Procedural Content Generation for Game Props? A Study on the Effects on User Experience

OLIVER KORN, Offenburg University
MICHAEL BLATZ, ADRIAN REES, and JAKOB SCHAAL, KORION GmbH
VALENTIN SCHWIND, University of Stuttgart
DANIEL GÖRLICH, SRH Hochschule Heidelberg

This work demonstrates the potentials of procedural content generation (PCG) for games, focusing on the generation of specific graphic props (reefs) in an explorer game.

We briefly portray the state-of-the-art of PCG and compare various methods to create random patterns at runtime. Taking a step towards the game industry, we describe an actual game production and provide a detailed pseudocode implementation showing how Perlin or Simplex noise can be used efficiently.

In a comparative study, we investigate two alternative implementations of a decisive game prop: once created traditionally by artists and once generated by procedural algorithms. 41 test subjects played both implementations. The analysis shows that PCG can create a user experience that is significantly more realistic and at the same time perceived as more aesthetically pleasing. In addition, the ever-changing nature of the procedurally generated environments is preferred with high significance, especially by players aged 45 and above.

Categories and Subject Descriptors: I.6.8 [**Types of Simulation**]: Gaming; Animation; K.8.0 [**General**]: Games; D.2.9 [**Management**]: Productivity

General Terms: Design, Human Factors, Performance, Management

Additional Key Words and Phrases: Procedural content generation, game design, algorithms

ACM Reference Format:
Oliver Korn, Michael Blatz, Adrian Rees, Jakob Schaal, Valentin Schwind, and Daniel Görlich. 2017. Procedural content generation for game props? A study on the effects on user experience. Comput. Entertain. 15, 2, Article 1 (February 2017), 15 pages.
DOI: http://dx.doi.org/10.1145/2974026

INTRODUCTION AND MOTIVATION

In video games, graphics can be generated in different ways: on the one hand, there is the traditional creation through skilled artists; on the other hand, graphics can be generated by algorithms, i.e., procedurally. Thus, the programmers of the algorithm partially substitute the work previously done by artists. This procedural content generation (PCG) is getting more and more attention from the computer games industry. Especially when many varied iterations of similar elements (e.g., trees, walls, swords) are required, PCG is an interesting solution.

Authors' addresses: O. Korn, Offenburg University, Badstr. 24, 77652 Offenburg, Germany; email: oliver.korn@acm.org; M. Blatz, A. Rees, and J. Schaal, KORION GmbH, Moempelgardstr. 16, 71640 Ludwigsburg, Germany; emails: {michael.blatz, adrian.rees, info}@korion.de; V. Schwind, University of Stuttgart, VIS, Pfaffenwaldring 5a, 70569 Stuttgart, Germany; email: valentin.schwind@vis.uni-stuttgart.de; D. Goerlich, SRH Hochschule Heidelberg, Ludwig-Guttmann-Str. 6, 69123 Heidelberg, Germany; email: daniel.goerlich@hochschule-heidelberg.de.

DOI: http://dx.doi.org/10.1145/2974026

However, does PCG reach the quality of the artists? To address this question, we need to look at the development of computer graphics. Just a few years back, the look of games was mainly limited by technical possibilities. Today games like *Minecraft* use a simple "pixel look" deliberately as a stylistic device [Persson 2011]. Therefore, the current spectrum in game graphics reaches from pixel graphics from the era of the Commodore 64 to high-definition 3D graphics [Jones 2013].

However, even the best graphics are no guarantee for appreciation. The *Call of Duty* series undoubtedly is a flagship of the game industry, produced with great financial and artistic effort. Still it is notoriously criticized for tube-like level maps with little freedom of choice. Such restrictions are typical flaws of manually generated levels, making this series a representative of state-of-the-art manual content generation. However, the richness of detail and the cinematic quality of some dramatic moments in the narration can only be achieved through a high level of control, at the cost of the players' freedom [Yannakakis and Togelius 2011].

When aiming at such cinematic experiences, more and more cost and effort are required for the development of high-quality assets. A way to save resources is not using the artists' capacity for the production of similar assets. PCG, graphics can be modified with a variety of randomized parameters at runtime, resulting in completely different looks of generic assets like rocks or trees. As a logical next step, whole buildings and eventually whole game worlds can be generated procedurally [Hosking 2013].

The successful game series *Borderlands* shows that PCG can not only be used to generate diversity in graphics, but also in gameplay: a countless number of different weapons, grenades and shields is randomly generated, creating the illusion of technical variety und fueling the players' passion for loot, resulting in a vivid game experience.

A holistic procedural approach is taken by *Minecraft*, which not only generates game elements but the whole game world procedurally [Persson 2011]. Generation takes place at runtime, making it possible to constantly discover new areas of the world. In combination with the many ways to manipulate the game world, this generates a very high replay value [Belinkie 2010]. Thus, the procedural approach is corresponding well to a paradigm in game development: "the game enables the experience, but it is not the experience" [Schell 2015].

Due to the many possibilities of PCG, it is difficult to create a universal formula for the frequency and type of its implementation. Most games are either created with PCG elements or without them—thus it usually is impossible to tell how the game experience would have been different if another type of content generation had been used. In order to create valid findings on the effects of PCG, a control condition is required—a game that is identical in all aspects but one: using or not using PCG.

In the case study presented in this work, we used the production of the documentary game *Conquest of the Seven Seas* [Korn et al. 2015b] by the game studio KORION to do just that: one implementation is based on PCG while the other one is using the traditional artist approach (Figure 1). Both alternatives were played and judged by 41 players. Based on the study, we examine the acceptance of PCG in different age groups and validate if claims regarding the positive effect of PCG on the replay value are substantiated.

BACKGROUND

For a better understanding of how PCG can be utilized in games, we provide a short overview of the most influential methods to create procedural content.

Algorithms for generating procedural graphics are frequent in computer games; yet there are others focusing on artificial intelligence (e.g., AI Director) or changes of state (e.g., Markov chains). Thus, the implementation of specific PCG methods strongly depends on the particular use case.

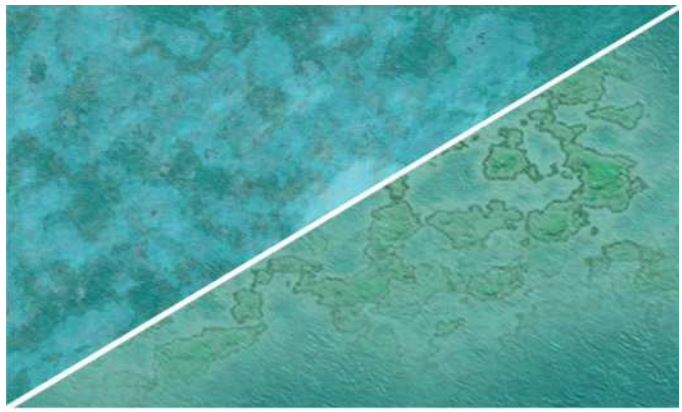


Fig. 1. Procedurally generated reefs (top) and reefs designed by an artist (bottom). A study evaluates players' preferences regarding aesthetics and change.

Fig. 2. Romanesco is a good example of how iterative algorithms can procedurally create aesthetically pleasing structures.

The most common use of PCG is probably generating terrain or landscapes [Shaker et al. 2016]. Typically, noise functions are used to create such content, but principally other solutions like fractal-based algorithms could also generate the required self-similar structures. To provide an overview, we will briefly describe some of the most common PCG methods.

Mandelbrot Sets

Mandelbrot sets (after Benoit Mandelbrot) are fractal structures. Fractals are defined as a mathematical set that exhibits a repeating pattern that displays at every scale [Boeing 2016]. The fact that several plants incorporate structures similar to fractals (Figure 2) indicates that they are well suited to represent natural structures. Mandelbrot stated, that the forms of nature, like trees, mountains, shore lines, or clouds, can only be adequately described using fractals [Ziegler 2013].

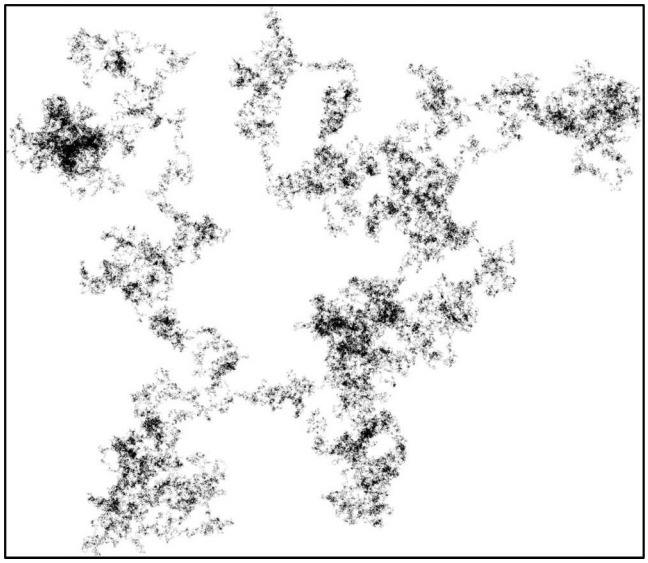


Fig. 3. Self-avoiding random walk in two dimensions. The darker the region, the more it has been visited. Created with MATLAB by Purpy Pupple, CC BY-SA 3.0.

As fractals are recursive functions with potentially infinite depth, the only technical restriction when using them is the increase in computing time that comes with increasing recursion depth. For the use in computer games, especially on mobile platforms, this makes a real-time implementation of fractals problematic.

Brown Fractals

The discovery of Brownian motion goes back to botanist Robert Brown, who observed the random motion particles of pollen grain particles in water. This motion is the result of non-visible molecules in liquids and gasses that move with random velocities, in random directions, colliding with the pollen grains. The mathematician Norbert Wiener was deeply impressed by this phenomenon and tried to reconstruct it in an algorithm. Later the French physicist and Nobel-prize winner Jean Perrin had the idea of describing the Brownian motion with continuous non-differentiable curves [Mandelbrot 1982], a method called "Wiener process." Mandelbrot used the Brownian motion to develop what he calls a "self-avoiding random walk" [Mandelbrot 1982]. It describes a function, laid out on a grid, not allowing the intersection of individual points (Figure 3).

While the implementation does not take into account the previous position in the grid, it employs a rule that already entered areas cannot be entered again. In addition, areas the function could not leave again are blocked. Brownian motion therefore is well suited to generate forms that are similar to land masses or jagged shorelines.

Perlin Noise

The algorithms mentioned above all require a considerable amount of computing power due to recursion. For practical purposes, it is mandatory to find alternatives that can describe natural forms with less computing effort. Noise algorithms not based on recursions are such alternatives. A noise function is a mapping, which assigns a random value to every natural number.

In the 80s, Ken Perlin addressed the question of how noise images can describe natural forms. He was frustrated with the pixelated look of graphics at that time and

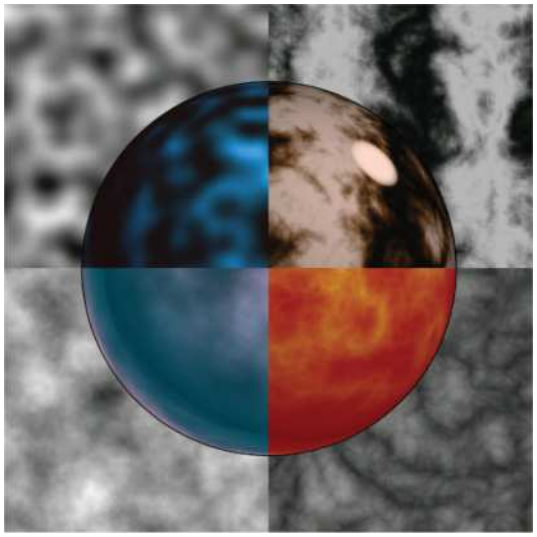


Fig. 4. Different versions of noise as described by Perlin.

searched for an "image synthesizer", generating "naturalistic visual complexity" [Perlin 1985]. He defined his noise as "a texturing primitive you can use to create a very wide variety of natural looking textures" and explained that "combining noise into various mathematical expressions produces procedural texture" [Perlin 1999]. This texture can be applied to any object, so Perlin noise can be used in 2D as well as in 3D (Figure 4). Another advantage of gradient and noise functions is that, even if the distribution is randomly generated, every configuration can be saved and reapplied so that the same conditions will always produce the same results.

Perlin used frequency and amplitude to describe the noise function. The texture is a single-channel image (usually greyscale) with pixel values represent frequencies between -1 and $+1$. A high frequency (up to 1) means smaller details, resulting in many small points in a specific area, while a low frequency results in mostly large points. The noise function's amplitude describes the color values' "height" between completely black (0) and completely white (255).

The implementation is described in three steps [Perlin 1999]:

1. Given an input point
2. For each of its neighboring grid points: pick a "pseudo-random" gradient vector.
3. Compute linear function (dot product)
4. Take weighted sum, using ease curves.

As this implementation only performed well in up to three dimensions, a revised version called "Simplex Noise" [Olano et al. 2003] was developed. Simplex noise does not use a grid based on quads, but a grid with equal-sided triangles, reducing the number of neighboring points and therefore computing time. While Perlin noise has the complexity of $O(2n)$, Simplex noise only requires $2(n^2)$, using $n + 1$ vertices instead of $2n$. This is most noticeable when working in dimensions with $n > 4$.

STATE OF THE ART

In the introduction, we described that procedural generation is a versatile tool to enrich computer games. In the following, we will describe approaches to structure procedurally generated contents and their impact on game design.

Table I. Structural Overview on the Use of PCG in Games by Hendrikx et al. 2013 (Copyright granted by Mark Hendrikx)

Games	Release	Game Bits	Game Space	Game Systems	Game Scenarios
Borderlands	2009	X			
Diablo I	2000		X		
Diablo II	2008		X		X
Dwarf Fortress	2006		X	X	X
Elder Scrolls IV: Oblivion	2007	X			
Elder Scrolls V: Skyrim	2011				X
Elite	1984		X	X	X
EVE Online	2003	X	X		X
Facade	2005				X
FreeCiv and Civilization IV	2004		X		
Fuel	2009		X		
Gears of War 2	2008	X			
Left4Dead	2008				X
.kkrieger	2004	X			
Minecraft	2009		X	X	
Noctis	2002		X		
RoboBlitz	2006	X			
Realm of the Mad God	2010	X			
Rogue	1980		X		X
Spelunky	2008	X	X		
Torchlight	2009		X		
X-Com: UFO Defense	1994		X		

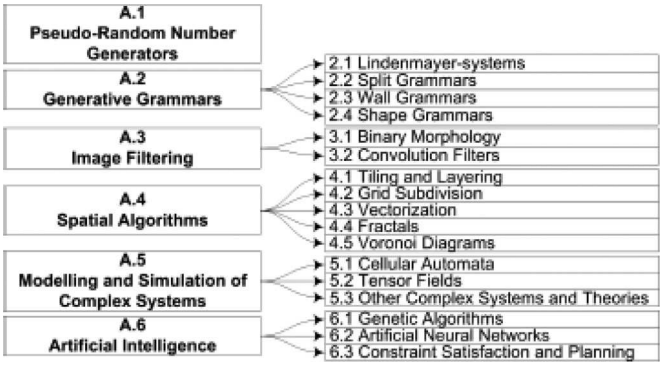
In most cases, not the whole game, but only specific parts are created procedurally. Besides landscapes and maps [Togelius et al. 2011], also game characters, artificial intelligence or stories, and quests can be generated procedurally [Hosking 2013]. On the structural level [Hendrikx et al. 2013] differentiate between "Game Bits" (e.g., textures, sound), "Game Space" (e.g., the game world), "Game Systems" (e.g., complex relations between game objects), and "Game Scenarios" (e.g., Story). Using this differentiation, the application of PCG in some of the most influential video games is analyzed (Table I).

We believe that this system is an approach well suited to describe PCG use in game. Especially, if it the content-oriented perspective is complemented by a technical perspective. Indeed, in a second step, Hendrikx et al. [2013] categorize PCG use according to different levels of technical complexity, beginning with pseudo-random number generators up to artificial intelligence (Table II).

In the field of artificial intelligence, Valve has drawn some attention with the series *Left 4 Dead*. Using the tool *AI Director* they changed the game world depending on the course of the game. This was done in a covert way, e.g., by enabling passages that were previously blocked. Contrary to many competitors, they not only changed the behavior of the players' opponents, but also the game map according to the flow of the game, a method described in detail in Thue and Bulitko [2012]. Additionally, according to the calculated stress level of the player, enemy waves were being adapted either to challenge the player or to give him time to relax.

This tactic builds on the concept of "flow" [Csíkszentmihályi 1975; Csíkszentmihályi et al. 2005], where users are confronted with an adequate level of challenge. In single player games, this usually means that there are times of stress and times of regeneration. This universal approach in the meanwhile is even mirrored in other domains: recent advances in context-aware systems for production environments [Korn et al. 2014; Korn et al. 2015a; Funk et al. 2014] use sensors to determine a user's stress

Table II. Categorization of Types of PCG by Hendrikx et al. 2013
(Copyright granted by Mark Hendrikx)



level and thus the position on the flow curve between excitement and control. Then the challenge level is either raised to prevent boredom or lowered to prevent overextension.

Such examples of user-centered adaption are impressive. However, when procedural content is used in games, there is no sensory feedback on the player's reactions, so the game sessions have to be carefully tested or evaluated [Togelius et al. 2012]. An influential model for modeling and evaluating player experience is presented by Pedersen et al. [2010]. It predicts certain key affective states of the player based on both gameplay metrics that relate to the actions performed in the game, and on parameters of the level that was played.

It is important, that players do not perceive a situation as unfair and at the same time can use experiences from previous game sessions in ensuing ones. To meet these demands, some games, like the *Super Mario* series, do not generate their levels at runtime. Instead, procedural level generation takes place in several steps: First, the basic level containing gaps for jumping passages is created. Then level decorations, like enemies, coins and tubes are put into place [Smith 2012]. This model of generation allows first checking the level without content, and re-generating it if necessary.

In this section, we mainly looked at PCG in console and PC games. For the fast-growing mobile market, to our knowledge there is yet no systematic investigation on PCG usage. However, the use of PCG for mobile platforms is especially rewarding, as mobile games often are small-scale productions. Games like *Doodle Jump* or *Tiny Wings* incorporate a simple gameplay and a high replay value—and both concepts are well suited for the use of PCG.

Regardless of the potentials of PCG, there are also downsides: for example visual repetitions or an unpleasant "boring" look. In the next section, we will examine this phenomenon based on two implementations of documentary games.

IMPLEMENTATION

To measure the impact of PCG on user experience systematically, we used the game production *Conquest of the Seven Seas*, a tactical strategy game by the studio KORION. The developers implemented two different versions: in one version, artists manually created the "reefs," a central game prop; the alternative version used a procedural approach (Figure 6).

In the game, players maneuver ships through the islands and the reefs to put themselves in superior firing positions (Figure 5). The ships' motion path is adjusted with a Bezier curve, which reacts to the direction and strength of the wind. The reefs were intentionally selected: like trees or rocks, they require a high visual diversity and still

Fig. 5. Explorers' ships maneuvering through dangerous waters. The reef (here: manually generated) is at the top left.

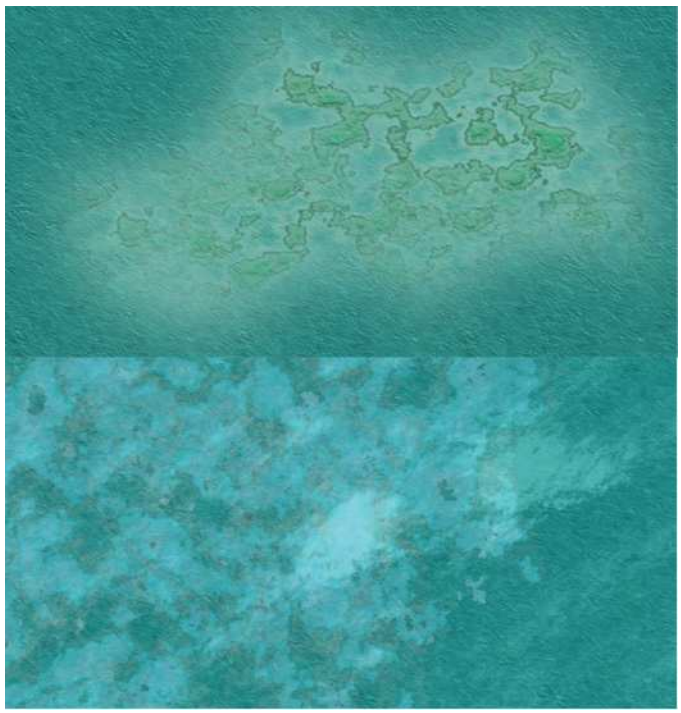


Fig. 6. Reef structures designed by an artist (top) and generated procedurally by Perlin noise (bottom).

need to "look the same". Moreover, they are not just a decorative prop but feed into a core game element: as the wind changes direction, it creates ever-new conditions for the players. Avoiding the reefs is a central element of the gameplay, as a ship takes severe damage if it hits a reef or an island.

Choosing the appropriate PCG method

The version without PCG featured nine manually generated reefs. Artists designed them based on pictures of the Australian Great Barrier Reef. The corresponding PCG implementation had to meet two requirements: respect the hardware limitations of mobile devices and produce realistic results when compared to photographs of actual reefs.

This scenario combined the typical limitations of PCG: shortage on resources (see Introduction) and achieving the highest possible graphical quality (see State of the Art). As the overall game had a realistic look consisting of very detailed sprites, the reefs had to be designed at an equal level of detail to match the appearance. With these demands in mind, the selection could be narrowed down to two methods of PCG: self-avoiding fractals and Perlin noise. Both methods are suited to produce structures similar to reefs. Finally, Perlin noise was selected for two reasons:

—The two-dimensional Perlin noise demands less computing effort.
—The generated noise image already offers a suitable structure making it possible to only draw the desired area and blending the remaining greyscales with the surroundings through tone addition. With self-avoiding fractals, the final structure would have to be calculated separately.

The developers from KORION implemented the procedural generation in *GameMaker Studio* by YoYo Games. This IDE allows fast prototyping and offers basic drawing operations as well as an HTML 5 export.

The pseudocode in Figure 7 shows an implementation of Perlin noise for the reef structures. The algorithm consists of two parts: initialization and calculation. First, the variables are set up to define the datatype, as the flooring of integer value calculations is required. The frequency of supporting discrete gradient points in both directions is doubled with each octave; the amplitude is multiplied with the persistence value. Typically, five octaves are iterated.

At the start of each octave, the discrete gradients are randomly chosen between -amplitude and +amplitude. Then every sample value is calculated in three steps:

(1) Calculate weight for each neighboring discrete gradient point;
(2) Bilinear interpolate these values (w00. . .w11) to the final result (for this sample at this octave) by the Blend Function;
(3) Each interpolation is added to the final Perlin noise.

Due to the grey values being adjustable in Perlin noise, it was possible to specify exactly, which values should be drawn, so, all values that were black or in dark grey were assigned an alpha value of 0, excluding them from being drawn. The resulting noise is normalized and cut off below 0.2. Removing this data resulted in the sharp outline around the remaining values, generating the reef's distinctive structure. This structure is a result of the remaining values between 0.2 and 1.0, which are mapped to grayscale color. Although Perlin noise typically is used for height maps with very soft edges, this straightforward approach matches the natural appearance in the contours of reefs very well.

A parametrized example for the function call would be: `perlin2D(5, 32, 32, 1, 0.5, 1024, 1024);`

We will shortly explain the parameters in this example to ease re-implementations based on the algorithm presented in Figure 7:

—*octaves [3-8]:* The number of octaves. A higher number produces a higher quality noise at the cost of performance. The amount of discrete gradients doubles with each octave, so performance costs grow exponentially. Normally, a value of 5 is sufficient;

values above 8 do not produce noticeable differences but cost a lot more processing time. Values below 3 produce very low quality noises.
—*startFrequencyX/Y [5-64]:* Higher numbers produce higher fluctuations in the resulting noise.
—*startAmplitude [0.1-1.0]:* This parameter's effect depends on the use of the noise. Higher values result in higher values in the generated noise and vice-versa. If the noise is normalized after generation, the value has no effect as long as it is above or equal to 0.1.
—*persistence [0.1-0.9]:* This parameter changes the amplitude of the octaves >1. A value of 0.5 is a good starting value. Values below 0.1 give octaves >1 too less value while values above 0.9 make them nearly as strong as the first octave.
—*sampleWidth/Height [64-4k_or_more]:* The width and height of the produced image/array/noise. Sets the required size of the output.

Note that these parameters are interrelated. For example, a higher amount of octaves means that the persistence has to be increased accordingly—otherwise higher octaves would become invisible.

A timer called at the beginning of the generation, showed that level generation (result in Figure 6, bottom) usually took less than one second. This shows that Perlin noise indeed allows a resource saving implementation.

As described in State of the Art, the degree to which PCG is applied in games can easily exceed the implementation shown in this case study. For example, even in the game presented here, clouds still consist of a number of handmade sprites, while they could have easily been procedurally generated with methods like Perlin noise. However, for the purpose of the study, the procedural implementation of just a single game element was essential.

STUDY

In this section, we describe a comparative study, laid out to determine the effects of the two alternative reef implementations (with and without PCG) on the players' experience.

Expected Results

As explained above, PCG is not limited to just affecting the look of a game—it can also affect the gameplay. While visual variety already reduces the frustration generated by redundant designs, PCG that affects the gameplay potentially increases replay value. In the selected game, the reefs differ in every map, constantly demanding tactical planning and flexible ship maneuvers. Thus, the reefs have a strong impact on the gameplay. However, while the manually generated reefs differ only in number, appearance, and position, the procedurally generated ones additionally differ in shape.

Through this constant variation of the map, players were to be motivated to continuously risk new battles; even lost battles should increase motivation as there is an element of chance—similar to gambling, but less intense. The prospect of more advantageous wind conditions and reef positions in the next battle are well suited to increase the replay value. Our hypothesis was that players would recognize and favor the greater variation of the PCG approach.

Population and Setup

Forty-one test subjects participated in the study. Twenty-six were aged 18–24, eight were aged 25–29, and seven were aged 45 and above. To determine if PCG has a different impact on middle-aged persons, we intentionally included test subjects older than 45. For the answers to remain as unbiased as possible, we did not indicate that

```
float[][] function perlin2D(int octaves, int startFrequencyX, int startFrequencyY, float
startAmplitude, float persistence, int samplesWidth, int samplesHeight){
        int frequencyX = startFrequencyX;
        int frequencyY = startFrequencyY;
        float amplitude = startAmplitude;
        float[][] perlin = new float[samplesWidth][samplesHeight];

        for(int i = 0; i < octaves; i++){
                float[][][] discretePoints = new float[frequencyX + 2, frequencyY + 2, 2];
                for(int m = 0; m<frequencyX + 2; m++){
                        for(int n = 0; n<frequencyY + 2; n++){
                                discreteGradients[m, n, 0] = Random.Range(-1f, 1f) * amplitude;
                                discreteGradients[m, n, 1] = Random.Range(-1f, 1f) * amplitude;
                        }
                }
                for(int k = 0; k<samplesWidth; k++){
                   float positionX = k/((float)samplesWidth) * frequencyX + 1;
                   int currentPosX = (int) positionX;
                   float x = positionX - currentPosX;
                   for(int m = 0; m<samplesHeight; m++){
                        float positionY = m/((float)samplesHeight) * frequencyY + 1;
                        int currentPosY = (int) positionY;
                        float y = positionY - currentPosY;

                        float w00 = discreteGradients[currentPosX, currentPosY, 0]
                         * x + discreteGradients[currentPosX, currentPosY, 1] * y;
                        float w10 = discreteGradients[currentPosX + 1, currentPosY, 0]
                         * (x - 1) + discreteGradients[currentPosX + 1, currentPosY, 1] * y;
                        float w01 = discreteGradients[currentPosX, currentPosY + 1, 0] * x
                         + discreteGradients[currentPosX, currentPosY + 1, 1] * (y - 1);
                        float w11 = discreteGradients[currentPosX + 1, currentPosY + 1, 0]
                         * (x - 1) + discreteGradients[currentPosX + 1, currentPosY + 1, 1] * (y - 1);

                        float w0 = (1 - blendFunction(x)) * w00 + blendFunction(x) * w10;
                        float w1 = (1 - blendFunction(x)) * w01 + blendFunction(x) * w11;

                        perlin[k, m] += (1-blendFunction(y)) * w0 + blendFunction(y) * w1;
                   }
                }
                amplitude *= persistence;
                frequencyX *= 2;
                frequencyY *= 2;
        }
        return perlin;

}
float function blendFunction(float x){
        return 10 * x^3 - 15 * x^4 + 6 * x^5;
}
```

Fig. 7. The code used to generate the Perlin noise. While it looks complex, this still is Pseudocode. However, it should provide enough detail for similar implementations, especially in Java and C#.

PCG was the research topic beforehand. The study was set up in three subsequent steps:

(1) *Collecting Participants' Data.* We asked for age, experience with different gaming systems and playing time per week. The aim was to determine, whether the test person according to their game experience would benefit from PCG.
(2) *Playing the Alternative Implementations.* The test subjects played the two alternative implementations in randomized order, each implementation for 10 to 15 minutes.
(3) *Determining the User Experience.* After the test, we used a questionnaire to measure the subjects' experiences during playing. The aim was to determine which implementation was favored and thus if PCG created additional value.

Previous Game Experience

Each of the test subjects had experience with games, mostly on the PC but also on consoles. 78% of the participants also played on mobile phones or tablets. 39% played more than 15 hours per week and therefore were considered frequent players. An additional

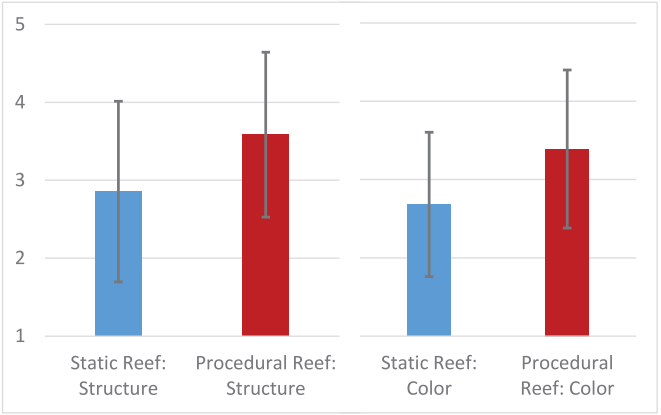


Fig. 8. Acceptance of the reefs based on perceived realism with standard deviations.

24% played 5–15 hours and 37% played less than 5 hours per week. Concerning genres, the users preferred shooters (26%) followed by strategy games (19%), role-playing games (16%), puzzle games (9%), Jump and Run (7%), and racing games (7%).

User Experience with and without PCG

We evaluated the user experience using a five-point Likert-scale and compared the results using Student's t-test. As a first step, we compared the acceptance of the manually generated and the procedural reefs (Figure 6), asking how realistic the reefs were on a scale from 1 (completely unrealistic) to 5 (completely realistic). To prevent misunderstandings about "realism" we asked separately about both the structure and the color (Figure 8).

In both criteria, the procedural version of the reefs was preferred to the manually generated one. With respect to structure, the static reefs' mean acceptance rate was $\bar{x} = 2.9$ (SD $= 1.2$), the procedural reefs were preferred with $\bar{x} = 3.6$ (SD $= 1.1$). This difference is highly significant ($p < 0.01$).

With respect to color, the static reefs' mean acceptance rate was $\bar{x} = 2.7$ (SD $= 0.9$), while the procedural reefs were rated $\bar{x} = 3.4$ (SD $= 1.0$). Again, the difference is highly significant ($p < 0.01$). Thus, the hypothesis that users perceive procedural contents as more realistic is strongly supported. The higher variation of the procedural reefs has successfully created the impression of a richer and more realistic world.

In a second step (Figure 9, col. 1, 2), we moved from an objective criterion ("realism") to a more subjective one and asked which reefs were "more beautiful." The underlying hypothesis was that while the users might perceive procedural reefs as more realistic, they still could consider them as unaesthetic, thus decreasing the user experience.

In the third step (Figure 9, col. 3, 4), we explicitly asked about the preference for persistency versus change: we assessed the statements "I prefer reefs A as the look remains constant" versus "I prefer reefs B as the look changes frequently."

When it comes to aesthetics, the users did consider neither of the solutions as beautiful, although "beauty" might be difficult to accomplish for abstract structures altogether. The static reefs' mean aesthetic quality was rated $\bar{x} = 2.2$ (SD $= 1.0$) while the procedural reefs were attributed $\bar{x} = 2.8$ (SD $= 0.9$). This difference is significant ($p < 0.015$). So users considered the procedural reefs as both more realistic and more aesthetically pleasing.

When we explicitly mentioned the concept of constancy versus change, the results are extremely clear: the static reefs' appeal with respect to a constant look was assessed

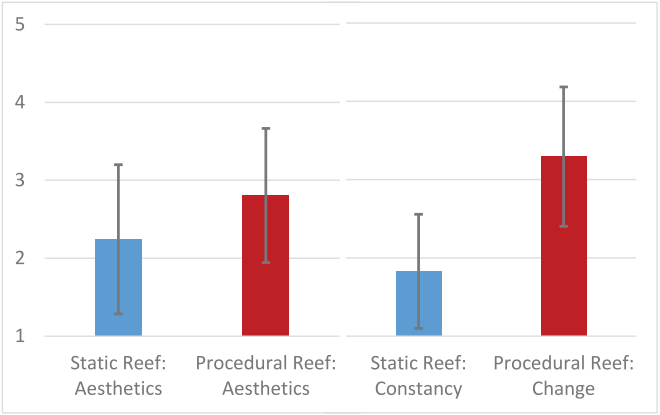


Fig. 9. Perceived aesthetic quality (columns 1 and 2) and preference with the explicit mentioning of constancy and change (columns 3 and 4) with standard deviations.

with $\bar{x} = 1.8$ (SD $= 0.7$) while the procedural reefs' appeal with respect to a changing look was assessed with $\bar{x} = 3.3$ (SD $= 0.9$). This difference is highly significant with $p < 0.01$ and even $p < 0.0000001$.

Discussion

It is interesting that 85% of the users favor PCG in the last question, as the differences when analyzing perceived realism and aesthetics were significant but less obvious: although about 60% of the players preferred the PCG-reefs, 40% of them still preferred the manually generated ones. We have two possible explanations for this: first, the changing look might be the most attractive element of the PCG-reefs—so when directly addressed, this advantage is more prominent; second, the words "constancy" versus "change" might create certain semantic priming effects, for example, a "modern" person might be more inclined to embrace change. Thus, the strong effect might also be attributable to a bias in the users' self-perception.

However, if such a bias existed, our data indicate that it is restricted to gamers aged 45 and older: there is both a negative correlation between age and the preference for static reefs ($r = -0.32$) and a positive correlation between age and the preference for procedurally generated reefs ($r = 0.28$). When comparing the affinity for a game experience with more variety by procedural reefs (Figure 10), the 26 players aged 18-25 embraced this change more ($\bar{x} = 3.3$, SD $= 0.8$) than the eight players aged 25-44 ($\bar{x} = 2.8$, SD $= 1.1$) – but the seven players aged 45 and above appreciated change most ($\bar{x} = 4.0$, SD $= 0.0$).

Even when comparing the players aged below and above 45 directly, this difference is highly significant ($p < 0.00001$). The stereotype intuition ("PCG creates experiences for young gamers") is not supported. It is the gamers aged 45 and above, who enjoy the effects of PCG most. This may be due to their advantage in life experience: they might prefer games mirroring the complexity of life—or they simply enjoy being surprised.

CONCLUSION

In this work, we first summarized the historical background of procedural algorithms in games. Then we described their use in various games and presented established methods to categorize and structure the use of PCG in games.

Based on the history of PCG, there are numerous ways to incorporate procedural algorithms in computer games. We showed that Perlin noise is an especially effective

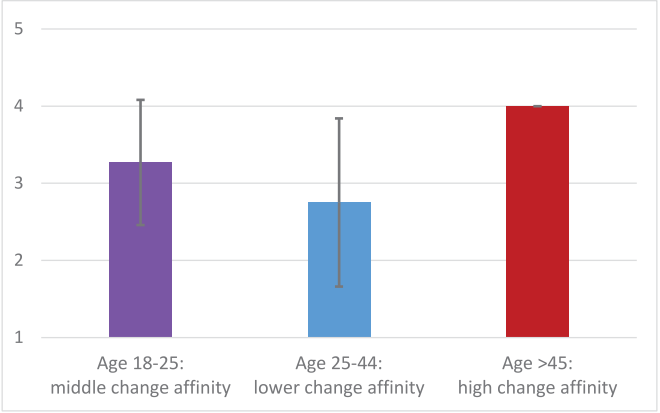


Fig. 10. Age groups and their affinity to change with standard deviations.

method when processing resources are limited. Within an ongoing game production, we developed an algorithm for generating reefs procedurally. We provided a detailed pseudocode implementation showing how Perlin or Simplex noise can be used efficiently.

Based on this procedural implementation and the traditional approach with manual designs by artists, we evaluated the effects of PCG on the user experience. The study shows that in comparable structures (here, manually versus procedurally generated reefs in exactly the same game) the use of PCG can be much more than a way to save money in production: it can create an experience that users perceive as both significantly more realistic and significantly more aesthetically pleasing.

Furthermore, users perceive the changing nature of the game environments as a great advantage. Changing environments are (highly significantly) preferred to traditional static environments. Interestingly, players aged above 45 enjoy this increased variation even more than the younger ones.

FUTURE WORK

The work presented here is a comparative study on the effects of PCG. To validate the findings, similar studies are required. These could be similar in scope, i.e., focusing on the effects of graphical elements on the gameplay. Ultimately, the scope should be broadened to examine the use of procedurally generated quests, characters or even artificial intelligence behaviors. However, such studies are difficult to implement: if actual games are to be used, the evaluation can only take place in the late phase of a game production, where the developers can still create alternative versions (at least for playable sub-aspects of the overall game design).

REFERENCES

Matthew Belinkie. 2010. What makes Minecraft so addictive? *Overthinking It* (November 2010).

Geoff Boeing. 2016. Visual analysis of nonlinear dynamical systems: Chaos, fractals, self-similarity and the limits of prediction. *Systems* 4, 4 (November 2016), 37. DOI: https://doi.org/10.3390/systems4040037

Mihály Csíkszentmihályi. 1975. *Beyond Boredom and Anxiety*, San Francisco, USA: Jossey-Bass Publishers.

Mihály Csíkszentmihályi, Sami Abuhamdeh, and Jeanne Nakamura. 2005. Flow. In *Handbook of Competence and Motivation.* New York, NY, USA: Guilford Press, 598–608.

Markus Funk, Oliver Korn, and Albrecht Schmidt. 2014. An augmented workplace for enabling user-defined tangibles. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'14)*. ACM, New York, NY, USA. DOI: https://doi.org/10.1145/2559206.2581142

Mark Hendrikx, Sebastiaan Meijer, Joeri Van DerVelden, and Alexandru Iosup. 2013. Procedural content generation for games: A survey. *ACM Trans Multimed. Comput Commun Appl* 9, 1 (February 2013), 1:1–1:22. DOI:https://doi.org/10.1145/2422956.2422957

Claire Hosking. 2013. Stop dwelling on graphics and embrace procedural generation. *Polygon* (October 2013).

Olly Jones. 2013. Polygons to pixels: The resurgence of pixel games. *Gaming Illus.* (September 2013).

Oliver Korn, Markus Funk, Stephan Abele, Thomas Hörz, and Albrecht Schmidt. 2014. Context-aware assistive systems at the workplace: Analyzing the effects of projection and gamification. In *Proceedings of the 7th International Conference on Pervasive Technologies Related to Assistive Environments (PETRA'14)*. New York, NY, USA: ACM, 38:1–38:8. DOI:https://doi.org/10.1145/2674396.2674406

Oliver Korn, Markus Funk, and Albrecht Schmidt. 2015a. Assistive systems for the workplace: Towards context-aware assistance. In *Assistive Technologies for Physical and Cognitive Disabilities*, Lau Bee Theng (Ed.). IGI Global, 121–133.

Oliver Korn, Adrian Rees, and Uwe Schulz. 2015b. Small-scale cross media productions: A case study of a documentary game. In *Proceedings of the ACM International Conference on Interactive Experiences for TV and Online Video (TVX'15)*. New York, NY, USA: ACM, 149–154. DOI:https://doi.org/10.1145/2745197.2755516

Benoit B. Mandelbrot. 1982. *The Fractal Geometry of Nature*, San Francisco: W.H. Freeman.

Marc Olano, John C. Hart, Wolfgang Heidrich, Bill Mark, and Ken Perlin. 2003. Real-time shading languages. *SIGGRAPH 2002 Course 36 Notes* (March 2003).

C. Pedersen, J. Togelius, and G. N. Yannakakis. 2010. Modeling player experience for content creation. *IEEE Trans. Comput. Intell. AI Games* 2, 1 (March 2010), 54–67. DOI:https://doi.org/10.1109/TCIAIG.2010.2043950

Ken Perlin. 1985. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'85)*. New York, NY, USA: ACM, 287–296. DOI:https://doi.org/10.1145/325334.325247

Ken Perlin. 1999. Making noise. (September 1999).

Markus Persson. 2011. Terrain generation, Part 1. *World Notch* (March 2011).

Jesse Schell. 2015. *The Art of Game Design: A Book of Lenses* Second edition., Boca Raton: CRC Press.

Noor Shaker, Julian Togelius, and Mark J. Nelson. 2016. Fractals, noise and agents with applications to landscapes. In *Procedural Content Generation in Games*. Computational Synthesis and Creative Systems. Springer International Publishing, 57–72. DOI:https://doi.org/10.1007/978-3-319-42716-4_4

Gillian Margaret Smith. 2012. *Expressive Design Tools: Procedural Content Generation for Game Designers*. Santa Cruz, USA: University of California.

David Thue and Vadim Bulitko. 2012. Procedural game adaptation: Framing experience management as changing an mdp. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Julian Togelius, Tróndur Justinussen, and Anders Hartzen. 2012. Compositional procedural content generation. In *Proceedings of the Third Workshop on Procedural Content Generation in Games (PCG'12)*. New York, NY, USA: ACM, 16:1–16:4. DOI:https://doi.org/10.1145/2538528.2538541

Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis. 2011. What is procedural content generation?: mario on the borderline. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games (PCGames'11)*. New York, NY, USA: ACM, 3:1–3:6. DOI:https://doi.org/10.1145/2000919.2000922

G. N. Yannakakis and J. Togelius. 2011. Experience-driven procedural content generation. *IEEE Trans. Affect. Comput.* 2, 3 (July 2011), 147–161. DOI:https://doi.org/10.1109/T-AFFC.2011.6

Günter M. Ziegler. 2013. *Mathematik - Das Ist Doch Keine Kunst*, Munich, Germany: Knaus.

Received March 2016; revised May 2016; accepted July 2016