UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

PROCEDURAL GENERATION OF VIRTUAL WORLDS

Sandro Henrique Duarte Correia

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Interação e Conhecimento

Trabalho de Projeto orientado por:
Prof. Doutor Luís Manuel Ferreira Fernandes Moniz

2020

Agradecimentos

Em primeiro lugar aos meus pais por cuidarem de mim estes anos todos, disponibilizando-me a possibilidade de escolher o caminho académico que pretendia. Sem o apoio incondicional dos dois não teria a oportunidade de chegar aqui.

À minha namorada Marta, pelo seu apoio e paciência sempre presentes ao longo deste ano complicado. Por todas as suas ajudas e revisões, certificando-se sempre de que me mantinha empenhado e no caminho certo neste grande desafio.

A todos os meus amigos que deram algum do seu tempo para me ajudar a testar o projeto, em particular aos meus colegas Nuno Rodrigues e Pedro Andrade pela ajuda proporcionada com os mais diversos temas ao longo de todo o desenvolvimento.

E por último mas não menos importante ao Prof. Luís Moniz, pela oportunidade de trabalhar sob sua orientação nesta tese.

Abstract

Procedural generation is a method of algorithmically generating data instead of manually doing so. There is an increasing opportunity for the use of procedural generation techniques, mainly in the ever growing video-game and movie industries, due to the necessity of creating virtual content. Even though the manual creation of such content offers an higher level of control, it is also usually a long process with the necessity of one or more technical experts, making the possibility of automation for these processes something desirable.

When it comes to the video-game industry, replayability is a term used to assess a video-game's potential for continued play value after its first completion. Using procedural generation it is possible for a game to offer very different experiences each time it is played, potentially increasing the game time, which is something commonly wanted in the industry. An example where this is implemented successfully is in the game *Minecraft* [12]. The possibility to generate different worlds each time offers unique experiences to each player, not only turning it into a more personal experience but also increasing the level of replayability as well.

In the case of procedurally generating terrain in a virtual world we must take into account not only its shape and height points but also the type of terrain being created. Whether it be a beach or a mountain, the decision of what type of terrain to generate depending on the context can be as important as its shape.

The intent of this dissertation is to develop a procedural generator of virtual worlds, so the results of the application of a decision tree evolved through genetic programming can be visualized. The decision to be done by the decision tree will be regarding the type of terrain to be generated. To perform the genetic evolution a small set of decision trees will be generated to be evolved simultaneously, with each generation the terrains resultant being shown to the user, allowing them to perform a choice according to their own criteria of which trees they wish to crossover for future generations.

Keywords: Procedural Generation, Virtual Worlds, Genetic Programming, Decision Trees, Artificial Intelligence

Resumo alargado

Geração procedimental é uma forma de gerar dados algoriticamente ao invés de o fazer manualmente. Cada vez mais se verifica uma maior oportunidade para utilização de técnicas de geração procedimental, principalmente nas indústrias crescentes dos video-jogos e dos filmes, devido à necessidade de criar conteúdo virtual. Embora a criação manual deste tipo de conteúdo ofereça um maior nível de controlo, é também um processo geralmente demorado com a necessidade de um ou mais especialistas técnicos, tornando a possibilidade de automatização destes processos algo desejável.

Para a indústria dos video-jogos existe o conceito de ”replayability”, isto é, o valor que se pode obter em jogar um jogo mais do que uma vez. Utilizando geração procedimental é possível fazer com que alguns jogos ofereçam experiências muito diferentes de cada vez que é iniciado o jogo, aumentando potencialmente o tempo de jogo, algo muito procurado na indústria. Um exemplo onde isto é implementado com sucesso é no jogo *Minecraft* [12], onde a possibilidade de gerar mundos diferentes permite oferecer experiências únicas a cada utilizador, não só tornando a experiência mais pessoal como aumentando o nível de ”replayability”.

No caso da geração procedimental dos terrenos de um mundo devemos ter em conta não apenas a sua forma e pontos de elevação, mas também o tipo de terreno a ser criado. Quer seja uma praia ou uma montanha, dependendo do contexto a decisão do tipo de terreno a gerar pode ser tão ou mais importante que o formato do terreno gerado em si.

Com esta dissertação pretende-se desenvolver um gerador procedimental para a criação de mundos virtuais, de modo a observar os resultados da aplicação de uma árvore de decisão evoluída com programação genética. A decisão a ser feita pela árvore de decisão será no que toca ao tipo de terreno a ser gerado. Para realizar a evolução genética será gerado um pequeno conjunto de árvores de decisão a serem evoluídas em simultâneo, sendo que em cada geração os terrenos resultantes serão exibidos ao utilizador, permitindo o mesmo efetuar uma escolha segundo os seus próprios critérios de quais as árvores pretende que sejam cruzadas para as gerações seguintes.

A abordagem escolhida nesta dissertação consiste de diferentes etapas. Numa primeira etapa foi desenvolvido um gerador de terreno procedimental, com a representação de terreno escolhida sendo a de uma grelha de vóxeis. De modo a poder representar alguma variedade em termos de características de terreno foi escolhido um pequeno número de diferentes tipos de blocos para os vóxeis, optando por blocos que representam: terra, relva, areia, neve, pedra e água. A geração procedimental foi feita através de uma função de ruído, sendo optada a função de ruído melhorada de Ken Perlin [24] devido aos seus resultados atraentes e bom desempenho. Nesta altura do desenvolvimento os tipos de blocos de cada voxel eram simplesmente atribuídos de forma aleatória.

De seguida, foi criada uma Interface do Utilizador de modo a permitir uma melhor visualização de múltiplas instâncias de terrenos. A função de ruído foi também alterada de forma a permitir a utilização de sementes fixas, para que as múltiplas instâncias sejam todas geradas com estruturas de terreno idênticas, facilitando a comparação entre resultados. Foi também durante esta etapa que foram implementadas árvores de decisão com o propósito de determinarem os tipos de bloco a serem escolhidos, com dois tipos de nós de decisão diferentes, um realizando decisões de acordo com a altura de um certo voxel e o outro de acordo com o número de voxéis vizinhos adjacentes de um certo tipo de bloco, bem como nós folha que ditam o tipo de bloco a ser atribuído.

Finalmente a Interface de Usuário previamente criada foi expandida, permitindo utilizadores escolherem um ou dois terrenos para evoluir. A evolução genética entre as árvores de decisão escolhidas foi então implementada. Quando dois terrenos são selecionados a geração seguinte consiste dos descendentes resultantes da recombinação dos terrenos em questão, sendo depois aplicadas as funções de mutação a cada um. Caso contrário, e apenas um terreno seja selecionado, a geração seguinte consistirá de clones da árvore de decisão respetiva ao terreno, sendo apenas aplicadas depois as funções de mutação a cada um dos clones. Foram implementadas três funções de mutação: a primeira, ”mutação de valor”, tem uma chance de alterar os diferentes valores dentro de um nó na árvore; a segunda, ”mutação de função”, tem uma chance de alterar o tipo de função de um nó de decisão, bem como os seus valores, e não pode ser aplicada a um nó folha; e, por último, a ”mutação de nova função”tem uma chance de mutar um nó folha num nó de decisão aleatório, gerando também dois novos nós folha descendentes dele próprio. Foi também adicionada uma interface que permite a visualização das árvores de decisão, na esperança de que a mesma pudesse ser de uso para utilizadores possuírem uma melhor compreensão de como cada instância de terreno foi gerada, no entanto a falta de poda das árvores de decisão fez com que esta adição não fosse tão útil como inicialmente planeado visto a legibilidade reduzir rápidamente à medida que as árvores de decisão evoluem para tamanhos significativos.

Após a ferramenta ser terminada, potenciais utilizadores foram solicitados a ajudar testar a mesma devido à elevada subjetividade possível quando se evolui e avalia um terreno. Foi pedido aos utilizadores para tentarem alcançar quatro terrenos objetivos utilizando o programa, sem especificar um resultado final de modo a permitir que os mesmos evoluam os terrenos de uma forma tão imparcial quanto possível. Após a realização das quatro tarefas os utilizadores preencheram um questionário acerca das suas experiências com a ferramenta. Os resultados obtidos destas sessões de teste permitiram-nos chegar a um melhor entendimento tanto dos pontos fortes do trabalho como dos fracos. Para começar, quando tentamos evoluir um terreno através das decisões de um utilizador, é importante que o utilizador em questão possua uma imagem clara do seu objetivo ao longo das gerações de evolução, de modo a permitir que as árvores de decisão convergem adequadamente. Se o utilizador fizer frequentemente escolhas inconsistentes ao seu objetivo pode acabar por abrandar ou até mesmo tornando impossível para o algoritmo alcançar o objetivo de terreno inicial.

Foi também possível recolher que alguns utilizadores sentiram-se limitados pela ferramenta e que provavelmente seria mais fácil criar um terreno de raíz. Os utilizadores também sentiram, no entanto, que as suas escolhas tiveram impacto nas alterações a serem feitas no terreno, e que não era um processo demorado até algum progresso ser aparente. Estes resultados levaram-nos a crer que, embora exista espaço para melhorias, utilizar algoritmos evolucionários para evoluir árvores de decisão consegue ser uma forma viável de gerar mundos virtuais de forma procedimental.

Existem alguns aspetos deste trabalho que poderiam ser de interesse aprofundar no futuro. Para começar, o visualizador de árvores de decisão de momento não possui muita utilidade devido ao quão rápido se torna ilegível para árvores de maiores proporções. Poderiam ser implementadas diferentes técnicas de poda de árvores de decisão de forma a amenizar o problema, limpando muita da redundância atualmente presente em árvores de decisão maiores e, por consequência, melhorando o desempenho. Com isto seria também mais prática a implementação de funcionalidades mais complexas no visualizador, tal como, por exemplo, a possibilidade de os utilizadores interagirem com a estrutura de uma árvore de decisão e fazerem ajustos manuais diretamente conforme acharem adequado.

Devido à natureza subjetiva da implementação escolhida foi dificil de comparar diferentes taxas de mutação e algoritmos de recombinação em termos de resultados objetivos. Isto é uma área que também poderia ser aprofundada, utilizando por exemplo uma função de aptidão que permita avaliar os diferentes terrenos de forma algoritmica e portanto automática, o que permitiria uma melhor comparação entre os resultados. As configurações encontradas que forneçam os melhores resultados poderiam depois então ser utilizadas com o algoritmo original.

Palavras-chave: Geração Procedimental, Mundos Virtuais, Programação Genética, Árvores de Decisão, Inteligência Artificial

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Both the video-game and movie industries have been ever growing the last couple of years. According to data gathered by analyst Pelham Smithers [14], the video game industry's global revenue reached \$138.5 billion in 2018, roughly \$100 billion more than 20 years before in 1998 (\$36 billion). As for the movie industry, according to a research done by the Motion Picture Association of America (MPAA) [23] the global entertainment market reached \$96.8 billion in 2018, which is still a very noticeable amount. Revenue numbers alone don't allow us to determine the profits of these industries, but it is easy to see the impact they have left in the economy and how companies might want to try and cash in, and with the increasing level of quality standards for both films and video games, producers as well as developers could benefit from reducing costs wherever they can.

One such way of reducing cost is through the use of procedurally generated content. Procedural generation is a method of algorithmically generating data instead of manually doing so, which can help relieve the burden of some tasks in both film and video game development. Usually long processes with the necessity of one or more technical experts, such as generating a large virtual world, can have a hefty reduction of development times with the aid of procedural generation, be it real-time or otherwise. While procedural generation is undoubtedly more predominant in video games, it is not unheard of being used in other mediums. MASSIVE [10] is a procedural generation software originally developed for use in Peter Jackson's *The Lord Of The Rings* film trilogy [16], later bringing this technology to both films and television, becoming the leading software for crowd related visual effects and autonomous character animation. Allowing a team to procedurally generate several hundreds of animated characters to a scene instead of manually doing so, MASSIVE is a great example of how procedural generation can save a developer both time and money. When it comes to video games, however, there have been records of almost every aspect of a game being procedurally generated, even before graphically oriented video games, such as the genre-defining *Rogue* [15], a dungeon crawler taking place on a square grid represented in ASCII characters. *Rogue* procedurally generates the dungeon levels, monster encounters and treasure on every playthrough, greatly increasing its replayability as each playthrough offers a unique experience each time the player decides to tackle it. The more recent indie video game *Dwarf Fortress* [4] takes it a step further and procedurally generates a whole ASCII world, dungeons, characters, back stories, civilizations, even legends and artifacts, simulating 250 years of the world's history before allowing the player to play. And while the Roguelike genre is definitely not to be undermined, and is clearly one of the biggest inspirations for procedural content generation in games, it would be remiss not to mention the single best-selling video game of all time, *Minecraft* [12]. *Minecraft* was officially released in 2011 and quickly became one of the most influential video games in the industry, with its fully procedurally generated and virtually infinite 3D world, as well as the ability to ”mine” blocks and place them wherever the player so desires, the game offers a unique experience for everyone, and even multiple unique experiences for each person every time they choose to create a new world.

Besides being useful for reducing costs, replayability is also a big factor when it comes to choosing whether or not to use procedurally generated content. Replayability is a term used to assess a video-game's potential for continued play value after its first completion. While by definition procedurally generated content isn't a necessity for a game to have high replayability, having content such as the world, the enemies or even the treasure be procedurally generated each time you play inherently increases the replay value as it makes each run feel more unique to the player. Replayability increases a game's longevity, and having procedurally generated levels is also a good way for a game to have virtually infinite content, as having a large number of set levels is not only time and resource consuming to the game developers, but the player will most likely eventually defeat the levels and be left with no new content to play, whilst with procedurally generated levels one can theoretically play a virtually infinite amount of levels, though how repetitive they feel depends on how complex the various degrees of procedurally generated content are implemented.

1.2 Goals

While there seems to be a lot of focus on the generation of the terrain per se, there isn't as much attention given to the decision of what type of terrain is generated. The main goal of this dissertation is to explore the possibility of using evolutionary algorithms to evolve a decision tree in order to procedurally generate virtual worlds without the need of a technical expert. For that purpose, the following steps were taken:

- First a procedural generator of voxel worlds (similar to those of the *Minecraft* video game) was developed in Unity[17]. These worlds were to be pseudorandom, each one always returning the same seed, and subdivided by chunks, in order for the generator to be more efficient and prevent constantly loading and unloading singular blocks when they get in or out of range. This first iteration of the world generator would not include a decision tree for deciding the type of terrain to be generated;

- Afterwards, a User Interface was be created to allow an easier visualization of multiple procedurally generated worlds simultaneously. These generated worlds all had random decision trees assigned for determining the type of terrain to be generated, but they would all be generated from the same seed, for better comparison between the different results;
- The third and final step was to apply genetic evolution to the previously generated decision trees. For this purpose, the previously developed User Interface was then to be made interactable, allowing a user to select their preferred decision trees as the ones who should crossover for the next evolutionary generation.

1.3 Structure of the document

Aside from this introductory chapter, this document is organised as follows:

- Chapter 2 - detailed overview on the state of the art regarding procedural generation, decision trees and evolutionary algorithms.
- Chapter 3 - description of the problem and methodologies used, as well as hurdles found and how they were overcome.
- Chapter 4 - discussion and analysis of the results found.
- Chapter 5 - final thoughts on the work, as well as future work that could be done in order to improve it.

Chapter 2

State of the art

2.1 Procedural Landscape Generation

Procedural generation has been a topic in the field for four decades now [26], with new procedural methods being regularly introduced ranging from offline procedural generation being used as a form of compression in games such as Elite [7] and Exile [8], to procedurally generating thousands of planets on the fly as can be seen in No Man's Sky [13]. While this project's main focus is that of the usage of procedural generation for virtual world development, it was thought relevant to elaborate on some of the common problems found in the field, and relevant methods introduced to surpass them.

2.1.1 Pseudorandom Numbers

Randomness is a key factor in procedural generation as it is what allows content to have some degree of variety each time it is generated. Ian Millington dedicates a section on this subject in their book *AI for Games* [22]. Generating true random numbers would require specialized hardware that can sample stochastic physical phenomena, so instead most of the time the following two random categories are used:

- Cryptographic random numbers - while not truly random, are unrepeatable and unpredictable enough to serve the purpose and can be used as the basis of cryptographic software;
- Pseudorandom numbers - these are the ones usually returned by the *random* functions provided. They are pseudorandom because they are based on a seed, and as such for a given seed they will always return the same value.

While most times randomness is something desirable, while working with procedural content generation it might be preferable to be able to repeat the generated values. If testing between different terrain generation algorithms, for example, utilizing the same seed for both testing scenarios will provide with a much more insightful observation on the differences between the algorithms. A seed based random number generator also allows users to save their favorite seeds for later usage, which might be beneficial in some cases.

Randomly generated values are very commonly calculated from the seconds in a computer system's clock. This makes it so that every time one generates a random value the seed is different, and as such the value returned should be different most of the time. However, there exist cases in which being able to generate the same result more than once would be advantageous. If random seed is generated instead, and then that seed is used from then onwards the generated content will be the same every time.

A single random value is not usually enough, however, and calling the number generator more than once is desirable. Each time a new random value is called, the seed is updated in a deterministic way, so the sequences of numbers generated beginning from a particular seed will always be the same. The tricky part to producing the repeatability, is making sure that the same number of calls are made each time. If using the same seed for procedurally generating content is desirable, the content is expected to be exactly the same every time, regardless of the actions taken in between. This can be accomplished by using several random number generators, with only a master generator being given the game seed, and all others being created in a specific order. This way actions that require calling a random number from a generator will not interfere with the other generators, maintaining the sequence for each one.

2.1.2 Terrain Representation

Before exploring the options of how to procedurally generate landscape one must first make the fundamental decision of how to represent the terrains themselves. The choice of data representation will determine not only the types of landscape that are possible to represent, it will also impact the tools available to generate said landscape. When deciding on what representation to choose one must take into account different factors such as how precise does the terrain need to be or if it is relevant to be able to represent terrain structures in which multiple surfaces can have the same horizontal coordinates such as overhangs and caves.

Ryan Saunders identifies some of the most relevant alternatives in their article *Terrainosaurus: realistic terrain synthesis using genetic algorithms* [19]:

Height-Maps

The most common terrain representation seems to be the height-maps, also sometimes known as height fields. Height-maps are two dimensional grids of surface features, typically associated with elevation, although they may contain other layers of data, such as surface texture and water flow.

One major advantage of height-maps is that their regular structure makes it easier to optimize operations such as rendering, collision-detection and path-finding. Even collision detection, which is usually a computationally expensive operation, can be done cheaply on height-maps seeing how given an (x, y) position only a few surrounding triangles need to be checked for collision.

Another advantage is the significant quantity of real-world terrain data available in height-map form, allowing an easier time working with real-world terrain and usage of simulation based approaches.

The greatest disadvantage of height-maps is that they fail the vertical line test, that is, due to how the elevation is a function of an (x, y) pair there must be exactly one elevation value per pair of coordinates, denying the possibility of more complex terrain structures such as overhangs and caves, in which multiple surfaces can have the same horizontal coordinates. How much of a disadvantage this proves to be depends on the use case, seeing how such structures are uncommon in a realistic environment and the exceptions can be treated separately (assuming an offline procedural generation).

A second disadvantage would be their finite, uniform resolutions. Because it is finite this means that there is an upper limit to how detailed the height map representation can be, which might result in an unnatural feel when the terrain is viewed from a too close proximity. Not only that, but since the resolutions are also uniform, height maps cannot smoothly handle areas of finer detail. If the resolution is chosen to match the average scale of features in the terrain, that means finer detailed areas will be simplified or removed. Alternatively, if the resolution is chosen to be high enough to capture those finer details, areas of the terrain which aren't as complex will be captured at the same resolution, resulting in an undesirable waste of both space and computational time.

Voxel Grids

Voxel grids are very similar to height maps, except they are three-dimensional instead. This means that instead of representing an altitude as a two pair coordinate it is represented as the third coordinate of a point in the grid instead. This three-dimensional grid consists of voxels, in which, in the simplest case, each voxel can individually be filled or not.

The main clear advantage of voxel grids over height maps is that they are not constrained by the vertical line test, allowing multiple surfaces to have the same horizontal coordinates and thus being able to represent more complex structures such as caves and overhangs.

The trade-off for this, however, is that operations in voxel grids are significantly more computationally expensive. They still suffer from the same disadvantage of height maps, having finite and uniform resolutions, however with a whole new dimension to be taken into account a lot more points have to be taken into account as well, specially considering how common it is for there to be large chunks of the grid either completely empty (in the case of air) or buried deep underground. These wasteful calculations can be somewhat optimized through the use of spatial subdivision techniques, but to what degree of usefulness depends on the scenario.

Meshes

It is worth mentioning non-uniform meshes as a possible terrain representation given that it is often assumed as the default representation. While modelling such meshes is dependant on the skills of artists who are able to do so manually, the level of control it allows is incomparable to the other possible representations.

Unlike the previous two representations, meshes naturally support varying levels of detail, allowing different numbers of vertices for finer details in the terrain while still maintaining relatively few vertices in flat areas. This allows meshes to represent complex terrain structures far more efficiently than regular grid methods, as it does not require a globally high resolution to achieve them.

The difficulty with using meshes for terrain representation comes from the fact that, as far as this project's research found, there exists no clear way to generate them automatically, being dependant on technical experts to manually generate said structures.

Continuous Functions

While not as commonly used in practice, according to Saunders, there can also exist terrain presentations based on continuous functions such as analytic and fractal functions.

Both kinds of continuous functions have the advantage of not losing resolution while being viewable at any scale, unlike grid-based representations. Fractals in particular offer the unique advantage of continuing to produce new details at progressively finer scales, which in turn means that a fractal terrain can have as much detail as the display system can render.

The main problem with this kind of representation is how difficult it becomes to model terrain with. If a single, global function is used, it is difficult to know how to modify the function in order to achieve a certain local effect, giving a fairly low level of control in comparison to other representations.

2.1.3 Terrain Generation Approaches

Even though there does not seem to exist a great variety of papers regarding terrain generation, the methods that have been proposed seem to generally fall into a few broad categories. The aforementioned *Terrainosaurus: realistic terrain synthesis using genetic algorithms* [19] article sorts these methods in 4 different categories:

GIS Based

While not necessarily a way of generating terrain, Geographical Information System (GIS) data is a simple but effective way of getting realistic looking terrain models. A lot of GIS data providers exist [1] for different parts of the world in different resolutions, being ideal when realism and accurate depiction of real life locations are the main concern.

What GIS data offers in realism and ease of use however, it lacks in variety and innovation. Since only real-world locations exist for this kind of data, it makes finding the ideal terrain for a project very difficult, if not impossible, being limited to what terrains the data sets have to offer. This may make this approach unappealing to those searching for a high degree of possible customization in their terrains.

Sculpting Based

The most common type of terrain generation is the sculpting of terrains and its features through the use of 3D modelling software (such as Maya [11], 3DSMax [2] or Blender [5]) or specialized level editors for particular games (for example, Unreal Tournament [18]). The big advantage of sculpting methods is the creative freedom that its tools offer with the ability to shape every node in the models, the only limitations being the artist's imagination, their skill and their effort.

Of course this strength is also the method's biggest disadvantage, as achieving great results through sculpting is heavily dependant on time and effort from the artist's part, not to mention the need of a technical expert in the first place. In a world where both games and films are getting bigger and bigger in scale, so are the costs for manually producing this kind of content.

Simulation Based

Simulation based generation methods consist of simulating the effect of the physical processes we see in the real-world, just as erosion and faults, and evolving the terrain with its applications over time. These methods have the capability of generating highly realistic terrain results to the extent that they accurately model the corresponding physical processes, while also being an automated process with fairly low input from the user.

One drawback of such methods is how computationally expensive they are, requiring a lot of processing time in order to produce realistic results; and as usual, as the resolution of the simulation increases, so does the time required to run it. Another issue is how little control these methods offer. The only real inputs that the user has control over are the initial terrain conditions and which physical processes are to be applied, which can make achieving specific terrain results complicated, if not sometimes impossible.

Procedural Based

The term procedural generation refers to the algorithmic generation of content instead of doing so manually. When it comes to terrain generation, simulation based generation could be described as such as well, but besides the fact that the latter requires a starting environment, there's also a difference in methodology as most procedural generation methods don't attempt to simulate physical processes. Instead, methods for procedurally generating terrain commonly rely on random number generation and the cases in which they are found to have a desired look.

Some of the most popular procedural generation techniques such as the midpoint displacement method and Perlin Noise are fractal in nature. That is, these techniques exhibit the fractal property of self-similarity at different scales, and generally involve randomly perturbing the height values of the terrain by increasingly smaller amounts at increasingly finer scales.

The main advantage of procedural generation is how it can generate interesting results with little to no human input. Terrain data can be generated from scratch, unlike in simulation methods, and because of the randomness some very interesting and unique results can be achieved that wouldn't be normally possible simulating the physical processes of the real world. However, this disconnection from realism can also be a big drawback in certain cases. Procedural methods only happen to coincidentally resemble real-world terrain, as achieving a desired effect of the terrain is mostly a trial and error experience, and most methods end up having a limited range of terrain types they can generate.

Perlin noise is a nowadays common approach put forward by Ken Perlin in 1985, for which they won an Oscar in 1997 for technical achievement. As Millington explains in their book [22], Perlin noise begins with a grid coarser than the target terrain (if a desired landscape has 1024 x 1024 cells, one might create Perlin noise on a 64 x 64 grid). At each location of the smaller grid a random gradient is created, which will represent the slope of the noise in three dimensions. For each location in the original grid, the gradient is interpolated from the four nearest locations on the small grid. This interpolated gradient is then used to calculate the height, which can then be remapped into a desired range.

The size of the small grid will determine how much detail is in the final noise and, as such, it is common to combine Perlin noise at successive doublings of scale. This approach is the one that the popular game Minecraft[12] uses, and the one used for procedurally generating terrain in this project as well.

2.2 Decision Making

When someone thinks of Artificial Intelligence, one of the first thoughts that may come to mind is that of a computer making a decision between multiple choices. While the execution of the choice itself is often taken for granted, in reality the decision making is more often than not a relatively simple part of the AI.

Decision making can be broken down into the processing of a set of information that is used to generate an action for carrying out. In a game this set of information can be further broken down into external and internal knowledge, with the first being the information that a character knows about its surrounding environment and game state, and the latter being the information known about the character's own state or thought processes. Millington talks about this in more detail in their book [22].

While lately there have been more sophisticated tools for decision making, this section will cover three of the most often considered technologies since the appearance of the field.

2.2.1 Decision Trees

Decision trees are one of the simplest decision making techniques, which contribute to their popularity and use. They have the advantage of being modular and easy to create, and can be used in a wide array of situations, whether it be for developing a strategic AI, multiple choice conversations or even animation control.

A decision tree consists of connected nodes, each node representing either a decision to be made or an action to take, in case the node has no children (a leaf). Starting from the root, each decision leads to either another decision or an action to take. Decisions in the tree should be simple and check only a single value, but boolean logic can still be performed through proper node placement in the tree. To perform an AND operation, for example, two condition nodes should be placed in a series on the tree.

A good advantage of the decision tree is how efficient it is in narrowing the search space when performing a search. Since the decisions are built into a tree, the number of verifications needed for a given task are usually much smaller than the total number of nodes. That is because the number of decisions to be made for a given action can only be as big as the biggest branch of the tree.

While it is more common to see binary decision trees, the possibility for a given decision to be made between more than two values exists, but ultimately it ends up being mostly a matter of preference. The underlying code is fundamentally the same, however some if not most of the common learning algorithms that work with decision trees require them to be binary.

2.2.2 State Machines

It is common for characters in a game to behave a certain way, until some event causes that behaviour to change indefinitely. We can cover this type of artificial intelligence with decision trees, however in most cases it is simpler to use state machines, as they were designed for this purpose and thus come with some added benefits.

With a state machine a character is always on a given state, with specific actions and behaviours associated with each different state. Assuming it remains in the same state, the character's behaviour shouldn't change. States are connected through transitions, each leading from a given state to another and depending on conditions. If the conditions of a certain transition are met, and the character is currently in the state from which the transition comes, then the character changes to the target state. This means that, unlike most decision trees, not every action can be reached at any time, as only transitions from the current state are considered.

Sometimes, however, there exist so called ”alarm behaviours” that need to be called regardless of which state the machine is currently in. A good example of this is that of cleaning robots. A robot with states for cleaning, searching or going back to its power station. Naturally, if its batteries start to run low, it should want to return ”home” regardless of his current behaviour. Similar transactions can be included in both states for this example, but with a more complex state machine this could quickly turn inefficient. Not only that, but in the case of multiple alarm behaviours, the question of which one should take priority becomes apparent. Instead of combining all the logic into a single state machine, it can be separated into several state machines on an hierarchy. For the previous example, a state machine of the cleaning robot could be going back to recharge, while the other is a state machine including the cleaning, searching and going back states. This allows a better structuration of the artificial intelligence.

2.2.3 Rule-Based Systems

Rule-based systems used to be some of the most popularly used artificial intelligence techniques before the ”deep learning” era, and even to date they have been used on and off in video games despite being known as inefficient and difficult to implement. The big advantage of these systems is their ability to justify their reasoning, since the rules are explicitly spelt out.

Rule-based systems have a common structure consisting of two parts: a database containing the knowledge pertaining to the artificial intelligence and a set of if-then rules. Rules will activate based on if the database contains information that allows the ”if” condition to be met. In the case of more than one rule being activated at the same time, some systems also have an arbiter to decide in which order they trigger, although most simply consider the first rule of the set that was activated. Following are some of the more common rule arbitration approaches, as described by Ian Millington in their book [22]:

- First Applicable - the rules are provided in a fixed order, and the first rule in the list to trigger gets to activate. This makes it so that earlier rules have priority over the later;

- Least Recently Used - a linked list holds all the rules in the system in a fixed order, just like in First Applicable. When a rule gets to activate, however, it gets removed from its position and attached at the end of the list, allowing the system to prioritize rules who haven't been activated recently. This rule prevents looping by making sure all rules have an opportunity to activate;

- Random Rule - if multiple rules are triggered, one is randomly selected to activate. Whereas the previous algorithms stopped at the first rule found who triggered, this one has to go through the full list of rules to properly check which ones triggered and choose one, being less efficient;
- Most Specific Conditions - often times the more complex the conditions of a rule the more likely it is to be specialized for the current situation. When multiple rules trigger, this algorithm chooses the one with the most specificity to activate. The level of specificity of a rule depends, but usually the number of clauses is a good measure when rules are expressed as Boolean combined clauses;
- Dynamic Priority Arbitration - identical to the First Applicable algorithm, but the order of the rules is changed according to their current priority. Sometimes it may make sense for some rules to lose their priority when in comparison for others, for example, in a First Person Shooter game if the character has full health it makes sense for his priority to use an health pack to be lower in comparison to shooting an enemy. An alternative implementation would be for the algorithm to go through the set of rules, returning all that trigger, and activating the one with the current highest priority (instead of constantly updating every rule's priority).

While rule-based systems can support fairly advanced AI, its major weakness is the difficulty of writing good rule sets. When compared to state machines and decision trees, which are easy to expose in a graphical editor, rule-based systems tend to be less frequently used, despite their power.

2.3 Evolutionary Algorithms

Evolutionary algorithms use mechanisms inspired by biological evolution to attempt global optimization of a given problem. Between such mechanisms are examples such as crossover, mutation, recombination and selection, with candidate solutions playing the role of individuals in an evolving population. These individuals are evaluated by a fitness function which determines how the evolution of the population as a whole should proceed.

There have been multiple evolutionary algorithm approaches, most differing in genetic representation, with some of the most commonly researched ones being genetic algorithms and an extension of genetic algorithms, genetic programming. Michael Affenzeller, Stephan Winkler, Stefan Wagner, and Andreas Beham wrote a very detailed article [20] about these two approaches, summarized below.

2.3.1 Genetic Algorithms

Genetic algorithms are iterative procedures which operate on a population of individuals of usually constant size. What specifically makes an individual depends on the problem at hand, but each individual represents a solution to the problem usually in the form of a fixed-size binary string.

At the start of the algorithm an initial population of individuals is generated either randomly or heuristically. Each individual presents a solution to the problem, and to evaluate how good of a solution each one is all individuals are fitted against a fitness function particular to the problem.

During each iteration, called a generation, the fitness of the individuals of the current population is measured. In order to form a new population for the next generation, individuals are selected according to their fitness (the better the fitness the better the chance of being selected) and crossover genetic operations are performed between pairs of the selected individuals. These pairs, called parents, generate one (or two, in some implementations) new individual from them, called the offspring. Depending on the implementation, sometimes only the offspring individuals make it to the next generation, sometimes both offspring and parents do and some even randomly generate new individuals to fill out the population size. Regardless of the method chosen, what's important is that each generation has a population of constant size for evaluation and that the genes of the better fit individuals are carried on.

The crossover genetic operation works by combining parts from both parents, with the results being the offspring. In its simplest form, the operator randomly selects a crossover point in the binary string and swaps substrings between both parents.

The second genetic operator to be applied is the mutation operation. In the case of binary string, a random mutation chance is applied to each bit which, when it succeeds, swaps the bit at the current position (if the bit was 1 it become 0, and vice-versa). This helps prevent premature convergence by randomly sampling new points in the search space.

As genetic algorithms are stochaustic by nature, convergence isn't a guarantee. Therefore, there is usually a trigger for a termination of the algorithm, which could be a maximum number of generations, finding an acceptable solution or more sophisticated criteria that indicates that continuing the algorithm will probably not yield better results.

2.3.2 Genetic Programming

While genetic algorithms are able to produce high quality results for a variety of problems, by themselves they are not able to handle the task of getting a computer to solve problems without explicitly programming it to do so. Genetic programming is an extension of genetic algorithms, working similarly on populations of candidate solutions for a given problem and being based on the Darwinian principles of survival of the fittest. Genetic programming provides a way to successfully conduct the search in the space of computer programs. This is of great value, as virtually all problems in artificial intelligence, machine learning, adaptive systems and automated learning can be recast as a search for computer programs.

Just like genetic algorithm-based problem solving processes, one of the key issues in genetic programming is the representation of problem instances and candidate solutions. While conventional genetic algorithms can solve a huge amount of problems with the use of fixed-length strings, the most natural representation for a solution in the context of genetic programming is that of an hierarchical computer program of variable size, most commonly represented by a point-labeled structure tree. In these trees, all nodes are either functions or leaf nodes. Leaf nodes are evaluated directly, that is, their return values can be calculated and returned immediately, while all functions have child nodes which are evaluated before using the children's calculated return values as inputs for the parent's evaluation. Evaluation of the functions is therefore executed recursively depth-first, starting from the left side of the structure tree.

As we've seen with genetic algorithms, crossover and mutation exist, with crossover taking two parent individuals and producing a new offspring by swapping parts from the parents. One of the major advantages of hierarchical tree representations here is how single-point crossover can be easily performed by replacing a subtree of one of the parents for a subtree from the other parent, both chosen at random. Mutation on the other hand can be applied by modifying a randomly chosen node of the respective structure tree, performing actions on the subtrees such as deleting or replacing with a randomly initialized subtree, or even just on the nodes themselves (for example changing a function's type or converting a function node into a leaf one).

Genetic programming has some advantages over genetic algorithms. The fact that evaluation is performed recursively in genetic programming allows newly generated or manipulated programs to be evaluated immediately without any intermediate transformation step. With another major advantage being how structure trees allow the representation of solutions whose size and shape change dynamically, unlike the usually fixed-size bit strings of genetic algorithms.

2.4 Summary

As demonstrated in this chapter, there are not only various aspects to take into consideration for the procedural generation of terrains, but also a large number of methods in which to approach each one. Despite the viability of all the different approaches researched, some were found to be more relevant to the work at hand than others.

When it comes to the different terrain representations available, **voxel grids** appeared to be the most appealing choice. As the goal was to have users evolve terrains based on their preference, it is important for the terrain features to be easily distinguishable from each other, which voxel grids' structure allowed with the drastic contrast existing between the voxels when compared to terrains generated in meshes or continuous functions. While this could also be achieved with an height-map representation, a voxel grid allows us to represent more complex structures due to not being constrained by the vertical line test, making it the most favorable representation out of the two.

As one of the main goals of this work is to lessen the need of technical expertise when generating terrain, **procedural based** generation was a clear choice due to its ability to generate interesting results with little to no human input. GIS Based generation could also be used, as it only requires access to GIS data, however its lack of both variety and innovation were considered too big of a drawback in exchange for realism.

Decision trees were the decision making approach chosen due to its relative simplicity and readability when compared to other approaches. As the results from the terrain generation aren't always clear regarding the decisions being made ”behind the curtain”, it was considered important for users to have the option of viewing the process behind the decisions being made to their terrains when choosing which ones they prefer. This conjoined with our goal of users not requiring technical expertise in order to generate interesting terrains, make the decision trees the most favorable choice.

Finally, **genetic programming**'s recursive evaluation was the deciding factor in choosing it as the evolutionary approach to this work, since the chosen algorithm would require evaluating ever growing decision trees.

Chapter 3

Methodologies

This chapter presents any relevant decisions, conclusions, approaches and algorithms that define the solution design.

3.1 Terrain

An important aspect to keep in consideration when choosing how to represent the terrain was the main purpose of this project. While using mesh terrains would most likely result in more realistic looking results, there was a bigger focus on withdrawing conclusions from the evolution of decision trees which determine the type of terrain to be generated, as well as verifying the feasibility of such a method. With these considerations in mind, a voxel grid world was chosen as the type of terrain representation. Representing the terrain with multiple individual voxel offers a more drastic contrast between features within the terrains than a mesh representation would, with the later resulting in a smoother terrain due to the interpolation required when procedurally generated. This contrast allows a better comparison between the different terrain results as more subtle details could go unnoticed when taking human error into account. This is especially true when using an approach similar to the game Minecraft, in which each voxel is of a given type of block, and each type of block has their own individual texture. This also helps distinguish the voxels apart more easily than, for example, an approach where each voxel would be colored in a gradient according to their elevation or surrounding voxels.

Using a voxel grid instead of height-maps also allows us to generate overhangs which, even though is not the focus, can result in more interesting and realistic looking terrains. This isn't always the case, however, as there are situations where being a voxel grid can result in odd looking features (i.e. flying mounds) and when looking for a flatter terrain such as some plains or a beach an height-map could likely generate better looking results.

The algorithm chosen for generating the terrain was a C# implementation [9] of Ken Perlin's improved noise function [24], adjusting the function call to take custom parameters of frequency and amplitude into consideration. This noise function uses a set permutation table in order to both boost the speed of the algorithm as well as include deterministic results for better comparison, as when evolving the decision trees it is important to generate the same terrain shape to achieve better comparisons. However, using the same permutation table every time makes it impossible to generate different terrain shapes. This was solved by allowing the user to assign a set seed before running the program. Then we can multiply each value from the noise function's permutation table by a random float value (while using the floor function to guarantee each value is an integer), as the usage of a set seed guarantees that querying a random value will return the same order every time. This way each different seed generates a unique permutation table, and thus an unique terrain.

In total there are five parameters (Figure 3.1) used for the generation of the terrain, these are the width/length (*maxDim*) and height of the terrain (*maxHeight*), used for defining the desired size for each generated terrain, the *amplitude* and *frequency* of the noise function, with the first referring to the range at which the results can be in and the later the period at which data is sampled, and a minimum noise value (*minNoise*) which determines whether a voxel on the grid should be empty or not, depending on the noise value assigned by the algorithm. The terrain dimensions chosen for the particular examples shown were $maxDim = 64$ and $maxHeight = 32$.

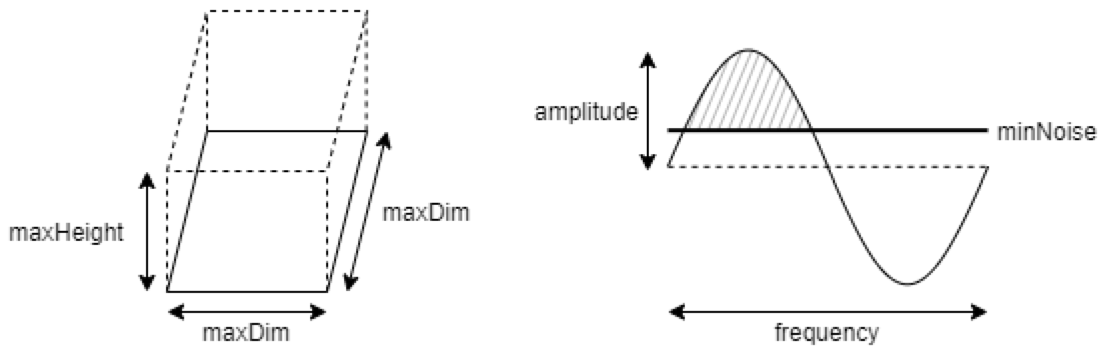


Figure 3.1: The five parameters used for the generation of the terrain.

Terrain generation is done in two main steps: first, the noise function will assign a noise value to each voxel on the 3-dimensional grid, instancing the voxel with a random block type if the noise value is above the set minimum. Afterwards, we traverse through the voxel grid a second time and apply the respective decision tree to each individual block, which will determine what block type each voxel in the terrain should be and change them accordingly. In order to improve rendering performance, during this step any voxel that would be enclosed by other voxels is also removed, as ultimately they would never be visible unless the user could in some form remove the surface blocks. This results in the generated terrains being completely hollow, improving the performance substantially. Figure 3.2 shows an example of the terrain results from each of the two steps side-by-side.

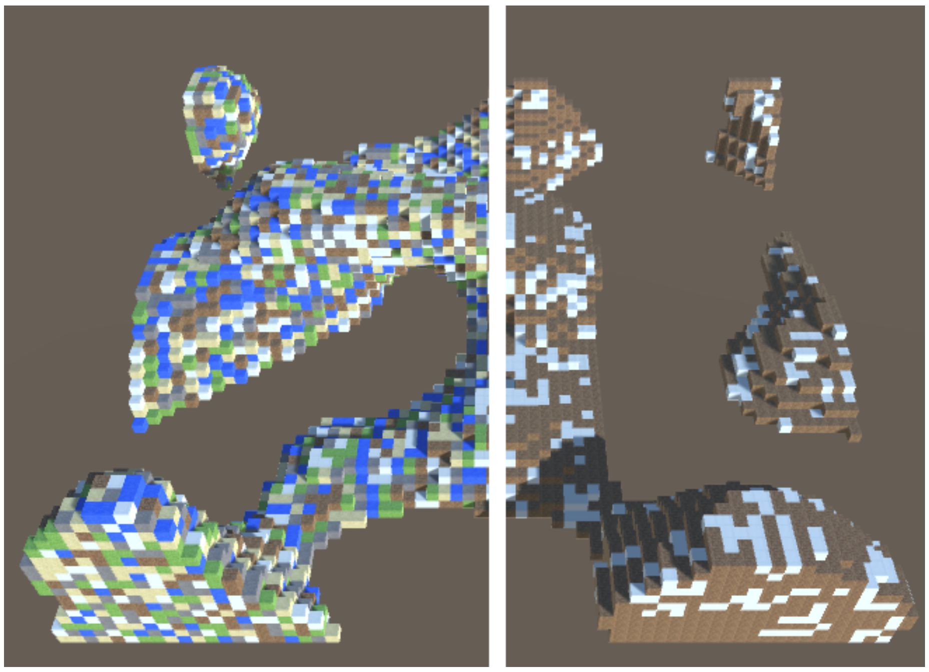


Figure 3.2: Comparison between the first terrain generation step (left half) and the second step (right half).

When choosing how many and which types of blocks to use in the terrain generation the goal was to have a good variety without an abundant amount of them. Having a short amount of block types would lead into little to no terrain variety, whilst too many could make it very difficult for the decision tree to converge. Ultimately the following six (Figure 3.3) block types were chosen: dirt, grass, sand, snow, stone and water. While water generation is usually considered different to normal terrain generation due to its complex and unique features, a water block type was considered in order to better gauge the project's robustness, as including relatively believable water in a terrain was thought to prove a more difficult task.

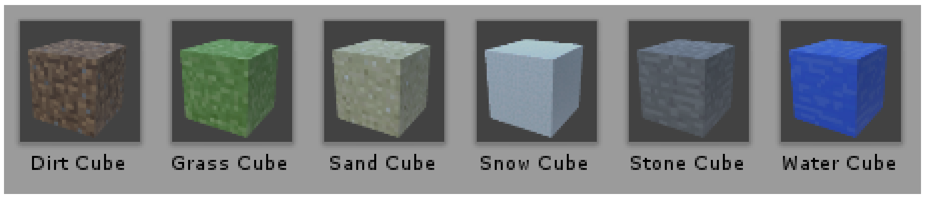


Figure 3.3: The six different chosen block types.

Chunks

When procedurally generating terrains it is possible for the world to be theoretically infinite, however this begs the question of how to both store and load the terrain's data. In the case of Minecraft the approach was to separate its world in 16x16 segments, called chunks [6]. Because generating a theoretically infinite world at once would take more processing time and storage than one would realistically be willing to afford, Minecraft opts to instead generate its terrain as the players move through the world exploring new chunks. Chunks are stored and loaded as they're needed according to their distance to a player, with the default being a radius of 12 chunks around each player, and if any chunks in that radius have not been generated yet that generation is done in runtime using the world's predefined seed. This concept has the advantages of only storing as much terrain data as is required by the players and being able to generate expansive worlds without a large initial terrain generation, but it also has the disadvantage of being more dependant on the user's own machine, as users with lower end machines could end up with a reduced overall experience due to the chunks not loading in time, resulting in wide gaps in the terrain.

While the option of using a chunk-like approach to this project was considered, as it could be useful for the user to better visualize the generated terrain, the idea was ultimately scrapped as the generated portion of the terrain was thought to already give enough of an overview of its features.

3.2 Decision Tree Structure

The chosen decision tree structure was that of a binary decision tree that would apply on each individual voxel in the terrain, with custom functions acting as the decision nodes and the leaf nodes dictating what type of block said voxel should be from the list of block types chosen (Figure 3.4). Two different functions were considered relevant for this project:

- Height - this function takes the voxel's height position in the grid and compares it to a given value, using a given comparison operator. The given value must belong to the $[0, maxH]$ interval, where $maxH$ represents the maximum terrain height, while any comparison operator from the $\{<, >, =, <=, >=\}$ set is valid;
- Neighbours - this function takes the voxel's position in the 3-dimensional grid, counts the number of its adjacent voxels of a certain block type and then compares that count to a given value, using a given comparison operator. As the maximum amount of possible adjacent voxels in a 3-dimensional grid (including diagonals) is of 26, the given value must belong to the $[0, 26]$ interval, while any comparison operator from the $\{<, >, =, <=, >=\}$ set is valid.

These two functions were the only ones chosen in an attempt to allow the generation of interesting terrain features while keeping the project as a proof of concept. Depending on the end user's goal additional functions could be modularly added, however for the purposes of this project the results found from testing these functions were satisfactory.

When initially generating a new decision tree a random function is generated as the root with two leaf nodes, with the tree being traversed through in a breadth-first search order afterwards. Each leaf node of a function has a $(x^{-2} * 100)\%$ chance of being mutated into a random function node with two random child leaf nodes, where x represents the number of function nodes currently present in the decision tree. This allows some initial variety in the decision trees generated, whilst limiting them to a small number of tree levels.

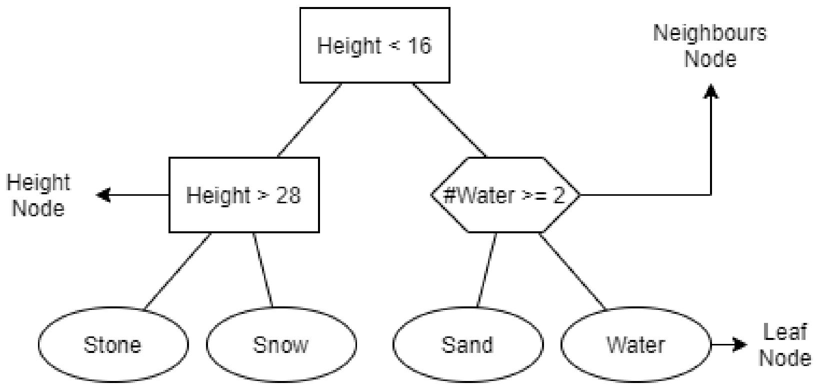


Figure 3.4: A visual representation of the different node types.

3.3 Genetic Evolution

3.3.1 Crossover

Since in this particular evolution problem we want a terrain to evolve according to the user's criteria and not a set algorithmic definition of a realistic terrain, some user interaction would be necessary to determine which terrains were fit to crossover. There ended up being two different approaches for the crossover algorithm.

The first approach consisted of a traditional fitness-based approach, in which the user would assign a fitness value from an interval to each terrain, followed by the algorithm crossing over the different decision trees with increasing chances for the better fit to be chosen to evolve into the next generation. In genetic evolution keeping some of the lower fit genomes might sometimes prove beneficial for innovation, as a poorly performing individual may contain good features that haven't been properly fleshed out yet and are currently outweighed by the rest of its genome. Since in this case we manually decide the fitness values, however, in theory we are free to eliminate the lower half scoring of genomes as any such beneficial feature would be noted and properly evaluated by the user. Then, the higher half would be crossed over, filling in the now free spots in the generation for their children. Even though in theory this approach made sense, after some analysis several flaws were found with this implementation. Since we are evolving the decision trees themselves, but the user's judgement is based on the terrain representation generated from those trees, the user can't always make absolutely correct decisions based on what they perceive, because there will always exist branches in the decision trees which are not visually represented. Allowing the user to visualize the decision trees themselves might help soften the problem, but considering how large the individual trees can grow it's not realistic as a permanent solution. The argument that any beneficial feature would be noted by the user does not then hold true in this problem, and thus removing the lower half of the population outright could prove more harmful than beneficial. Another flaw found with the implementation was that even though by keeping the higher half of the population between generations we guarantee that the progress made is not lost, it also means that only the lower half of the population is innovating in each generation, severely slowing the evolution. There was an iteration of the algorithm in which this problem was addressed by removing the parent genomes from the population, and instead crossing over each pair twice, but since the user's goal for the terrain may be somewhat abstract, resulting in an inaccurate fitness function which can drastically change each generation, this makes it difficult if not outright impossible for the population to converge towards an exact goal, and ultimately results in an overall loss of progress.

Due to the aforementioned flaws a second non fitness-based approach was attempted, and an approach based on a research article [21] by Frade et al. (2009) was found suitable. In this approach the user can select one or two individual terrains to create the next population. If a single individual is selected, the following generation will consist of clones of said individual with only the mutation operators being applied to each decision tree, allowing the generation to present few variations from the selected terrain and evolving slowly. If instead the user opts to select two individuals the next generation's population will consist of the results from crossing over the decision trees from the selected terrains with mutation operators applied afterwards, allowing the population to present more diversity. While this approach doesn't fix the problem of the user's judgement being based on the terrain representation, it helps tackle the problem of their likelihood to score individuals inconsistently thanks to the continuous generation of new forms based on the fittest from the previous generation.

The crossover algorithm (Figure 3.5) itself starts by first taking both parent decision trees and selecting a random node in each. Afterwards, the selected nodes and their respective subtrees are swapped. Finally the mutation operators are applied to each tree, and the two resulting decision trees are returned as the crossover offspring.

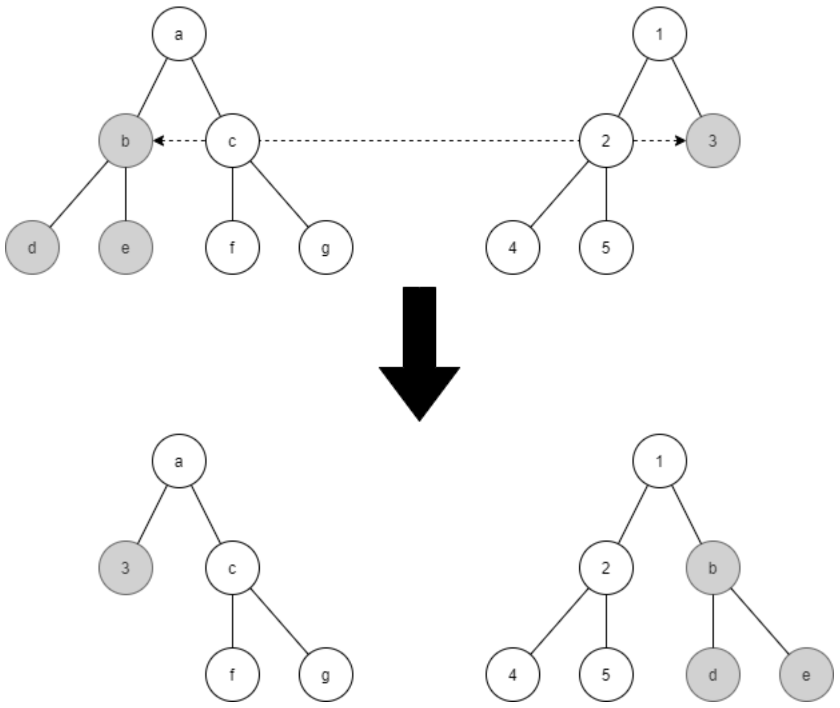


Figure 3.5: Example of the crossover between two decision trees.

3.3.2 Mutation

There are three mutation operators that can take place in each child decision tree after crossover. These mutation operators each have their own chance rate of being applied to each node in the decision tree, and the tree is traversed through in a breadth-first search order.

Value Mutation

The first and simplest mutation was dubbed the Value Mutation (Figure 3.6), as it has a chance to change the different values present in a node: for a leaf node this means that the corresponding block type may be changed; for a height node both the comparison operator and the value being compared against can be changed; finally for a neighbours node the comparison operator, the value being compared against and the block type being checked can all be changed. The rate for this mutation is of 20%.

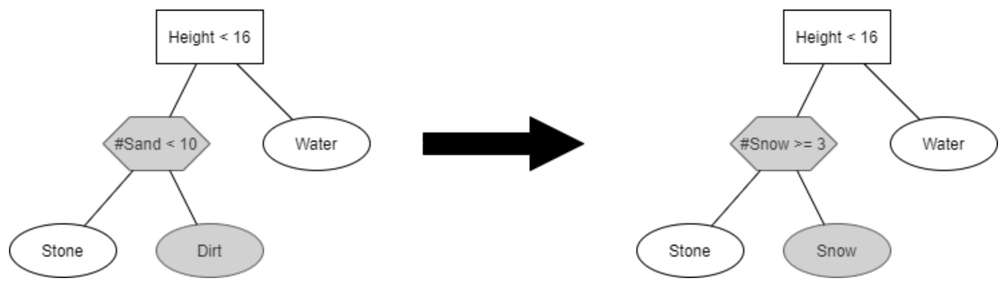


Figure 3.6: Example of Value Mutation being applied to both a neighbours and a leaf node of a decision tree.

Function Mutation

The second mutation was dubbed the Function Mutation (Figure 3.7) and it cannot be applied to leaf nodes. This mutation may change a decision node's function from height to neighbours or vice-versa, but it can also maintain the function type and change only the function's values in a fashion similar to the value mutation. The rate for this mutation is of 1.5%.

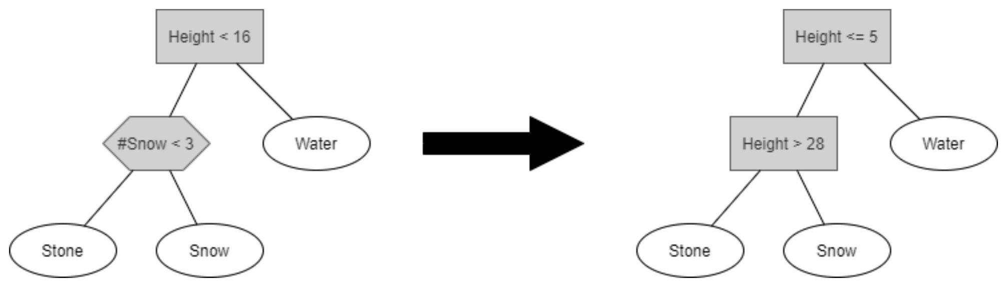


Figure 3.7: An example of Function Mutation mutation being applied both to a neighbours and height node of a tree.

New Function Mutation

Lastly there's the New Function Mutation (Figure 3.8), which can only be applied to leaf nodes, mutating them into either an height or neighbours type decision node and generating two random leaf nodes as its children. In theory this could be implemented as part of the function mutation, as they both change the type of a node. Defining each mutation individually, however, was considered a better option as it allows us to apply differing mutation rates to each one if necessary, as the impact of both operators on the decision tree can be vastly different. Since when applying the mutation operators the tree is traversed in a breadth-first search order, the newly generated nodes from this mutation can also have mutations applied to them. The rate for this mutation is of 20%.

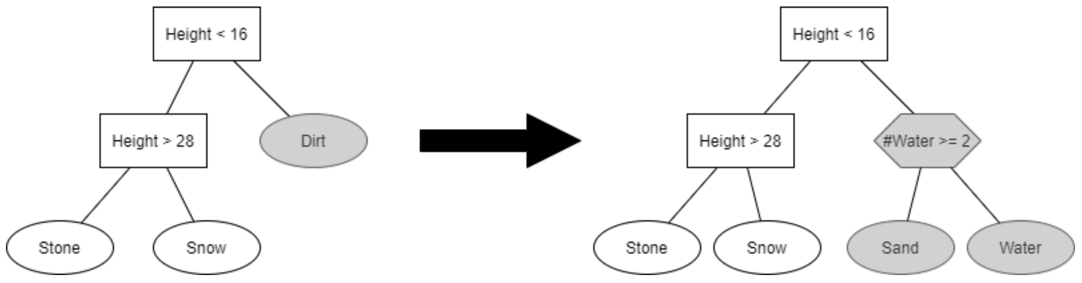


Figure 3.8: Example of New Function Mutation being applied to a leaf node, replacing it with a neighbours node and two new leaf nodes.

When a new random function is chosen in either Function or New Function mutations, there is only a 20% chance for the function generated to be an height one. This decision was made due to how much more impactful to the terrain height functions are, when compared to neighbour functions which might only affect a few blocks at a time.

Some tests were performed in order to try and find adequate rates for each mutation, however concrete optimal values were difficult to find due to the high subjectivity of the results, leading to the search for optimal mutation rates ending up as a process of trial and error. With low rates terrains evolve more slowly which, even though it might allow more control to the user, leads to scarcer opportunities for innovation, resulting in larger amounts of generations taken for relevant progress to be made. If the rates are too high on the other hand the terrains may evolve drastically, making it harder to maintain desired features.

While for the Value Mutation different mutation rates between 1% and 50% did not seem to affect the results much, the same cannot be said for the other two mutations. Since both Function and New Function mutations are able to drastically change the structure of decision trees they were also the ones found to have the most restrictions when searching for satisfactory mutation rates. For the Function Mutation, rates above 5% were found to be already too high to be able to maintain desired terrain features without them being drastically mutated, while for the New Function Mutation rates above 30% caused a great number of new nodes to be generated since mutations are also applied to them, leading to the algorithm becoming unresponsive in a matter of a few generations due to its recursiveness.

3.4 User Interface

3.4.1 Terrain Visualization

The program's interface was made to be as simple as possible while trying to be intuitive to the users. While the plan initially was to have the multiple terrain instances on screen simultaneously, Unity's engine limitations made it so such an attempt would severely reduce the performance, with the interface becoming sometimes downright unresponsive. This is due to the fact that each individual voxel is considered as a different entity that needs to be rendered, the sheer amount of entities being rendered took up a great chunk of Unity's CPU usage. In order to circumvent this problem the approach was changed so that only one instance should be rendered at a time (Figure 3.9), but allowing the user to quickly swap between each instance view without losing significant performance. This approach has the added benefit that since we're only rendering one instance at a time it can freely take up a bigger portion of the screen, allowing for a better terrain visualization without the need of a zoom option.

The number of instances being evolved is configurable in the project settings between a maximum of 30 and a minimum of 2, with the default of 11. For each instance being generated there is a corresponding button on the top-left corner that allows the user to easily swap views without having to cycle through every terrain. The buttons are also color coded to help the user keep track of the terrains: a blue button indicates an instance that has not been visualized yet; a yellow button indicates an instance that has already been visualized; and a green button indicates an instance that has been selected for genetic evolution. Two arrow buttons were also included left and right of the terrain visualization to allow cycling through the instances, with each having their corresponding arrow key as a keyboard bind.

Controlling the evolution of the terrains can be done through: the Select/Unselect button on the middle bottom of the screen (or with the "S" key on the keyboard), which allows the user to select the current terrain to be considered for evolution into the next generation; and the Next Generation button on the bottom-right of the screen (or with the "N" key on the keyboard), which, after having selected the desired instances, allows the user to evolve the decision trees onto the next generation. The number of the current generation is also indicated above the Next Generation button, starting with generation 0.

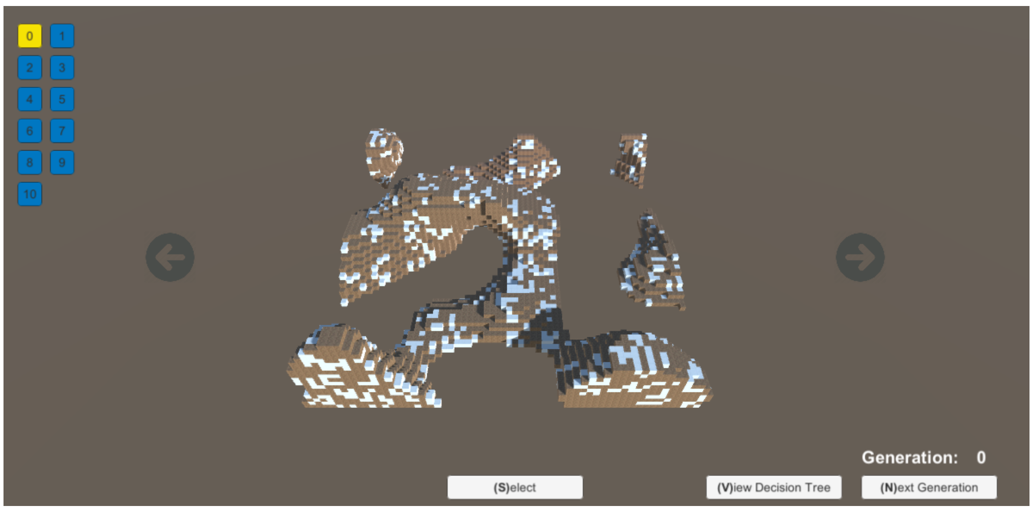


Figure 3.9: The terrain visualization window.

3.4.2 Decision Tree Visualization

In addition to the previously stated interface elements, a button which allows the user to visualize a given terrain's decision tree was also implemented in the bottom right of the screen (or with the "V" key on the keyboard).

This view (Figure 3.10) allows the user to drag the screen with their mouse in order to move through the decision tree as well as zoom in and out with the mouse's scroll wheel. Every non-leaf node represents a decision in the tree in which the next node is either the left child if the decision statement is found to be false or the right child if it is found to be true, colored blue or red respectively. Each node has a text label to indicate the type of node as well as its values, with a legend on the top right of the screen explaining the different types to remind the user.

Initially the goal was to allow the user to be able to manually change the decision tree by interacting with it, in hopes of being able to perfect more precise details that would otherwise be difficult to specify during the genetic evolution, as well allowing them to save the decision tree for later usage. However, as the main focus of this project was on the feasibility of using evolutionary algorithms for procedural generation the virtual worlds, this was deemed as having a low priority and thus ended up not being a topic that was expanded upon.

The algorithm used for drawing the decision trees was an adaptation of an existing [3] approach to the Reingold-Tilford algorithm. In summary, the algorithm follows these steps:

1. Start with a post-order traversal of the tree;

2. Assign an initial X value to each node of 0 if it's the first in a set, or previousSibling + 1 if it's not (in our case the tree is binary so only 0 or 1);

3. Find the desired X value that would center it over its children: if it is the left node its X is set to that value; otherwise, set a Mod property on the node to (X - centeredX) in order to shift its children so they're centered under this node. The last traversal of the tree will use this Mod property to determine the final X value of each node;

4. Determine if any of this node's children would overlap any children of siblings to the left of this node, that is, for each Y get the largest and smallest X from the two nodes, and compare them:

    - If any collisions occur, shift the node over by however much needed. Shifting a subtree only requires adding to the X and Mod properties of the node;
    - If a node was shifted, also shift any nodes between the two subtrees that were overlapping so they are equally spaced out.

5. Do a check to ensure that when the final X is calculated, there are no negative X values. If any are found, add the largest one to the Root Node's X and Mod properties to shift the entire tree over;

6. Do a second traversal of the tree using pre-order traversal and add the sum of the Mod values from each of a node's parents to it's X property.

While viewing the decision tree directly can be useful, it also becomes increasingly difficult each generation due to its fast growth, with even simple looking terrains generating expansive trees that would realistically take too long for a human to analyze. The fact that every individual node and node connection represents an individual object for Unity to render also ends up significantly slowing the generation of the visual representation, eventually reaching a point where the program becomes unresponsive. While these problems were not addressed, they could perhaps be toned down substantially with proper tree pruning, of which there is currently none.

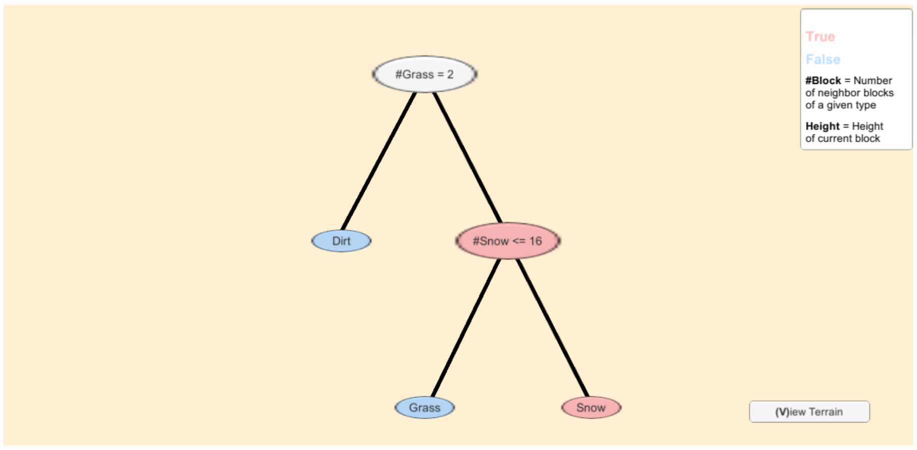

#Grass = 2
Dirt
#Snow <= 16
Grass
Snow
True
False
#Block = Number of neighbor blocks of a given type
Height = Height of current block
(V)iew Terrain

Figure 3.10: The decision tree visualization window.

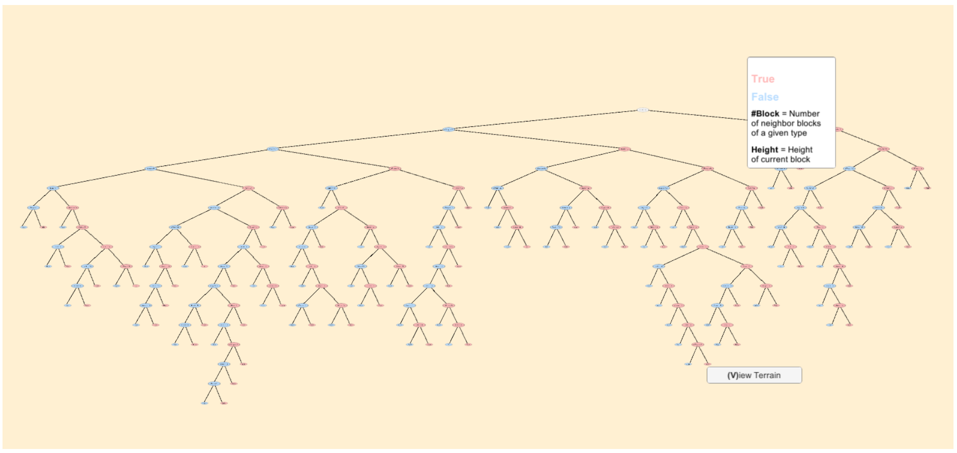

True
False
#Block = Number of neighbor blocks of a given type
Height = Height of current block
(V)iew Terrain

Figure 3.11: Showcase of a more complex decision tree visualization.

Chapter 4

Experimental Results

In this chapter we will discuss the tests that were performed in order to evaluate the viability of the implemented solution as well as their results. User testing was conducted in two parts in order to attempt to achieve results in the most efficient way, with there being seven users in total. The first part of testing consisted on giving the users an hands-on experience with the project, in hopes that the results would allow us a better understanding of the intuitiveness or lack thereof that the terrain evolution possesses, regardless of academic background, as well as whether or not desirable terrains could be generated regardless of subjectivity. For the second part users were then asked to take part in a three-part survey regarding the hands-on experience that they undertook, in order to be able to grasp concrete data regarding user opinions and the project itself from these experiments.

	Gender	Age	Experience
1	F	24	No
2	F	23	No
3	M	15	No
4	M	24	Yes
5	F	27	No
6	M	23	Yes
7	M	25	Yes

Table 4.1: User statistics regarding their gender, age and whether or not they have any experience with genetic evolution.

4.1 Hands-on

In order to be able to gather some data from the hands-on experience, users were all given 4 different terrain goals to try and achieve, each starting off a respective set seed so that results can be more accurately compared. A soft limit of 30 generations was given, so as to have a reasonable time limit for the users, and they were all given the option to either skip or retry the test in case they were not satisfied with the results they got. While trial and error is often a necessity in evolutionary algorithms, one cannot expect the user's patience to be limitless, and giving them this freedom of choice lets us get a better grasp of how plausible the program could be in a more critic environment. In addition, all testing users were given a basic explanation of the project, as well as its goals, and were allowed to try the program before any of the test challenges to familiarize themselves with both the interface and the methodology at work.

The 4 different terrain goals given were the following: "Snowy Mountains", "Grassy Fields", "Beach" and "Regular Mountains". Some of these terrain labels are vaguer than others, but the idea was to let each user come up with their own definition instead of limiting their choices, which is why these labels were given without any corresponding visual representation. In these tests we measure the number of generations taken for each user to achieve a result they find desirable, as well as their self-assessed satisfaction level with the chosen terrain on a scale from 1 to 10.

Snowy Mountains

	# Generations	Satisfaction
1	9	8
2	15	7
3	24	7
4	8	9
5	22	7
6	6	9
7	13	8
Avg	13.86	7.86

Table 4.2: Results for each user when creating a "Snowy Mountains" terrain.

In the first test there seemed to exist two main approaches on the terrain regarding the snow that is present. Terrains from users 1, 2 and 3 appear to focus on placing heavy layers of snow exclusively on higher altitudes with the use of Height Function nodes, whilst the rest of the terrains have heavy snow all around, with some other block types sprinkled in with Neighbour Function nodes. The most common block choices for this challenge are that of snow and stone, which users considered fitting of their image of a "Snowy Mountain", with some cases including dirt and grass as well.

The users' satisfaction level ratings are consistent in the $[7, 9]$ interval, with an average of 7.86, however the number of generations taken fluctuates between a minimum of 6 and a maximum of 24, averaging out at 13.86. This fluctuation is also present between similar-looking terrains, as not only can different evolution steps lead to similar looking results, but there is also the fact that decision trees can be completely different regardless of similarities between resulting terrains. There seems to be, however, a correlation between the number of generations taken and an user's self-assessed satisfaction, with users 2, 3

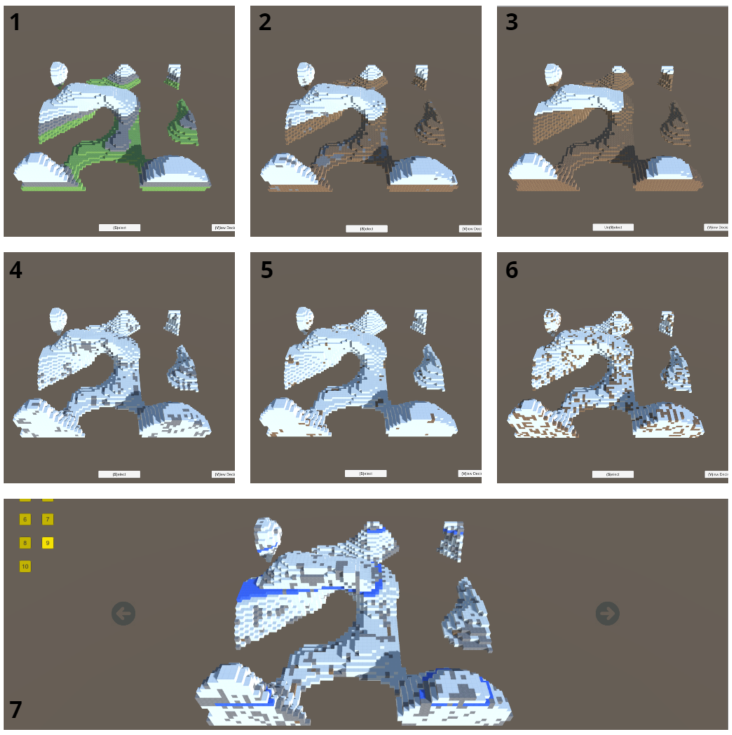


Figure 4.1: Terrains generated by the users when attempting to create a ”Snowy Mountains” terrain. Seed = 73735.

and 5 having taken the biggest amount of generations to achieve a desirable result, whilst also having the lowest levels of satisfaction, whereas users 4 and 6 took the least amount of generations and also assigned the highest levels of satisfaction.

Grassy Fields

	# Generations	Satisfaction
1	4	9
2	0	9
3	23	6
4	2	9
5	9	9
6	5	8
7	0	10
Avg	6.14	8.57

Table 4.3: Results for each user when creating a ”Grassy Fields” terrain.

For this test’s results not a lot of terrain variety exist, with all the terrains consisting mostly of grass blocks, sometimes including small features such as dirt, water or even some snow. This is most likely due to the fact that the image of ”Grassy Fields” usually entails that of vast and green landscapes without many significant terrain features except perhaps some flora, which cannot be properly portrayed with our small pool of block types.

With an average of 8.57, all of the users' satisfaction ratings are consistent in the $[8, 10]$ interval, with the sole exception being user 3, whilst the number of generations taken to evolve the terrain presents similarly positive results. Only the aforementioned user 3's terrain ended up being an exception at 23 generations, as all other terrains took under 10 generations for the users to achieve desirable results, with an average of 6.14. Also worthy of note are the terrains resulting from users 2 and 7, in which they found a desirable result without the need of any genetic evolution, as they considered a fully made of grass terrain to represent their image of "Grassy Fields" the best with the block types present.

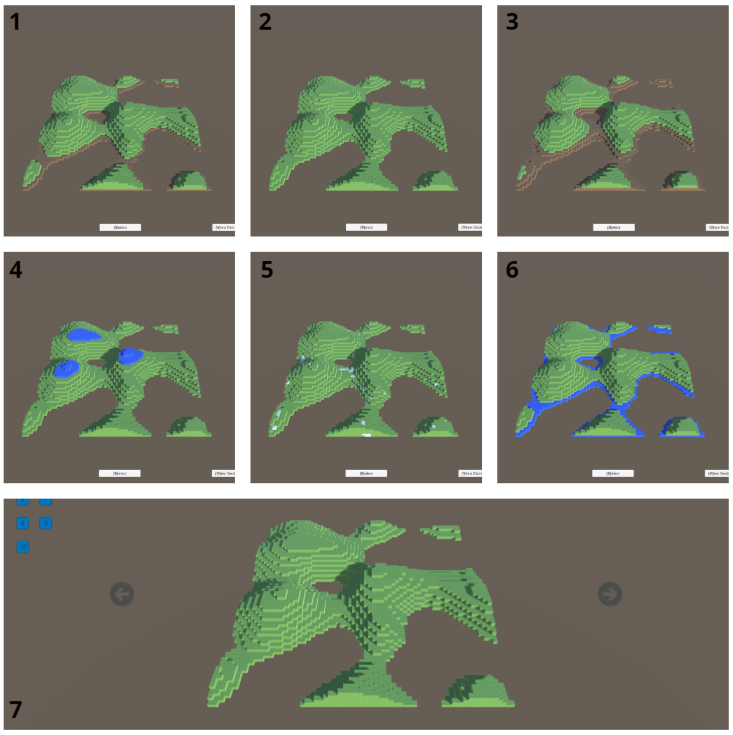


Figure 4.2: Terrains generated by the users when attempting to create a "Grassy Fields" terrain. Seed = 21312.

Beach

Similarly to "Grassy Fields", this test's results do not contain a lot of terrain variety, with almost every terrain consisting of a few water layers on the bottom and sand dunes on top. The only two exceptions to this are user 7's terrain, with some grass on top of the sand, and user 5's terrain, who chose to use dirt instead of sand, as an attempt to create something akin to a volcanic beach with the existing block types.

	# Generations	Satisfaction
1	10	9
2	3	10
3	10	6
4	11	10
5	30	7
6	18	9
7	14	10
Avg	13.71	8.71

Table 4.4: Results for each user when creating a ”Beach” terrain.

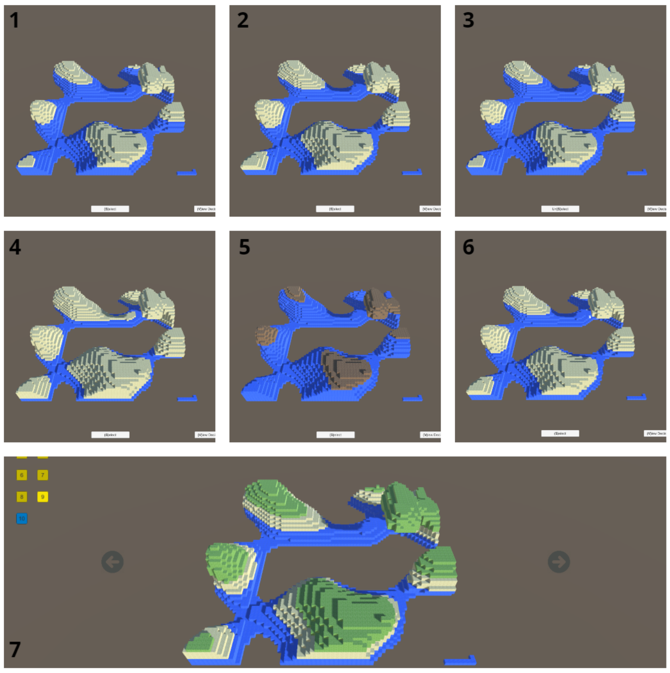


Figure 4.3: Terrains generated by the users when attempting to create a ”Beach” terrain. Seed = 11037.

In terms of user satisfaction the results were very positive. With the exception of users 3 and 5, who claimed slightly above average satisfaction levels, all other users claimed to have either a 9 or 10 level of satisfaction, averaging out at 8.71. This is, according to most users, due to the fact the terrains generated were very close to, if not exactly, what they had in mind. When it comes to the number of generations taken there seems to be a pattern in the $[10, 20]$ interval, with a single positive outlier taking 3 generations to achieve their desired result, and one negative outlier taking 30 generations, averaging out at 13.71 generations taken.

Regular Mountains

	# Generations	Satisfaction
1	-	-
2	-	-
3	1	7
4	2	9
5	17	8
6	18	9
7	33	6
Avg	14.2	7.8

Table 4.5: Results for each user when creating a ”Regular Mountains” terrain.

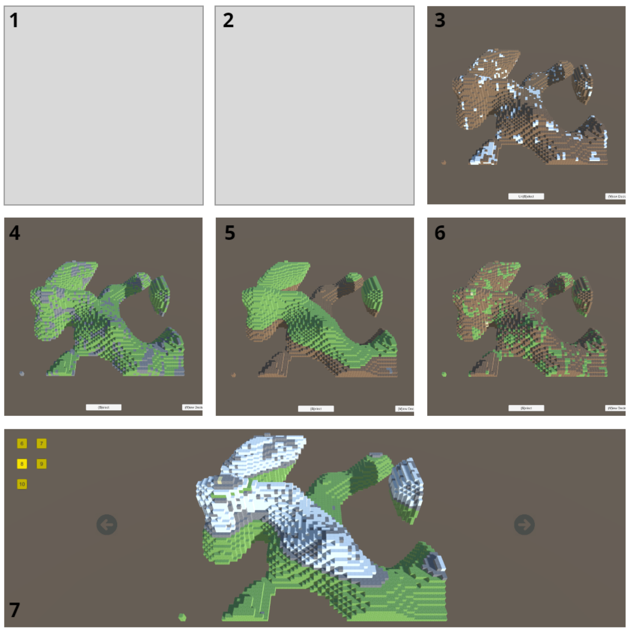


Figure 4.4: Terrains generated by the users when attempting to create a ”Regular Mountains” terrain. Seed = 912384.

Out of all the labels given, the ”Regular Mountains” one was the vaguest, and the impact on the results are noticeable. While the block types present don’t have much variance, the resulting terrains certainly do, with no two terrains looking the same. The most common feature is the presence of grass, intertwined with either stone, dirt or snow blocks in different ways each time.

This test was the only one in which participants gave up on trying to evolve a terrain, with users 1 and 2 not being able to achieve a terrain that would suit their tastes, even after attempting multiple times in the case for user 2.

They claimed that while they were close to their goal terrain, they weren't able to quite get there and the terrains ended up evolving away from the desired solution. As for the other results, there did not seem to be any obvious patterns. The number of generations taken fluctuates in a wide range between a minimum of 1 and a maximum of 33, with an average of 14.2, whilst the satisfaction levels are spread in a $[6, 9]$ interval, at an average of 7.8. These inconsistent results can be considered a direct result of the vagueness of the goal label for this test.

Overview

When comparing the results obtained from the tasks performed we can conclude that users had an harder time evolving terrains in which the goal they intended to achieve wasn't as well defined. Both the "Grassy Fields" task, which had the lowest average number of generations taken to achieve a result, and the "Beach" task, which had the highest average satisfaction value, were considered well defined goals, with most users having a clear image of what kind of terrain they intended to generate and their results all being very similar. On the other hand, terrains resulting from the "Regular Mountains" task had both the highest average number of generations taken to achieve as well as the lowest average satisfaction values, whilst also possessing the most diversity between different user results.

While the number of generations taken clearly has a correlation with the satisfaction levels of users, we can also deduce from the results that it wasn't always the most impactful factor. Taking a look at the individual satisfaction levels of users for the "Regular Mountains" task, for example, there's the case of user 3's terrain taking a single generation to evolve, but still attaining a satisfaction level of 7, whilst user 6's terrain took 18 generations and had a satisfaction level of 9 assigned. Examples such as these are common throughout all tasks performed, but is ultimately a cause of the subjectiveness of each user.

Also worthy of note is the fact that out of all the satisfaction values, the lowest assigned values were of 6, when the possible value interval is $[1, 10]$. While this could indicate that the generated terrains were on average very satisfactory to users, it is just as likely of a possibility that there existed some sort of bias in the self-assigned values.

4.2 Survey

After trying the project hands-on each user was then asked to fill out a survey regarding their experience. This survey was split into 3 sections: the first has questions focused on their overall experience with the program; the second section asks the users how hard they found each individual test to be; and, finally, the third section has pictures of varying terrain types generated by the program, with users being asked to evaluate each one in terms of realism.

User's Experience

In this section users were asked to answer questions regarding how they felt about certain limitations that may have been found during their hands-on experience. Users could reply with a rating from 1 to 10, with higher ratings indicating that they felt said aspect too frequently.

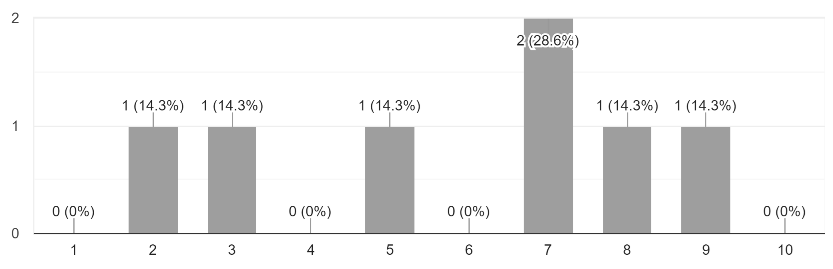


Figure 4.5: User results when asked ”How often did you feel that the tool was limiting you?”.

The first question inquires the users on whether or not they found the program to be limiting their actions. Ideally, this wouldn't often be the case and the user would be able to express themselves freely when generating the terrain, but unfortunately results suggest otherwise. Out of the 7 users 4 often felt limited by the program's constraints. The results weren't completely one sided however, as some users seem to have rarely felt that way.

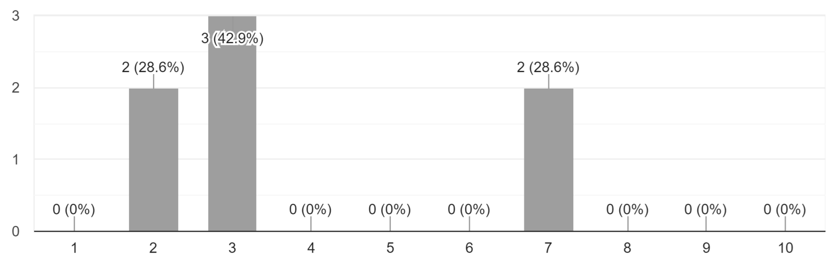


Figure 4.6: User results when asked ”How often did you feel that your choices didn't matter to the changes being made in the terrain?”.

In a worst case scenario, the genetic evolution of the generated terrains would take little to no consideration to the choices made by the user, making the whole process pointless. The second question asks the users how often they felt that ultimately their choices didn't matter to the changes that were being made in the terrain. While there were two cases in which the users felt above average in that that was indeed the case, the greater majority disagreed. It is worthy of mention, however, that the fact that the users felt like their choices mattered doesn't necessarily mean that they thought the impact felt on the terrain to be beneficial.

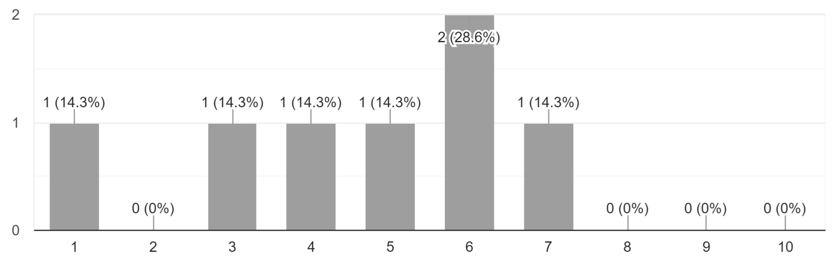


Figure 4.7: User results when asked "How often did you feel that it took time for progress to become apparent?".

Another important aspect to take in consideration is how long it can take for progress to become apparent, as the main purpose of this approach to procedurally generating terrain is to save both time and human resources. When asked if users felt this during their tests, the results were fairly spread out, but mostly around average. This could indicate that while users found the program taking longer to progress than what they'd hope for, it still managed to achieve results in reasonable amounts of time.

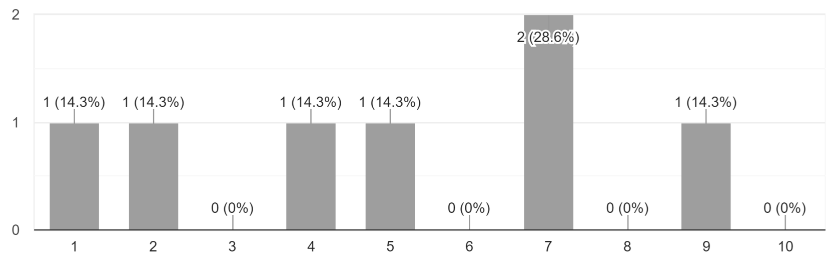


Figure 4.8: User results when asked "How often did you feel that it would be easier to make a terrain from scratch?".

The last question for this section asks if the users often felt that it would be less trouble trying to generate a terrain through any other known means. There did not seem to be a general consensus in regards to this, as can be seen by the fluctuation between answers.

This means that even though some users were satisfied with the results produced with the current tool, there is definitely room for improvement.

Task Difficulty

In the second section users were asked to rate each task they performed on a scale from 1 to 5 in terms of difficulty to achieve a desirable result, where a higher number represents a higher difficulty. Below are the difficulty histograms of the results obtained:

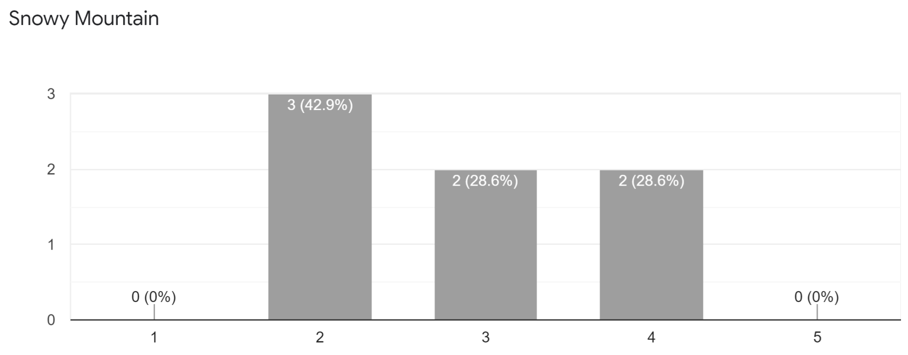


Figure 4.9: User ratings regarding the difficulty of the ”Snowy Mountain” task.

The difficulty levels assigned to the first task (Figure 4.9) are consistently average, with all ratings belonging to the $[2, 4]$ interval, leading the task to be considered as having a medium difficulty, leaning slightly towards the easier end of the spectrum. The average difficulty was of 2.86, with a median of 3.

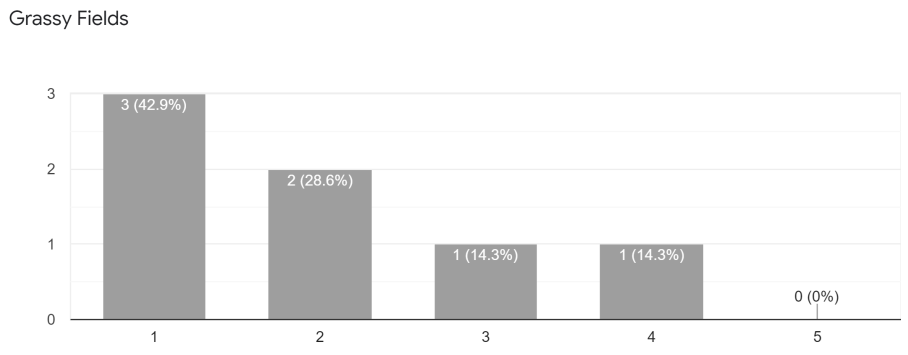


Figure 4.10: User ratings regarding the difficulty of the ”Grassy Fields” task.

The ”Grassy Fields” task had consistently low difficulty values assigned to it (Figure 4.10), with 3 out of 7 users giving it the lowest difficulty score. This is most likely due to the low number of defining features required for users to achieve such a terrain. Both its average difficulty and its median values were of 2.

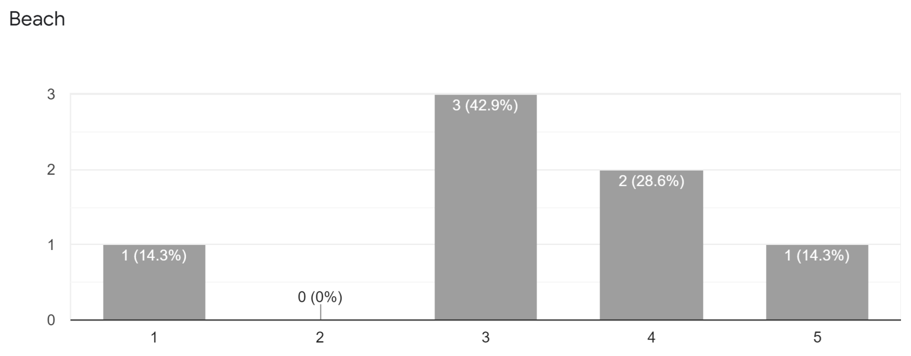


Figure 4.11: User ratings regarding the difficulty of the ”Beach” task.

On the other hand, most users assigned between average and high difficulty values to the ”Beach” task (Figure 4.11), with only a single user finding it to be of low difficulty. Even though the goal terrain of the users for this task seemed to be very well defined, this is also most likely what caused the users to have an higher difficulty achieving a desirable result as it also means that the user will be more critical regarding what qualifies as a desirable result. It had an average difficulty assigned of 3.29, and a median of 3.

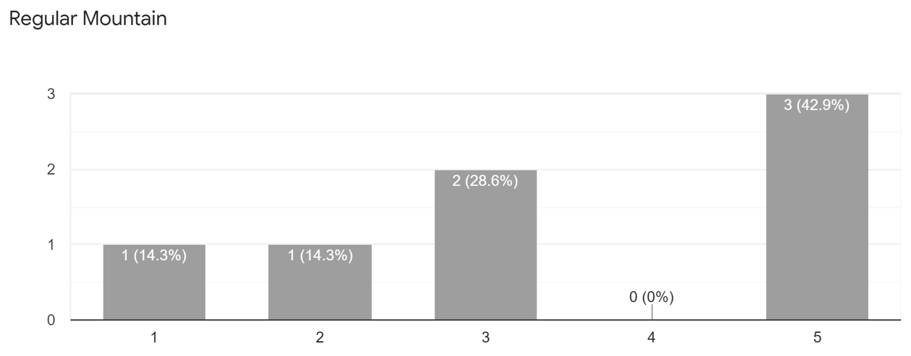


Figure 4.12: User ratings regarding the difficulty of the ”Regular Mountain” task.

The ”Regular Mountain” task being the most vague one is very well represented in the assigned difficulty levels by the users (Figure 4.12), as they were revealed to be fairly inconsistent. While 3 out of 7 users attributed the highest level of difficulty to this task, the other 4 all assigned low to medium difficulty levels, resulting in a very subjective task. Its average difficulty was of 3.43, and its median was of 3 as well.

Difficulty Overview

When comparing this section’s answers to the results obtained from the hands-on experience, it is easy to draw a correlation for most tasks between the difficulties assigned by the users and the average number of generations taken for them to achieve a desirable result in said task. The higher the average number of generations taken, the harder the average difficulty assigned by users. The same could almost be said for the average satisfaction levels, however the ”Beach” task is an exception to this by possessing the second highest average difficulty, whilst being the task with the highest average of user self-assigned satisfaction. This helps drive in further the point that regardless of an higher number of generations taken, which is indicative of an harder difficulty, a user may be satisfied as long as a desirable terrain is met.

Terrain Evaluation

For the last section of the survey 15 terrain images were shown to the users and they were asked to rate them, from 1 to 10, in regards to how realistic they found each terrain to be, where a higher number represents a higher level of realism. The goal with this evaluation was to try and find some less subjective aspects when considering what makes a procedurally generated terrain acceptable. In order to perform some level of test control, terrains 2 (Figure 4.16) and 13 (Figure 4.27) are similar, to make sure we obtain consistent results, while terrains 4 (Figure 4.18) and 14 (Figure 4.28) were randomly generated as a negative test control. Other noteworthy examples are terrains 10 (Figure 4.24) and 12 (Figure 4.26), which instead of generating water as any other block type, always set a global sea level at height 2 (filling all open voxel positions with height below or equal to water blocks).

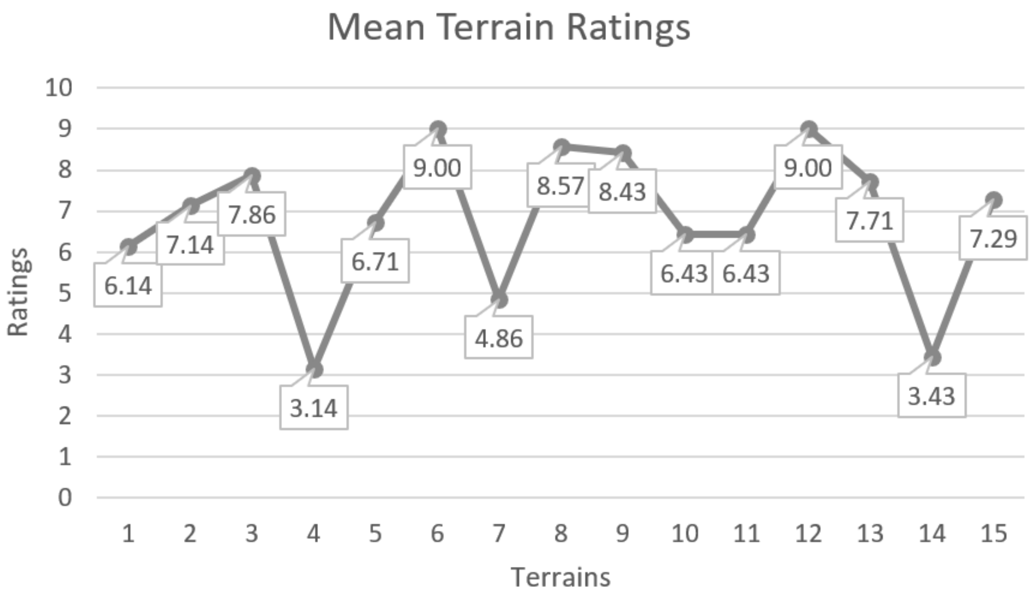


Figure 4.13: Mean ratings of the presented terrains.

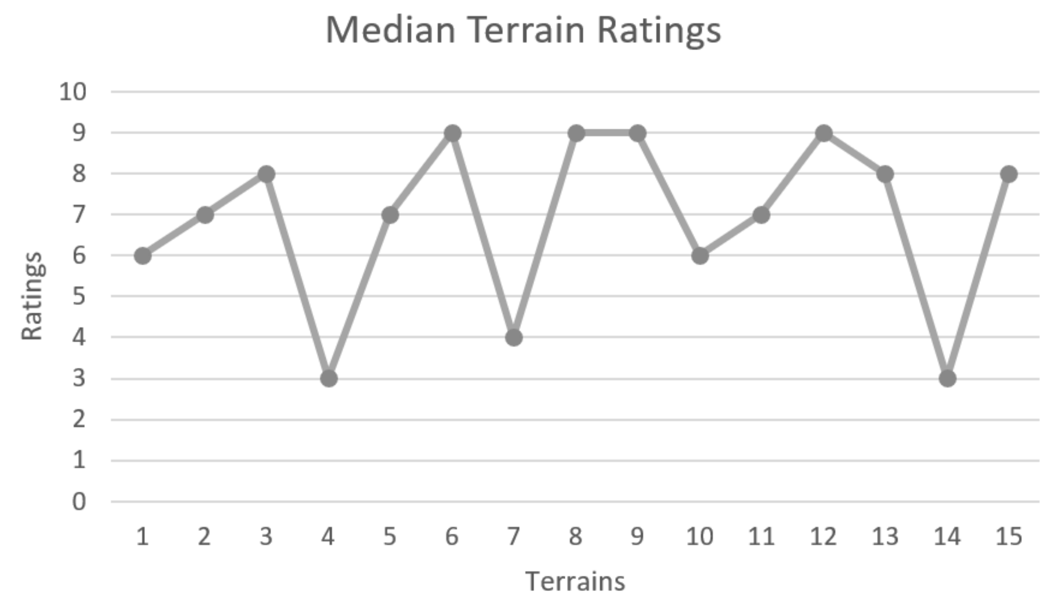


Figure 4.14: Median ratings of the presented terrains.

As can be seen in figures 4.13 and 4.14, when measuring both mean and median values only 3 terrains had below average results. Out of these terrains the two randomly generated ones used for negative control testing are the lowest scoring, while the other is terrain 7 (Figure 4.21). The common link between these three is that they all use the neighbours function to extensive amounts, resulting in a lot of scattered blocks in the terrain, however this also happens in some other terrains such as terrains 9 (Figure 4.23) and 15 (Figure 4.29) while possessing good mean and median results. The main difference between these seems to be the types of blocks being spread across in regards to the rest of the terrain. For example, terrain 7 consists mostly of grass blocks, however, sand blocks can be found spread all across the grass, which is not something that one would commonly find in the real world. On the other hand if we take a look at terrain 9, we'll again see a terrain consisting of mostly grass blocks, but this time there are dirt blocks being spread across, which can look much more natural next to grass when compared to sand.

Regarding the similar terrain control test, all users save for one evaluated both terrains with the same results, indicating a good consistency for most users. There does not seem to be any significant bias towards particular terrain shapes either, as can be seen even between the two terrains with a set global sea level. There also weren't any significant disparities found between the mean and median values of the test results, which might be an indicator of uniform results. However, since this is a rather small data set, no guarantee can be taken away from this information.

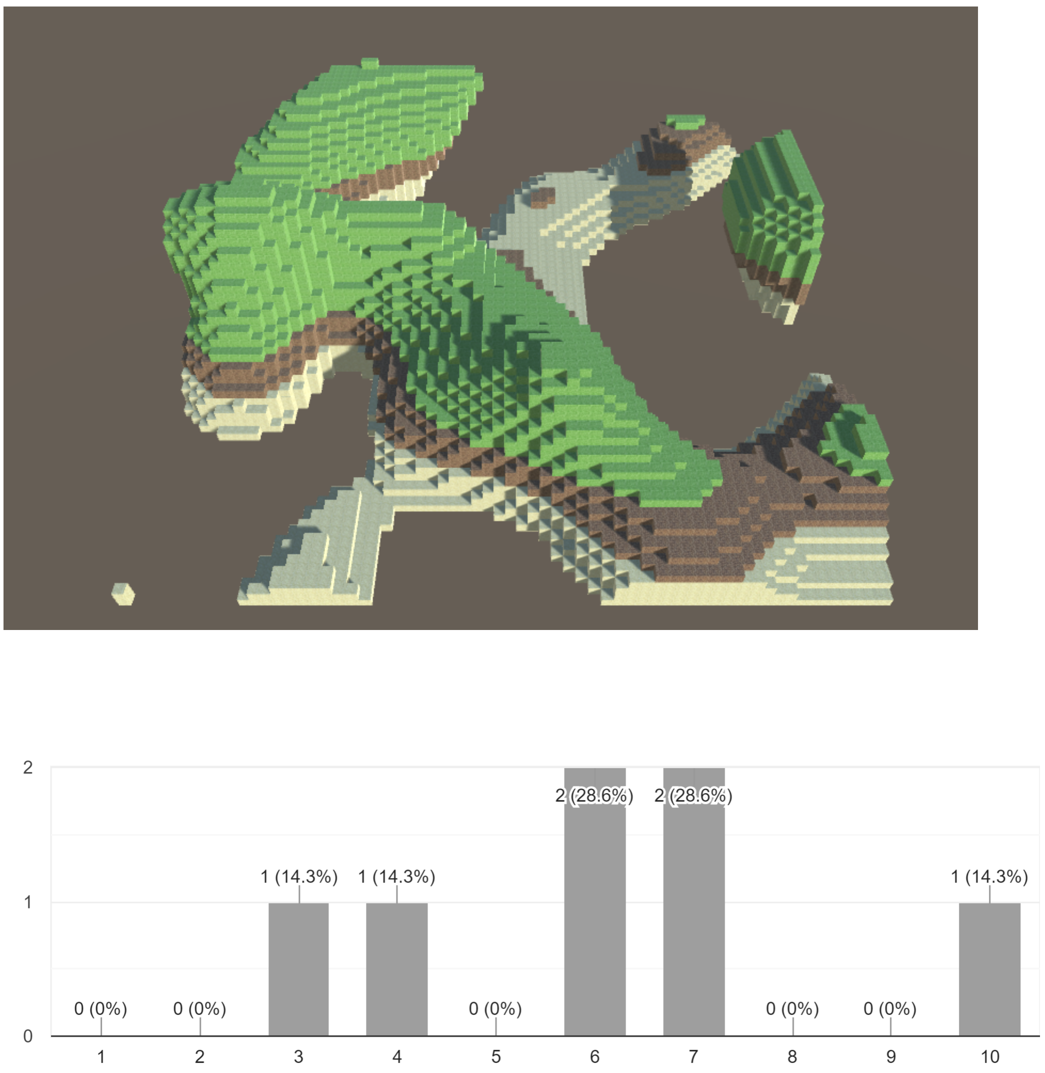
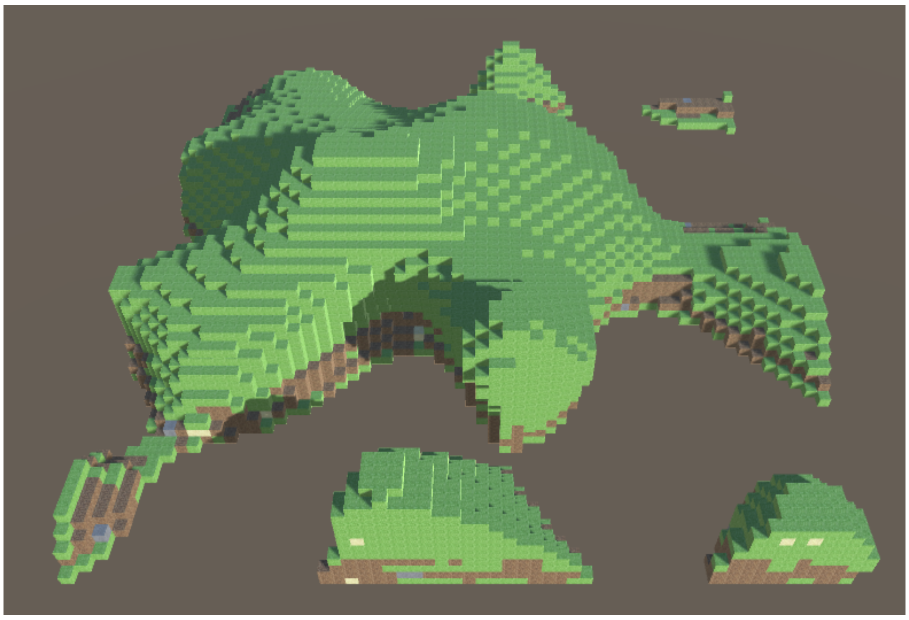


Figure 4.15: Terrain 1 and its results.



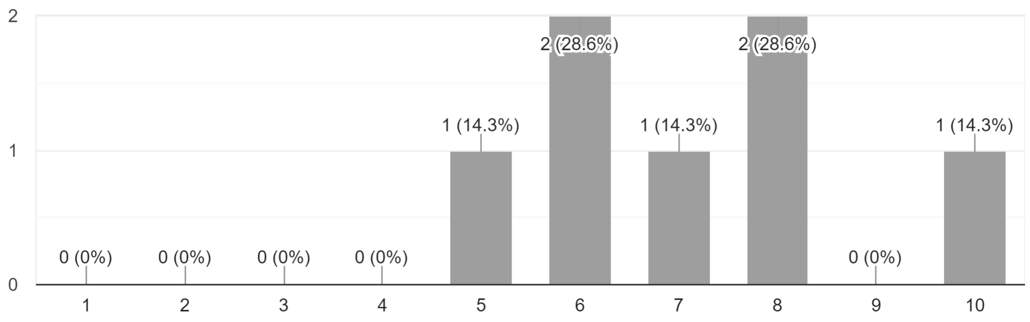
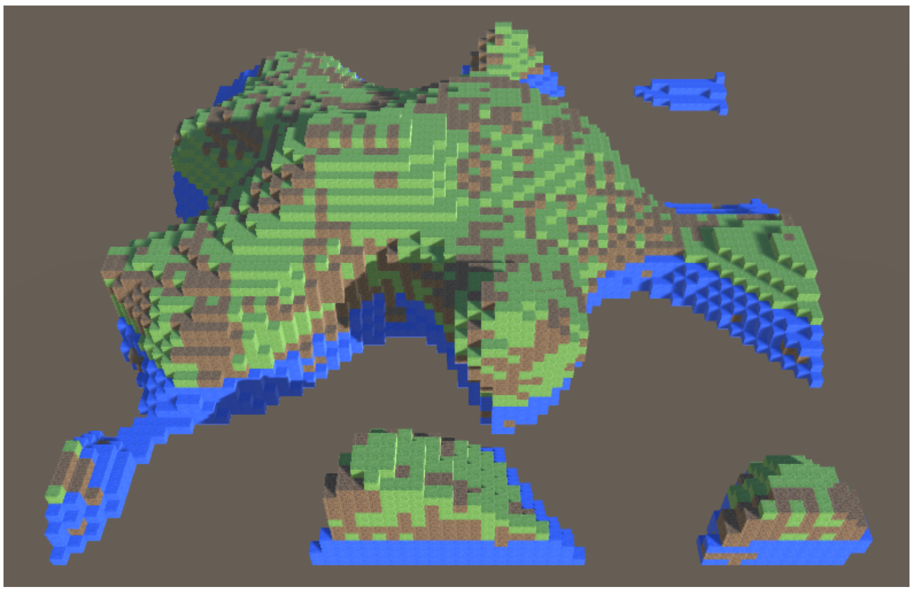


Figure 4.16: Terrain 2 and its results.



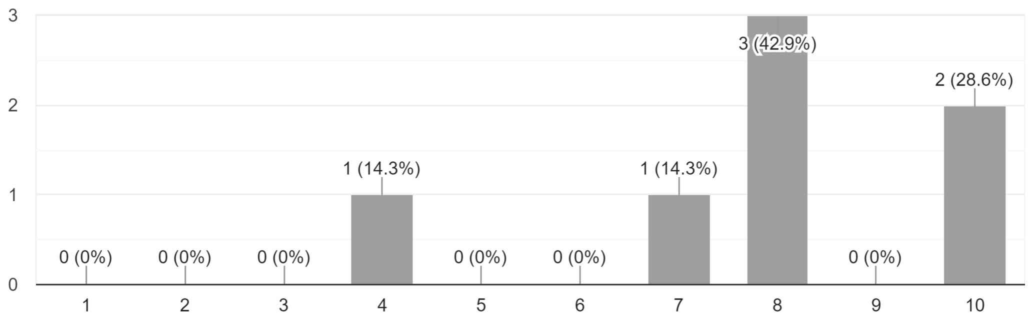
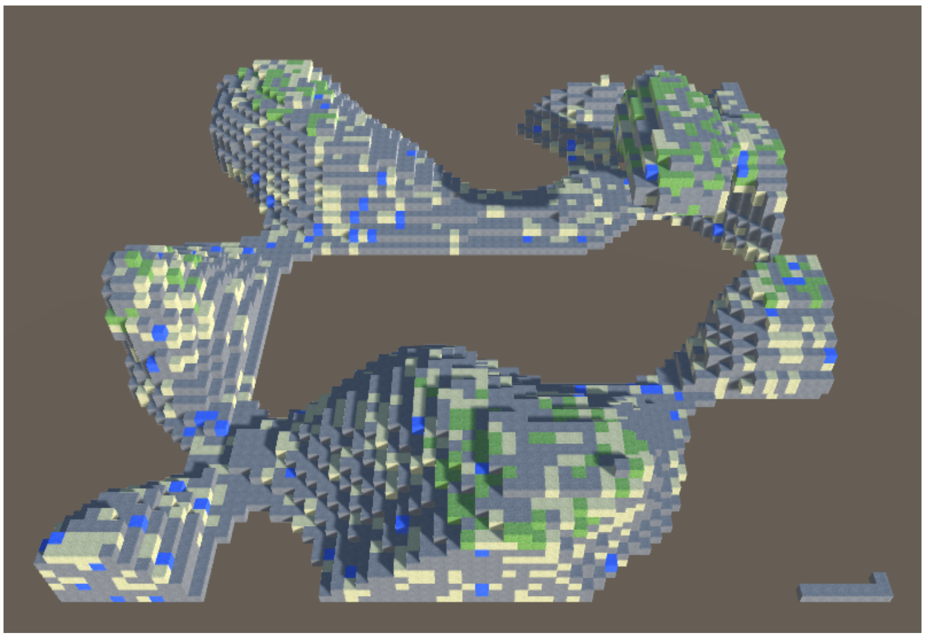


Figure 4.17: Terrain 3 and its results.



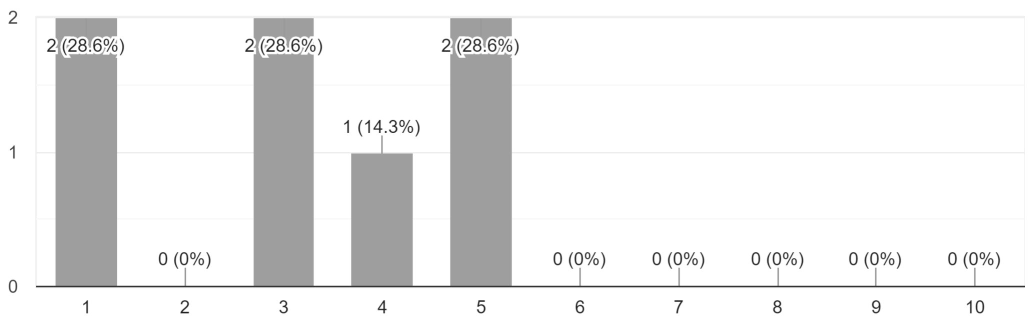
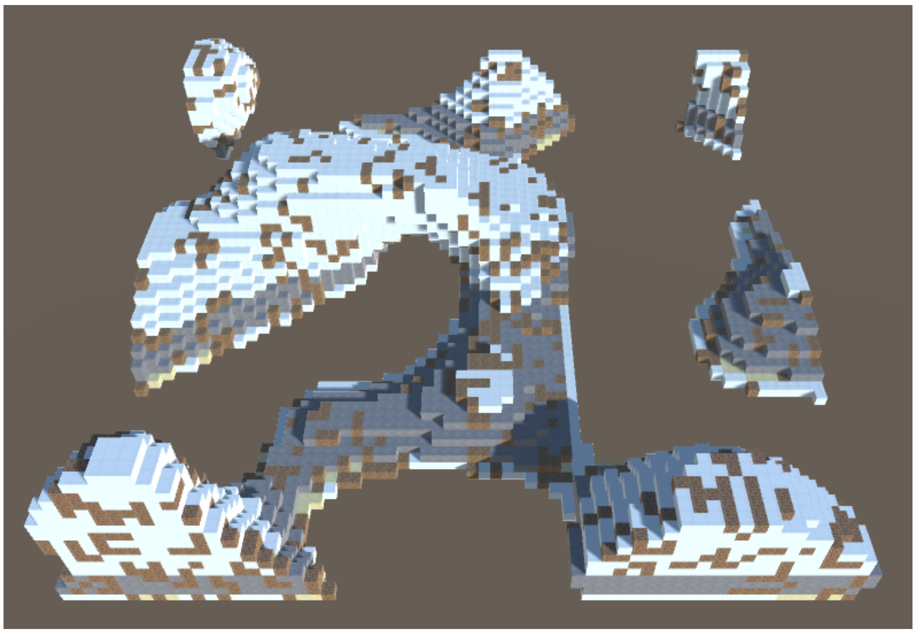


Figure 4.18: Terrain 4 and its results.



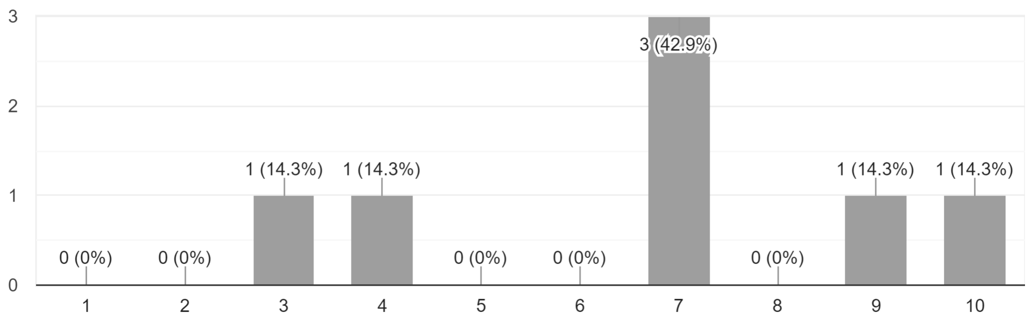
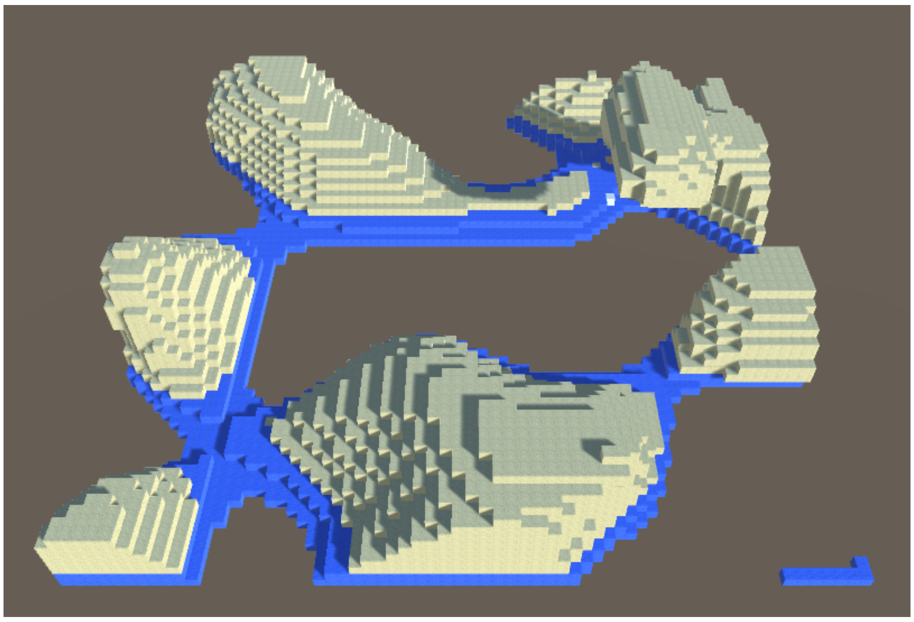


Figure 4.19: Terrain 5 and its results.



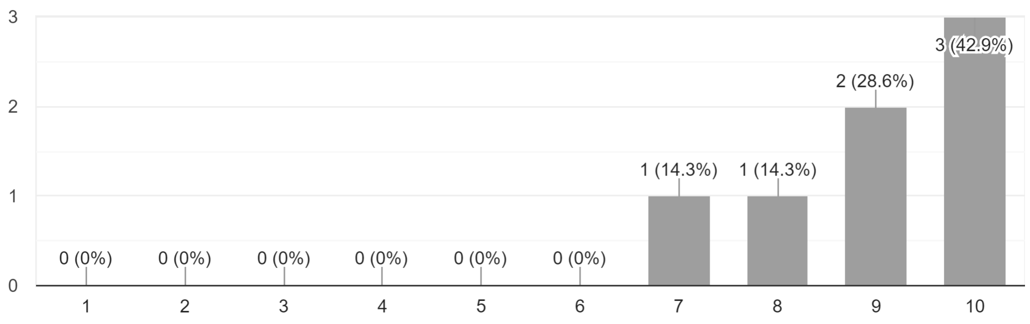
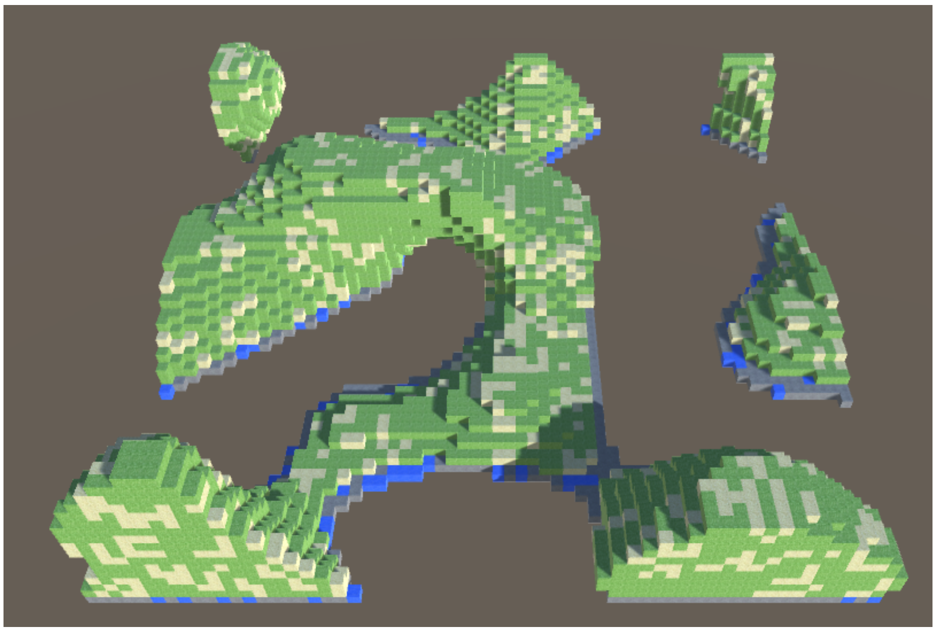


Figure 4.20: Terrain 6 and its results.



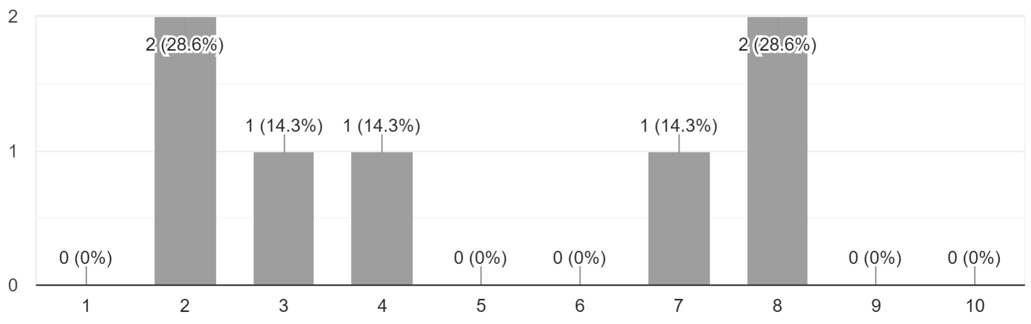
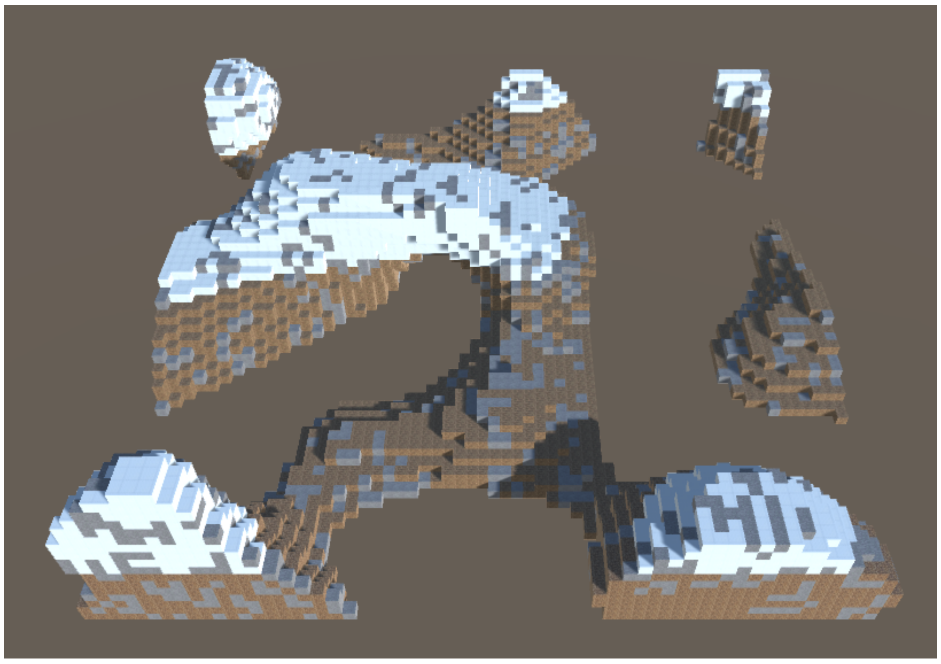


Figure 4.21: Terrain 7 and its results.



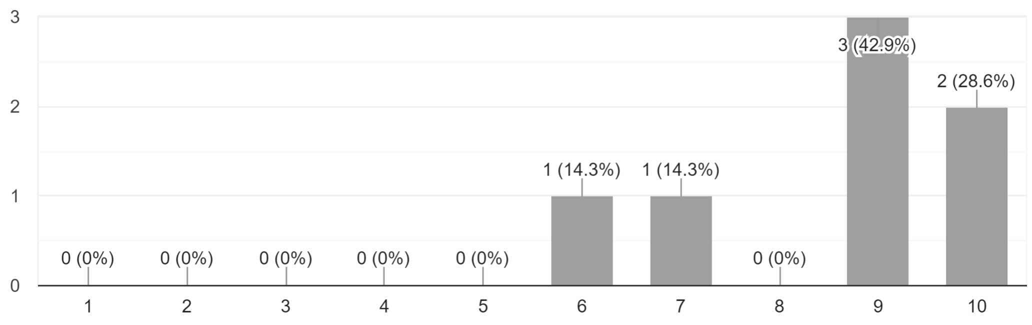
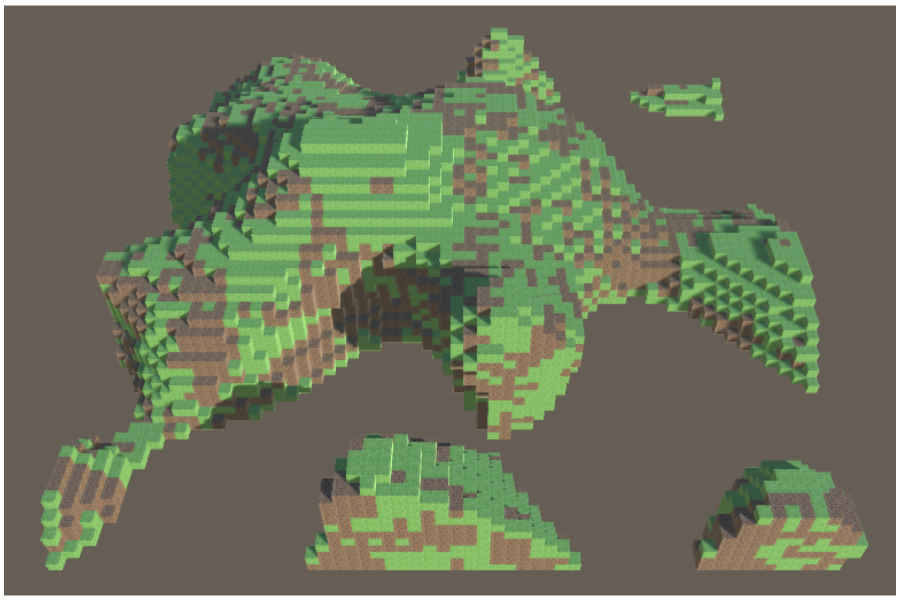


Figure 4.22: Terrain 8 and its results.



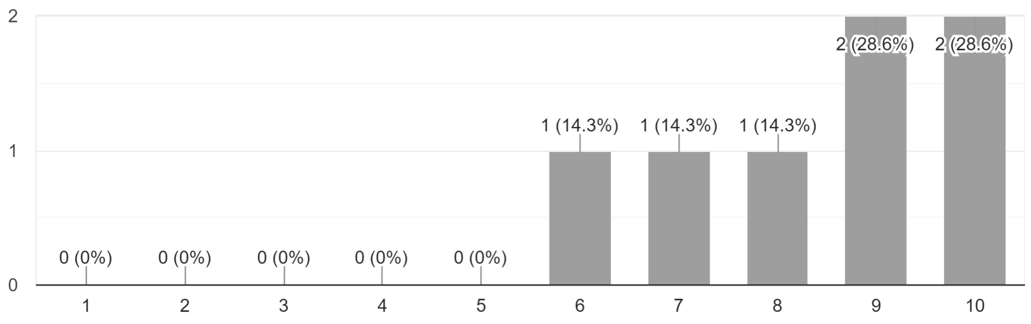
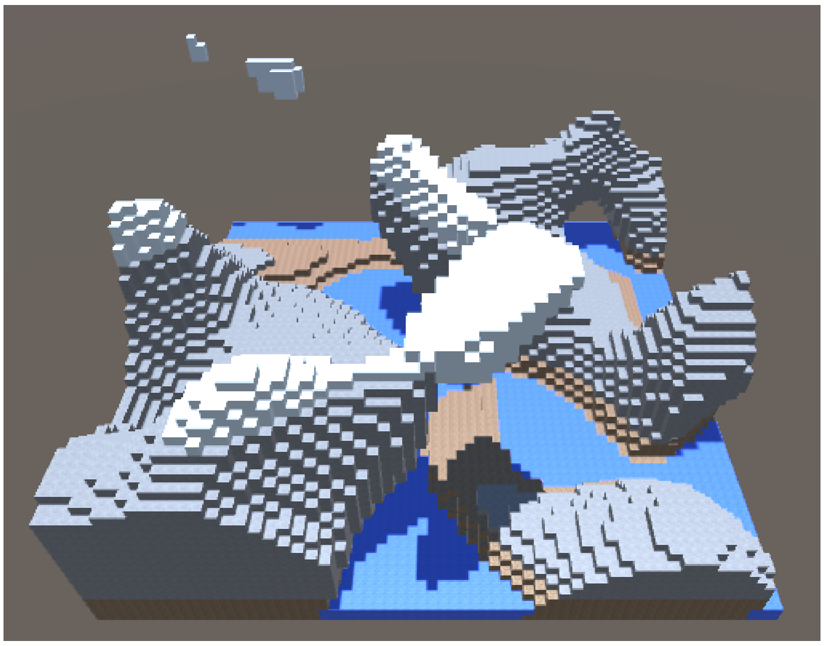


Figure 4.23: Terrain 9 and its results.



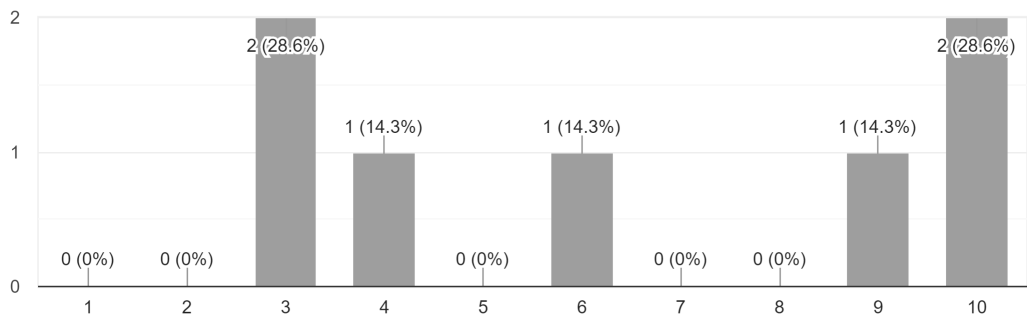
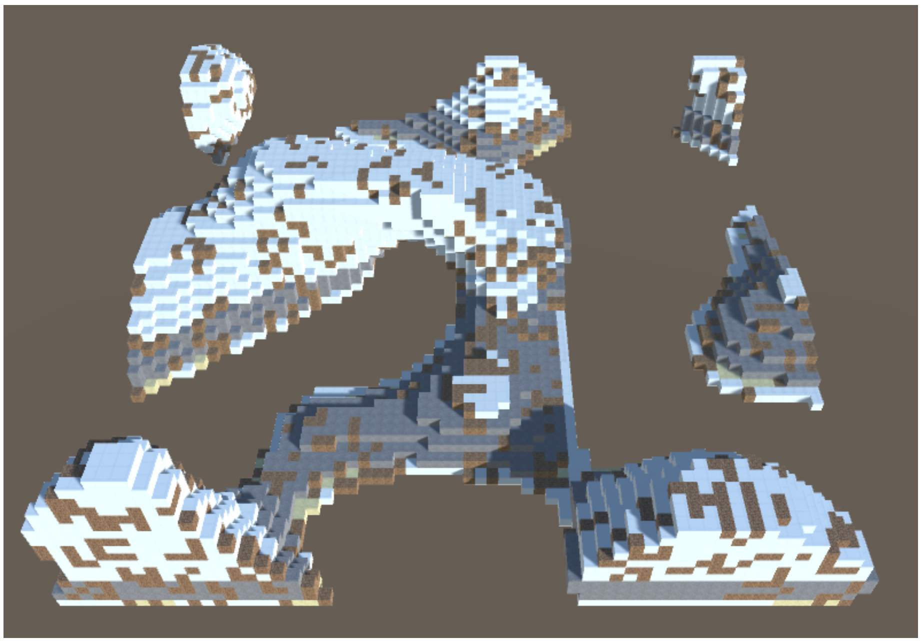


Figure 4.24: Terrain 10 and its results.



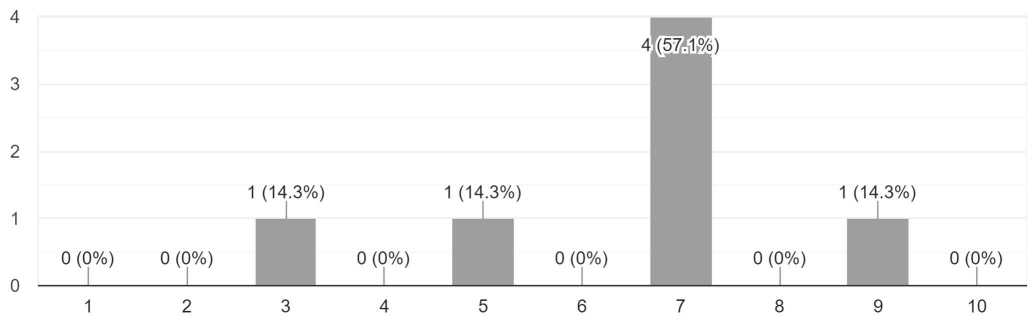
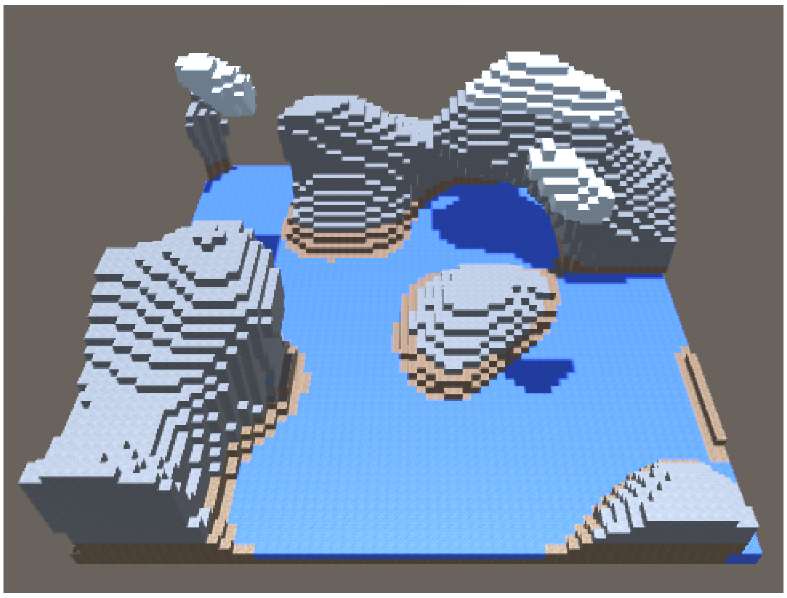


Figure 4.25: Terrain 11 and its results.



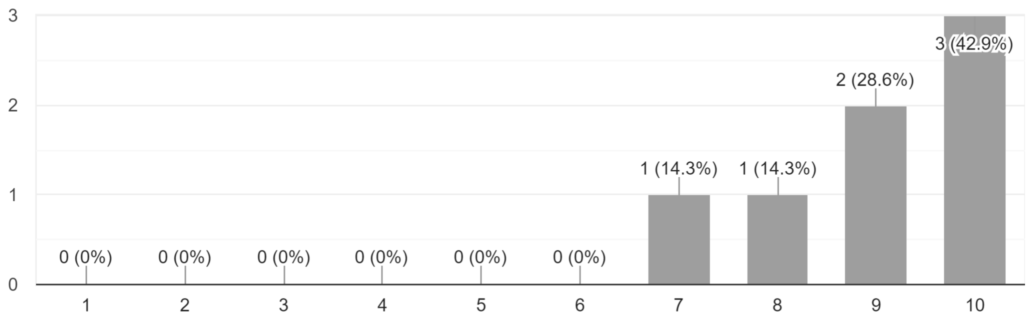
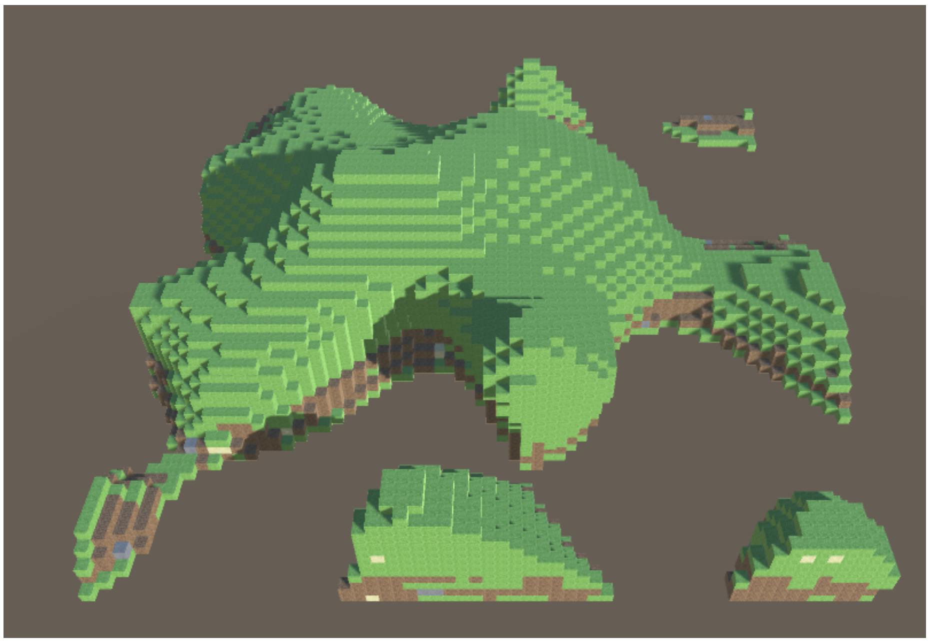


Figure 4.26: Terrain 12 and its results.



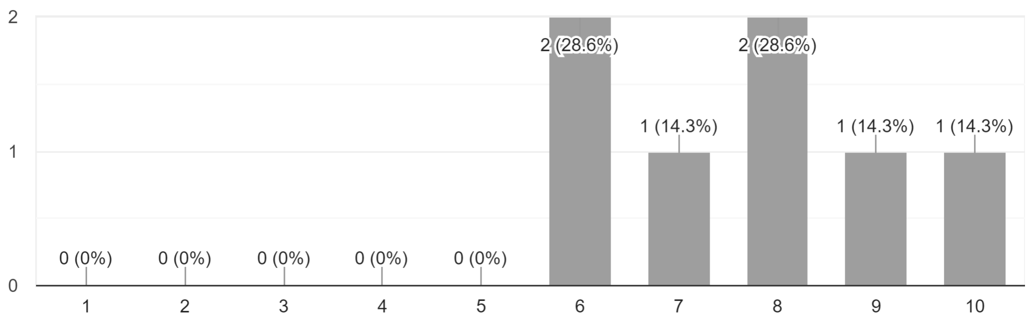
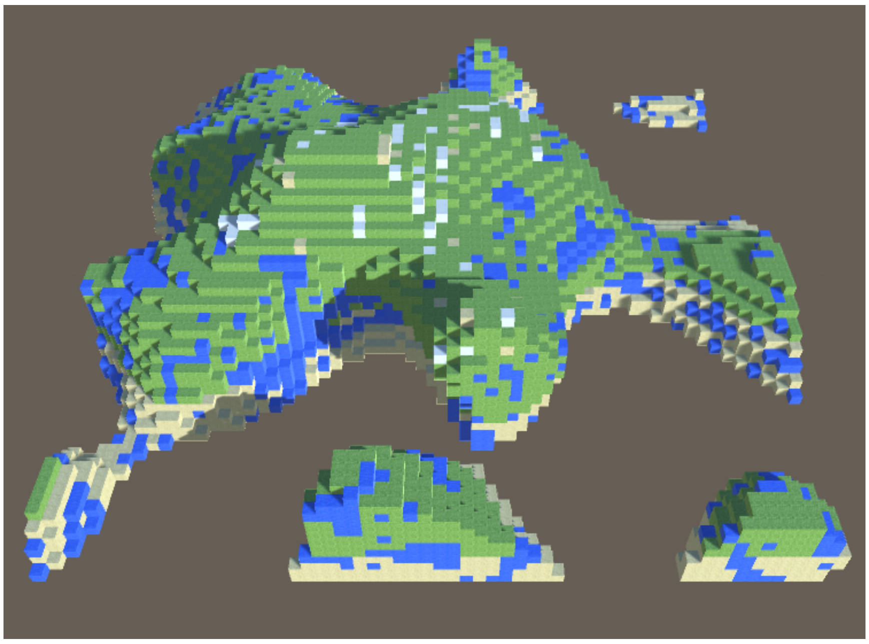


Figure 4.27: Terrain 13 and its results.



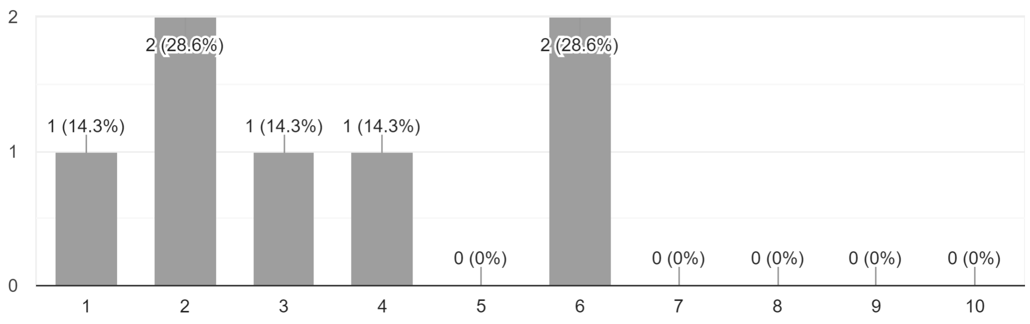
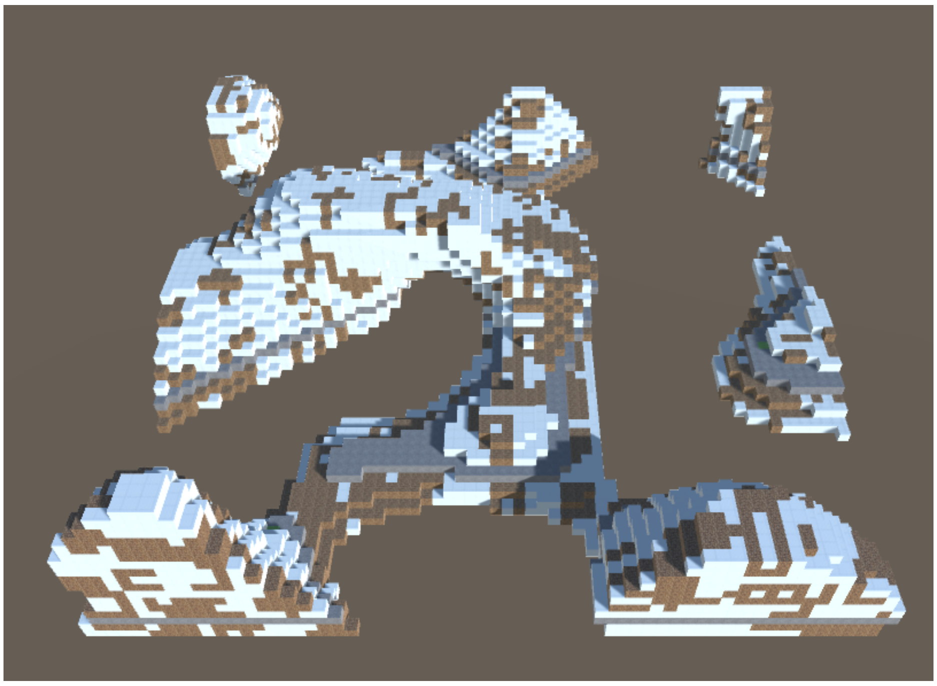


Figure 4.28: Terrain 14 and its results.



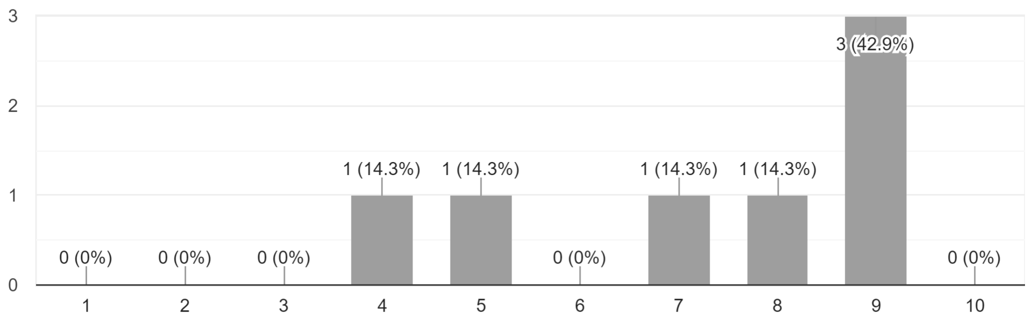


Figure 4.29: Terrain 15 and its results.

Chapter 5

Conclusions

In this chapter we briefly go over the results obtained from this work as well as the goals that were proposed. We also discuss ideas that could be implemented in the future to further improve the work.

5.1 Conclusions

In the first stage of this work a procedural terrain generator was developed, with the chosen terrain representation being that of a voxel grid. A small number of different voxel block types was chosen in order to be able to represent some variety of terrain features, and the procedural generation algorithm was ultimately implemented using Ken Perlin's improved noise function [24] due to both its better looking results and faster performance. At this time the block types of each voxel in the terrain were randomly set. Afterwards, the User Interface was created in order to allow a better visualization of multiple terrain instances. The algorithm was also changed to allow the use of set seeds, so that the multiple terrain instances are all generated with equal terrain structures, allowing for better comparison between results. It was also during this stage that decision trees determining the block types in the terrain were implemented, with two different node functions, making decisions according to either the height of a certain voxel or the number of its adjacent neighbours of a certain type, and leaf nodes defining the block type to be used. Finally the previously developed User Interface was expanded upon, allowing users to select one or two of their preferred terrains to evolve. Genetic evolution between the chosen decision trees was then implemented, applying crossover in the case of two terrains being selected, and always having a chance to apply the three chosen mutation operators. An interface for viewing the decision trees was also added, in hopes that it could aid users in having a better grasp of how each terrain instance was generated, however the lack of decision tree pruning resulted in the feature not being as helpful as initially thought since readability diminished quickly with decision trees evolving to sizable amounts.

After the tool was finished, potential users were requested to help test it due to the high subjectivity possible when evolving and evaluating a terrain. Users were asked to try and achieve four goal terrains using the program, without specifying an end result so as to allow them to evolve the terrains in the most unbiased approach possible. After the four tasks users filled in a survey regarding their experiences. The results obtained from these testing sessions allowed us to reach a better understanding of both the work's shortcomings and strong points. For starters, when attempting to evolve a terrain with user decisions, it is important for the user in question to have a clear image of their goal throughout the evolution generations, so as to allow the decision trees to converge properly. If the user keeps making choices inconsistent to their goal it may end up severely slowing or even ultimately making it impossible for the algorithm to reach the initial terrain goal. We were also able to gather that some users felt both limited by the tool and that it would probably be easier to make a terrain from scratch instead. The users did feel, however, that their choices had an impact on the changes being made in the terrain, and that it didn't take too long until progress was made apparent. These results lead us to believe that, while there is room for improvement, using evolutionary algorithms to evolve a decision tree can be a viable method to procedurally generate virtual worlds.

5.2 Goal Completions

As previously mentioned, the main goal of this work was to explore the possibility of using evolutionary algorithms to evolve a decision tree as a means to procedurally generate virtual worlds without the need of a technical expert. While we feel that this goal was completed, some originally planned features that could perhaps impact the results were not.

Given the engine limitations, generating terrains with bigger dimensions did not prove feasible, which could've impacted a user's opinion on the existing terrain features. Subdividing the generated terrains with a chunk approach could've solved this problem by, for example, allowing the users to control a moving camera that would travel across the terrain and its features. While for the terrain generation algorithms and parameters being used the generated portion of the terrains were thought to give enough of an overview of their features, this could be an important addition to take into consideration when using other terrain generation methods.

The initial plan of allowing the visualization of multiple procedurally generated terrains simultaneously was also not possible to due the engine's limitations, as the interface proved to become unresponsive while rendering such a large amount of individual objects on screen at the same time. This did not prove to be a major hindrance to the work's results, as users were still able to view each terrain individually, it most definitely slowed down their process of evaluating the terrains, which could've lead to increased frustration and/or impatience while using the tool.

5.3 Contributions

This work's main contributions are the following:

1. Studying the viability of evolutionary algorithms being applied to procedurally generated content based on user feedback. In particular, providing the user with increased control over the generated content whilst requiring minimal user input by shifting the burden onto the algorithm;

2. Expanding the analysis of using procedural generation as a means to reduce virtual content development costs. In the case of video-games this can also be used to increase replayability, providing more content to its players and possibly increasing the video-game's value;

3. Development of a terrain generation tool which does not require any area expertise in order to achieve desirable results. This would also allow its use as an introductory procedural generation tool for a broader audience.

5.4 Future Work

In the future, there are some aspects in this work that could be of interest to improve. For starters, the tree decision viewer as it stands currently does not possess much use due to it quickly becoming unreadable with larger trees. In order to soften this problem different decision tree pruning techniques could be properly implemented, cleaning much of the redundancy currently present in bigger decision trees and ultimately increasing performance. Doing this would also allow the implementation of more complex features in the viewer, such as the initially mentioned goal of allowing users to interact with the decision tree structure and performing manual adjustments as they see fit (a feature often requested by users during their hands-on experience with the tool).

Due to the subjective nature of the evolutionary algorithm chosen, different mutation rates and crossover algorithms were difficult to compare in terms of objective results. This is an area that could be expanded upon, perhaps by utilizing an algorithmic fitness function to evaluate the different terrains automatically, which would allow us to better compare results between different rates and crossover algorithms. The configurations found to provide the best results could then be used with the original user-dependant evolutionary algorithm.

Bibliography

[1] 10 Free GIS Data Sources: Best Global Raster and Vector Datasets (2019) - GIS Geography. `https://gisgeography.com/best-free-gis-data-sources-raster-vector/`, visited on 2019-12-22.

[2] 3ds Max — 3D Modeling, Animation & Rendering Software — Autodesk. `https://www.autodesk.com/products/3ds-max/overview`, visited on 2019-12-22.

[3] Algorithm for Drawing Trees — Rachel Lim's Blog. `https://rachel53461.wordpress.com/2014/04/20/algorithm-for-drawing-trees/`, visited on 2020-07-24.

[4] Bay 12 Games: Dwarf Fortress. `http://www.bay12games.com/dwarves/`, visited on 2019-12-31.

[5] blender.org - Home of the Blender project - Free and Open 3D Creation Software. `https://www.blender.org/`, visited on 2019-12-22.

[6] Chunk – Official Minecraft Wiki. `https://minecraft.gamepedia.com/Chunk`, visited on 2019-07-24.

[7] Elite (video game) - Wikipedia. `https://en.wikipedia.org/wiki/Elite_(video_game)`, visited on 2019-12-22.

[8] Exile (1988 video game) - Wikipedia. `https://en.wikipedia.org/wiki/Exile_(1988_video_game)`, visited on 2019-12-22.

[9] GitHub - keijiro/PerlinNoise: 1D/2D/3D Perlin noise function for Unity. `https://github.com/keijiro/PerlinNoise`, visited on 2020-07-24.

[10] Massive Software – Simulating Life. `http://www.massivesoftware.com/index.html`, visited on 2019-12-31.

[11] Maya — Computer Animation & Modeling Software — Autodesk. `https://www.autodesk.com/products/maya/overview`, visited on 2019-12-22.

[12] Minecraft. `https://www.minecraft.net/en-us/`, visited on 2019-12-31.

[13] No Man's Sky. `https://www.nomanssky.com/`, visited on 2019-12-22.

[14] Peak Video Game? Top Analyst Sees Industry Slumping in 2019 - Bloomberg. `https://www.bloomberg.com/news/articles/2019-01-23/peak-video-game-top-analyst-sees-industry-slumping-in-2019`, visited on 2019-12-31.

[15] Rogue (video game) - Wikipedia. `https://en.wikipedia.org/wiki/Rogue_(video_game)`, visited on 2019-12-31.

[16] The Lord of the Rings Trilogy - IMDb. `https://www.imdb.com/list/ls072068350/`, visited on 2019-12-31.

[17] Unity Real-Time Development Platform — 3D, 2D VR & AR Visualizations. `https://unity.com/`, visited on 2019-12-31.

[18] Unreal Tournament. `https://www.epicgames.com/unrealtournament/`, visited on 2019-12-22.

[19] Terrainosaurus: realistic terrain synthesis using genetic algorithms. (December):119, 2007.

[20] Michael Affenzeller, Stephan Winkler, Stefan Wagner, and Andreas Beham. Genetic algorithms and genetic programming: Modern concepts and practical applications. *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications*, pages 1–365, 2009.

[21] Miguel Frade, F. Fernandez De Vega, and Carlos Cotta. Breeding terrains with genetic terrain programming: The evolution of terrain generators. *International Journal of Computer Games Technology*, 2009.

[22] Ian Millington. *AI for Games THIRD EDITION*. 2014.

[23] Motion Picture Association of America. Theme report. *Motion Picture Association of America*, pages 1–55, 2018.

[24] Ken Perlin. Improving Noise. *ACM Transactions on Graphics*, pages 681–682, 2002.

[25] Steve Rabin. *Game Ai Pro2: Collected wisdom of game AI professionals*. 2015.

[26] Ruben Smelik, Tim Tutenel, Klaas Jan De Kraker, and Rafael Bidarra. Integrating procedural generation and manual editing of virtual worlds. *Workshop on Procedural Content Generation in Games, PC Games 2010, Co-located with the 5th International Conference on the Foundations of Digital Games*, 2010.