artwork to change after it's creation. However, the less predictable techniques are more difficult to control and may lead to unwanted results.

As a consequence of this research, the decision has been made to use genetic programming in order to compute the genotypes of the artworks, that will be created for this thesis. It has been shown, that the high level of variance fits the requirement to show many different artworks to a user base, that may vote for these very artworks in order to create new ones. In 3 it will be shown, that this approach already leads to interesting results.

3 Related Work

In the following, two related works will be discussed. Both works use genetic programming in order to create aesthetically pleasing objects. In 3.1 3D objects are altered and users can vote for their favourites, which leads to a new generation of objects. A similar approach is taken in 3.2, but the user votes are replaced by a programmatic way to determine the object's aesthetic value. These works give an example for how the techniques discussed in 2.2 can be used in order to generate artworks. In addition, it is shown, how the related works differ from the approach that is taken in this thesis.

3.1 "Design and Implementation of a Procedural Content Generation Web Application for Vertex Shaders"

In their paper [10] Juan C. Quiroz and Sergiu M. Dascalu write about a project that is very close to the one discussed in this thesis. For their experiment, they created a web application that displays 3D renderings and lets users vote for their favourites. After the user voted, a genetic algorithm is used to compute new renderings. Every 3D rendering consists of the same model, that is altered by different vertex shaders. Vertex shaders are used to alter the positions of a 3D model within the rendering pipeline. Each vertex of a 3D model passes the vertex shader. The input of a vertex shader is the original position. The output is the altered position. Quiroz and Dascalu use mathematical equations that are added to the original position. In their paper, they use the equation "$x/(x + z)$" as an example. During the rendering pipeline, each value of the original position will be increased by the result of this equation. As soon as the user voted, a genetic algorithm uses the favoured equations in order to generate new equations.

Although Quiroz and Dascalu's work is very close to this thesis' project, there are some differences. First of all, Quiroz and Dascalu position their work in the field of procedural content generation. What makes this different to a generative art project is that procedurally generated content is supposed to fulfil a practical purpose. In their paper, they make the proposition to use the generated content for video games. Secondly, the application doesn't accumulate the votes of all users but creates a new experiment for each user from scratch. This leads to the outcome being tailor-made for one single user. This is opposed to the goal of using the sum of user votings for the genetic algorithm in order to create something that may please mass appeal. Furthermore, for the use of Quiroz and Dascalu's web application, the user may provide a 3D model or use a predefined model of a human body. This is opposed to the goal of creating something new and unique. In addition, Quiroz and Dascalu use the genetic algorithm for the vertex shader only. This is opposed to the project discussed within this paper because it aims at computing the whole composition of the artwork with the help of a genetic algorithm. Finally, the user experience differs from the goals discussed within this thesis. The web application displays nine renderings at once and lets the user choose more than one. In their paper, Quiroz and Dascalu mention that the users weren't engaged enough by the user interface. It is desired to use their findings in order to create a more engaging experience.

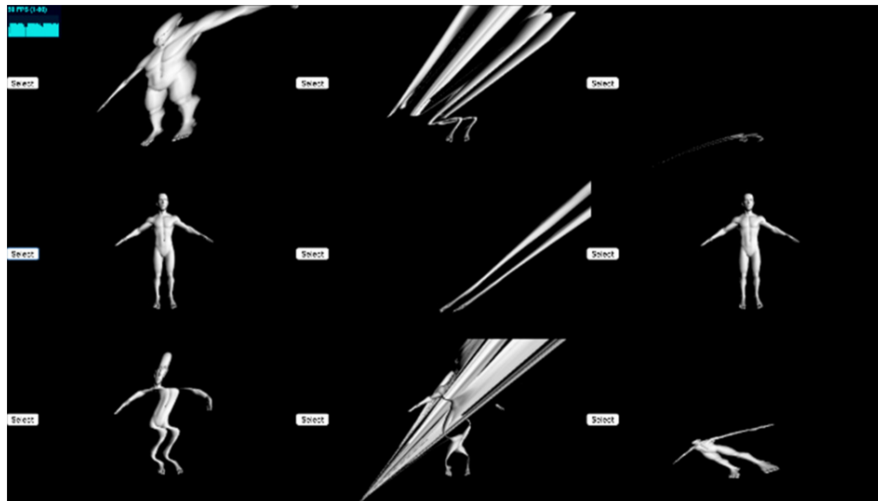


Figure 16: Screenshot of the provided user interface

3.2 Aesthetic-Oriented Generation of Architectonic Objects with the Use of Evolutionary Programming

In their paper [11] Prof. Ewa Grabska and Agnieszka Mars provide another example for the use of genetic programming and the generation of aesthetically pleasing objects.

The paper aims at creating prototypes for architectonic objects. The objects are composed of a set of solid basic objects, as seen in figure 17 and evaluated by Biederman's visual perception model [12]. Based on this evaluation a genetic algorithm computes new compositions. Like the work of Quiroz and Dascalu and like the project presented in this thesis, the ultimate goal of this paper is to create the most pleasing outcome possible. The outcoming 3D objects are composed within the course of the program. This is comparable to the goals of this thesis' project and opposed to Quiroz and Dascalu's use of predefined renderings.

What separates this work significantly from Quiroz and Dalascu's work and also from this thesis' intent is the fact, that the approach doesn't use any kind of user interaction. The fitness values are taken from the result of the computer-based evaluation only.

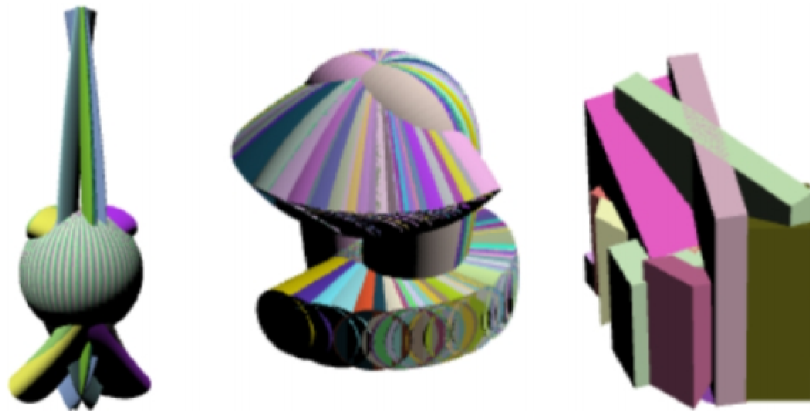


Figure 17: Example results generated by Mars and Grabska's algorithm

4 Conclusion

The purpose of this chapter was to find suitable techniques for a generative art project. Furthermore, it was desired to find examples of the recent past, that made use of these techniques. As a first step, it is needed to discuss the question "What is generative art?". This is done in 1. It has been shown that there is a need for a system, that helps with the creation of an artwork. For the sake of finding a suitable technique that can be the basis of said system, the most common techniques were discussed in 2. It has been shown, that most of these techniques aim at simulating or mimicking nature's laws. All of the shown results can lead to aesthetically pleasing artworks. However, the techniques discussed in 2.1 don't seem to qualify for a generative art project in 2018. The reasoning behind that is, that these techniques have been used too many times in the past. If an artwork is created using, for example, the Mandelbrot set, it doesn't meet the requirement of creating something new and unique. Therefore, in 2.2 the less predictable techniques were introduced. These techniques promise to enable a much higher level

of detail and variance. This leads to the conclusion, that these techniques are more suitable for reaching the goals formulated for this thesis. Furthermore, in 2.2.3 it has been shown, that genetic programming has means to create generations of artworks, that can iteratively increase their quality. This has been underlined by the related works discussed in 3. As a conclusion, for this thesis artworks are generated with the help of a genetic algorithm, that computes the individual's fitness on the basis of user votes.

Requirement analysis

In the following, the requirement analysis will be presented. In 2 the steps, that compose the creation of artworks will be discussed. In 3 the nonfunctional requirements are described. In 4 the requirements are summarized in order to reference them in the following chapters of the thesis. As stated in chapter 2.4, the system will create the artworks with the help of a genetic algorithm. The fitness value of each artwork will be determined by the number of votes it received.

1 Roles

There are two roles involved in the process of creating new artworks: The user, who votes for artworks and the system, which creates new artworks based on these votes.

2 Functional requirements

There are four functions, that the system has to provide. As seen in 1 the user can use the system by voting. This triggers the system to process the vote, which may lead to the generation of new artworks, which is separated into computing the artwork's genotypes and rendering the according phenotypes.

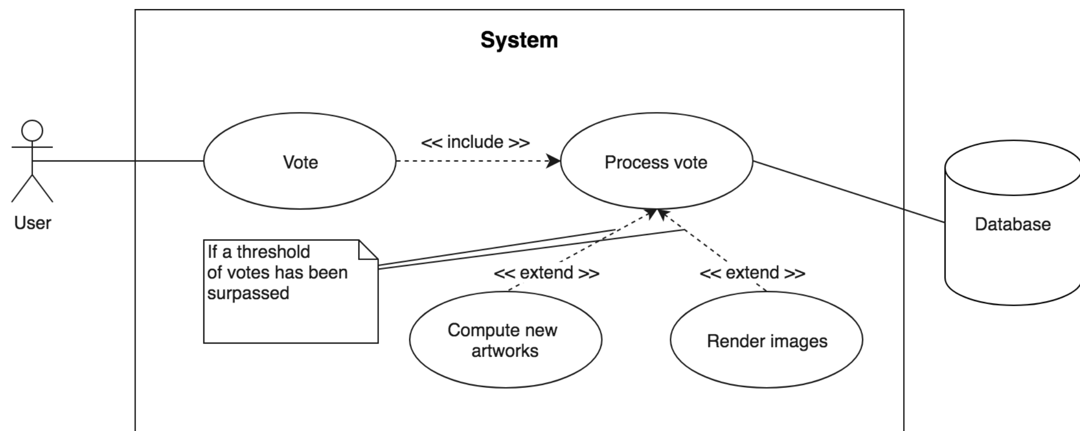


Figure 1: This use case diagrams shows the functions required in order to create new artworks

2.1 Vote

Actors: User, System The system provides an user interface as seen in 2

2.1.1 Success scenario:

1. The user interface requests a pair of images. It is made sure by the system that the user hasn't seen these images yet and that the images are from the newest generation of artworks
2. The user is asked to vote for his favourite artwork
3. As soon as the user chooses an artwork the system processes the vote
4. The user is taken back to step 1

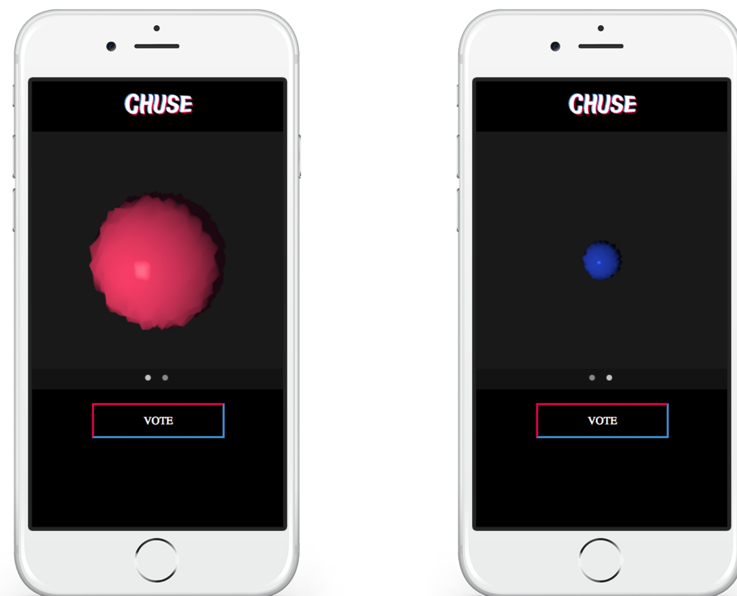


Figure 2: The user interface on a mobile phone. The images can be browsed via a swipe gesture. A touch of the button will vote for the image. Afterwards, the system will display two new images.

2.2 Process vote

Actor: System

2.2.1 Success scenario:

1. The system gets the result of the vote
2. The system increases the vote count for the winner
3. If a defined threshold of votes is surpassed, the system triggers the creation of a new generation of artworks
4. As soon as the system has computed the new generation of artworks, the images are rendered by the system

2.3 Compute new artworks

Actor: System

2.3.1 Success scenario:

1. As soon as the system triggers the computation of new artworks, the system gets the voting results
2. The system gets the current generation's genotypes
3. The system starts the genetic algorithm, which uses the provided information as a basis for the fitness function
4. The genetic algorithm computes a new generation of genotypes
5. The system saves the new generation
6. As soon as the new genotypes are saved the system triggers the rendering process

2.4 Render images

Actor: System

2.4.1 Success scenario:

1. As soon as the system triggers the rendering of new artworks, the system reads the genotypes of the new generation
2. The system renders the 3D model of each individual within the new generation
3. The rendered models are saved by the system

4. From this point on, the system will display images of the new generation via the user interface

3 Nonfunctional requirements

3.1 Accessibility

Since it's desirable to have a large number of votes in order to make better assumptions on mass appeal, the whole experience should be easily accessible for the user. As a consequence, the application shall be accessible from as many operating systems as possible. In addition, the application should be accessible from as many devices as possible. Especially mobile devices are important because it seems more convincing, that a user might want to vote in order to pass the time on a subway ride, than voting from a desktop computer at home.

3.2 User engagement

In addition to the accessibility, it is important to make the experience for the user. The reasoning behind this is to increase the time a user spends with the application, as well as the chance of the user using the application multiple times within the run of the voting period.

3.3 Variance

The system shall create generations of artworks, that are well distinguishable from each other. Therefore it is helpful to create them using an unpredictable approach. On the other hand, it is important that the approach is not completely random, because the system shall be able to embody the features, that it determined as pleasing to the users. In conclusion the function "Compute new artworks" 2.3 may use unpredictable means in order to compute new generations with a high level of variance.

3.4 Extensibility

It should be easy for another developer or artist to add a function to the primitive set, for example, a new effect that can be applied on objects. This is desirable, because this allows fast testing of new ideas for me and others.

3.5 Loose coupling

There are some steps that are necessary for almost every 3D application, like creating a rendering pipeline, setting up a data structure for the scene graph and so on. Therefore, the components shall be coupled as loosely as possible, so that the components may be reused for different projects.

4 Conclusion

Table 1 lists the discussed requirements. The next chapter will show, which design decisions were made in order to fulfill these requirements.

R1	The user can vote for artworks
R2	The system can process votes
R3	The system can compute new generations of artworks
R4	The system can render artworks
NFR1	The system is as accessible as possible
NFR2	The system is as engaging as possible
NFR3	The system produces artworks with a high level of variance
NFR4	The system is easily extensilbe
NFR5	The system is loosely coupled

Table 1

Design and implementation

In this chapter a software system is introduced, that is supposed to be a proof of concept, showing that a computer can learn to create aesthetically pleasing artworks by implementing the functionality defined in chapter 3. The system will make use of a genetic algorithm. This is done as a consequence of the research presented in chapter 2. Firstly, in 1 the reasoning for a distributed system is made. In 2 the components of this very system are introduced in detail. In 3 the infrastructure that is required to run the system is shown. In 4 an example run demonstrates how the previously introduced system works in order to create a new generation of artworks. A conclusion is given in 5 followed by an overview of the pitfalls emerged during the implementation in 6.

1 Reasoning for a distributed system

The first design decision was to build a distributed system. This is done in order to combine the accessibility of a web application and the performance of a native application written in C++. In order to achieve the nonfunctional requirement of accessibility, as discussed in chapter 3.3.1, a web application is the obvious choice: Web applications are accessible from every operating system, desktop computers and mobile devices. Furthermore, a web application doesn't require any kind of setup, which eliminates another obstacle for any user who wants to take part. This also helps the requirement of user engagement, as discussed in chapter 3.3.2. However, web applications are limited in terms of performance. In order to avoid these limits, premade 3D renderings are displayed via the user interface.

These renderings will be made by a native application that is written in C++. This enables the system to achieve a much better performance. In addition, this also adds to the fulfilment of the requirement of accessibility, because this way possible computation limits of mobile devices become irrelevant. Instead, renderings can be made on a server with state of the art GPUs, which enables the system to create artworks of a high complexity. Since it would go beyond the scope of this thesis and be very expensive to provide these servers, as well as other needed services that will be listed further below, a service provider needs to be chosen. The service provider needs to offer cloud computing solutions, that support GPU usage. This will be further discussed in 3.

2 Components

As stated in 1 the `Rendering` component and the `Web Application` will be separated within the distributed system. In addition, there is a third component for the `Genetic Algorithm`. This is done for two reasons: Firstly, this grants the freedom to choose freely between differenet genetic programming frameworks written in a arbitrary language. Secondly, this helps to achieve the requirement of loose coupling, as discussed in chapter 3.3.5. Since the voting results and the renderings have to be stored online, at least a database and any kind of web storage are required. In order to avoid implementing the interaction with these services for each component, there will be a fourth component, that handles these demands. The results will be made available via an API that can be used by HTTP calls. This makes it easy to interact with the services for every component because almost every programming language has some built-in mechanics for HTTP calls. Furthermore, the components get smaller, since there is no need to add the service provider's SDK to the components. A full overview of the components is shown in figure 1.

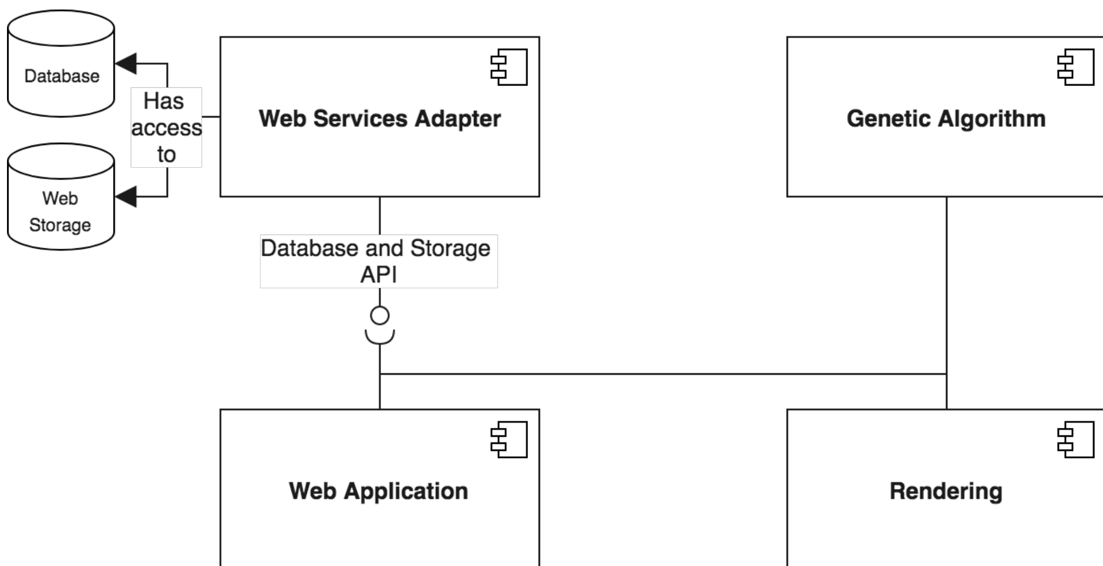


Figure 1: An overview of the components

2.1 Web Application

The main purpose of the `Web Application` is to provide a user interface, as shown in figure 2. Via the `Web Application`, users can vote for images. Therefore, the `Web Application` displays two artworks. The user can vote for one of the artworks. Afterwards, two new artworks are shown to the user.

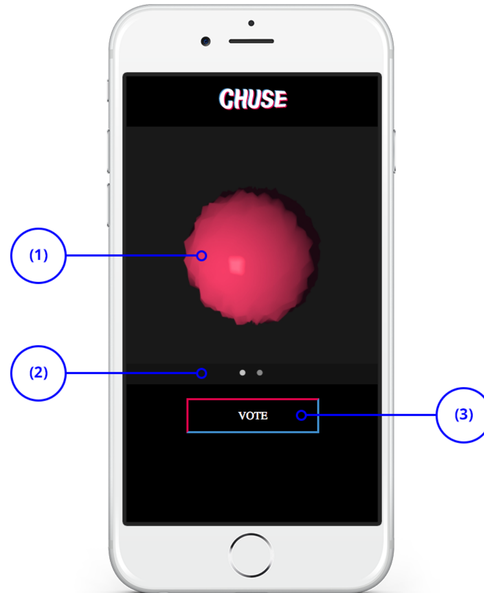


Figure 2: In (1) the artwork is displayed. Via the controls shown at (2), the user can trigger the system to show the next artwork. A tap on the button (3) triggers the vote.

2.2 Voting process

The `Web Application` is accessible via every browser on desktop computers or mobile devices. Once started, the application will request two unseen images from the `Web Services Adapter` component. In order to determine which artworks have been seen already, the `Web Application` saves the IDs of each seen artwork to the local storage of the device. An array of these IDs is sent in the body of the request. The body of the response contains two URLs of unseen artworks, that the `Web Application` will display, once received. In the next step, the user will be asked to vote for one of the displayed artworks. Once done, the `Web Application` sends another request to the `Web Services Adapter` component, that processes the vote accordingly. This will be further discussed in 2.5

2.2.1 Realization of the mobile-first approach

In order to fulfil the requirement of accessibility, the `Web Application` is designed under a mobile-first approach. That means, that every feature is built for a mobile device at first, to make sure the functionality works within the limits of a mobile device, and adapted for bigger screens afterwards. In order to make this as easy as possible, the Web Application is responsive[1]. This means, that the appearance of the user interface changes for different screen sizes.

2.3 Rendering

The `Rendering` component's task is to create 3D models. In order to do so, the component expects a scene graph as an input value. The component processes the scene graph and renders an according 3D model. Finally, the model is saved by the `Rendering` component. Naturally, this fulfils the requirement "Render images".

In addition, the rendering component is designed to fulfil two nonfunctional requirements: Firstly, the requirement of "Loose coupling". This is done by strictly limiting the component to rendering, as opposed to adding functionality that is part of creating new generations of artworks. This makes the `Rendering` component reusable for any kind of generative art project that uses 3D graphics. Secondly, in order to fulfil the requirement of "Extensibility", a custom-made scene graph was implemented, that is based on inheritance. It will be shown, how this leads to an easy entry point for artists or developers, who want to add new nodes to the scene graph.

2.3.1 Choosing a framework

In order to avoid low-level tasks like setting up a window that can be used for rendering, it is advisable to use a framework that provides this features. There are multiple choices for anyone who is interested in any kind of creative coding. There are three frameworks that are frequently used: openFrameworks[2], Processing[3], which positions itself as a programming language, that runs on a JVM[4], and Cinder[5]. As discussed in 1, the decision for a distributed system was made because it promisses a better performance for the `Rendering` component. Therefore, Processing doesn't seem like a valid choice, because it works on very high abstraction level, which produces an overhead that can be avoided by choosing one of the C++ frameworks. In conclusion, openFrameworks was chosen over Cinder, because it has a larger user base. This usually leads to more ressources that may help to get to know the framework. openFrameworks uses openGL[6]

[1] `https://en.wikipedia.org/wiki/Responsive_web_design`
[2] `http://openframeworks.cc`
[3] `https://processing.org`
[4] `https://en.wikipedia.org/wiki/Java_virtual_machine`
[5] `https://libcinder.org/`
[6] `https://www.opengl.org/`

in order to render 3D graphics.

2.3.2 Loop

The application runs in a continuous loop, that has three main phases:

Setup Before the loop starts the setup phase takes place. Everything that has to be done only once, like initializing the rendering pipeline, should be done during this phase.

Update The update phase takes place every loop. During the update phase, values may change in order to change the output of the draw phase. For example, if it is desired to create an animation of an object that moves from left to right, during the update phase the object may be moved a small amount in the desired direction.

Draw The draw phase takes place directly after the update phase. During the draw phase, the 3D models are rendered.

2.3.3 Scene graph

As in most computer graphics applications, a scene graph[7] is used. The scene graph consists of multiple nodes that define the composition of the artwork. For example, a node can contain the mesh of a 3D model, as well as modifications for a 3D model that range from changing the model's appearance, changing the model's position, to changing the model's mesh by changing the position of the mesh's vertices.

Composition of the scene graph The scene graph is basically composed of five node types: The `RootNode`, the `ModificationNode`, the `ShaderNode`, the `MatrixModificator-Node` and the `ObjectNode`. All of these nodes inherit their basic functionallity from a `Node` class, as seen in figure 3. In the following all of these node types will be discussed in detail.

Apart from the `RootNode`, all of the nodes are abstract classes. In order to create a node, that is usuable within the scene graph, a node class has to be created that inherits from the `ModificatioNode`, the `ShaderNode`, the `MatrixModificatorNode` or the `ObjectNode`. Each of these classes is designed to make it as easy as possible to create a new node that is usuable within the scene graph. This helps to fulfill to the requirement of extensibility. Further below, it will be discussed in detail, how a concrete class can be created and which tasks the abstract class takes care of.

[7] `https://en.wikipedia.org/wiki/Scene_graph`

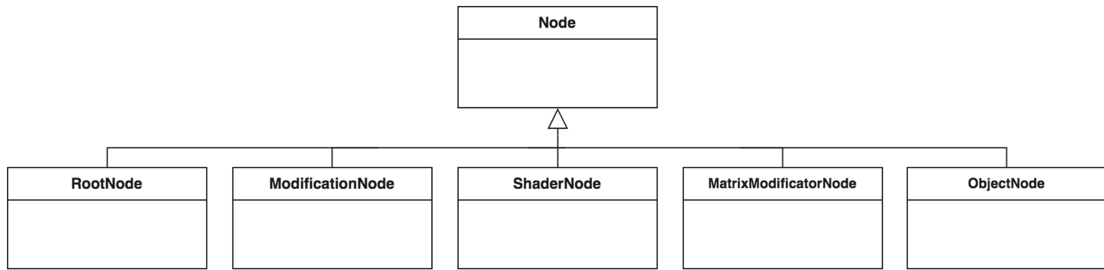


Figure 3: An UML diagram showing the inheritance between the node types

Node The `Node` class provides basic features that are needed in order to build a tree, such as adding children, getting a collection of children, getting the parent and so on. Furthermore the `Node` class provides means to identify nodes. This is done by using a predefined name for every node plus an ongoing number. For example, a `Node` that contains a sphere and was the second node to be created within the process of building the scene graph, will be named "SphereNode_02". This is useful for debugging purposes and a crucial feature for serialising the scene graph. In addition, the `Node` class provides methods for other nodes to register as an observer. There are hooks that notify observers right before the node is drawn, after the node is drawn, after the setup is done and so on. For example, shader nodes make use of these hooks in order to know when to `begin()` or `end()`. This will be discussed in detail further below. Every class that inherits from `Node` needs to implement the methods `draw()` and `setup()`.

RootNode The main purpose of the `RootNode` is to mark the root of each scene graph. Each scene graph should consist of exactly one `RootNode`. This is helpful for internal methods, that want to process the whole scene graph.

ObjectNode The `ObjectNode` is the base class for every node that contains a 3D object. An `ObjectNode` will be initialized with the required information. For example, a `SphereNode` will receive its radius and its resolution. The `ObjectNode` contains a model of the 3D object, that can be requested by other classes. The `ObjectNode`'s `setup()` method initializes the mesh that is within the node. The `ObjectNodes`'s `draw()` method renders the mesh.

ModificationNode `ModificationNodes` are used to change the meshes of nodes in it's subtree. In order to get every mesh in the subtree, a `ModificationNode` sends a request for a set of models to all of its child nodes, as seen in figure 4. If a node receives the request, there are two possible responses: If the node is an `ObjectNode`, it adds its model to the set, if it's any other node, it just passes the request to all of its children. Once all models are collected, the set is returned back upwards to the original request. For this, there are also two possibilities: If the node is another `ModificationNode`, it will modify every model in the set, before it passes the response further upwards. If it's

any other node, it will just pass the model further upwards.

In order to create a concrete class, all a developer has to do, is to implement the modification itself. Therefore, the abstract method `modify(...)` is defined in the `ModificationNode` class, that gets called for every model in the node's subtree and receives the model as a parameter.

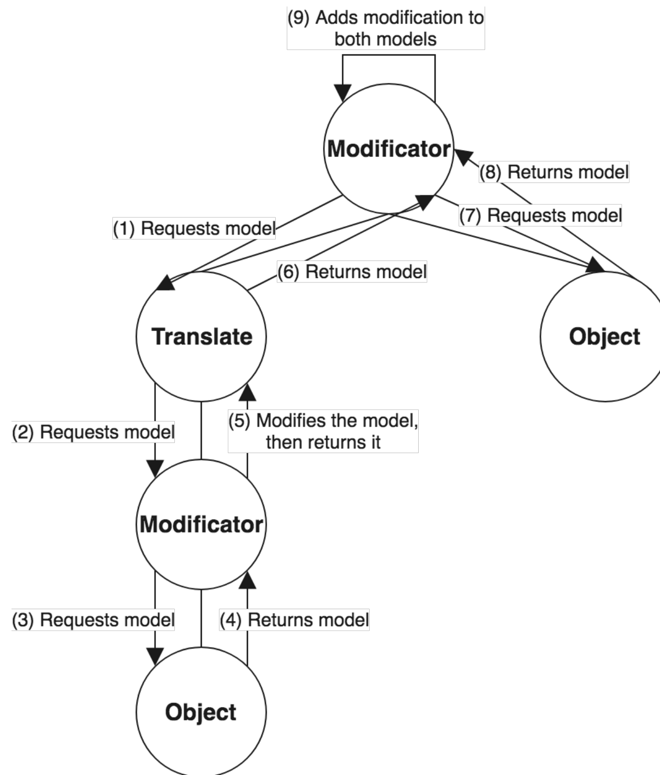


Figure 4: An example subtree containing two `ObjectNodes`.

ShaderNode As the name suggests, a `ShaderNode` encapsulate a shader. A shader is a programm that is executed on the GPU of a computer. It is part of openGL's rendering pipeline to apply shaders if specified. Therefore, a shader is activated by calling it's `begin()` method. From this point on, everything that will be drawn is affected by the shader. Each shader has an `end()` method, that stops the effect of the shader. Since shaders are usually stored in separate files, the `ShaderNode` takes in the name of the shader. During the setup phase, it will load the according shader from a predefined location.

The `ShaderNode` is designed to begin the shader's effect before it's subtree is drawn and end it, after the all of the leaves are processed. Therefore, the `ShaderNode` registers

itself on every leaf in its subtree. When the leaves are done drawing, the `ShaderNode` gets notified. Then, it increases the count of drawn leaves. If this equals the total number of leaves, the `ShaderNode` ends the shader.

All an inheriting node has to do, is to provide the name of the shader. The `ShaderNode` expects to find a directory in a configurable location, where the shader's files can be found. Once this is done, the shader will be loaded, during the setup phase, begin before it's subtree is processed and end, once everything in its subtree is drawn. A shader can receive arguments from the program that uses the shader. These arguments are called uniforms. If a developer wishes to pass uniforms into the shader, this can be done by implementing the method `setupUniforms()` within the inheriting class.

A mentionable concrete class is the `MaterialNode`. As the name suggests, the `MaterialNode` defines a material. A material defines the appearance of an object's surface. This is crucial in order to expose the object. Since most lighting algorithms, such as the Blinn-Phong shading model[8], require three colours, the `MaterialNode` consists of these three colours: The ambient colour, the diffuse colour and the specular colour. It is crucial, that every `ObjectNode` is in the subtree of at least one `MaterialNode`, because otherwise the object would look flat.

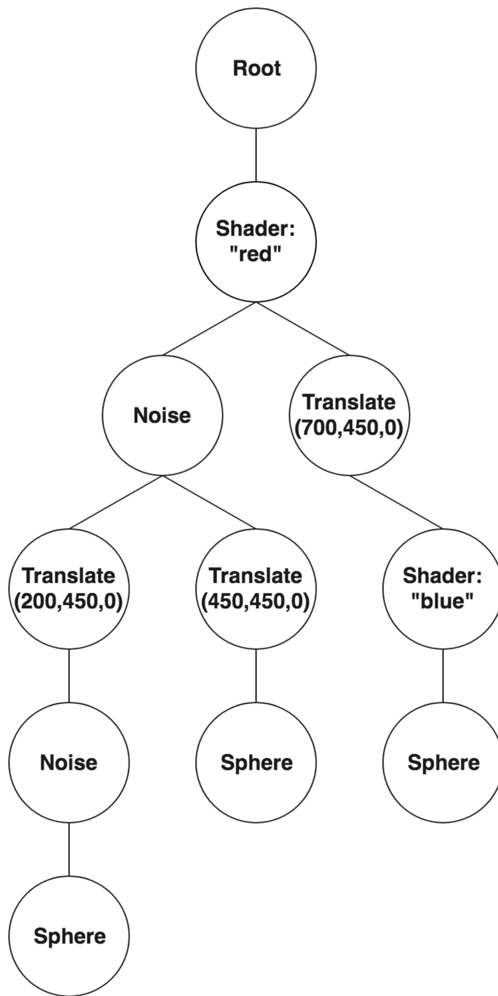
MatrixModificatorNode The `MatrixModificatorNode` encapsulates changes to the ModelViewProjectionMatrix[9]. This is desirable, everytime a model is supposed to be translated, scaled or rotated. A `MatrixModificatorNode` applies the changes to every object in its subtree. Before applying the modifications, the current ModelViewProjectionMatrix gets pushed onto a stack. After every leaf has been drawn, the `MatrixModificatorNode` pops the previous ModelViewProjectionMatrix back from the stack. In order to create a matrix modification an inheriting node needs to implement the method `motdifyMatrix()`. Within this method, the changes to the ModelViewProjectionMatrix may be done.

Combining nodes Finally, the scene graph comes together by combining nodes. In order to keep the scene graph flexible, there should be as few constraints regarding the possibilities of combination as possible. As a matter of fact, the only constraint is, that the scene graph starts off with a `RootNode`. Some combinations may make no sense. For example, a `MatrixModificatorNode` as a leaf node won't change anything, but it's still possible.

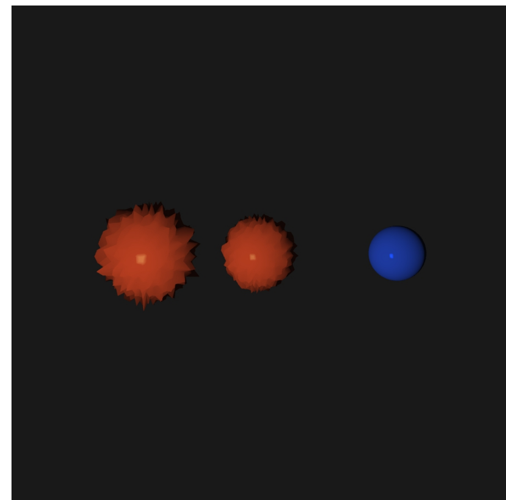
Each node affects every node in its subtree. For example, if a `ShaderNode` sets the colour to blue, every object that is a child of this node, will be blue. Some effects are added up, some override each other, as illustrated in figure 5. For example, a

[8] `https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model`
[9] `http://www.codinglabs.net/article_world_view_projection_matrix.aspx`

`MatrixModificatorNode` that changes the position of an object to the centre of the window, a `ShaderNode` that sets the colour to red and an `ObjectNode` that contains the model of a sphere, will lead to a red sphere in the centre of the window. Other nodes replace each other. For example, if in the same tree a `ShaderNode` setting the colour to blue will be between the `ObjectNode` and the `MatrixModificatorNode`, the `ShaderNode` that sets the colour to red will be overridden, therefore, have no effect. However, it is imaginable that the red `ShaderNode` has another child that is an `ObjectNode`. This object will be red again.



(a) A scene graph, that consists of `SphereNodes`, `TranslateNodes`, `NoiseNodes` and `ShaderNodes`. The `TranslationNodes` move the spheres within the three-dimensional space. The `NoiseNodes` roughen up the surface of a sphere, by moving the vertices away from the center by an configurable amount that is multiplied with a 3D noise function. The `ShaderNodes` set the colour of the spheres.



(b) The sphere on the left-hand side is an example for the addition of `ModificationNodes`. This sphere is affected by two `NoiseNodes`, while the sphere in the middle is affected by only one. This leads to the left-hand sphere having bigger "spikes" compared with the sphere in the middle. The node on the right hand side is an example for overriding: The `ShaderNode` setting the colour to red is overridden by the `ShaderNode` setting the colour to blue.

Figure 5

2.3.4 Renderer

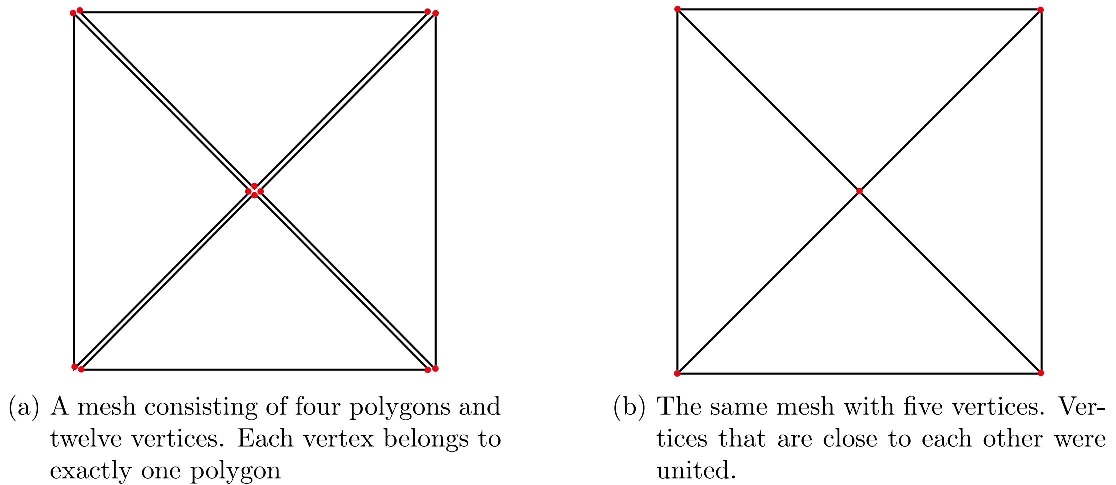
In order to render the scene graph, a `Renderer` class has been implemented. The `Renderer` traverses the scene graph using a depth-first search algorithm. During the traversal, each node will be processed, until every node was reached. During the setup phase, the `Renderer` will call `setup()` on every node, during the draw phase, it will cal `draw()` on every node. Each node has to implement the effects that these calls will have by itself. As discussed in 2.3.3 a `ModificationNode`, that was reached by the `Renderer`, triggers every other `ModificatorNode` within its subtree before the Renderer reaches these nodes. In order to avoid that the effect will be applied twice, nodes can be skipped.

2.3.5 Data structure for meshes

The `Rendering` component contains a data structure for meshes. The data structure stores vertices, edges and faces of a 3D model. The data structure provides methods to change vertices, add polygons and to get information about the mesh, like the number of faces, vertices and edges. The data is organized as a doubly connected edge list[10] ("DCEL"). The data structure can be initialized empty or by passing in an instance of the openFrameworks mesh class. If a mesh is passed in, the data structure will transform the mesh into a DCEL. In order to make use of openFrameworks' rendering methods, the data structure can be converted (back) into the mesh class that openFrameworks uses.

Under the assumption, that one has a predefined mesh that is passed into the data structure, reproduced and converted back into the openFrameworks mesh class, it may seem unnecessary to use the DCEL at all. But, there are two mechanisms that make modelling with the data structure way easier than using the raw mesh provided by openFrameworks. Firstly, it unites vertices that are very close to each other. Some meshes are organized in a way, that crossing points of polygons consist of multiple vertices, as seen in figure 6a. This makes the modification of these meshes very difficult because, in order to move the centre point of the shown square, the algorithm would need to find and move four different vertices. In order to avoid this, the data structure checks for every vertex it processes, if there is already a close vertex within the data structure. If so, the processed vertex gets dropped and its edges are connected with the already existing vertex instead. The result is illustrated in figure 6b. Secondly, the DCEL helps to identify adjacent faces, which is very useful for many modifications that may be applied on the mesh. The data structure provides methods to find neighbours of a given face. The method can be called with the face only, which will return the direct neighbours of the face. There is an optional parameter, that defines the depth. For example, if the depth is 2, in addition to the direct neighbours of the face, the neighbours' neighbours will be returned as well.

[10] `http://www.cs.sfu.ca/~binay/813.2011/DCEL.pdf`



(a) A mesh consisting of four polygons and twelve vertices. Each vertex belongs to exactly one polygon

(b) The same mesh with five vertices. Vertices that are close to each other were united.

Figure 6

2.3.6 Serialisation

Lastly, the `Rendering` component has means to serialise and deserialise a scene graph. The chosen format for the serialisation is JSON, because it is readable, which makes it easily debuggable and almost every programming language has built-in means or well-tested libraries to convert JSON strings into an object. Every node within the serialised scene graph consists of the same three key-value pairs: Firstly, there is the name of the node, that is unique and the same among all components. Secondly, there is the data of the node, which contains the information that is needed in order to reproduce the node. For example, for a sphere, the data field contains the radius and the resolution of the sphere. Finally, the last key is called children. The value of this key is an array of nodes. This is used to save the hierarchy of the scene graph.

2.4 Genetic Algorithm

As most genetic algorithms, the one implemented for this thesis works with trees as a genotype. Since every artwork is the product of rendering a scene graph, there is already a tree-like representation for each artwork. This will be further discussed in 2.4.1. This makes it very easy to transform the artwork into a form that is usable for the genetic algorithm. What makes the nodes of the tree special, is the fact, that the nodes contain key-value pairs that need to be processed by the genetic algorithm as well. This can range from a single value, for example, the intensity of an effect, to more complex structures. For example, a `MaterialNode` consists of three colours. Each of these colours is defined by using the RGB model, that requires three values. This complexity requires the genetic algorithm to handle each node differently.

The fitness of an artwork is determinable by its fitness values, which is rather sim-

ple: The more votes an artwork received, the better it's fitness value. Therefore, the fitness evaluation of each individual is completed, before the algorithm was started. What is more difficult to determine is what made an artwork more pleasing than another. In order to get a better picture of that, global tags are used. Global tags are criteria that describe attributes that influence how an artwork is composed like if an artwork is multicoloured, monochrome, or black and white, the number of objects in the artwork, the used colour families and so on. The tags are set programmatically during the initialization, in an additional field of every artwork's tree representation.

2.4.1 Transforming the scene graph into a genotype

The desired outcome of the transformation process is a genotype that is usable by the `Genetic Algorithm` component. The genotype needs to have a tree structure and consist of functions and terminals. Since the scene graph is already in a tree-like structure, all that needs to be done to transform it into a genotype is to categorize its nodes into functions and terminals. The set of terminals consists of every node that inherits from `ObjectNode`. Everything else will be handled as a function. While handling the functions, the genetic algorithm makes sure, that every genotype has a `RootNode` and every `ObjectNode` has a `MaterialNode`, because they're needed for every artwork.

2.4.2 Design of the algorithm

Initialization The initialization phase creates a new generation of individuals. The initialization algorithm works like the grow method, as shown in chapter 2.2.2.3 with some modifications. The grow method is a good choice here because it promises more variance than the full method. The grow method may produce short trees if there are significantly more terminals than functions. Since that is not the case here, this can be ignored. The maximum depth is predefined before runtime via a config file. The assembly of the tree is divided into four phases: At first, the `RootNode` is added as the root of the tree. Secondly, a random amount of terminals is added to the tree. In the third phase, for every terminal, a `MaterialNode` is added between the `RootNode` and the terminals, so that every terminal can be exposed correctly. Finally, random functions are added between the `RootNode` and the `MaterialNodes`. Since the initialization uses the grow method, the amount of nodes added to the tree in the fourth phase ranges from 0 to $MAX_DEPTH - 3$. The MAX_DEPTH is decreased by three here, because `RootNode`, `MaterialNode` and a terminal are already part of each path from the `RootNode` to one of the leaves at this point.

Selection The selection is an exact implementation of the roulette-wheel selection algorithm. The probability of an individual being chosen depends on the number of votes for the individual in relation to the total votes.

Crossover During the crossover phase, the two parents' nodes are sorted into three collections: One for the `ObjectNodes`, one for `MaterialNodes` and one for every other node. In order to avoid that the trees become too bloated, there is a maximum depth for each new tree. The recombination works similar to the uniform crossover algorithm. At first, the number of `ObjectNodes` gets computed. The according number of `ObjectNodes` is then added to the child. Secondly, for every `ObjectNode` a `MaterialNode` is taken from the parents. Finally, the child tree is filled with the remaining function nodes from the parents, until the max depth is reached, or before.

Mutation The mutation algorithm works like the point mutation algorithm. It swaps nodes with newly computed nodes. Every function node can be swapped with another fuction node, with two exceptions: Firstly, since it would have no effect, the `RootNode` isn't mutated. Secondly, since it's not desirable to substitute a `MaterialNode` with any other node, the mutation algorithm will replace them with new `MaterialNodes`, that may differ in colour. The terminals can be swapped with different terminals.

2.5 Web services adapter

The `Web Services Adapter` is made to bundle functionality, that at least two of the other components need. There are two benefits gained by this. Firstly, the functionality has be implemented only once. Since the components are written in different programming languages, it's not possible to copy and paste functionality or make it available as core libraries for the project. Therefore, the functionality is made available via HTTP calls. Secondly, the components get smaller and more manageable, because the service provider's SDK doesn't need to be added to the other components. Every component, that wants to use the web services, needs to be able to make HTTP calls, which is usually a comparatively easy task.

The main task of the `Web Services Adapter` is to process the votes. As stated in chapter 3.2.2, this may lead to the generation of new artworks. Therefore, the `Web Services Adapter` may trigger the `Genetic Algorithm` and the `Rendering` component. Furthermore, the `Web Services Adapter` handles everything, that has to do with saving and reading from the database or web storage. This is especially needed by the `Genetic Algorithm` in order to get the genotypes of the current generation and to save the newly computed next generation.

2.5.1 API

The methods provided by the API can be accessed by HTTP calls. Every method defines a "route" within the API. The functionality of many routes is the same, that one would expect from basic `get` and `set` methods. For example, they may be used to get the current generation's phenotypes as a JSON string, or the fitness value of an

individual. Therfore, they won't be discussed in further detail. The routes listed in table 1 are essential for the interaction of the components, as shown in 4.

Name	Input/Output	Description
vote	**Input:** Object that contains two IDs. One ID is marked as the winner, the other as the loser of the vote **Output:** -	In this route, the votes are processed. This ranges from increasing the vote counts for the artwork and the global tags to calling routes that trigger the run of the `Genetic Algorithm` and the `Rendering` component
pair	**Input:** Array of every artwork's ID, that the user has seen already **Output:** Array consisting of two URLs that belong to two unseen artworks	The pair route provides two URLs that belong to unseen artworks
generation	**Input:** Object containing every individual of the coming generation **Output:** -	This route is used to post new generations as genotypes. This is used by the Genetic Algorithm, after the initialization or evolution of new generations. Once saved, the `Web Services Adapter` triggers the rendering of the phenotypes.

Table 1

3 Infrastructure

In order to make the system accessible online, it has to be hosted on servers. Thus, a service provider needs to be chosen, since it's easier and cheaper to use a service provider than building a custom made solution. Apart from basic hosting services, that are offered by nearly every service provider, like web storage and databases, the system developed for this thesis has special requirements , because the `Rendering` component will need GPUs to work on. Therefore, in 3.1 the benefits of cloud computing in general are listed. Afterwards, in 3.2, it will be discussed why Amazon Web Services[11] ("AWS")

[11] `https://aws.amazon.com/?nc1=h_ls`

was chosen as a service provider. The UML diagram in figure 7 provides an overview of the system within the AWS ecosystem. The services will be further explained in 3.3. Finally, in 3.4 it is shown which additional means had to be taken in order to run the system in the provided ecosystem.

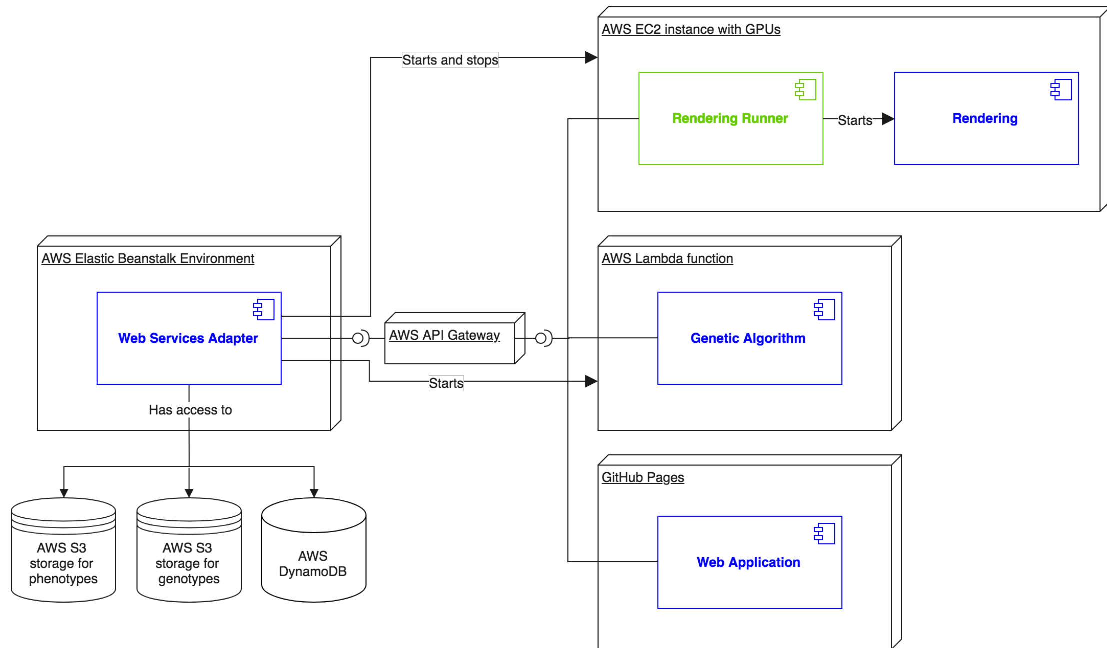


Figure 7: UML diagram showing the system's components within the AWS ecosystem. For a better overview, the already existing components are marked as blue. New components that will be introduced in this chapter are marked green. The black boxes represent AWS services, that will be discussed in detail further below.

3.1 Cloud Computing

The huge benefit of cloud computing[12] is, that there are no hard limits regarding performance and scalability. In addition, the costs of cloud computing are usually lower than renting a server, because only the time that the system is actually doing something is billed. This is very important for this project because the `Rendering` component needs GPU computing, which is very expensive.

3.2 Amazon Web Services

For this project, the main requirement for a service provider is to offer cloud computing with GPU support. In addition, a service provider is favoured that offers every service

[12] `https://aws.amazon.com/what-is-cloud-computing/?nc1=h_ls`

that is needed to run all of the components because that promises less overhead that may be produced by leaving the ecosystem and getting data into the ecosystem of another service provider. Finally, it's desirable that the services are well documented and there are already many people using them since it's helpful to have as many resources as possible in order to learn about the services and the ecosystem in general.

All of the above criteria are met by multiple service providers. It's imaginable to run the software system on services provided by Google Cloud Platform[13] just as well. The choice in favour of AWS was based on the fact, that AWS is the market leader[14], which leads to the biggest community around it.

3.3 Chosen services

For each component, a service was chosen that promised the best performance, scalability and cost control. In the following, an overview is given on which services were used per component.

3.3.1 Web services adapter

The `Web Services Adapter` is written in TypeScript[15]. Therefore, it needs to be run in an node.js[16] environment. Furthermore, it needs to be accessible via HTTP calls. The chosen service is AWS Elastic Beanstalk[17]. Apart from the cloud computing environment that supports node.js, AWS Elastic Beanstalk sets up an API Gateway[18] that handles the HTTP calls. In addition, the Elastic Beanstalk Environment provides the feature to scale the application up, by providing additional cloud computing instances and a node balancer, that divides the load between the instances. This is helpful because the `Web Services Adapter` is used by every component. If the system will be used by many people at once, no changes would need to be made in order to handle the increased traffic load.

3.3.2 Rendering component

The `Rendering` component needs to run in a cloud computing environment that supports GPU usage. AWS's Elastic Cloud Computing[19] ("EC2") provides cloud computing solutions for many use cases, with 3D rendering in the cloud being one of them. The rendering component runs on a G3[20] instance. This instance type is chosen because it has the needed GPU power, while being relatively priceworthy.

[13] https://cloud.google.com/?hl=en
[14] https://www.gartner.com/doc/reprints?id=1-2G205FC&ct=150519&st=sb
[15] http://www.typescriptlang.org/
[16] https://nodejs.org/en/
[17] https://aws.amazon.com/elasticbeanstalk/?nc1=h_ls
[18] https://aws.amazon.com/api-gateway/?nc1=h_ls
[19] https://aws.amazon.com/ec2/?nc2=h_m1
[20] https://aws.amazon.com/ec2/instance-types/g3/

3.3.3 Genetic algorithm

The `Genetic Algorithm` runs in an AWS Lambda function[21]. AWS Lambda lets applications run without the need of provisioning servers or cloud computing instances. Applications run via Lambda are called "Lambda functions". Lambda functions are triggered by other services. For this project, the Lambda function can be started via HTTP calls. Once triggered, a Lambda function executes the application for a maximum duration of five minutes. Lambda functions are a good solution for repetitive tasks, that can be executed quickly. This is applicable for the `Genetic Algorithm`, which usually takes a few seconds to compute a new generation.

3.3.4 Web application

The `Web Application` is hosted outside the AWS environment via GitHub Pages[22]. This is done because the source code is hosted on GitHub. The `Web Application` doesn't need to interact with anything, but the API Gateway for which it makes no difference, if a call comes from inside the AWS ecosystem, or not. The benefit of GitHub Pages is, that once the source code is updated, the new version of the `Web Application` is available immediately without the need for an additional deployment.

3.3.5 Database and storage

For the database, AWS's NoSQL[23] solution DynamoDB[24] is used. A NoSQL solution is suitable because the data is already available in a JSON format, which can usually be passed into a NoSQL database without any further transformations.

Since the genotypes are relatively large and the information on the scene graph is not needed until the point where the genetic algorithm gets triggered, the database contains the ID, the global tags and a numeric fitness value for each individual. The full genotype is stored in additional web storage. Therefore, the Simple Storage Solution[25] ("S3") was chosen. Within the S3 service, virtual storage units can be created, that are called "buckets". In addition to the bucket that contains the genotypes, there is a bucket where the phenotypes are saved.

3.4 Putting components into service

In order to run the system within the AWS ecosystem, some additional functionaliity had to be implemented. This regards the `Rendering` component especially. As discussed earlier, the `Rendering` component is supposed to be loosely coupled. Therefore, the

[21]`https://aws.amazon.com/lambda/?nc2=h_m1`
[22]`https://pages.github.com/`
[23]`https://aws.amazon.com/nosql/?nc1=h_ls`
[24]`https://aws.amazon.com/dynamodb/?nc2=h_m1`
[25]`https://aws.amazon.com/s3/?nc2=h_m1`

`Rendering` component itself makes no calls to any other service. Since it's needed to trigger the rendering process automatically, within the EC2 instance there is an additional component: The `Rendering Runner`. The `Rendering Runner` is started, as soon as the EC2 instance boots. This is done, once the `Genetic Algorithm` component has computed the genotypes of the next generation. In order to trigger the rendering of the phenotypes, the `Rendering Runner` downloads the genotypes to the hard drive of the EC2 instance and starts an executable of the `Rendering` component, that takes in the JSON string of the genotypes as an argument. Then the `Rendering` component itself saves the phenotypes locally. Once this is done, the `Rendering Runner` uploads them to a S3 bucket. Finally, it is made sure by the `Rendering Runner`, that the EC2 instance gets shut down again, in order to avoid paying additional minutes, although the `Rendering` component is idle.

4 Example run

Putting it all together, in figure 8 an example run of the system is shown. It all starts with the `Web Application` requesting an array of two URLs from the Web Services Adapter, which are then displayed. Afterwards, the system waits for the user to vote. Once the user voted, the vote is processed within the `Web Services Adapter`. Under the condition, that a given threshold was surpassed, the `Web Services Adapter` will send an HTTP request to the Lambda function containing the `Genetic Algorithm`, which triggers its execution. Once the execution is done, the Web Services Adapter gets notified. As a final step, the `Web Services Adapter` starts the EC2 instance that contains the `Rendering` component. Within the EC2 instance, the `Rendering Runner` gets started by a hook within the operating system's booting procedure. The `Rendering` Runner executes the `Rendering` application. Once the application exited, the `Rendering Runner` uploads every phenotype to an S3 bucket. Once this is done, the `Rendering Runner` notifies the `Web Services Adapter`. From this point on, the `Web Services Adapter` will provide phenotypes of the new generation when the `Web Application` requires them.

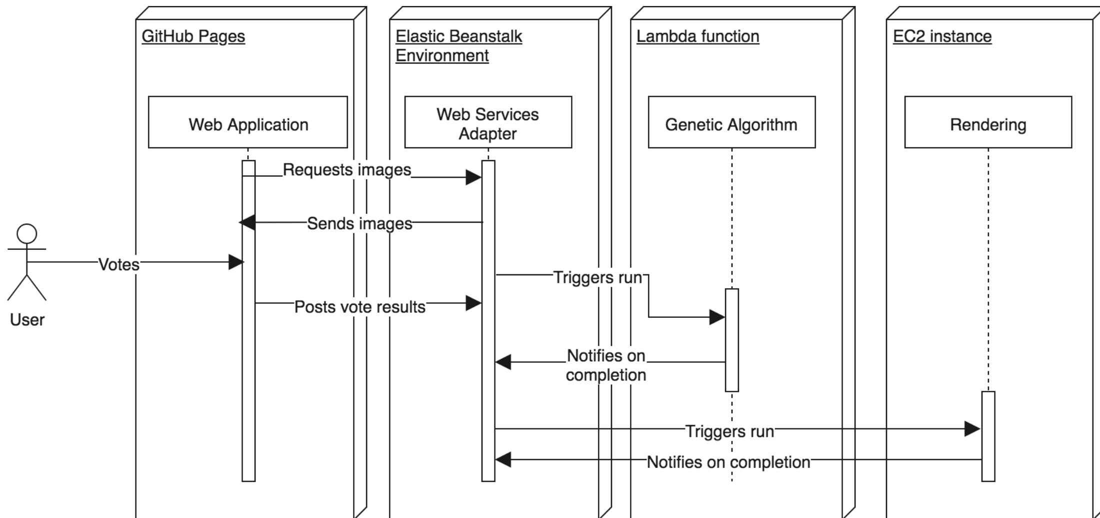


Figure 8: An interaction diagram illustrating the steps that lead to new artworks

5 Conclusion

The intent of the work discussed in this chapter was to fulfil the requirements set in chapter 3. First of all, in 1 it is shown, that a distributed system is a good choice in order to create a software system that fulfils the requirement of accessibility, without worrying about a possible lack of performance. In 2 the four components of the distributed system are introduced. Each of these systems fulfils one functional requirement. In addition, it is shown, how the design of the components leads to the fulfilment of nonfunctional requirements. In 2.1 it is shown, that the `Web Application` fulfils the requirement "The use can vote for artworks". Furthermore, in 2.3 it is shown, how "The system can render artworks" is fulfilled. In addition, the introduction of the scene graph fulfils the nonfunctional requirement of extensibility. Since another requirement aims at loosely coupled components, the `Rendering Runner` was implemented, as described in 3.4. To fulfil the requirement of creating artworks with a high level of variance, a genetic algorithm is used, as discussed in 2.4. Finally, 2.5 gives an overview of the `Web Services Adapter`. In addition, the routes needed to to fulfil the requirement "The system can process votes" are discussed in further detail.

Since time was running out during the end of the project, the `Web Application` is not as engaging as planned. The reason for this are some unexpected pitfalls, that will be discussed in 6. An outlook on what yet has to be implemented is given in chapter 5.1.

6 Lessons learned

In the following, the pitfalls that emerged during the design and implementation will be presented. Although the goal of creating a proof of concept that can fulfil the functional requirements was reached, it consumed significantly more time than estimated. This is especially due to two things: Firstly, the overhead of a distributed system, which will be discussed in 6.2. Secondly, setting up the project on AWS was more time consuming than expected, as shown in 6.1.

6.1 Amazon web services

The first pitfall using AWS is the huge amount of services. As this thesis is written, there are close to 100 different services within the ecosystem of AWS. Some are used for specific cloud computing solutions, like EC2 or Elastic Beanstalk, some are needed in order to configure the ecosystem like the AWS Identity and Access Management[26] ("IAM"). This requires a lot of reading. More pitfalls occur, once the services are supposed to work with each other. For example, if the `Web Services Adapter` has to trigger the EC2 instance that contains the `Rendering` component, it needs some configuration beforehand in order to ensure thtat the `Web Services Adapter` has sufficent rights. Furthermore, the database solution DynamoDB can be run locally. AWS promises that an application developed with the local version of DynamoDB will work with the remote version in the exact same way. Sadly, it didn't. Instead of two days, it took two weeks to get the components running within their environments and make the components work with each other.

6.2 Overhead of the distributed system

Building a distributed system is very complex. Firstly, for the `Genetic Algorithm` as well as for the `Rendering` component, a mechanism needed to be implemented, that transformed the object that the JSON reader provides into a usable genotype or scene graph. Secondly, as most distributed systems, the system interacts with the other components via HTTP calls only. This is more complex than calling methods with arbitary data as one would in a monloithic solution. Although creating a distributed system comes with the benefits discussed in 1, it produced more overhead than estimated. The advantages and disadvantages of an approach that uses a monolithic architecture are listed in chapter 5.2.

[26] `https://aws.amazon.com/iam/?nc2=h_m1`

Conclusion

In this thesis, it was explored, how to generate artworks that please mass appeal. Therefore, it was desired to create a system, that generates artworks that show a high level of variance and present these artworks to users. Based on the user's votes, the system shall learn how to iteratively create artworks that are even more fit to mass appeal. In chapter 1.2 it was stated that these artworks will be composed of 3D objects.

Chapter 2 conveys an overview of generative art and it's most used techniques. Firstly, it is defined what can be considered generative art in chapter 2.1. Afterwards, the most commonly used techniques are categorized in more predictable methods (chapter 2.2.1) and less predictable methods (chapter 2.2.2). By comparing these techniques, the conclusion has been made, that the less predictable methods carry more potential for generative artworks that are supposed to have a high level of variance. In addition, it has been shown that genetic programming is especially promising when it comes to creating multiple artworks of high variance and iteratively increase their level of mass appeal. This is underlined by projects that already used genetic programming in order to create aesthetically pleasing artworks, as discussed in chapter 2.3.

Based on these findings, a requirement analysis has been put together in chapter 3. The requirement analysis set four functional requirements and five nonfunctional requirements that are listed in chapter 3.4. In 4 it is shown how these requirements are fulfilled. Therefore, in chapter 4.1 it is shown, that a distributed system promises to combine the performance of a native application and the accessibility of a web application. In chapter 4.2 the four main components of this very system are introduced in detail. In chapter 4.3 it is discussed how the components were put into service with the help of AWS. As a conclusion, the thesis has shown that the combination of the custom-made scenegraph and genetic programming can lead to interesting results that offer a high level of variance.

Until this point, the system aims implements a proof of concept that shows how the components can work with each other in order to create artworks. As this thesis is published, the artworks will consist of a sphere in different colours. This can be used to determine the users' favourite colour. In addition, with the scene graph, a foundation has been implemented that can be used to create more complex modifications within a short span of time. Unfortunately, there was no time left to present more complex artworks in this thesis. Nevertheless, it is further discussed in 1 what ideas are yet to implement and where the system can go from here. Lastly, in 2 the advantages and

disadvantages of a monolithic approach are compared to the approach taken for this thesis.

1 Outlook

In the following, it is presented what has yet to come per component. In 1.1 new features for the `Web Application` that focus on increasing the user engagement are shown. In 1.2 it is shown what functionallity could be added to the `Rendering` component in order to create more complex artworks. Finally, in 1.3 it is discussed how the `Genetic Algorithm` can be improved.

1.1 Web application

For the `Web Application` there are two main aspects that may be improved: The user engagement and the display of the renderings.

1.1.1 Better user engagement

As stated in chapter 3.3.2 the software aimed at creating a pleasing user experience. In order to motivate the user to vote multiple times, a counter could be shown on top of the displayed artwork. The counter could display a text like “2 out of 5 votes cast”. Once, the user cast the given number of votes, he could receive an evaluation of his voting behaviour along the lines of “Your favourite colour seems to be blue.”. Furthermore, the desktop experience needs to be improved. As stated in chapter 4.2.1 the `Web Application` was built under a mobile-first approach. Right now, on a desktop computer, the website feels like a scaled up mobile application. This can be changed by implementing a layout for bigger screens. The layout may display the artworks next to each other instead of making the user swipe to see the next artwork.

1.1.2 3D renderings within the browser

Right now, the artworks are displayed as images. For the future, it would be better, if the artworks were presented in a way that the user could rotate them in 3D. If it’s not desired to render the artworks within the browser, because that could lead to performance issues, one could use a solution that is based on videos.

1.2 Rendering

For the `Rendering` component, more nodes should be added in order to create more complex artworks. This could range from more simple `ObjectNodes` to very complex ModificationNodes that change the object’s mesh. In addition, algorithms may be added that determine how to position the objects. For example, one algorithm could order the objects symmetrically and another one could order them asymmetrically purposefully.

Finally, more options for the lighting could be implemented. For example, the light could differ in colour and brightness, or more lights could be added.

1.3 Genetic algorithm

Once, a new node is introduced in the `Rendering` component, the `Genetic Algorithm` needs to implement a mechanic that can handle the node accordingly. Apart from that, the `Genetic Algorithm` needs to be further tested and improved. Changing small details can have a great impact on the algorithm. Among these details are the size of the population, how many of the new generation's individuals are mutated, how the fitness values shall be weighted. Right now, the genotypes are rather simple. Once this changes, the settings that work well now, may not be fitting anymore.

2 Monolithic approach

In chapter 4.6 it was stated, that implementing the distributed system caused a lot of overhead. Therefore, it is debatable if a monolithic solution would have lead to better results. In order to keep the accessibility of a web application, the monolithic application would use a WebGL[1] framework like three.js[2]. This would set the programming language to JavaScript. Rendering the artworks within the browser would eliminate the overhead of rendering the images, uploading them and displaying them in another component. Furthermore, the user could rotate and zoom into the artwork out of the box. On the other hand, the rendering would be dependent on the device's GPU. In the case of a mobile device, this may lead to a bottleneck.

A monolithic solution would require the genetic algorithm to run in JavaScript as well. Although there are frameworks for genetic programming in JavaScript, at a first glance, they didn't seem as advanced as comparable frameworks in other programming languages. This may lead to more custom code that needs to be written in order to achieve the same functionality. But, since there would be no more need to serialise and deserialise anything, it could reduce some overhead.

In conclusion, a monolithic solution is beneficial, when it comes to prototyping. For an experiment on mass appeal, it may be sufficient to keep the complexity of the artworks low. Nevertheless, a single threaded application that runs in a browser may not reach the performance of a C++ application. In addition, the `Rendering` component is reusable for different projects within the field of generative art.

[1] `https://www.khronos.org/webgl/`
[2] `https://threejs.org/`

Bibliography

[1] Philip Galanter. What is generative art? complexity theory as a context for art theory. In *In GA2003–6th Generative Art Conference*. Citeseer, 2003.

[2] Hans Haacke. "condensation cube". `https://www.macba.cat/en/condensation-cube-1523`, 2006. [Online; accessed 26-February-2018].

[3] Daniel Shiffman. *The Nature of Code: Simulating Natural Systems with Processing*. Daniel Shiffman, 2012.

[4] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[5] Benoit B Mandelbrot. *The fractal geometry of nature*, volume 173. WH freeman New York, 1983.

[6] Robert Devaney. Unveiling the mandelbrot set. `https://plus.maths.org/content/unveiling-mandelbrot-set`, 2006. [Online; accessed 26-February-2018].

[7] Eric W. Weisstein. "complex plane." from mathworld–a wolfram web resource. `http://mathworld.wolfram.com/ComplexPlane.html`. [Online; accessed 26-February-2018].

[8] Stephen Wolfram. *A new kind of science*, volume 5. Wolfram media Champaign, 2002.

[9] Charles Darwin and William F Bynum. *The origin of species by means of natural selection: or, the preservation of favored races in the struggle for life*. Penguin, 1859.

[10] Juan C Quiroz and Sergiu M Dascalu. Design and implementation of a procedural content generation web application for vertex shaders. *arXiv preprint arXiv:1608.05231*, 2016.

[11] Agnieszka Mars and Ewa Grabska. Aesthetic-oriented generation of architectonic objects with the use of evolutionary programming. 2017.

[12] Irving Biederman. Recognition-by-components: a theory of human image understanding. *Psychological review*, 94(2):115, 1987.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 09. April 2018 Jonas Seegers