gameBITS Framework: A procedural content generation framework for digital games

Marcus Vinícius da Silva* Fernando Marson Vinícius Cassol

Universidade do Vale do Rio dos Sinos - UNISINOS, Jogos Digitais, Brazil

Figure 1: Applications built with the *gameBITS Framework*.

ABSTRACT

The use of procedural generation techniques has emerged as a growing trend in the field of digital games due to the numerous applicability in content generation, as this has been shown as the main bottleneck in high-quality commercial titles production process, affecting mainly small developers with limited budget. Faced with this problem, this paper presents a general purpose procedural content generation framework for digital games, called *gameBITS Framework*, relying on a modular and hierarchical architecture for use in different environments and purposes. The objective of this work is to provide a simple, robust and extensible solution for small developers of digital games in low and medium complexity applications, and through this ensure cost reduction and production time. For this purpose, a review of studies in the field of procedural generation is presented to clarify the distinction between the various types of content and techniques applicable in this context as well as other existing architecture proposals that corroborate for the proposition process of the new architecture. The effectiveness of the approach adopted for the architecture is demonstrated by three applications built based on the proposed framework, highlighting the performance in the context of a game, and the code reusability potential.

Keywords: procedural modeling, content generation, digital games.

1 INTRODUCTION

The techniques of procedural generation has been presented as a solution to many problems in the field of digital games, due to the numerous applicability in generating content. One of the main reasons perceived for its application, in general, is due to the fact that the game industry has faced in recent years a major challenge generated by the increase in costs for the production of content for their products [13], which depends essentially on the manual creation of the game elements, thus limiting the scope of the projects and direct impact on the user experience quality. This scenario is observed mainly in small companies or independent developers, since they do not have vast financial resources to afford large teams or more development time. As stated by Hendrikx et al. [5], the content production has become a bottleneck in the development process of high quality commercial titles.

Faced with this problem, this work presents itself as a contribution to the solutions of dynamic content generation, proposing a general purpose procedural content generation framework for digital games applications, called *gameBITS Framework*, based on a modular and hierarchical architecture. The solution introduces a level of generalization that allows its use in a wide range of content generation applications, providing reusability of features between different applications that share similar demands, and exposing a set of simple, robust and extensible resources for use by small developers in low and medium complexity applications, and aimed at reducing costs and production time.

This paper introduces the studies in the field of procedural generation that helped in the elucidation of the differentiation between distinct types of content and techniques applicable in this context, as well as other existing proposals that corroborated for the *gameBITS Framework* architecture proposition, followed by the presentation of some applications built on top of the framework, aiming to demonstrate the efficacy of the approach adopted in terms of performance, versatility and reusability of features. The paper is organized as follows. Section 2 presents the theoretical basis required to support the discussion around procedural content generation in digital games. Section 3 presents existing solutions in the field of procedural generation, pondering on these in relation to the proposed solution. Section 4 presents the framework architecture proposed by this work. Section 5 exposes the applications developed on top of the framework. Section 6 discusses the results obtained. Section 7 concludes the paper.

2 THEORETICAL BASIS

As well synthesized by Parberry [18], the process of procedural generation has three main features, the first being the rapid, computationally feasible and efficient generation of outputs, the second one is characterized by the ability to dynamically generate content while maintaining certain quality standards, and lastly to provide a comprehensive and friendly configuration interface. Based on these statements, there has been the need to understand the basics of procedural generation, as well as the techniques applied in the frame-

*e-mail: mav.jed@gmail.com

work structuring process.

2.1 Procedural content generation in digital games

Procedural content generation applied to the context of digital games can be described as a set of techniques that aims to generate different types of game elements through the use of specifically shaped algorithms to control various parameters, which jointly determine the dynamics of the composition of these elements. The main purpose of the application of content generation by procedural methods is to eliminate or at least reduce the need to produce all artifacts manually, in order to reduce the development time of this or even the reduction of labor costs at the same time as it provides an abundance of elements with pseudo-infinite variations.

The artifacts that comprise what is called content refers to all elements capable of affecting the gameplay of a game, with the exception of the NPC's behavior, non-player character and the game engine itself [23]. Based on this assumption, it can be characterized as content elements such as terrain, maps, levels, story, dialogue, quests, characters, set rules, music, items, among others.

Hendrikx et al. [5] proposes a hierarchical structure which comprises the various possible types of content that can be generated procedurally, which exposes several layers, being the lower level composed by what he calls *Game Bits*, that consists of basic features such as sounds and textures that may or may not be used by upper layers for composing elements that result in final form, while the upper layers adopts higher-level concepts, and are characterized according to the type definition of the elements that composes it. Figure 2 presents this proposed taxonomy.

One of the first games to use procedural content generation techniques is the game *Rogue*[1], where the player controls an adventurer through dungeons generated dynamically by an algorithm of procedural generation, so every time a new game starts the adventure is a new experience different from the previous one. More recently, the game *Minecraft*[2] made use of procedural generation to build entire worlds made of 3D blocks, with the presence of various biomes, each with varieties of flora and fauna [12]. The game *No Man's Sky*[3] goes even further, creating an entire galaxy from a macro scale, such as stars and planets, to a micro scale, creating plants and animals.

Despite the multitude of possibilities derived from procedural generation, it can also end up generating inconsistent and chaotic results, which stands as one of the challenges to the implementation of a dynamic content generation system. A given artifact may have a huge set of variables that define what constitutes a multidimensional problem of high complexity, so an adequate knowledge regarding the techniques used in this solution is needed.

2.1.1 Pseudo-random number generation

Pseudo-random number generators are specific, sequential and deterministic algorithms that use mathematical foundation to generate sequences of numbers from an initial value called seed, therefore, given a seed, the output is always the same.

These have many applications, from statistical experiments, numerical analysis with methods of *Monte Carlo*, probabilistic algorithms, encryption and even applications related to entertainment, being also ideal for digital game applications, but different applications require adequate levels of quality according to the context.

It is considered a good sequence number generator algorithms that are able to guarantee a high level of randomness, which operate uniformly distributed chains of values, that are portable among different platforms, with reproducible and homogeneous output in relation to the bits randomness, and that have periods long enough that are not repeated within the same execution cycle [17].

There are several types of approaches used to generate pseudo-random numbers, and the most commonly used, developed and tested techniques are the *Linear Congruent Generators*, the *Lagged Fibonacci Generators*, the *Shift Registers Generators* and *Hybrid Generators* [21]. Among the currently existing solutions, the *Mersenne Twister* algorithm shows itself capable to provide adequate functionality to the context of this work, specifically for dynamic values attribution purposes for the generation of certain artifacts control parameters.

2.1.1.1 *Mersenne Twister*

The *Mersenne Twister* algorithm is a pseudo-random number generator proposed by Matsumoto et al. [11], it is a variant of the *Shift Registers Generators*. The algorithm is able to generate sequences of numbers with a period of $\rho = 2^{19937} - 1$, and its operation can be described by the linear recurrence (1):

$$x_{k+n} = x_{k+m} \oplus (x_n^u | x_{k+1}^l)A, (k = 0, 1, ...) \quad (1)$$

Where: n is the degree of the recurrence, m is the middle word, $1 \leq m \leq n$, u and l additional tempering bit shifts, and a matrix $A_{(m \times m)}$. The initial seeds are x_0, x_1, ..., $x_{(n-1)}$, then it generates x_n by the linear recurrence presented with $k = 0$. The other values $x_{(n+1)}$, $x_{(n+2)}$, ... are determined by making $k = 1, 2, ...$

Nandapalan et al. [15] points out some studies that criticize the algorithm in a more rigorous statistical perspective, however for certain purposes, such as this work, these cons may be disregarded by not significantly compromise the end result.

2.1.2 Coherent noise generation

For certain applications, randomly generated numbers with a high degree of randomness, as in the case of generating pseudo-random numbers, do not present an ideal solution because they create sequence values without any correlation with each other, making their use infeasible due to lack of control over what is generated as output. Faced with these issues, coherent noise generation techniques emerge as a solution to provide methods of generating sequences of pseudo-random numbers that allow some level of control over the frequency distribution of these numbers given distance, area or dimensional space of higher order [7]. Objectively, coherent noise generation must meet three basic criteria:

- Given an input value, the generated output is always the same.
- A small variation in the input generates a small variation in output.
- A large variation in the input generates a random variation in output.

Based on this assumption, you can generate a value, given a point in a space n-dimensional, consistent with its neighbors, in the same space, that is, a coherent noise generator is capable of producing sequences of numbers with a smooth transition. This feature allows these techniques to be particularly useful in generation of the procedural textures and height maps for the creation of terrain. Lagae et al. [9] describes the following benefits of procedural noise techniques:

- A procedural noise function is extremely compact, it does not need to store texture data since generates at run time.
- The noise can be generated for any resolution, that is, it is possible to approach a point indefinitely without losing definition.

[1] *Rogue*, 1980, developed by Michael Toy e Glenn Wichman

[2] *Minecraft*, 2011, developed by Mojang, 2011: `https://minecraft.net`

[3] *No Man's Sky*, 2016, developed by Hello Games: `http://www.no-mans-sky.com`

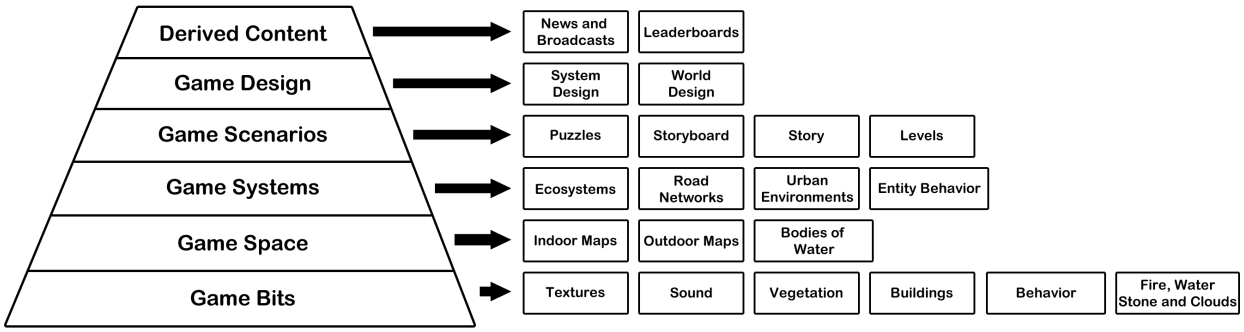

Figure 2: Hierarchical structure of the taxonomy of procedurally generated content. Adapted from Hendrikx et al. [5].

- It does not present periodicity for any space n-dimensional.
- Procedural noise function is parametrized, which allows control over the pattern noise by adjusting these variables.
- Allow dynamic access to different portions of the space n-dimensional, which allows the use of parallelism by CPU or GPU.

One of the most known techniques has been introduced by Perlin [19] known as *Perlin Noise* algorithm, which is capable of generating procedural noise for a wide range of applications in computer graphics. There are also other techniques that use the outputs of coherent noise generators to increase the variety of final results, such as based of *Fractals* and *Multifractals* functions [2], using a combination of noises generated in different scales to create a final result of greater complexity.

2.1.2.1 *Perlin Noise*

The *Perlin Noise* algorithm was developed using the concept of procedural noise generation, aiming to simulate a degree of realism in textures in order to render clouds, fire, water, marble, wood and stone. Miranda et al. [14] summarizes the algorithm stating that noise at a point in space is given by the result of inner product determined by (2):

$$G \cdot (P - Q) \tag{2}$$

Where: P is the position of the point where the noise is being calculated, Q the position of one of the neighbor points, and G the value of a pseudo-random gradient vector. Thus, the inner product is calculated for all neighbor points, followed by a spline interpolation [9], ensuring a smooth transition between adjacent values.

Since its creation the *Perlin Noise* algorithm had been improved to contemplate more efficient methods of generating procedural noise, but with different results, known as *Simplex Noise*.

2.1.2.2 *Worley Noise*

Worley Noise [22] is a type of noise generator used in procedural texture generation, it is based on the concept of *Cellular Texturing* and is capable of producing similar results to various types of surfaces such as sponge, scales, pebble and tiles [2], being also useful in generating height maps due to their cellular characteristics that can be used in the creation of mountain ranges, and may be further combined with other noise generation techniques, such as *Perlin Noise*, in order to enhance the naturalness and realism of the final result.

The purpose of the *Worley Noise* algorithm is based on the *Voronoi Diagram*, which in a simple definition can be described as the division of space into regions or cells, so that all the points contained in each region are closest to a certain predefined point, called *feature point*, a set of other points distributed in a space of dimension R^2 or R^3, that is:

- A two-dimensional or three-dimensional space is defined.
- A certain amount of *feature points* are distributed in this space.
- For each *feature point*, all points closer to itself in relation to other *feature points* are found.
- The noise $f_n(x)$ is the distance to the n-th point closer to x.

Where: n is a *feature point* belonging to the first set of points generated, and x is an arbitrary point in space, different from any *feature point*.

2.1.2.3 *Fractals* and *Multifractals*

Fractals are defined by Ebert [2] as geometrically complex objects, where the complexity emerges through repetition of a certain geometry across a range of scales. Comparatively, the complexity of the generation methods of non-fractal objects arises from the combination of artifacts generated over time through unrelated events, that is, the actual process of generating non fractal object is complex in itself, while the fractal complexity starts from a simple assumption, defined as the combination of products resulting from the same process carried out in different scales.

Algorithms based on this concept are especially useful for generating textures and height maps, because they add complexity to the final result, making it more realistic and believable to the user, simulating synthetically a similar behavior as observed in nature, such as mountains, clouds, water and planetary surfaces. *Fractal Brownian Motion*, or simply *fBm*, is an example of a very useful fractal based technique used on height maps generation.

However, methods of fractal generation as *fBm*, have certain limitations in situations where the environment size is relatively large, since fractal algorithms are said statistically homogeneous, as the results show repetitive patterns over the entire surface, and isotropic, because the resulting surface has repeated physical characteristics in all directions. In other words, realism is limited to more restricted scenarios, and its application in vast scenarios, such as a planetary surface, for example, can generate monotonous and unnatural results, allowing to characterize such techniques as *Monofractais*.

A relatively simple solution is to apply *Multifractals* techniques, which heuristically can be defined as heterogeneous fractals, whose heterogeneity is invariant according to the scale, that is, it captures the heterogeneity of large-scale terrains [14] allowing the variability of the results generated by presenting, in a context of terrain generation, both mountain ranges as great plains in different places. One of the notable techniques used in generation of terrains and employing this concept is the *Ridged Multifractal*.

2.2 *LOD* systems

LOD or *Level of Detail* systems are techniques typically used in the optimization process of interactive graphics applications, where the objective is to obtain performance gain at the expense of the level of complexity presented by the elements that compose a scene and therefore reducing the computational load for execution. As emphasized by Luebke et al. [10], despite constant advances in graphics processing hardware, the need to use these optimization techniques still exists, because as computational resources evolve, the applications themselves also grows in complexity and establish new requirements and limits to be met.

In the context of digital games, *LOD* techniques are commonly used to simplify 3D objects meshes, reducing its complexity as its distance increases relative to the observer within the virtual environment, textures, applying equivalent versions of lower resolution, or even controlling aspects of shading and lighting, alternating simple and advanced techniques to eliminate the cost of processing and memory.

Basically there are three *LOD* management methods: *Discrete LOD*, *Continuous LOD* and *View-dependent LOD*. *Discrete LOD* is characterized by an offline approach to the generation of the elements, where the artifacts representing an object are defined in advance, so at run time, as they are presented on the screen, the *LOD* system, based on predefined rules, determines the appropriate representation of the object to be displayed, saving, in the case of 3D meshes, for example, a considerable amount of polygons, and allowing more objects to be displayed without the loss performance.

Continuous LOD works similarly to *Discrete LOD*, however, the simplification system creates a data structure that encodes a continuous range of detail, so that the quality of the object is defined at runtime without the need to pre-process the object to be displayed on the screen. *View-dependent LOD* is a variation of the *Continuous LOD* technique, where the level of detail observed is dependent on the current point of view, so that even a single object presents a range of different levels of complexity through its mesh.

3 Related Work

Although the techniques of procedural content generation is being used for decades, there is no single reference able to determine an enough generic solution to meet all the needs, but several proposals have been made, and new techniques come up from time to time. The challenges involved in this work address both basic resources generation issues and high level organizational problems, so it is essential to be aware of previous work on procedural content generation so that a critical view on this field of knowledge can established.

3.1 State-of-the-art

In Hendrikx et al. [5] a search is conducted for the purpose of organizing the data known about procedural content generation focused on digital games, motivated by the fact that there is not a solid and comprehensive bibliographic references in this area, noting that the works are spread across different areas of knowledge, which makes the process of research on the subject very complex, since there is not a consolidated and final generalization of the approach of the techniques in the context of games. Faced with this situation, it was proposed a taxonomy of the types of content that can be generated using procedural techniques, where the content types were arranged in a layered structure, allowing an overview of content groups according to the level of complexity. Then presented a taxonomy of existing techniques for procedural content generation for digital games, in order to clarify the applicability of each technique with the respective layers of the taxonomy of content defined in advance. After the taxonomic definition of content and techniques, an investigation regarding the level of maturity of the techniques were performed, where it was observed that the higher the content complexity, the lower the evolution level of the technique used in that context is. Finally, the research highlights five key points to be best studied in future work, particularly highlighting the need for research on techniques that allow the creation of most complex and abstract content as shown on the proposed taxonomic structure.

Keane [8] leads a study to propose a framework capable of procedurally generating and displaying in real time the surface of planets, allowing developers to focus on gameplay rather than invest time in manually creating these elements. Several assumptions are adopted, such as performance, aesthetics and versatility of the tool, using as a development platform the *Unity*[4] game engine. The research involves the use of techniques such as *Perlin Noise* and *Ridged Multifractal* for texture composition, as well as optimization techniques for presentation of content, such as *LOD* and *Multithreading* to control generation of some elements more efficiently. In general the work shows promising results, being able to generate planetary surfaces, but does not address certain framework customization requirements, which can be problematic in certain cases.

Greuter et al. [4] introduces a framework for generating pseudo-infinite virtual worlds, towards a more generalist approach. The structure of the system is based on three main components, namely: filling the field of view, limiting the generation of content to the limit of the *frustum*; caching geometry, which provides a certain amount of memory space for the storage of elements already generated, thus avoiding the need to regenerate them; and geometric generation itself, which combines a number of techniques to generate the elements that compose the scene. The framework is applied to the context of a procedurally generated city as a way of validation, generating also procedurally the buildings that compose the scene, which is highlighted as a counterpart to the best known commercial solutions, which generally use pre-designed elements. The framework proved to be able to manage the resources and generate the virtual environment, however certain technical aspects were not considered, such as the use of a physics system for collision management, but the architecture design proved to be valid.

Yannakakis et al. [23] presents the *EDPCG* framework, *Experience Driven Procedural Content Generation*, with a general and effective approach to optimized content generation based on user experience. The justification is based on the fact that despite the dynamic generation of content show up as an obvious solution, it is most effective when the configuration parameters are determined according to the specific characteristics of the target audience of the product. The solution relies basically on the capture of user experience quality, using it as a form of metrics to decide what and when to generate more or less a given element, which in this case is treated as a search space to be computed either by exhausting means or with the adoption of heuristics to rely on existing dimensions to solve the problem. The study demonstrates practical scenarios with the proposed approach, but points out the difficulty of the application of the technique because there is a high degree of subjectivity to be captured for the result to be effective.

In Merrick et al. [13] a structure of a procedural generation system of wide range aimed for generation of content for digital games comprising an evaluation process of the result based on *Wundt Curve* is proposed, which defines, in a simple way, the level of interests of the subject in relation to the degree of innovation of the generated artifact, so potentially less interesting results to the end user can be eliminated in advance. To apply this principle, it was decided that a design task for a given device can be understood as a set of variables, characterizing a multi-dimensional search space. Thus, given a predefined design by a person, its decomposition is carried by the tool, and applying the concept adopted, similar results are generated based on this initial model, that is, the problem

[4] *Unity* is a cross-platform game engine developed by Unity Technologies: `https://www.unity3d.com`

is deconstructed in a computable search space to generate different artifacts.

Rudzicz [20] proposes to aggregate in a single framework, called *Arda*, a variety of procedural content generation techniques of various levels of detail through a system of interchangeable generation modules, potentially solving problems of generating content in different contexts. However, for functionality verification purposes in a viable time, restrictions were adopted about the content type, in this case terrain, cities and buildings. Subsequently, implemented over the framework, it presents a tool that provides a control interface for the generation of specific content adopted as a basis for the solution. The proposal was successful, being able to generate content in a satisfactory manner within the restrictions, but it highlights the noticeable lack of realism and complexity of the scene, and the absence of some more advanced technical optimization, able to parallelize certain parts of the procedural generation system to ensure optimum performance.

Carpenter [1] exposes two recurring problems in the process of procedurally generating vast worlds for digital games, the first is the need to produce content beyond the storage capacity of memory in a way capable of handling such resources to be created only when they are accessed, that is, without previously being generated, and the second addresses the problems involved in the generation of artifacts existing in larger structures without the need to generate the entire structure in which it is contained. Faced with this problem, a framework prototype that is able to solve these problems is proposed, based on a spatial subdivision logic that allows the management of low levels of granularity within larger and more complex systems, without the first is directly attached to the generation of the second. The structure adopted uses a hierarchical model of nodes and defines four types of interfaces that make up the entire system architecture. In order to validate the framework, the proposed architecture is applied to a specific context for the generation of a procedural galactic structure, however without proper generating each planetary system in detail for each star in this structure. In terms of performance and use of hardware resources the project had been successful, only in relation to the application generality an observation is raised, it is said to be complex to measure the level of flexibility without certifying the applicability in practice, but is positively accepted as theoretical assumption.

3.2 Contextualization in the state-of-the-art

Among the studied work it is perceived a certain consensus on the importance and applications of using procedural content generation. In a summarized manner, technical limitations are pointed, in the sense that the computational resources of today are still very limited to ensure the representation of very large virtual environments despite all the technological advances of recent decades, and limitations of financial resources, that ensure the structural maintenance of the companies for the production of content, which agrees with the scenario clarified by this work.

The survey conducted by Hendrikx et al. [5] is of utmost importance for the whole area of procedural content generation, serving to provide an overview of the maturity of the game content generation systems, and is very relevant to the scope of this paper, offering good guidance on the use of procedural techniques in each specific area and allow to situate the contributions of this work within the presented taxonomic structure of content.

The framework proposed by Keane [8], as the author points out, suffers from some flaws in regards to the ability to customize some elements, which may lead to greater restrictions for different applications, such as this proposal. An interesting question raised by the author points out certain limitations of the platform used, in this case the *Unity* game engine, related to the representation of floating point numbers, which causes the adoption of appropriate strategies for the representation of vast environments.

Greuter et al. [4], Rudzicz [20] and Carpenter [1] have similar contributions, since they use hierarchical organizational methods on their framework models, in order to meet in a generalist way the solution of procedural generation resources management for the production of content for different genres of games. Despite all the solutions prove promising and appropriate according to most of the assumptions made by each author in its scope of work, there is a certain lack of practical evidence of each proposal in the real context of a game, as in a game there are several other issues that should have a negative impact on performance, which added to the resources consumed by each solution may come to affect the players experience. However, the similarity between the architectural models observed through the studied work demonstrated that a hierarchical structure of nodes appears to be quite efficient and easier to maintain, which contributed to some important choices in this work. Another important consideration relates to the adoption of optimization techniques which indicates the necessity of the use of parallelism depending on the intended application.

Yannakakis et al. [23] and Merrick et al. [13] address issues related to the evaluation process of the generated content, generating data that turns into improvements for the results themselves subsequently. The biggest problem perceived in these proposals is the difficulty of capturing the subjectivity involved in this process, which is confirmed by both, but it is also true that the quality of results tends to be higher, if the data is worked consistently in each domain. Highlighted these difficulties, a more viable alternative to the adoption of advanced content evaluation techniques is a more simplistic approach, however valid, that is to validate the artifacts directly with the target audience.

Among the solutions studied, despite having positive and useful results for use in content generation, all serve a more experimental purpose just aiming to prove their hypotheses. However, there is a certain lack of concern for conditioning their solutions to a more favorable environment for a practical implementation of an interactive application based on its framework, which implies a lack of a more emphatic validation and restricts the result itself to a more theoretical field. Keane [8] comes closer to this ideal, since it takes a solution approach within the *Unity* game engine, which allows easier access by other developers, but having specialized his architecture for a particular purpose, it does not prove to be as useful to different contexts. Facing this scenario, this work presents itself as a counterpart, as presupposes the use of a user-friendly development environment for the implementation of games from the beginning, and consider a more generic framework architecture.

4 Framework Architecture

Through the understanding of the procedural generation techniques according to their classification and applicability, and having studied some formulation methods of frameworks for content generation already proposed, the architecture of the *gameBITS Framework*[5] was structured, as shown in Figure 3. The framework takes the following assumptions:

- The existence of a hierarchical organizational node system to manage the generated artifacts.
- The ability to generate content synchronously or asynchronously.
- The existence of a *LOD* system based on a set of simple rules..
- The distinction between concrete and abstract artifacts.

[5] A *C#* implementation of the *gameBITS Framework* for *Unity* can be found at `https://github.com/Carnicero/gameBITS-Framework`

- The characterization of artifacts through a components system.

- The complete separation between characterization and generation of artifacts.

The structure makes use of some design patterns of general use, as well as some more specific patterns for digital games. The procedural generation resource management core is defined by a *Singleton* pattern, as shown in Johnson et al. [6], ensuring that only one instance of the class resides in memory during the same application execution cycle and providing a global point of facilitated access to the managed resources. Other patterns are used to define the characterization and generation interfaces of artifacts, such as the *Builder* and *Abstract Factory* patterns also exposed by Johnson et al. [6], which ensure a high degree of generalization at the same time that provides decoupling between the artifact definition and creation.

In the concrete artifact and *LOD* system classes, stands out the use of the *Update Method* pattern, presented within the *Sequencing Patterns* demonstrated by Nystrom [16], which allows a centralized update control of the resources through the management class. The architecture also makes use of an adaptation of the *Game Loop* pattern, also clarified by Nystrom [16], which offers a way of coupling between the internal update process and the game engine update process, providing easy integration between them.

The *gameBits Framework* architecture has fifteen different classes that compose the framework. The *gBManager* class is responsible for centralizing the management of generation requests, it also holds the root artifact of the artifact hierarchy, grants registration and access to tools, as well as providing integration with the update cycle of the game engine and transmitting to each artifact the internal update order of each and its *LOD* systems. The *gBObject* class characterizes the interface that defines an object within the context generation system. *gBTool* defines an interface for classes that provide specific services, such as access to the game engine features, for example. *gBSensor* is a specialized class tool that aims to define a capture interface of any state of the virtual environment or performance data, providing access to this information mainly by the *LOD* system. *gBGenerator* is a specialized tool class that specifies an interface for accessing the generation of a particular resource, targeting itself to isolate the complexity of the same generation process. *gBRequest* is the class responsible for mediating requests between specific devices and generators synchronously or asynchronously. The class *gBResource* defines the basis of a resource generated procedurally, and is the one who contains the seed that determines the outcome generated. *gBParameter* is the class that features a parameter of an artifact, that is, it is what defines the characteristics of a particular artifact. *gBArtifact* describes a generic artifact, and is characterized by a collection of parameters. *gBAbstract* is the class definition of abstract artifacts, in general it characterizes resources such as meshes and textures, for example. The *gBHelper* class defines auxiliary structures for storage and manipulation of data, in general used to handle data not manipulated by the game engine. The *gBConcrete* class characterizes what is defined as concrete artifact, it consists of abstract artifacts, characterization parameters, may have a *LOD* control system, and is the class that characterizes the concept of node, it may be linked to a parent node and several child nodes. The *gBLODSystem* class defines an interface for *LOD* systems. The *gBLODRuleSet* class features a *LOD* system based on a *Discrete LOD* system, which manages the rules that define the presentation behavior of an artifact. The *gBLODRule* class describes a level of detail modification rule of an artifact, being able to dynamically change characteristics of this to suit a particular situation detected by sensors.

5 APPLICATION

For efficacy validation purposes, having adopted as principles to be provided by the architecture performance, flexibility, extensibility and reusability, three scenarios in which the framework was used as a basis for the development of the final solution were studied. The first scenario uses the framework for building a procedural planetary systems generator, called *StellarGen*, in order to evaluate the performance and the extensibility potential of the basic interface provided by the framework. The second scenario uses the framework for the implementation of a procedural generation system of pseudo-endless tracks for *Endless Runner* games, called *TrackGen*, aimed at assessing the reuse potential of modules developed in the first application at the same time as evaluates the framework flexibility for totally different purposes. The third scenario aims to validate the use of the proposed solution in the context of a game, called *Stellar Empire*, in order to study the overall behavior of the system stability in a real environment of competition for hardware resources.

In the three case studies the *Unity* game engine was used as a development environment. The framework and applications were implemented in *C#* language. For carrying out the performance tests, a computer equipped with an Intel Core i5-750 2.66GHz processor, 2x4GB RAM memory at 1333MHz in dual-channel mode and a Nvidia GPU GTX560Ti with 1024MB of RAM were used. All tests were performed three times, so that the results presented are an average of the results obtained in each test.

5.1 *StellarGen*: A procedural planetary systems generator

StellarGen[6] is an *asset* developed for *Unity* which has the functionality of generating planetary systems containing a number of elements present in real systems, such as stars, rocky planets, gas planets, dwarf planets, asteroids, meteoroids and satellites, as well as generating other elements that help to build these artifacts like barycenters, orbits, belts, atmosphere and oceans. The application is able to generate textures and high quality and high complexity meshes, especially using algorithms of coherent noise generation like *Perlin Noise*, *Worley Noise* and *Ridged Multifractal* controlled with a parameter configuration interface, whose values are also generated dynamically within a pre-determined range by the use of pseudo-random number generators based on the *Mersenne Twister* algorithm. Figure 4 shows an image generated by the *asset*. Although the generation process is complex, the generation of each system originates from a single initial seed, which is an unsigned integer value of 64-bit, providing a wide range of possible unique results.

The *asset* was created in order to test the relationship between all parts of the framework, also assuming as a principle the pursuit of full performance during execution. Initially all elements are generated in a minimum quality to be displayed, and during the execution more complex versions of each artifact is generated asynchronously by using the combination of the requisitions system with the *LOD* system, which allows the obtaining of a visual experience of high quality without the need to generate each element previously, which in extreme cases render impossible the application execution on systems with limited hardware, and without major impact on the flow of execution, since it does not stop user interaction with the virtual environment. Every concrete artifact of higher complexity consists of abstract artifacts, which in general can be described as textures, meshes and sets of coherent noise modules, each with its specific characteristics depending on the type of artifact to be generated.

[6] *StellarGen* can be found at `https://www.assetstore.unity3d.com/en/#!/content/65062`

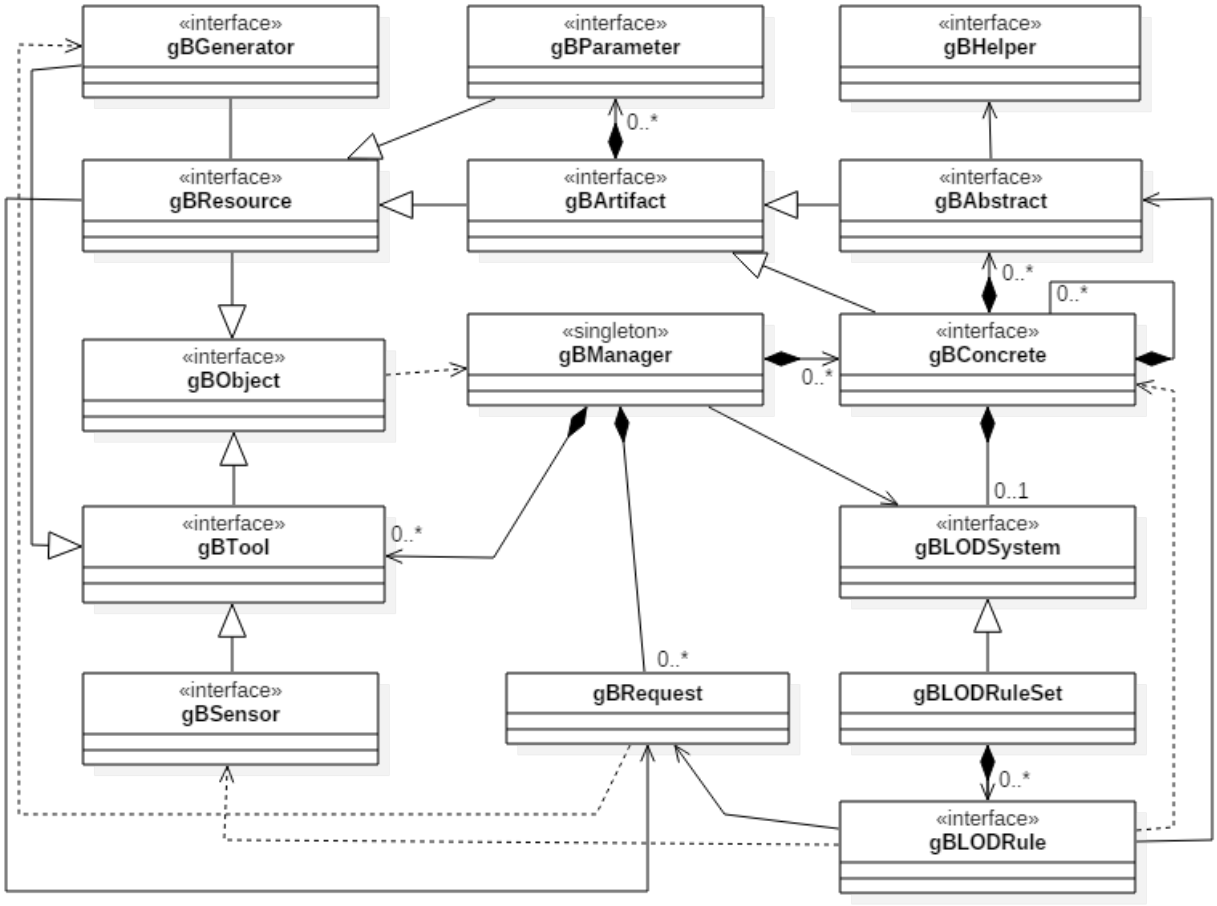

Figure 3: The *gameBITS Framework* architecture.

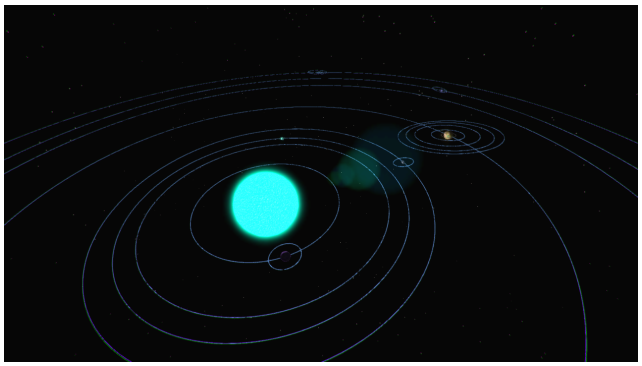

Figure 4: Planetary system generated by the *asset*.

Table 1: Elements contained in the generated system.

Artifact	Amount
Asteroids	21
Atmospheres	17
Barycenters	582
Dwarfs	2
Gas	5
Meteoroids	479
Oceans	3
Orbits	37
Rings	3
Rocky	2
Satellites	27
Stars	1

As a reference for performance evaluation, some tests were performed. Thus, a system was selected with a large number of elements. For an initial seed equal to 7777, a system with 1179 elements generated procedurally was created with a configuration as shown in Table 1.

Table 2 presents the results of time spent on the initial generation, and minimum and maximum memory consumption during execution, considering the generation of three quality levels, so that each level of quality is defined by the maximum texture size as follows: 512x256 pixels for low quality, 1024x512 pixels for medium quality, and 2048x1024 pixels for high quality. This evaluation is important because it mainly emphasizes the management efficiency of the resources generated and stored in memory.

Table 3 shows the average generation time and memory consumption by a common artifact at different levels of detail, with the same texture formats presented by the previous test, but with

Table 2: Time spent on the initial generation, and minimum and maximum memory consumption.

Quality	Time (ms)	Memory consumption (MB)	
		Min	Max
Low	1711	251	525
Medium	4053	260	838
High	13256	402	2788

the difference that for each quality level the complexity of the mesh also presents changes in the amount of polygons as follows: 320 polygons on low level, 1280 polygons on medium level, and 81920 polygons on high level. In this case, a rocky planet has been used as the subject to run the tests. This evaluation helps to provide an important reference for comparison with the consumption of global resources demonstrated by the previous test.

Table 3: Generation time of a concrete artifact.

LOD	Time (ms)	Memory consumption (MB)
Low	134	2,8
Medium	531	11,2
High	2445	45,5

By comparing the data in Table 2 and Table 3, it is possible to note the importance of dynamics in the generation and handling of artifacts, as by a simple math one can see that for keeping all the data loaded in memory during the entire time, it would be required more memory than the amount present in the hardware used, and on the other hand, if only the highest quality data remained loaded, there could be enough memory, but the processing would be completely compromised by the excessive amount of visually complex elements in the environment at the same time.

5.2 *TrackGen*: A procedural track generator

TrackGen[7] is an *asset* developed for the *Unity* game engine for the purpose of generating pseudo-endless tracks, commonly used in *Endless Runner* games. Each track is generated by combining a series of blocks fitted to each other, where each block suffers a deformation in its front face by changing rotation of its vertices relative to the center of frontal face itself in the x, y and z local axis, so that the next block to be generated fits the vertices of its back face on the vertices of the front face of the previous block, and thus repeating the process of deformation and fitting indefinitely to form the track. In order that the deformation control is consistent over time, coherent noise generation modules based on *Perlin Noise* were used, controlled by a parameters interface with a pre-determined range of values generated procedurally through a *Mersenne Twister* algorithm, just like the planetary systems generator. Figure 5 shows a generated track with the *asset*.

The implementation of this application allows to critically analyze the framework versatility matters, proving in this case the ability to use it in completely different contexts and with distinct levels of complexity. It is also important to highlight the reusability of resources, due to the proposed modularized solution induced by the framework architecture, about 80% of the track generator *asset* code comes from the modules built for the planetary systems generator, usually with just a few specializations, but without the need to change the original code.

The impact on performance while generating tracks proved negligible, as highlighted in Table 4. Each track section takes less than

[7] *TrackGen* can be found at `https://www.assetstore.unity3d.com/en/#!/content/65400`

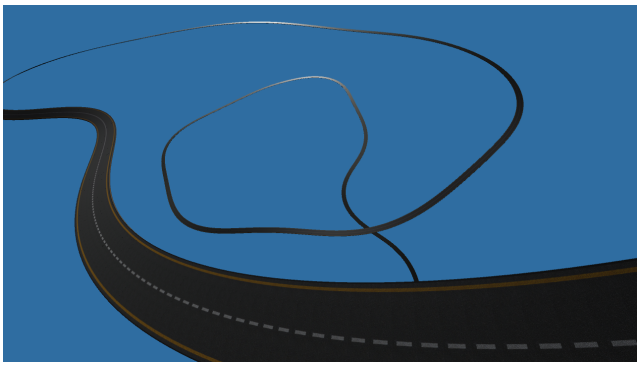

Figure 5: Track generated procedurally with the *asset*.

1 millisecond to be generated. A game prototype came to be developed in order to evaluate the performance of generating tracks process in a real scenario, where the track was generated at a rate of 20 blocks per second, but running the application both inserted in the game and outside of context, the rate of frames per second kept oscillating between 300 and 450 frames per second, which is well above the minimum acceptable as standard for interactive applications, ranging from 30 to 60 frames per second.

Table 4: Average generation time per track block.

Blocks generated	Time (ms)	Time per block (ms)
1000	859	0,859

5.3 *Stellar Empire*: An application in the context of a digital game

The game implementation was based on *StellarGen*, as the *asset* itself, as already highlighted, uses intensively the entire framework resource structure, which allows to clearly evaluate its potential inserted within the context of a game.

The developed game, called *Stellar Empire*[8], is based on a space economy system, where the player must grow his empire and expand its domain to other planets, satellites or asteroids present in the planetary system where it is located. The game is inspired by titles like *Sim City 4*[9] and *Imperium Galactica II*[10]. The gameplay is similar as regards the administration of resources, construction of buildings and units system. Figure 6 shows an in-game screen capture.

The game is slightly focused on visual aspects, and relies more on micromanagement matters of each celestial body dominated by the player, and on the management of spacecrafts that transport units and resources between each planet. In terms of hardware resource consumption, these characteristics tend to require more processing and memory than the GPU, which in this case is positive, since potential performance issues tend to stand out if there is a conflict between the use of resources for the game and the *asset*, which could provide clues on a bad design decision regarding the framework architecture. Table 5 exposes the performance results comparing a scenario where the *asset* runs in isolation, and in a second scenario where its already inserted into the game. The seed

[8] *Stellar Empire* can be found at `http://gamejolt.com/games/stellar-empire/162250`

[9] *Sim City 4*, 2003, developed by Maxis: `http://www.simcity.com/en_US/product/simcity4`

[10] *Imperium Galactica II*, 2000, developed by Digital Reality: `http://www.imperiumgalactica.com`

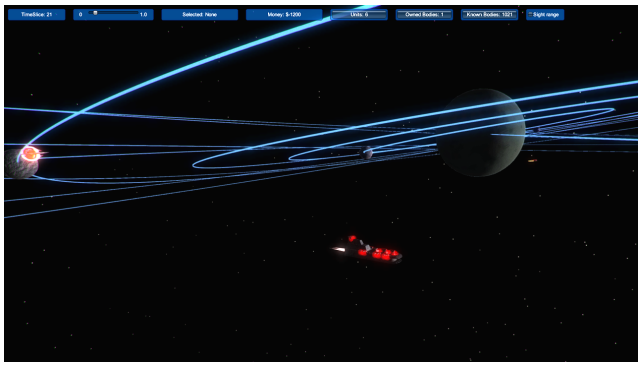

Figure 6: In-game image of the game *Stellar Empire*.

used was equal to 7777, and the quality setting was set to high with a maximum size of textures equal to 2048x1024 pixels.

Notably, while out-game the *asset* has full performance within acceptable standards for interactive applications, as well as in-game compared to average performance, but it is evident that during the game there are certain performance drops, but when compared to the minimum out-game performance it takes to assume that the poor performance responsibility is related more to the game itself than the *asset*, or even some internal process of the game engine, since the average tends to remain close to 60 frame per second.

Table 5: *Asset* performance in-game and out-game.

Context	Taxa de FPS		
	Min	Average	Max
Out-game	113	162	238
In-game	27	58	78

6 Discussion

The architecture model presented by the *gameBITS Framework* shown to be able to manage different sources of content generation, such as textures, mesh and pseudo-random numbers sequences, for example, while maintaining an internal organizational structure of the content generated, allowing resources to be handled and stored in nodes in a hierarchical structure analogous to a tree, similarly as employed by Carpenter [1].

The robustness of the solution is ensured by the high level of generalization adopted, allowing it to be flexible and extensible so that it can adapt to different contexts, to avoid specialization fashion solution as is observed in Keane [8], and by the modularized content generation that enhances the reusability of code through granularization of the solutions built on top of the framework.

The modularization is based on the analysis of the functional definition of artifacts generated procedurally introduced by Hendrikx et al. [5] combined with an adaptation of components system exposed by Gregory [3], as shown in Figure 7, in this context an object can be seen as an artifact, where components are different characterization parameters of the artifact or even abstract artifacts, if the first is a concrete artifact, for example. The framework also provides some management features of asynchronous requests with priority assignment, especially connected to the *LOD* control system, which allows a simple way to use *threads* for the parallelization of content generation via CPU, solving issues raised by Rudzicz [20].

Both *assets* developed corroborate for validating the applicability of the proposed framework in different scenarios and under different perspectives already emphasized, however for complete validation within the context of procedural content generation for digital games, it is considered necessary to introduce a solution based on the framework in an actual game, which was a clear gap in some of the studies reviewed. Thus it was proposed a game that could take advantage of the features offered by the planetary systems generator, so that it was possible to observe the behavior of the *asset* built based on the architecture presented by this work, competing with a medium complexity game for hardware resources, and thus be able to provide an analysis in a real environment and not just hypothetical, allowing the search for bottlenecks in the proposed architecture. In general, all cases showed positive results, although it was noticed certain underperforming situations within the game, these tends to manifest themselves more isolated and without a clear correlation with the solution, potentially without direct relationship with the *asset* or with the framework as suggested by the comparative performance tests.

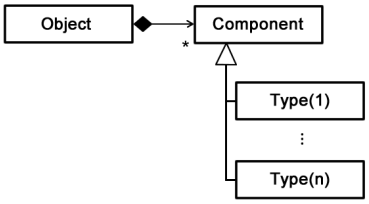

Figure 7: Components system, with an object containing several components. Adapted from Gregory [3].

However, the performance drop can also highlight the limits of the architecture in more complex scenarios, which does not affect the result obtained, as the proposal had always been targeted to low and medium complexity applications. Possibly, based on the results obtained in the context of the game, in more complex applications the introduction of more advanced *LOD* techniques would be necessary, like *Continuous LOD* or even *View-dependent LOD*, in order to provide greater processing performance and better use of memory without a big impact in frame rate, and therefore releasing more hardware resources to the game.

A possible gap in the proposed framework, when compared to Yannakakis et al. [23] and Merrick et al. [13] is related to the absence of an approach to a quality evaluation method, so as to provide a process which is able to quantify in a single metric the quality of an obtained result in the generation of artifacts from solutions built on top of the framework. These metrics could serve as a form of input for subjectivity capture system based on various artificial intelligence techniques, in order to treat high complexity matters, like individually setting each parameter within a parameter set, through simplified and more comprehensible metrics. An approach to solve this gap could allow the introduction of more abstract matters such as the application of *Game Design* concepts directly in the process of procedural generation, becoming ever more modeling results a rules-based design task, which would be much simpler from a human point of view, and less manual and low-level.

7 Conclusion and Future Work

In this paper several concepts related to procedural content generation for digital games were studied, trying to understand the techniques most commonly used in the generation of coherent noise for creating textures and height maps, and pseudo-random numbers generation techniques capable to provide number sequences with long periods for applications of various levels of complexity, also raising a number of works aligned with this proposal, serving as a reference for the proposition of the *gameBITS Framework* architecture.

The effectiveness of the framework has been demonstrated through the implementation of three applications built on top of

its structure, which were evaluated from the perspectives of performance, flexibility, extensibility and reusability. The results were positive, and corroborate for the validation of the proposal, mainly because of having employed the solution in a real game. Essentially, it can be stated that the *gameBITS Framework* is able to put it as a general purpose solution for small developers looking for ways of saving development time and financial resources in applications related to procedural content generation for digital games.

As future work, it is seen the need to extend the architecture towards an approach to manage and apply artificial intelligence techniques capable of providing means to capture the subjectivity connected with the process of evaluating the quality of the generated artifacts, turning it into simple metrics that allow their treatment at a higher level. Future works should also study the feasibility of extending the architecture to be used in more complex scenarios, also aiming at its validation in different contexts, in order to offer a solution to a wider range of applications of procedural content generation for digital games.

REFERENCES

[1] E. Carpenter. Procedural generation of large scale gameworlds. Master's thesis, University of Dublin, Trinity College, 2011.

[2] D. Ebert. *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann series in computer graphics and geometric modeling. Morgan Kaufmann, 2003.

[3] J. Gregory. *Game Engine Architecture, Second Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2014.

[4] S. Greuter, J. Parker, N. Stewart, and G. Leach. Undiscovered worlds–towards a framework for real-time procedural world generation. In *Fifth International Digital Arts and Culture Conference, Melbourne, Australia*, volume 5, page 5, 2003.

[5] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, Feb. 2013.

[6] R. Johnson, E. Gamma, J. Vlissides, and R. Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] B. Kachscovsky. Interactive methods for procedural texture generation with noise. Master's thesis, Uppsala University, Department of Information Technology, 2015.

[8] J. Keane. Procedural generation of planets in real-time. Undergraduate thesis, Teesside University, Middlesbrough, England, UK, 2014.

[9] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D. Ebert, J. Lewis, K. Perlin, and M. Zwicker. A survey of procedural noise functions. *Computer Graphics Forum*, 29(8):2579–2600, 12 2010.

[10] D. Luebke, B. Watson, J. D. Cohen, M. Reddy, and A. Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[11] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, Jan. 1998.

[12] A. Meer. Biomeshock: The new minecraft worlds. http://www.rockpapershotgun.com/2010/10/27/biomeshock-the-new-minecraft-worlds, October 2010. Accessed: 2015-10-21.

[13] K. E. Merrick, A. Isaacs, M. Barlow, and N. Gu. A shape grammar approach to computational creativity and procedural content generation in massively multiplayer online role playing games. *Entertainment Computing*, 4(2):115 – 130, 2013.

[14] F. Miranda, C. Cordeiro, and L. Chaimowicz. Um sistema para geração procedural de terrenos pseudo-infinitos em tempo-real utilizando gpu e cpu. In *Proceedings of VII Brazilian Symposium on Games and Digital Entertainment*, pages 113–116, Rio de Janeiro, RJ, BR, 2009.

[15] N. Nandapalan, R. Brent, L. M. Murray, and A. Rendell. High-performance pseudo-random number generation on graphics processing units. *Parallel Processing and Applied Mathematics*, 7203:609–618, 2012.

[16] R. Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[17] N. Pandeey. Implementation of leap ahead function for linear congruential and lagged fibonacci generators. Master's thesis, Florida State University, 2008.

[18] I. Parberry. Designer worlds: Procedural generation of infinite terrain from real-world elevation data. *Journal of Computer Graphics Techniques (JCGT)*, 3(1):74–85, March 2014.

[19] K. Perlin. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, pages 287–296, New York, NY, USA, 1985. ACM.

[20] N. E. Rudzicz. Arda: A framework for procedural video game content generation. Master's thesis, McGill University, School of Computer Science, 2009.

[21] C. E. C. Vieira, C. C. Ribeiro, R. de Castro e Souza, and P. U. C. do Rio de Janeiro. Geradores de números aleatórios, 2004.

[22] S. Worley. A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 291–294, New York, NY, USA, 1996. ACM.

[23] G. N. Yannakakis and J. Togelius. Experience-driven procedural content generation. *IEEE Trans. Affect. Comput.*, 2(3):147–161, July 2011.