Multimodal Technologies and Interaction

MDPI

Article

A Survey on the Procedural Generation of Virtual Worlds

Jonas Freiknecht * and Wolfgang Effelsberg *

Department of Computer Science IV, University of Mannheim, 68131 Mannheim, Germany

* Correspondence: jfreikne@mail.uni-mannheim.de (J.F.); effelsberg@informatik.uni-mannheim.de (W.E.)

Received: 30 August 2017; Accepted: 20 October 2017; Published: 30 October 2017

Abstract: This survey presents algorithms for the automatic generation of content for virtual worlds, in particular for games. After a definition of the term *procedural content generation*, the algorithms to generate realistic objects such as landscapes and vegetation, road networks, buildings, living beings and stories are introduced in detail. In our discussion, we emphasize a good compromise between the realism of the objects and the performance of the algorithms. The survey also assesses each generated object type in terms of its applicability in games and simulations of virtual worlds.

Keywords: virtual worlds; procedural content generation; multimedia content creation; serious games

1. Introduction

In the context of games and simulations, virtual worlds gain more and more importance. For example, when we look at Bethesda's *Skyrim* [1], one can see that the complexity and effort to create a credible and realistic 3D world is a task of several months for a large team of professional designers. However, it is not only the overall visual quality that plays an important role during the design process but also the sheer amount of different objects necessary to avoid a repetitive look where items like furniture, plants or buildings frequently reoccur [2]. As a result, game development studios are facing two challenges:

- to create a realistic and credible environment in an adequate period of time,
- to keep the project in budget to make the result affordable for the end user.

The famous game designer Will Wright (*The Sims* [3], *Sim City* [4]) has proven that these challenges not only exist in theory but also in practice. He named it *The Mountain of Content Problem* [5]. *Procedural generation* is an approach to face these issues. For years, developers have created methods and patterns to generate textures, 3D models or even entire game levels by applying algorithms to achieve a unique-looking environment that needs no or very little adjustment. Another reason for the growing interest in procedural content is to make a game worth playing again by changing levels or quests and hence offering new stories and impressions in each new session. The game mechanisms remain the same so that the player can make use of acquired skills and abilities. Examples for games that make use of these possibilities are *Elite: Dangerous* [6] or *Minecraft* [7], creating new worlds at the beginning of each game if the player wishes. The principle of procedural content generation (PCG) does not serve exclusively the domain of digital games. Board games like *The Settlers of Catan* [8] require the players to create a world map by distributing raw material fields randomly on the table [9], offering an entirely new experience to the player in each game. This approach has led to an enormous success of the game.

Furthermore, PCG has had an influence on the creation of *serious games*, focusing on a learning experience in addition to entertainment as in classic games [10]. In the area of education, changing environments have the effect that students not only repeat knowledge or motion sequences

they have learned but also apply their abilities to new situations. These situations can be randomly created using procedural generation mechanisms. Taking a virtual driving school as an example, generated road networks might teach the player to orientate himself in the real world in an unknown area. Other scenarios for randomly generated environments can be derived easily for serious games in diverse industries such as transportation, health and marketing.

This survey provides an overview of current research in the area of procedural content generation. It focuses on the creation of realistic modern virtual environments. After a short historical introduction and a definition of the term, each category of objects that form a virtual world will be presented and discussed regarding progress, usability and level of detail in research and application. The survey ends with a tabular comparison and evaluation of these classes of objects in terms of research status and practical application. We conclude with an outlook, also discussing features that can be expected in next-generation game engines and game development tools. The procedural generation of sounds and music is not part of this survey.

2. History of PCG in Digital Games

Already, in 1978, Don D. Worth used simple algorithms in his game *Beneath Apple Manor* [11] to create dungeons for this RPG (Role Playing Game) [12]. A more thrilling sensation was caused by the game *Rogue* [13], which also used algorithms to create levels for this very famous dungeon crawler [14]. The game came out in 1980. The name *Rogue* served as an eponym for the following generation of Rogue-like games that had the following characteristics:

- turn-based gameplay,
- procedurally generated levels,
- permanent death (no load/save functionality).

Other than expected, the developers neither chose the generative approach for the level creation—in the interest of making the game worth playing again—nor did they face the *Mountain of Content Problem*. In fact, in these days, the memory requirements of a game with many different levels were too high so that the decision was made to generate the levels on the fly instead of writing them to disk [15].

From a research perspective, Darwin R. Peachey's paper Solid Texturing, published in 1985, was one of the first publications explicitly discussing procedural content generation [16]. Similar to today's *normal mapping*, Peachey proposed a technique that enabled two-dimensional textures to look three-dimensional. This early paper was followed by various other publications discussing e.g., terrain generation [17] or design and animation of plants making use of fractals [18] and hence established procedural content generation as a scientific research area.

The derived methods were not only used in the digital game industry but also in the animation movie sector. For instance, Pixar's animation tool *RenderMan* [19] offered procedural functions to define textures and materials algorithmically, or to generate a variety of simple primitives by handwritten subroutines. With a growing maturity in research and practice, the generative procedures were used more and more in AAA games (AAA game is a widely used term for games with a high development and promotion budget.) such as *Command and Conquer: Red Alert 2* (2000) [20], *Diablo* (1996) [21] or *The Elder Scrolls II: Daggerfall* (1996) [22]. In those games, developers deliberately implemented PCG as a concrete game element, not because of a resource bottleneck as in former days. In 2000, EA Games (Redwood City, CA, USA) published Maxis' *The Sims* [3], introducing not only a completely new gameplay concept but also a highly innovative yet simple to use character editor. Players were able to create an in-game character in the form of a detailed 3D model composed of customizable body parts, face, clothing and personality. Using the editor did not mean to move single vertices and adapt a UV map (A UV map is the projection of a 3D model's surface to a 2D image.), as it is done in modeling tools like *Blender* [23]; Maxis designed the editor to offer a set of parameters to edit single characteristics of a human to e.g., adapt the waist, size or the interocular distance. Reasonable limits

supported the players while designing their characters to guarantee a credible and believable look of the result. Many 3D tools implemented a similar workflow to create humanoid models, like Autodesk's (San Rafael, CA, USA) *Character Generator* (formerly known as Autodesk Pinocchio) [24], *Fuse* [25] or the open source application *Makehuman* [26]. We consider this approach to be procedural content generation with a strong user interaction. One year later, in 2004, the group .theprodukkt published a simple shooter called *.kkrieger* [27], having its origin in the demo scene (see Figure 1). This scene's goal is to create visually or aurally impressive applications, being often denoted as digital art since they show simple, moving scenes—similar to a painting [28].

Figure 1. The game .kkrieger has a file size of 96 kb.

A particular category focuses on demos with a maximum file size of 4, 8 or 64 kb. This small footprint does not allow the usage of prefabricated assets such as graphics, music or models. Instead, those assets are created procedurally during runtime or before the demo starts. Keeping in mind that rogue-like games were also determined by memory, one can definitely find certain parallels here. Taking *.kkrieger* as an example, the development team .theprodukkt generated animations, levels, textures, shaders and music based on a tool set offering procedural methods. With a great interest in the final demo (which had a size of 96 kb), the developers decided to publish their tools in the form of the editor *.werkkzeug* [29].

Compton, Osborn and Mateas determine Computer Graphics as the origin of procedural generation and state that the discipline was connected to computer games with a publication of Intel (Santa Clara, CA, USA) [30]. This particular paper surveyed generative techniques such as L-systems, Perlin noise or fractals [31].

Smith dedicates an entire article to the history of PCG and speaks of a development from modularity in design to algorithmic assembly of content. She furthermore summarizes the motivations to use PCG such as replayability, PCG as an expressive medium or as assisting creative technology [32]. Nowadays, most of the discussed utilities have arrived in modern game development tools and make procedural content generation easier. 3D models are constructed by deformation, cropping or merging primitives instead of assembling them triangle by triangle (e.g., in *ZBrush* [33]). Thus, developers can focus on the procedural algorithms to generate content instead of tackling the technical challenges some tiers below. Examples include the game engines *Unreal Engine 4* [34], *Unity 2017* [35] and *Cry Engine V* [36] provide tools to simply manipulate terrains and *paint* vegetation like trees or flowers onto the ground (see Figure 2) and thus speed up the level creation workflow impressively. Hence, developers can lay their focus on finding an algorithm to distribute vegetation objects over the level without taking care of colliding flowers or repetitively formed trees.

A remarkable achievement of today's game development tools is the procedural generation of textures and materials. Similar to what *.theprodukkt* introduced in *.werkkzeug* some years ago, the procedural generation of textures and materials is now applied by most of the big players in the game engine market. In addition, instead of writing shaders by hand (in GLSL or HLSL), modular tools were offered to graphic designers to construct materials by combining images, mathematical operations or shader functions to e.g., create a normal mapping effect as shown in Figure 3. The next game engine

generation has begun to implement features to create even more complex objects. *Epic*, for instance, included a simple tool to generate 3D buildings [37].

Some very specific applications like Side Effects Software's (Toronto, ON, Canadian) *Houdini* [38] or the *Esri CityEngine* [39] have an even longer experience in the area of building generation, but, in the end, they still create non-random worlds without a lively simulated population.

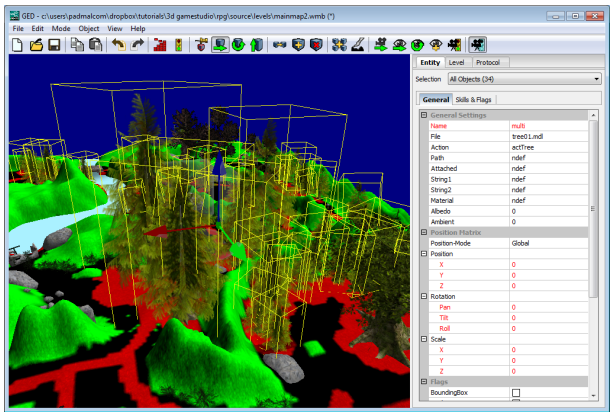


Figure 2. Modern editors offer functions to place objects by using brushes or color maps allowing an easy integration of procedural methods.

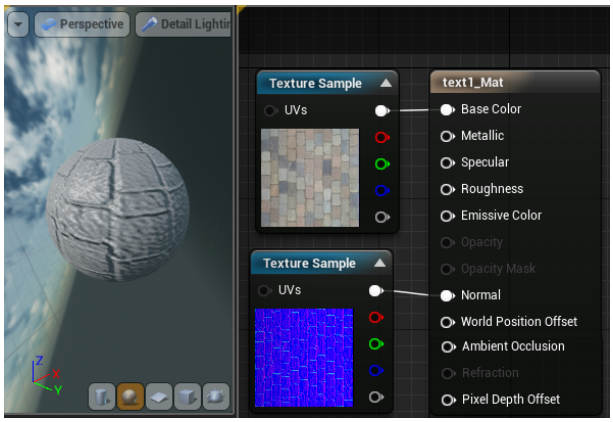


Figure 3. The Unreal Engine's (version 4, Epic Games, Inc., Cary, NA, USA) material editor allows for concatenating mathematical operations, textures and shader functions.

3. Definition of PCG

Ruben M. Smelik et al. define procedural content generation as "any kind of automatically generated asset based on a limited set of user-defined input parameters" [40]. They furthermore refer to Roden and Parberry [41] who call these kinds of algorithms *amplification algorithms*, taking a small set of input parameters to transform them to a larger set of output data. Togelius formulates a definition by an antithesis, saying that procedurally generated content does not correspond to content that is generated by users even if they make use of procedural algorithms since they have to be manually parameterized [42]. Hendrikx et al. see procedural generation as an alternative to manual design but stress the need for a possible parameterization so that designers can take an influence on the generated object [43]. Shaker et al. are more concrete and define PCG by giving examples what PCG is (e.g., a software tool to generate random dungeons without any user input) and what it is not (a map editor that lets users place items) [44]. At this point, we would like to give our own definition of PCG:

> Procedural content generation is the automatic creation of digital assets for games, simulations or movies based on predefined algorithms and patterns that require a minimal user input.

PCG is not only a subject of research in computer science. Prusinkiewicz and Lindenmayer emphasize the growing interests in other communities caused by the interdisciplinarity, affecting natural sciences like biology [45]. This strong interest in other fields of research is an indicator for the topic's presence. However, harmonizing all those disciplines like biology, architecture, urban studies, psychology, etc. and finding the right formalisms and data structures is a huge effort. Finkenzeller [46] narrows the affected areas of computer science to:

- grammars,
- L-Systems,
- shape grammars,
- programming languages.

He furthermore points out that programming is the most flexible but yet error-prone method to automatically generate procedural content. Hendrikx et al. introduce the abbreviation PCG-G (procedural content generation for games) [43] in order to delimitate PCG for games from further fields such as simulations or (animated) movies. This shows that PCG-based procedures, algorithms and tools can be applied to a large variety of fields like urban planning (e.g., the Esri CityEngine) or to the (animation) movie industry. Pixar has already been named as the company making use of procedural content generation in *RenderMan* [47]; additionally, Disney Research (https://www.disneyresearch.com/) provides publications mentioning procedural techniques, e.g., for virtual terrain editing [48], showing that there is an awareness of the advantages of automated content generation in the movie industry.

3.1. Theoretical Considerations

In computer science, the efficiency and maturity of software or algorithms are frequently measured to assure their quality and applicability. A widely used metric is a simple subjective estimation to what degree the generated content looks realistic; this is not the case when it can be easily identified by a human observer as automatically generated. We propose balancing between performance and fidelity in PCG.

When an algorithm returns a precise result (e.g., a natural-looking forest), the algorithm requires more processing power, more memory or more storage to generate more variations of trees, textures in higher resolution, more detailed meshes or a denser planting. Depending on the desired outcome, the user has to choose between either performance or realism to reach the optimum for the given system or the requirements for the virtual world. Often, the generated objects can either be categorized as handcrafted or as made by nature. A central question is if one could tell if either one or the other category can be created with lesser effort than the other. As an example, the complexity of the tasks to generate 3D trees can be compared to the one to generate 3D buildings. When producing much of the visible content automatically, the order of its generation becomes relevant. Therefore, we compare a highly simplified projection of the natural creation of the earth (see Figure 4b) to the procedural generation of a virtual world (see Figure 4a). In many PCG developers' conception of natural creation of a virtual world, the landscape (including water) on a planet serves as the basis of a world, followed by vegetation such as trees and plants. Later on, humans construct buildings on this landscape and connect them via road networks. Then, a settlement grows or dies over the years. If there is vegetation or mountains in the way, mankind tends to remove them in order to build roads or buildings.

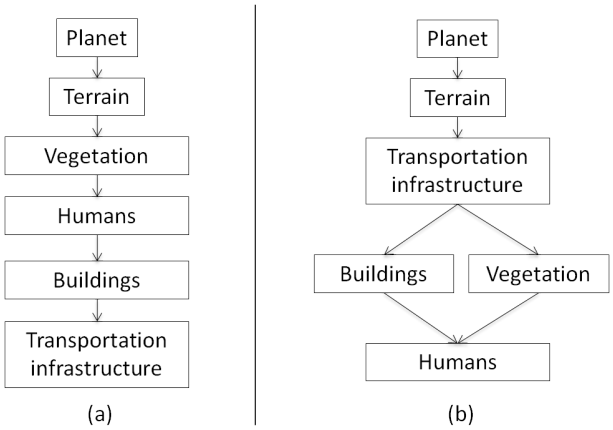


Figure 4. The construction of a virtual world can either happen in a simplified natural order (**a**) or in an optimized procedural one (**b**).

Mapping PCG of a virtual world to the simplified nature-like creation of the earth results in some extra iterations in which forests are removed to build roads, a terrain is flattened to place cities on it, or rivers get re-routed to let a city grow in the desired direction. As an alternative, we propose to use the order presented in Figure 4b where the terrain is created followed by a transport infrastructure, followed by buildings and vegetation. This proceeding will most likely not comply with the idea of natural growth, but it can ease the computer-aided generation of virtual worlds.

3.2. Random Number Generators

Random Number Generators (RNG) can either be a hardware device or a software [49], producing a deterministic and periodic sequence of (pseudo) random numbers [50]. Their existence and functionality is assumed in many publications dealing with PCG [32]. Despite the fact that not all researchers agree with their omnipresence in PCG. Some point out that pure random generation would result in chaos [42]. We think that the application of randomness depends on the context of each generation process. Taking the road network in Section 4.2.1 as an example, the placement of streets and intersections can occur randomly, but the algorithm to generate the streets meshes and textures has to be adequately implemented upfront.

4. Classification

We now provide a classification for the types of objects that are most frequently the subject of procedural content generation (see Figure 5). We took the *CityGML* specification (especially the *CityGML Core* schema) as the basis and added living beings to it to fit the needs to describe objects in a procedurally generated world [51]. In the remaining chapter, we will present vegetation, water, road networks, buildings, creatures, humans and stories as typical examples for these classes.

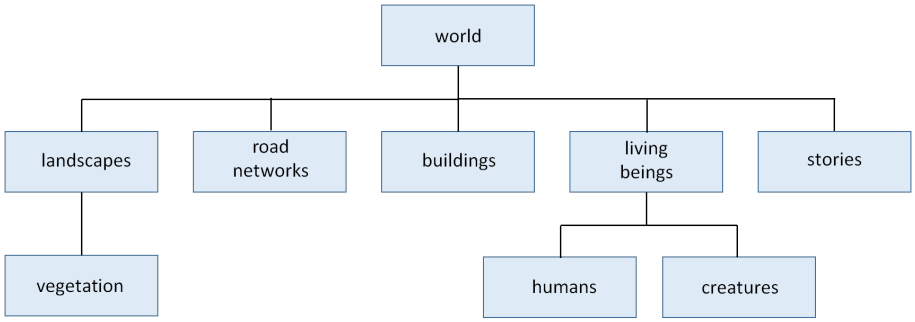


Figure 5. Classes of procedurally creatable objects discussed in this survey.

4.1. Vegetation and Landscape

The procedural creation of objects occurring in nature, like terrains or plants, belongs to the most explored areas in procedural content generation. Theories and software tools have existed for many years, and they have reached a high level of maturity.

4.1.1. Generation of Vegetation

Looking at plants in particular, one of the reasons for the strong interest in their structure comes from theoretical biology: inspired by their beauty, researchers have tried to find mathematical models for their growth. The Hungarian researchers Prusinkiewicz and Lindenmayer were pioneers in this area. As early as in 1968, they proposed a grammar called *L-Systems* (Lindenmayer Systems) to describe the structure of plants with mathematical methods.

Since we are interested in the graphical representation of trees and plants, we have to find a mapping of L-Systems to graphics. This interpretation is often called the *turtle model* as it is the basis of the language LOGO and the turtle used there. Our system consists of a two-dimensional grid and the following grammar:

F move one step forward, drawing a line,
f move one step forward without drawing a line,
\+ turn right by δ degrees,
\- turn left by δ degrees.

An example is shown in Figure 6.

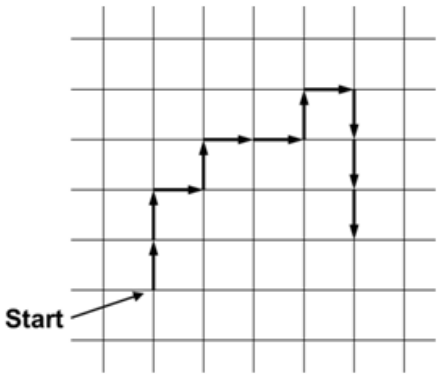


Figure 6. A structure produced with the Turtle model, position (1,1) is the starting point, initial direction upwards and $\delta = 90°$. The structure corresponds to the word FF-F+F-FF+F-F-FFF.

Now, we introduce re-writing, the fundamental idea of L-Systems. A *re-writing rule* or *production rule* defines that the left side of the production can be replaced by the right side, and that replacement can be repeated as often as necessary. If we take characters as the elements of a language of words, a set of re-writing rules might be the following:

$$a \rightarrow ab,$$
$$b \rightarrow a.$$

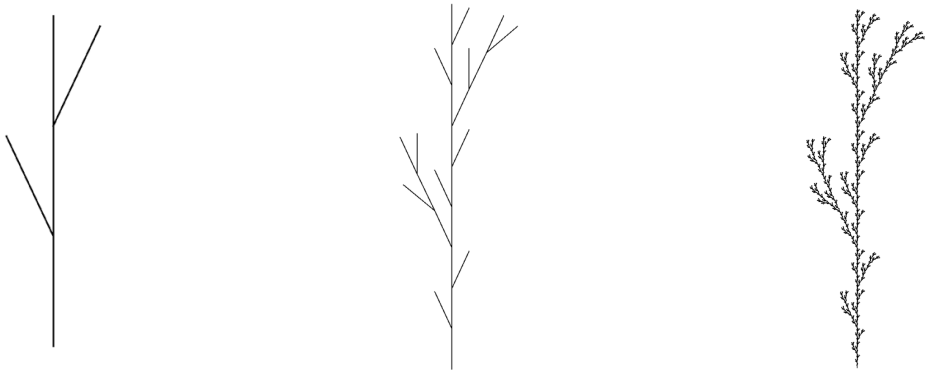
The language created by our grammar consists of all the words that can be created out of an initial character string and our re-writing rules. The attentive reader might notice that this principle was originally introduced by Chomsky to describe programming languages [52]. In contrast to Chomsky's languages, L-Systems require every re-writing rule to be applied in every round. The reason is that the growth of plants is based on cell division, and this happens in parallel for all cells. If we take our example from above with 'a' as the initial string, we can generate the following words in our language:

a
ab
aba
abaab
abaababa.
...

In an L-System, this production process describes how a plant grows. In a more common notation, we mark possible replacements with brackets around the corresponding substring. For example, we could define an L-System with re-writing as follows (n is the number of applications of the re-writing rule, 'F' in the second line is the initial string):

$$\text{n} = 1, \delta = 25^\circ$$
$$\text{F}$$
$$\text{F}\rightarrow\text{F[+F]F[-F]F.}$$

For a graphical representation, we again use the turtle model. A '[' is interpreted as a push-down on a stack, a ']' as a pop from the stack. The above L-System then produces the plant shown in Figure 7a.



(**a**) (**b**) (**c**)

Figure 7. Simple plants generated with an L-System with brackets and different re-writing rule iterations. (**a**) one iteration; (**b**) two iterations; (**c**) five iterations.

We now apply the re-writing rule twice, i.e., we change our production system as follows:

$$\text{n} = 2, \delta = 25^\circ$$
$$\text{F}$$
$$\text{F}\rightarrow\text{F[+F]F[-F]F.}$$

We then get the plant shown in Figure 7b. If we use $n = 5$, we get the realistic plant shown in Figure 7c. If we use the deterministic L-Systems, we have derived so far to produce a large number of plants for our virtual world that all look the same. This seems to be unnatural. Thus, we need to introduce a stochastic component: each step in the construction of a plant is taken with a specified probability. Let us thus introduce probabilities into the L-System from above:

$$\text{n} = 5, \delta = 25^\circ$$
$$\text{F}$$
$$p_1 = 0.33\text{: F}\rightarrow\text{F[+F]F[-F]F}$$
$$p_2 = 0.33\text{: F}\rightarrow\text{F[+F]F}$$
$$p_3 = 0.34\text{: F}\rightarrow\text{F[-F]F.}$$

The p_i are the probabilities for the productions; they add up to 1. This stochastic L-System might produce the plants shown in Figure 8. Note that they all seem to be of the same species, just variations at different levels of growth.

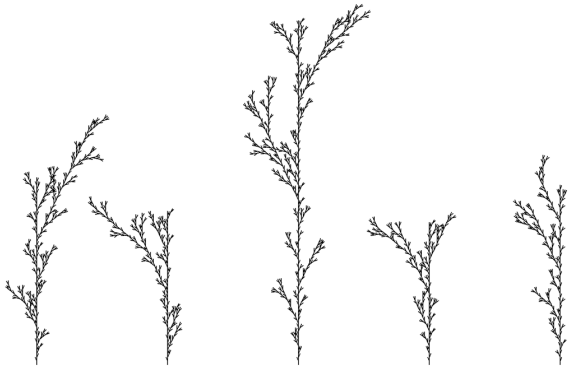


Figure 8. A number of plants generated with the stochastic L-System above.

Thus far, we have described how 2D plants can be generated. It is easy to extend our L-System grammar to 3D plants: we simply add operations to 'pitch down', 'pitch up', 'roll left' and 'roll right' to the initial 'turn right, turn left' operations at every decision point. This allows us to describe a large variety of 3D plants with short rules. The reader can imagine that it is also easily possible to extend L-Systems with different widths and colors for the different branches. Similarly, L-Systems were adapted to bushes and other types of plants. Many papers were published addressing the details e.g., [45,53–57]. Until today, variations of the early L-Systems are the most widely used methods for the procedural generation of plants.

Besides pure L-Systems, there exist various other systems. For example, Chen et al. propose a sketch-based tree modeling system that takes advantage of a tree database for the 3D layout [58]. The user sketches the basic branch structure of the desired tree and optionally the contour of the crown with a few strokes in 2D. The system then looks up matching 3D tree structures in a database of 20 tree models. The best match is extracted and used to generate the desired tree. A similar approach is proposed in [59]. In related work, other authors propose using photographs of trees to find the appropriate model in the database [60–62].

The self-similarity of plants is also often used to automatically generate them. The *Mandelbrot set* [63], the *Koch snowflake* or the *Pythagoras tree* are examples of mathematical models that can be visualized as shapes resembling natural structures (see Figure 9). When we take a look at the practical application of methods for tree and plant generation, one can see that the most commonly used tools *Xfrog* [64] or *Speedtree* [65] reach impressive results.

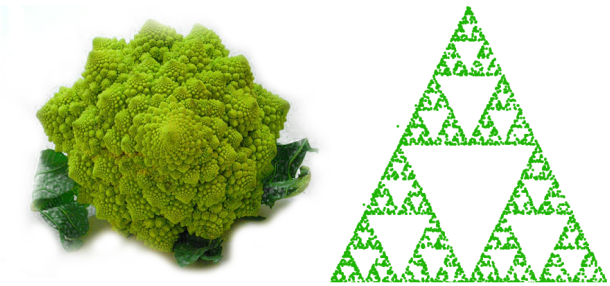


Figure 9. Structural similarities between a Romanesco and the Sierpinski triangle (source: fourmilab.ch).

In [66], the authors propose a procedural branch graph (PBG) approach that creates diverse trees with the same branch structure at different LODs (Level of Detail). Further state-of-the-art research such as [58,67,68] confirms the high maturity level of this research area.

4.1.2. Landscapes

Like the procedural generation of vegetation, the generation of landscapes in the form of height maps belongs to the more advanced topics in PCG. In most cases, height maps consist of grayscale bitmaps in which the elevation is represented by the shade of grey of the bitmap's pixel (see Figure 10a). The calculated height is then projected to a flat 3D mesh [69]. Usually, the whiter the pixels are, the higher is the elevation. After the mesh has been created, the terrain needs to be colored. For this task, there are three established techniques:

- manually drawing a texture,
- using a manually created color map to project textures to specific regions,
- generating a texture by analyzing slopes and heights of the terrain's mesh.

These three approaches differ in quality and applicability. The process of manually drawing a texture is simple, but it results in a huge bitmap or an insufficient resolution. Drawing a color map (see Figure 10b) that is used in the game or simulation to map a set of textures (see Figure 10c) to specific colors (in general by making use of a shader) is a common practice and returns visually impressive results (see Figure 10d) but is still a manual process.

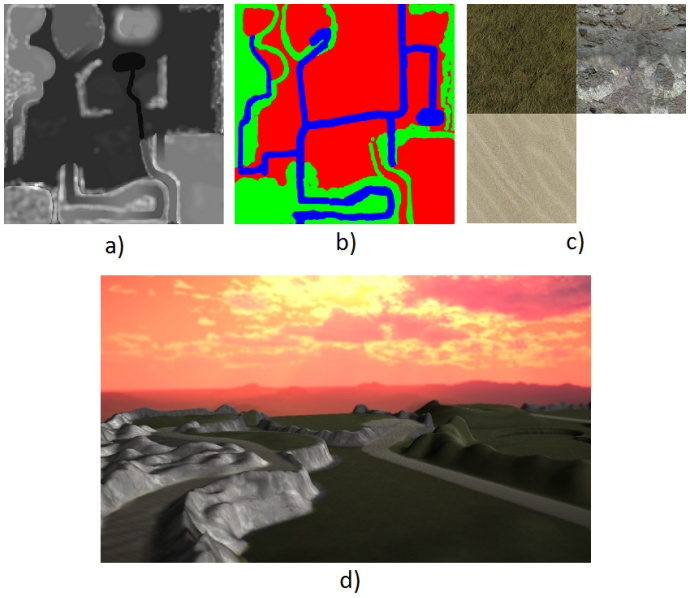


Figure 10. Creation of a terrain using a height and color map, (**a**) a height map , (**b**) a color map, (**c**) textures, (**d**) terrain.

The third method makes use of the mesh data and utilizes heights and slopes to calculate the appropriate texture mapping. Very low areas are typically seen as ocean and are hence textured with an ocean ground texture, average heights are textured using grass, areas with a high elevation are seen as mountains and receive a rock texture, and very high regions receive a snow texture. Slopes can be used to identify steep areas that are frequently represented by stone (e.g., cliffs) and hence receive a stone texture. Since the first two methods require the designer to create either the color map or the texture [70], they are not considered to be applicable for a purely procedural approach. In contrast, the third procedure fits well since it relies completely on the terrain's mesh. In recent games such as Minecraft [7], biomes come into play and contain climate information like humidity and temperature [71]. Biomes are regions of land, and, depending on their characteristics, the terrain is formed regarding elevation and textures. Adjacent biomes can either be blended or have abrupt borders depending on design decisions. A question that arises again is how to procedurally generate a height map. There exist various techniques to do that:

- fractal noise, e.g., Perlin Noise [72],
- midpoint displacement (mostly used to generate 2D landscapes) [73,74],
- diamond-square algorithm (adapts midpoint displacement to generate 3D terrains) [75].

These are only three examples for a huge variety of algorithms. For a practical implementation, there exist some open libraries like *libnoise*, containing modules to produce noise or other patterns like checkerboard or *Voronoi diagrams* [19]. The latter can be used to achieve a non-homogeneous appearance of a terrain shape, in contrast to noise algorithms [76]. Keeping in mind that a virtual world requires millions of square meters, it is indispensable to agree on a way to reach the optimal performance for the rendering process. The use of LODs is recommended [77]. It should be used to add more terrain detail in important and frequently visited areas of a terrain that are close to the user, and to reduce detail in less important regions, e.g., in far mountain areas. As an enhancement to a static solution, a real-time optimization algorithm called *Real-time Optimally Adapting Meshes* (ROAM) is proposed in [78], optimizing the mesh's triangulation during runtime depending on the player's view frustum. Lee, Jeong and Kim created a maze terrain authoring system (MAVE) to calculate a finite maze terrain based on different maze patterns [79].

A limitation to common algorithms for terrain generation is the creation of caves [69] or overhangs, which can either be addressed by layered terrains [80] or by voxel terrains [81]. Cui et al. not only propose a technique to create caves with different characteristics but also how to store their data efficiently in an octree data structure [82]. Boggus and Crawfis make use of 3D models to generate pattern based caves using prefabricated pattern images [83].

4.1.3. Placing Vegetation in a Landscape

Once a set of plants and trees has been created, the question of proper placement in the virtual landscape has to be answered. In general, there should be a differentiation between placing plants in large numbers in a given area and an individual placement [54]. Similar to the height maps described above, a technique based on a grey-scale image can be used in which the density of the vegetation is defined by the shade of grey. Similar to the approach of texturing a terrain by analyzing heights and slopes, Hammes [84] proposes a procedure to place plants and trees based on the grayscale of the terrain beneath. Another idea is to use color maps where each color stands for a type of vegetation [70]. An interesting alternative was presented by Alsweis and Deussen in 2006 [85]. They propose to model the natural resources available for the plants and the competition between them to determine their density. In the *FON model* (Field of Neighborhood, see [86]), each plant has a circular zone of influence on the neighboring plants. The size of that zone can depend, for example, on the humidity of the ground, the fertility of the soil and the type and size of the plant. They compose a landscape of tiles (the *Wang tiles* [87]), each representing a specified density of the plants according to the FON model. Transitions between the tiles are smoothened by relaxation methods. The property of the ground and thus the Wang tiles chosen depend on the elevation, nearby water, etc. In this way, they can produce very realistic areas of vegetation with different densities automatically.

An integrated system for modeling terrains and plants is described by Deussen et al. in [54]. They describe an entire toolkit for the process. It allows both the manual editing of height maps and plants and their procedural generation. An interesting idea is to provide an initial distribution of plants manually and then model their growth and death algorithmically over some time, taking plant competition into account; the final result is then represented graphically. Another idea is to reduce the geometric complexity of the scene by *approximate instancing*, replacing similar plants, groups of plants and parts of plants by representative objects before rendering. A number of impressive examples shows the realism of their approach. *Poisson distribution* is another approach for placing plants. Here, a probable number of plants is distributed in a partial area of a grid. *Poisson disk distribution* avoids plants growing too close to each other by defining a outer radius in which no other distribution point can be placed [88].

4.1.4. Water

Although water as an element is always the same, the creation of rivers, oceans, lakes and waterfalls differs in many ways. Where oceans and lakes are more or less calm, rivers and waterfalls move constantly. The creation of rivers is often discussed in two ways: either they are generated during the creation of the terrain, or river courses are placed later in the landscape in a separate step [40]. Another option is to refer to the sea level and assume the presence of water everywhere in the virtual world beneath this predefined height [84]. In contrast, Kahoun proposes a procedure of natural growth by the spreading of the flow of water [77]. This flow then iteratively forms the river courses. Ebert refers to the use of *dilation symmetry* to achieve realistically looking rivers where each smaller river branch looks exactly like the larger branch on a smaller scale [19]. The procedural creation of seas and rivers is rarely explored [40], and it focuses mainly on the shape and course of riverbeds [89]. In a few papers, the authors frequently differentiate between a grid-based and a mesh-based approach when creating rivers; the mesh-based approach reaches the more visually impressive results [90] since a 3D mesh is generated individually along a river, whereas the grid-based approach focuses on an existing layout.

Doran and Parberry mention coast line agents [91] to generate realistic island shapes. The creation of coast lines or surf and wave action can be found in frameworks in the form of concrete implementations. The tool *Mystymood*, for instance, generates shore lines, underwater caustics and shore break automatically using a simple collision algorithm and color maps [92].

4.2. Road Networks

This section discusses the efforts made to procedurally create a traffic infrastructure. In this context, the focus lies on the generation of road networks including pedestrian paths. Since air and sea traffic requires only little physical infrastructure, both topics are excluded from our discussion.

4.2.1. Intersecting Streets

The creation of road networks can be done in several ways, for instance, by creating a set of intersections to which the roads are connected [93]. A parameter limits the maximum number of roads connected to one single intersection. During the creation, it makes sense to only allow roads that do not overlay with others (i.e., no bridges). Figure 11 shows an implementation from the game development framework TUST [94] in which road networks are created by placing streets in a 3D environment.

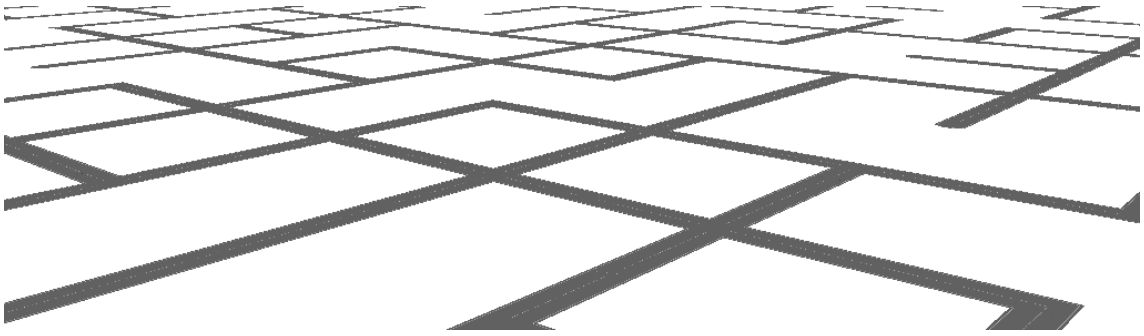


Figure 11. Road network based on junction points.

Figure 11 illustrates the usage of a Manhattan style road network that creates a grid of streets and randomly deletes a parameterized number of streets in the resulting grid. Streets contain at least a start point and an end point. An algorithm detects roads with similar start and end points and creates an intersection there. Each intersection counts the number of connected streets and will later on be replaced with a corresponding prefabricated intersection model with either one, two, three, or four connections. More than four connections form a roundabout. To achieve a realistic but yet simple road

placement, several algorithms can be used. If a street is defined by more than two points and if those points are not positioned on a straight line, then a street mesh is generated by expanding a simple spline to both sides, as can be seen in Figure 12.

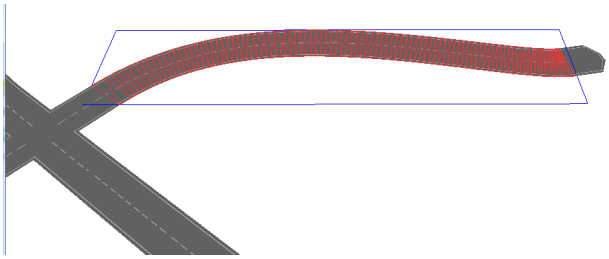


Figure 12. Spline-based road mesh generation.

Pedestrian paths can be attached to streets and/or around a parcel of land. In the case that roads surround such a parcel, sidewalks can be calculated by creating secondary polygons with smaller sizes adjacent to the street polygons [95]. Figure 13 shows a single sidewalk adjacent to a street.

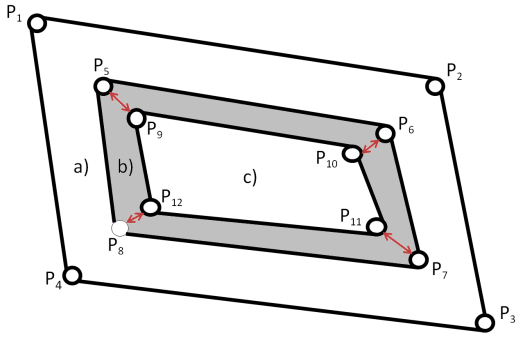


Figure 13. Creation of a smaller polygon within a street parcel results in a sidewalk (**a**) street; (**b**) sidewalk (grey area); (**c**) parcel of land.

We find many patterns in nature, math or computer science that can result in an interesting road network structure, like *Voronoi diagrams*, tree maps, or binary trees (see Figure 14).

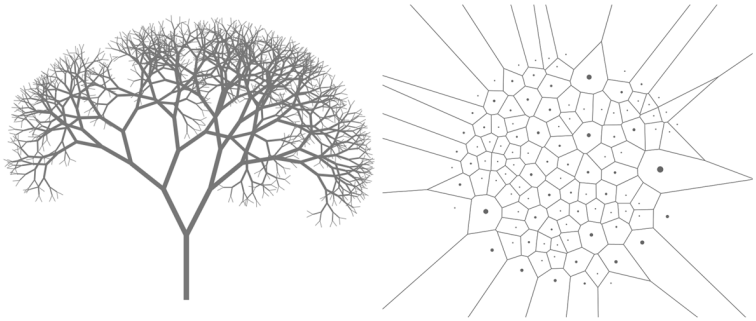


Figure 14. Binary tree (**left**) and *Voronoi diagram* (**right**).

Another approach is to break the road network generation down to a few reoccurring shapes like grids or radials [96]. The authors focus more on the creation of primary and secondary streets where the first ones handle heavy traffic and the latter ones lead to buildings or other facilities. As introduced before, L-Systems play an important role in PCG; they can also be used for road network generation. Extended L-Systems [97] are a hierarchical and adaptable method that allows for modifying the L-System modules during the road generation process.

The authors of [97] point out that their own system is not only applicable to road network generation but also to e.g., buildings. A tile-based system, as shown in Figure 15, receives less attention in current publications; this might be a consequence of the artificial look of the result. Nevertheless, tile-based systems are frequently implemented in games such as *Trackmania Nation* [98], *Ridge Racer* [99], or *Re-Volt* [100]. Furthermore, more complex street forms such as highways exits, non-standard crossroads and interchange roads are not covered in PCG-related publications.

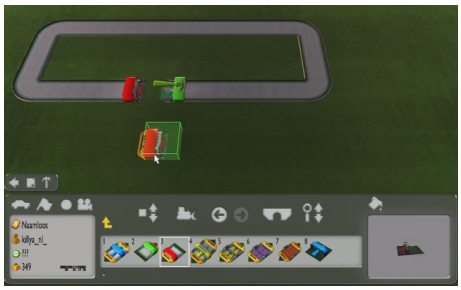


Figure 15. Modular road building system.

Traffic simulation is frequently discussed in the literature [101–103] but is beyond the scope of this survey.

4.3. Buildings

Regarding the construction of virtual buildings, there exists an overwhelming amount of research papers [104–107] but only a few concrete implementations [108,109]. In this context, we mainly discuss two topics:

- room arrangement on a fixed floor,
- shape and facade creation of the outer appearance of the buildings.

The synergy of both—namely the generation of entire accessible buildings—is only vaguely discussed, as well as further steps like the placement of doors or windows or the connection of floors using stairways. Furthermore, the goal to create buildings for a real-life simulation of a virtual world requires another factor: time. Time influences the outer appearance of buildings regarding

- the age (construction time, inhabited, renovation, decay),
- time of day or time of the year (illuminated windows and switched-on outer lights at night or in the Winter, smoking chimney in colder times of the year and open windows in the Summer).

While rendering several buildings in one scene, the Level Of Detail (LOD) plays an important role to manage details of the model. When the distance of the viewer is low, lamps, signs and window sills are visible. Furthermore, the area behind the entrance is supposed to be visible through the windows from outside. If the player enters the house, the model of the interior has to be loaded in full detail; if he/she is three hundred feet away the house can only be visible as a coarse outer shape. An example is shown in Figure 16.

Podevyn describes five different LOD stages within the CityGML (Geography Markup Language) schema [110], which include definitions for entire cities, not only for buildings:

- LOD0: regional, landscape,
- LOD1: city, region,
- LOD2: city districts,
- LOD3: architectural models (exterior), landmarks,
- LOD4: architectural models (including interior features).

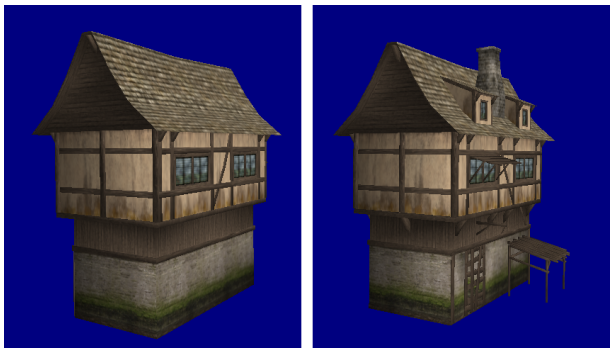


Figure 16. A medieval house model with low (**left**) and high details (**right**). The different appearance for the far-away view is achieved by a mesh reduction algorithm.

4.3.1. Residential Buildings

The general understanding of the term *residential building* does not only include single-family houses but is much more flexible in its interpretation so that row homes, apartment towers, semi-detached houses or student dormitories are included. Our survey discusses only more general algorithms for floor planning and facade creation. Floor planning is seen as a sub-topic of the Spatial Allocation Model and has been present in research since the early 1970s [111]. Mueller et al. attempt to develop a sequential grammar to create building shells [112] by taking arbitrary volumetric shapes as a basis and apply certain rules to them (e.g., split, scope (rotation, position, scale), or repeat). There are several other approaches like the application of a *Squarified Treemap Algorithm* [113] based on the original *Treemap Algorithm* [114], which has been adapted to be used to create rooms in a given rectangular building area. The former method was to display tree structures in a two-dimensional area.

As one can see in Figure 17, the rooms displayed in the squarified treemap (c) have an improved aspect ratio between width and height compared to the treemap (b), which is a result of the *Squarified Treemap Algorithm*. We assume a rectangular floor with a width of 6 and a length of 4 units. This floor is supposed to be split up into seven rooms with a size of 6, 6, 4, 3, 2, 2 and 1 square units. The rooms should be placed so that each room achieves an aspect ratio (width/length), which is as close to 1 as possible to avoid rooms that are very long but narrow. The *Squarified Treemap Algorithm* first determines a start half of the floor. Since the width is larger than the length, the room is placed in the left half (see Figure 18, step 1). Otherwise, it would be placed in the upper half. The aspect ratio of the first room is 8/3 (or 4/1,5). In a second step, the next room is placed above the first (step 2). Its aspect ratio is 3/2, which is nearer to 1 than 8/3, so we continue placing the third room above the second. The aspect ratio of the third room worsens to 4/1 (step 3) so the third room is moved to the right (free) half of the floor. The aspect ratio then improves to 9/4 (or $3/1,\overline{3}$) (step 4). The room also moves when the next room with a size of 3 is placed (step 5). Placing room 5 next to room 4 results in a worse aspect ratio (step 6), and it is hence placed above in the free top right corner (step 7). Rooms 6 and 7 reach the best aspect ratio if placed next to each other (steps 8, 9 and 10).

At first sight, the result lacks practical usability since each room is only directly connected to another room, meaning that there is no corridor. Mirahmadi and Shami propose an improvement to the Squarified Treemap Algorithm, which is able to find a corridor path connecting all individual rooms in the building [115]. Based on a set of rules (bathrooms and bedrooms may not be connected to a kitchen, bedrooms may not be connected to each other, etc.), it is first determined if a corridor is needed. Then, the authors propose selecting all inner edges (or practically spoken, walls) in the floor plan and connect them using a shortest-path algorithm. This path is then used to generate a corridor along the selected edges. By shifting and extending the path, the corridor is generated. Mirahmadi and Shami emphasize that any generated corridor needs intersection points with the rooms to allow the placement of doors.

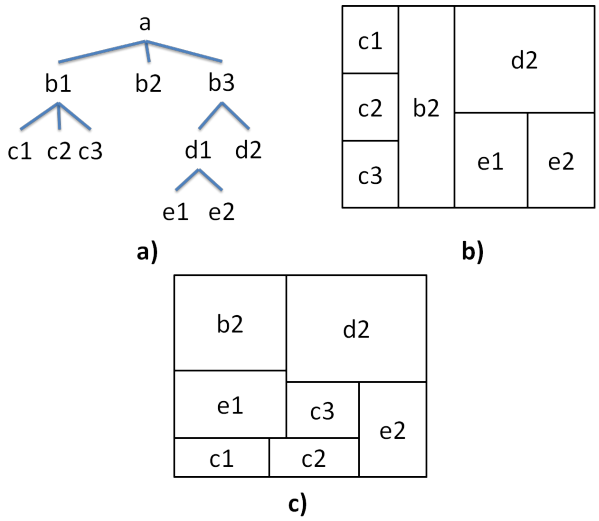


Figure 17. Tree structure (**a**); treemap (**b**) and squarified treemap (**c**).

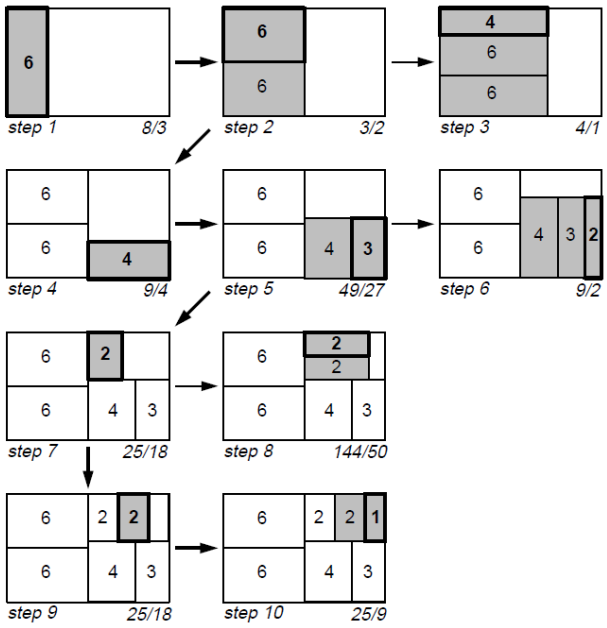


Figure 18. The squarified treemap algorithm [113].

Another known limitation of the treemap algorithm is that it can only handle rectangular areas. This does not hold for another approach, which can literally grow a room on any initial area [116]. The only precondition is that this area is tiled before the algorithm is applied. In the grid of tiles, each room is initially assigned to one cell, which serves as a starting point. One by one, each room grows in one direction in turn until its predefined target size is reached (Figure 19, top left and top right rooms in step e and f). In a second iteration, all cells that have not yet been filled are assigned to a connected room (Figure 19, bottom room, step f).

Due to the fact that there is no minimal cell size, the algorithm can be used to also fill e.g., a circle or a triangle by reducing the cell size to a minimum. The smaller the cell size is, the more accurate is the result for non-rectangular areas.

A possible next step of in the creation of houses would be to fill the created rooms with furniture and accessories. This could be done on the basis of interior design guidelines [117], defining the placement of furniture depending on accessibility, symmetry or adaptability to the room's structure.

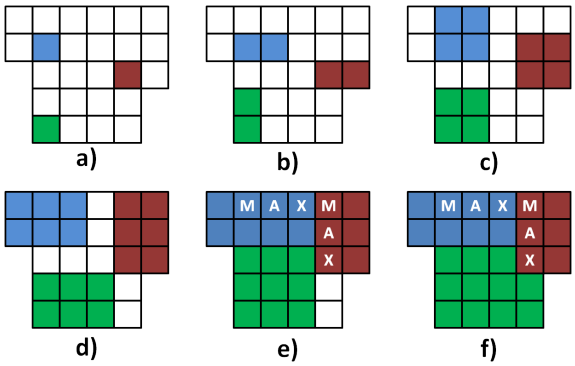


Figure 19. The growth algorithm of [116] is capable of populating even non-rectangular areas with rooms in different stages: (**a**) Initial room start cell placement, (**b**–**e**) growth of rooms, (**f**) connection of remaining cells to adjacent rooms.

We now briefly discuss the creation of facades, which seems to be quite simple at first sight since it only contains the tasks to create a wall and add a texture to it. Finkenzeller might disagree with this point of view; he offers a very comprehensive and competent overview over the diverse techniques to create facades algorithmically, like grammars, L-Systems or shape grammars [46]. Not only facades, windows and doors are part of his work. He rather develops a semantic model for cornices, ornaments and wall structures (e.g., made from baking stone). By that not only the modeling of facades but also the procedural generation of textures for them is addressed—another huge branch in the theory of computer graphics that has an influence on terrain, character and object creation.

Another approach to create believable outer facades was published under the name Split Grammar [118]. It introduces a technique to hierarchically split a wall and label the resulting parts as e.g., door, window or wall. In the next hierarchical layer, these parts can be split again, and they can also be described by grammar to receive more and more detail the more often the splitting is applied. To let virtual buildings appear more realistic and aesthetic, an extensive use of ornaments and decoration is recommended [119].

4.3.2. Other Buildings

Other constructions, like industrial or office buildings, might appear to be similar to the generation of residential buildings. This may be true regarding the basic requirements such as rooms, windows, doors and corridors; even some room types match like office rooms, kitchens and rest rooms. There are mainly three differences: industrial or office buildings have larger rooms, the number of kitchens and toilets has to be higher than in a residential building since there are more people per square foot, and the shape is more regular, making the generation easier [111]. As for residential buildings, shape grammar can be used to generate and place buildings in the virtual world [112]. The details of the procedural generation of other buildings are beyond the scope of this paper.

4.4. Living Beings

The generation of living beings can be divided into two parts. On the one hand, there are humans which are biped and walk upright. On the other hand, there are animals that can either be vertebratae or invertebratae. These classes can once again be broken down to e.g., fish, birds, insects or arachnids. We address those beings as creatures here. Barreto and Roque specify that the PCG of creatures includes the generation of meshes or 3D models, animations, behaviors and sounds [120]. We focus here on the creation of static meshes and skins for humans and creatures.

4.4.1. Humans

The generation of a flexible human model—in contrast to a static one—is characterized by some additional steps, including the generation of a skeleton, the rigging process and the animation [5].

The rigging takes care of assigning bones and joints to a model's limbs. Each level of detail of a model requires a separate rigging. The advantage of using a skeletal animation compared to a vertex animation is obvious: once an animation sequence (walking, running, talking, etc.) has been created, it can simply be assigned to hundreds of models. The motion only differs with age and gender of the human. Minor changes (e.g., stride length, body tension, and corpulence) may provide an individual look and behavior for each human. At this point, one could already guess that there might be some limitations when it comes to the question if a virtual world can entirely be created procedurally. On the one hand, a detailed humanoid model formed by a realistic skeleton, skin tissue and muscles is difficult to create from scratch, and, on the other hand, it is equally difficult to find an algorithm to generate a believable bipedal human motion.

As already mentioned in Section 2, there exist tools to create and customize a generic character model, like Autodesk's *Character Generator* [24], *MakeHuman* [26] (see Figure 20) or *Unity Multipurpose Avatar (UMA)* [121]. The use of a framework acting in the same way might be a good way to quickly create human models, and it is worth further research. To achieve a higher degree of realism, it is not only required to move the extremities according to the real-world but also to reflect the mimic and lip movement during interactions. The latter topic is addressed by the proposal of a text-to-speech engine, which is not only capable to translate written text to audible speech but also to automatically calculate the corresponding lip movements (see below).

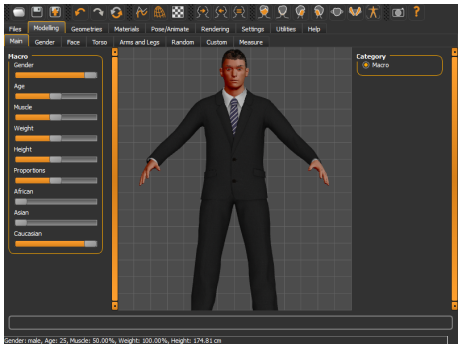


Figure 20. MakeHuman (version 1.1.1) [26] allows for creating individual characters by parameterization of a base model.

4.4.2. Creatures

As already mentioned, research on the generation of creatures is few and far between. An outstanding game called *Spore* [122] was developed by *Maxis*, designed by *Will Wright* and published by *EA Games* in 2008, introducing gameplay featuring the development of a microscopic organism into a highly intelligent and social creature. To create a huge amount of different-looking creatures, designer Will Wright proposed procedural generation as the means to address the generation of thousands of assets that Maxis had during the development of *Sims 2*. Based on the development for *Spore*, Hecker et al. introduced a novel system to animate creatures with an unknown body shape [123] in which generalized motion data could be applied during runtime to achieve an unexpected but realistic-looking animation. Hudson [124] introduced a three-step system:

1. a user defines a set of variables for the creature generation (referred to as *genes*),
2. a tool then translates these genes into a visual model,
3. the model is rigged to have a skeleton ready for animation.

4.4.3. Simulated Motion

The primary aspect of motion simulation in a virtual living environment focuses on the visual impression of life that can be summarized to inhabitants and traffic. Of course, many other simulations

can run simultaneously, like economic development, entertainment (sports, television), weather or evolution of beings. Controlling the behavior of hundreds or thousands of inhabitants belongs to the most challenging tasks. Some papers avoid talking about AI (Artificial Intelligence) in this context [125] since the routines used in games are generally not meant to imitate a realistic human being. Crowd simulation is a separate, well researched topic in the area of simulating large numbers of people. Crowd simulation is the coordinated movement and acting of multiple characters with and within a given environment [126].

Movement in and interactions with a virtual world are equally important. Early open world games—games in which the player can access the entire world from the beginning of the game—like *Grand Theft Auto 3* [127] introduced virtual citizens to make the world look more lively, but, due to the lack of interaction with each other and their missing personality, they appear to be a bit dull [128]. Better approaches of today's games like *Watch Dogs* prefer to give each NPC (Non-Player Character) a personality, a job, and special character traits.

Nevertheless, the basics like moving through a city, paying attention to traffic, using sidewalks, etc. work well in AAA games. They have shown that a simple path-finding algorithm is not enough to let the behavior appear to be realistic. Curtis et al. name four disciplines a crowd simulator has to cover [126]:

- goal selection,
- spatial queries,
- plan computation,
- plan adaptation.

In a continuous process, the goals of each NPC are set (*Goal Selection*), chances of action and movement are determined (*Spatial Queries*), and the path to achieve the selected goal is calculated (*Plan Computation*) and iteratively adapted after each action (*Plan Adaptation*). The resulting simulation can be enhanced by taking two other factors into account, which significantly improve the overall impression:

- credibility,
- possibilities to interact.

The factor *credibility* is seen as a superficial requirement. It can be split into several sub-conditions like personality, emotionality, determination and outer appearance [129]. Some authors make careful steps towards a procedural animation. Horswill describes his motion framework *Twig*, based on physical simulation, to be able to create motions like moving towards a target, or hugging. However, its goal is not to generate realistic animations but just those that make a character seem to be alive [130]. Karim et al. present a locomotion system for multi-legged characters, which is based on an algorithm that places footprints along a path and then calculates the position of the character's feet along the path [131]. The authors stress in their conclusion that the character model—especially the motion apparatus, down to the shape of the feet—has to be very detailed to generate a believable motion.

4.4.4. NPC Interaction

The possibility to interact with other humans, creatures or NPCs is now discussed in detail. An interaction can be broken down to verbal and non-verbal communication (Here, the classic axiom of the communication scientist Paul Watzlawick is broken saying that one could not not communicate; this is actually possible for an NPC if he is not explicitly programmed to do so.). It can include two or more actors. In this context, it is irrelevant if a human player (represented by an avatar) is involved or not. In verbal communication, the dialogue planning and management signify a central challenge [132]. One of the easier and vivid ways to address it is by using Finite State Machines (FSM) [133], known from classical computer science. Figure 21 shows such an FSM in which an NPC

asks the player a simple question, namely *What is 2 times 2?* and lets him pass when the player answers correctly. A set of two pre-defined choices is given—*4* and *other number*. If the answer is *other number*, the NPC will repeat the question. If the answer is *4*, the NPC will tell the player to pass and the dialog ends.

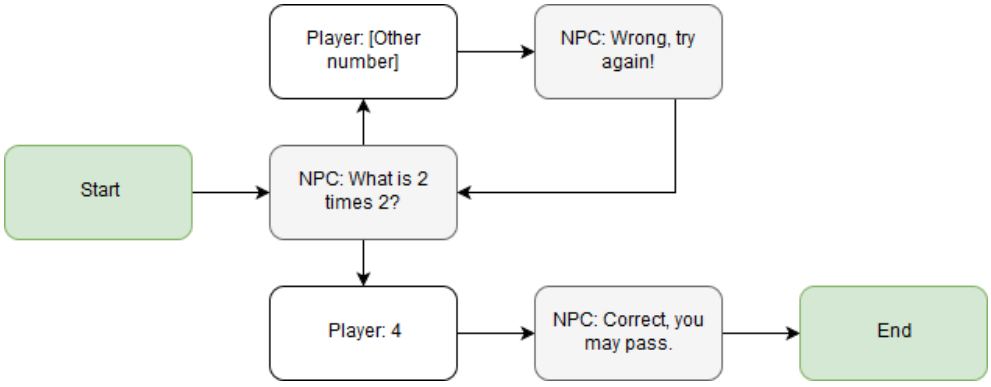


Figure 21. Flow chart illustrating a simple dialog.

This idea can be enhanced by the aspects of personality, relations and moods [134]. Having a look at tools like *articy:draft*, one can see that visual editing of dialogs (and entire story lines) has reached a high level of maturity [135]. Maxis' *The Sims* shows vividly how to simulate several NPCs with a strong focus on the interactional aspects. All characters in the game, including those controlled by the player, have a weighted relation to each other, and they have an individual personality. The limitation lies in the verbal interaction which still (since the first release of the game) takes place using a pseudo language called *Simlish*. Nevertheless, emotions are very well expressed through intonation, mimic and gesticulation, allowing an immediate interpretation of moods (see Figure 22 for the projection of moods on an NPC's model).

Figure 22. An NPC created in Mixamo Fuse (version 1.3, Mixamo, San Francisco, CA, USA) expresses emotions by mimic (neutral, angry and happy).

The Sims is limited to a certain number of actors so that an investigation on how to scale the approach to many interacting NPCs is valuable. The term Level of Detail, as it has been used describing a 3D model's details, can also be applied to the area of AI and actor behavior [136]. In the paper, a procedure is proposed in which the diverse behavior patterns are structured hierarchically. These patterns can be reduced or extended by certain layers, depending on the level of detail. A disadvantage of this approach is the additional cost for modeling the corresponding behavior tree. The author describes a tavern as an example in which the bartender and several guests are simulated at different levels of detail. If the player is absent, the scene in the tavern changes roughly; glasses get empty in an instant, guests do not move from table to table but instantly leave or enter the bar. If the player enters the tavern the simulation level of detail raises, and guests begin to interact with each other, the bartender moves around and cleans tables.

Cassel et al. developed the BEAT toolkit (Behavior Expression Animation Toolkit) to generate nonverbal behavior and synthesized speech for a virtual character based on a typed text. Their extensible rule-based system was derived from actual human conversations [137] (see Figure 23).

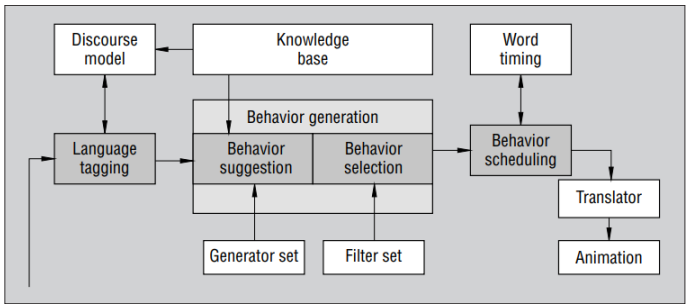


Figure 23. Behavior Expression Animation Toolkit (BEAT) converts text to nonverbal behavior [138].

In [138], Gratch et al. go more into detail and show how BEAT annotates text with hand gestures, eye and eyebrow movement and intonation and schedules these expressions along a speech timeline. De Carolis et al. present a conversational agent, which imitates human-like behavior during a conversation. A so called Mind–Body interface enriches a discourse plan with meaning and allows the projection of the corresponding emotions to the character's body [139]. Bickmore and Picard speak of the usage of *relational agents* to establish a long-term relation between a human (e.g., a player) and a virtual personality (e.g., an NPC) [140]. They place their focus on situations (such as education or business) in which the human interacts with his counterpart and provide insights in their motivation.

4.5. Procedurally Generated Stories

Many games are based on stories. They provide a natural narrative structure, helping the players to understand the mindset and the goals to be reached, establishing the context and motivating the player. This is especially true for roleplaying games (RPGs). The player typically has a main storyline, guiding them to the goal, and additional side stories that increase the fun in playing. Examples for such games are *The Elder Scrolls V: Skyrim* [1] and the *The Legend of Zelda* series [141].

The procedural generation of stories has many advantages [142–144]. First of all, creating stories is expensive; many artists, content designers, programmers and audio engineers are usually involved. This effort can be reduced considerably if PCG for stories is employed [145]. Second, when new story variations are created at runtime, this can keep players motivated when replaying the game [144,146]. Third, it becomes possible to generate stories that automatically adapt to the players' skills and preferences [147].

In narrative games, the notion of a *quest* is widely used. Quests are pieces of the story. A quest has a well defined beginning and ending, and a specified goal must be fulfilled. The main story of a game can then be subdivided into many quests, main quests and side quests. The main story (i.e., the series of main quests) is harder to generate procedurally. The reason is that the creativity of the storyteller cannot yet be replaced entirely by an algorithm using computational creativity (CC) [148]; a good narration and thus exiting quests are an essential part of a good story-driven game. CC is actually seen as a remaining challenge in the field of generating games [149]. Different side quests can be generated more easily. Different paths can then be taken through the game, even dependent on the performance and the personal taste of the player. Thus, we focus on the procedural generation of such side quests.

Any procedural generation must be based on a *model*. This model is always more abstract than the assets created. For example, the model for the generation of plants is often an L-System, and the model for a street system is a set of interconnected points, both with parameters. So, what would be a good model for a story? We look at three examples for such models: a planning language, a Petri net model and StoryTec.

4.5.1. Planning Algorithms

A first approach is to use a *planning language*. Such languages are know from artificial intelligence, for example to describe robot movements. An example is *ADL, the Action Description Language* [150].

It is based on actions that have a pre-condition and a post-condition. The pre-condition has to be fulfilled in order for the action to be performed, and the post-condition reflects its effects. Also allowed are quantified variables, i.e., the $\exists$ and $\forall$ operators. Conditions can be formulated as Boolean formulas with the operators $\wedge$, $\vee$ and $\neg$. Typed variables are supported. ADL assumes an *open world*, i.e., what is not contained in the specification is undefined. An *initial state* is defined as the basis, in our case the beginning of the story, and a *goal state* as the final result of the story.

Let us now look at an example. A person named Bob is supposed to move from his house "Bob's Shack" to the palace "Royal Palace". The house and the palace are both locations. Three alternative storylines allow him to either walk, fly or teleport. The result of our random number generation is available at runtime in the variable *random*. The story can then be described as follows:

Initial State Representation:
Person(bob) $\wedge$ location(bobsShack) $\wedge$ location(royalPalace) $\wedge$ At(bob, bobsShack),

Goal State Representation:
At(bob, royalPalace),

Action Representation:

1. Action(walk(p:person, from:location, to:location)
   Precondition: At(p, from) $\wedge$ random = 1
   Effect: $\neg$At(p, from) $\wedge$ At(p, to)
   ),
2. Action(fly(p:person, from:location, to:location)
   Precondition: At(p, from) $\wedge$ random = 2
   Effect: $\neg$At(p, from) $\wedge$ At(p, to)
   ),
3. Action(telePort(p:person, from:location, to:location)
   Precondition: At(p, from) $\wedge$ random = 3
   Effect: $\neg$At(p, from) $\wedge$ At(p, to)
).

At runtime, this will lead to Bob moving from his shack to the royal palace in one of the three ways, depending on the current value of the random variable. In our example, one of the three paths is chosen at random but other preconditions can also be defined easily, for example, to take the personality of the player or his current performance in the game into account. More details on ADL can be found in the original paper by Pednault [150].

More recent work is presented, e.g., by Riedl and Young [151,152]. The *Intent-Driven Partial Order Causal Link* (IPOCL) planning algorithm used in [152] simultaneously reasons about causality and character intentionality and motivation in order to produce narrative sequences that are causally coherent and have elements of character believability. The authors present a tool that generates a narrative plan meeting the outcome objective. It ensures that all character actions and goals are justified by events within the narrative itself. An overview of newer story planning concepts can be found in [153].

4.5.2. Petri Nets

Petri nets are a well known extensions of finite state machines; they allow an easy description of parallel activities. A standard Petri net is defined as a 4-tuple (S, T, W, M) where

- S is a set of places, marked graphically by circles,
- T is a set of transitions, marked graphically by bars,

- $W : (S \times T) \cup (T \times S) \rightarrow \mathbb{N}$ is a multiset of arcs, i.e., W assigns to each arc a non-negative integer *arc multiplicity* (or weight); note that no arc may connect two places or two transitions. The elements of W are indicated graphically by arrows.
- M_0 is an initial marking, consisting if tokens, indicated graphically by dots.

A transition fires if and only if $W_{(s,t)}$ tokens are at the input place, and it will produce $W_{(t,s)}$ tokens at the output place. A major advantage of Petri nets is the availability of tools to edit the model and to proof properties such as liveliness (i.e., the Petri net does not lead to deadlocks), and the reachability of a specified marking M from the initial marking M0. A very useful enhancement for Petri nets is to allow *colors* for tokens. In a standard Petri net, tokens are indistinguishable. In a colored Petri net, every token has a value ("color"). In popular tools for colored Petri nets such as the CPN tools [154], the values of tokens are typed, and they can be tested (using guard expressions) and manipulated with a functional programming language.

An obvious way to employ Petri nets for game quests is to interpret the places as game locations and the tokens as players [155]. The Petri net describing our example from above might look as in Figure 24. Our example is simplified: it contains conditions for the transitions. For real Petri nets, such conditions (and their variables) are not allowed; they must be expressed in terms of tokens. A possibility, described in [155], is to use colored Petri nets and have a colored token for each variable used in a condition. Thus, we do not only represent players by tokens but also variables. However, this (correct) notation makes the graph much more complex. The notation shown in Figure 24 can be translated 1:1 into the correct notation. Again, the random variable in the condition can be replaced by other variables, taking the context of the game and the player(s) into account.

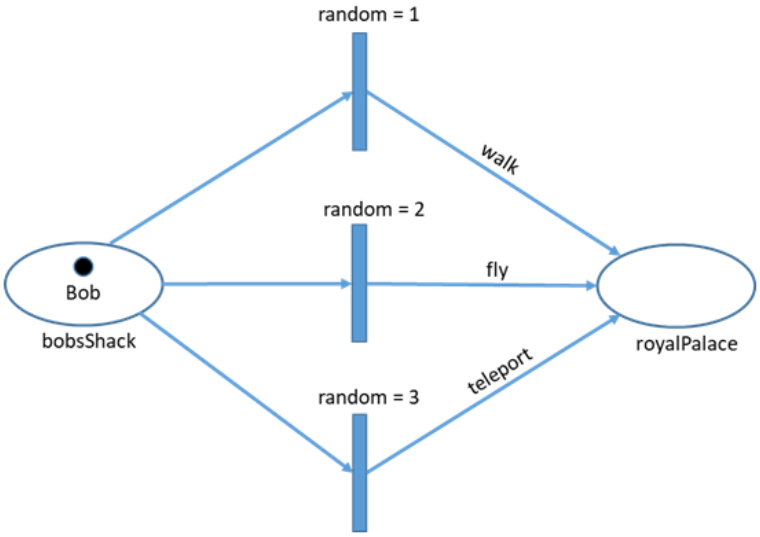


Figure 24. Petri net for the shack-to-palace example.

A different use of Petri nets is described in [156]. They extend the standard Petri Net model by three new constructs, *conditions*, *items* and *locations*, and they model stories with these modified Petri nets. They also describe how to infer the player type from his/her playing history, and they automatically generate game variations from their Petri net model for a real game (*Neverwinter Nights* [157]). A major drawback of their approach is that they lose the power of formal verification for standard Petri nets because they introduce new constructs.

4.5.3. StoryTec

A less formal model is the basis of StoryTec, a system for the specification of story-based games [143]. A StoryTec specification distinguishes the *game structure model* and the *game logic model*.

Game Structure Model

The game structure model describes the data part of a story. A story consists of scenes, similar to a theater play. Each scene models a small part of the game. Scenes are interconnected by *transitions*. A scene consists of a set of *objects*, including physical elements, interaction elements such as buttons or text fields, and avatars. Thus, the overall game has the narrative structure of a theater play. Scenes and objects can be configured with *parameters*. They can have the types boolean, color, composite, enum, file, float, scene, skill, stimulus and string.

Game Logic Model

When the data part of a story has been defined, activities are added. This is done with the game logic model. Its basic construct are *actions*. Similar to ADL, conditional actions can be specified. For example, a virtual character's move from one location to another is defined as an action. Actions also have parameters. Typically, actions are at a high level of abstraction; for example, the details of animations are not part of their specification. A *stimulus* is an event that triggers an action. Stimuli are also specified at a high level of abstraction. Unlike in Petri nets, parallel actions are not supported by StoryTec.

The StoryTec Editor and the StoryTec Runtime

A powerful StoryTec editor supports easy graphical editing of both the structure model and the logic model. Figure 25 shows the graphical interface of the editor. The main editing tools for the game structure and the game logic can be selected on the left side. The central part shows the currently edited scene in the upper part and below the overall scene structure. Note that we have created extra scenes for the walking, flying and teleporting activities since we want to show them graphically, and we want to assign parameters to them. The scene *Bob's Shack* contains several objects shown as small squares. The right side has windows for the objects with their parameters. Extra windows are opened if we want to edit conditions, as in our case with the variable *random*. When editing of the story is completed, we store the result in an ICML file (INSCAPE Communication Markup Language) file, similar to an XML file. Listing 1 shows an extract from this file for our example.

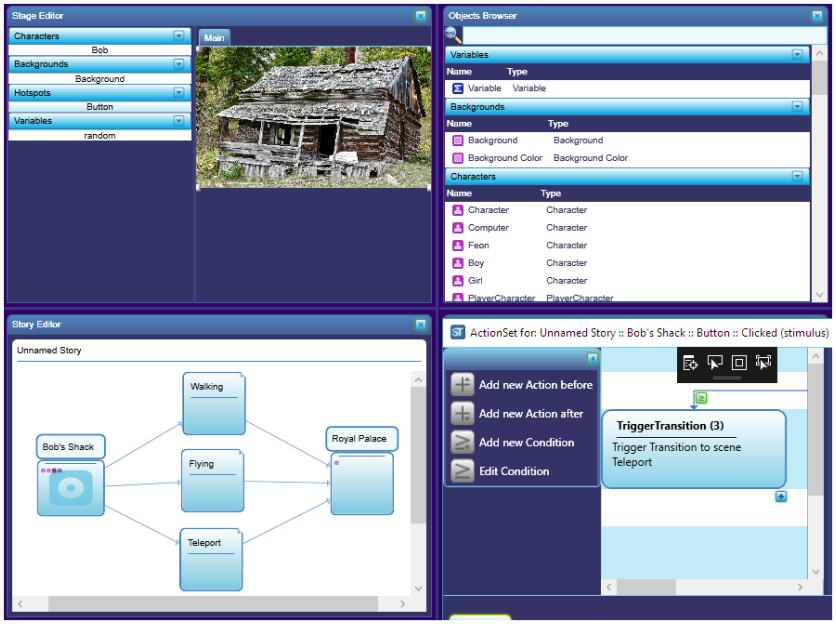


Figure 25. The StoryTec editor with our example.

Listing 1: Extract from the ICML file for our example.

```
<transitionSet scenarioID="StandardScenario">
  <transition nameID="Transition" name="Transition" fromScene="Bob's Shack"
    toScene="Walking" transitionType="automatic" transitionTarget="TriggeringPlayer">
    <condition name="default" typeOfCondition="triggered">
      <sequenceElement>
        <action name="changeToWalking" typeOfAction="changeScene" />
      </sequenceElement>
    </condition>
    <effect name="Loading bar" />
  </transition>
  <transition nameID="Transition (2)" name="Transition (2)" fromScene="Bob's Shack"
    toScene="Flying" transitionType="automatic" transitionTarget="TriggeringPlayer">
    <condition name="default" typeOfCondition="triggered">
      <sequenceElement>
        <action name="changeToFlying" typeOfAction="changeScene" />
      </sequenceElement>
    </condition>
    <effect name="Loading bar" />
  </transition>
  <transition nameID="Transition (3)" name="Transition (3)" fromScene="Bob's Shack"
    toScene="Teleport" transitionType="automatic" transitionTarget="TriggeringPlayer">
    <condition name="default" typeOfCondition="triggered">
      <sequenceElement>
        <action name="changeToTeleport" typeOfAction="changeScene" />
      </sequenceElement>
    </condition>
    <effect name="Loading bar" />
  </transition>
</transitionSet>
```

A runtime system called *story engine* is provided that connects to a game engine. The output of the editing process is fed into the story engine and executed. The overall architecture is shown in Figure 26.

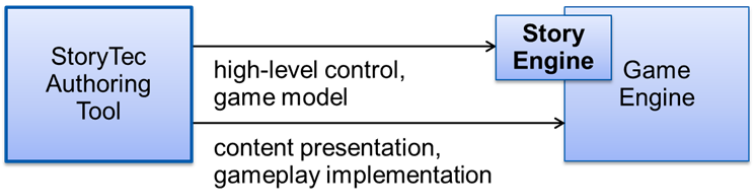


Figure 26. The story engine, the runtime of StoryTec (adapted from [143]).

4.5.4. The Procedural Generation of Game Content from a PCG Story

It is possible to combine the semi-automatic generation of stories or quests with the automatic generation of other content. For example, landscapes and buildings can be generated automatically from a procedurally generated story. When pre-specified location keywords such as *shack*, *palace*, *field* or *forest* are found in a story, the corresponding game objects can be created without human intervention, as described in Sections 4.1 and 4.3 above. Details can be found in [145]. The locations are mapped to a *space tree* where each node specifies a portion of the world. That space tree is mapped to a *2D grid*, and the landscapes, plants and buildings are then procedurally generated. Care is taken that neighboring landscapes are plausible; for example, in a mountain area, caves are more probable than gardens. In order to generate nice-looking landscapes, the main features of each region in the grid (i.e., the rocks in a mountain) follow a Gaussian distribution, and they spread over into their

neighboring regions. More details on the automatic derivation of content from a PCG story can be found in [142,145].

5. Conclusions

We conclude this work with a table showing the most important contributions to the field of PCG in research and in practice (Table 1) and rank their presence from very high (++) to very low (- -). As can be seen, there are many areas where practitioners have overtaken the research community, and, in other fields, theories exist that never made it into practice. The two worlds seem to be fairly separated. We thus recommend that researchers and game developers talk to each other more often, for the benefit of both.

Table 1. Presence of diverse topics in research and practice in the context of procedural generation from very high (++) to very low (- -).

Topic	Research	Implementation	Implementation Example
Creation of textures and materials	++	++	.werkkzeug, Unreal Engine 4
Generation of floor plans	++	-	
Creation of buildings without interior	++	+	BuildR, Building Generator
Creation of multi-story buildings	- -	- -	
Creation of buildings with interior	o	- -	
Creation of public buildings	- -	- -	
Creation of public places	–	- -	
Generation of road networks	++	++	Road Network Generator, SUMO [158]
Multi-track road network generation	o	++	Road Network Generator, SUMO
Traffic simulation	++	++	SUMO, MATSim [159]
Humanoid model generation	o	+	UMA, MakeHuman
Animation of humanoid models	- -	-	Munty Engine [160]
Artificial personality generation	-	+	Seventh Sanctum [161]
Simulation of NPCs with individual daily routines	–	-	Open Simulator [162]
Simultaneous simulation of many NPCs	++	+	PEDSIM [163]
Generation of plants and trees	++	+	Speedtree, Xfrog
Generation of stories and quests	++	o	Rogue, Skyrim, Zelda

An analysis of the research papers and implementations in the field of PCG clearly shows its immense complexity and variety. Results were contributed from many different areas of computer science, computer graphics, architecture, town planning, psychology, traffic planning and mathematics. We explicitly want to reference the good classification of procedural modelling methods by Smelik et al. [164]. It also became obvious that practitioners have contributed a great deal to the progress of the field without publishing their work in research papers. The interest and the awareness of the game development community in the context of PCG is shown by the recent foundation of diverse communities settling in the academic, hobbyist, commercial and independent sector. Some of the most active and valuable are:

- *PCG Wiki*— A wiki presenting PCG in theory and practice, collecting games that make use of content generation algorithms and listing related links,
- *procjam.com*—An online contest inviting developers to create graphical demos explicitly making use of PCG,
- *Reddit Procedural Generation*—Topic on Reddit that serves as a panel for discussions and presentations of projects.

Unfortunately, large companies are thrifty about how they model or automatically generate cities. Especially for open world games like *Grand Theft Auto 3-5* [127], *Watchdogs* [165] or *L.A. Noire* [166], let us assume that procedural techniques play an important role. *Assassin's Creed Unity* recently presented a technique to generate entire, flexible streets including buildings [167]. As impressive as this demonstration was, it is sad that there is so little industry knowledge about PCG at conferences like the GDC (*Game Developers Conference*). The knowledge cannot only be used to create games—there are so many other areas of applications as *Visitor Ville* [168] shows exemplarily on their website where

they visualize web traffic abstractly in a virtual world. One can say that the automatic generation of digital content is one of the next big things in game design. State-of-the-art engines already offer good workflows to create games in which the development teams rarely lack programming skills but time and money to buy or create models, music and levels. Furthermore, generated worlds can stimulate the game designers' creativity and show them worlds they would never have created on their own [96].

Acknowledgments: We gratefully acknowledge the help of Benjamin Guthier with the programming of the L-System examples for Section 4.1.1. We are also thankful for the assistance of Robert Konrad in the creation of the StoryTec example.

Author Contributions: J.F. and W.E. outlined the structure of a virtual world and provided an overview over current research results for each individual object type that can be found in such an environment.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bethesda Game Studios. *The Elder Scrolls V: Skyrim*; Bethesda Game Studios: Rockville, MD, USA, 2011.
2. Burgess, J. Modular Level Design for Skyrim. 2013. Available online: http://blog.joelburgess.com/2013/04/skyrims-modular-level-design-gdc-2013.html (accessed on 2 October 2017).
3. Maxis. *The Sims 1*; Maxis: Redwood Shores, CA, USA, 2000.
4. Maxis. *Sim City*; Maxis: Redwood Shores, CA, USA, 1989.
5. Bach, E.; Madsen, A. Procedural Character Generation: Implementing Reference Fitting and Principal Components Analysis. Master's Thesis, Aalborg University, Aalborg, Denmark, 2007.
6. Frontier Developments. *Elite: Dangerous*; Video Game; Frontier Developments: Cambridge, UK, 2014.
7. Mojang. *Minecraft*; Video Game; Mojang: Stockholm, Sweden, 2009.
8. Teuber, K. *The Settlers of Catan*; Franckh-Kosmos Verlags-GmbH & Co.: Stuttgart, Germany, 1995.
9. Wang, M. Java Settlers Intelligente agentenbasierte Spielsysteme für intuitive Multi-Touch-Umgebungen. Ph.D. Thesis, Free University of Berlin, Berlin, Germany, 2008.
10. Dörner, R.; Göbel, S.; Effelsberg, W.; Wiemeyer, J. *Serious Games: Foundations, Concepts and Practice*; Springer International Publishing: Gewerbestrasse, Switzerland, 2016.
11. Worth, D. *Beneath Apple Manor*; The Software Factory/Quality Software: Los Angeles, CA, USA, 1978.
12. Carreker, D. *The Game Developer's Dictionary: A Multidisciplinary Lexicon for Professionals and Students*; Cengage Learning: Boston, MA, USA, 2012.
13. Michael Toy, G.W. *Rogue*; Epyx: San Francisco, CA, USA, 1980.
14. Olivetti, J. The Game Archaeologist: A Brief History of Roguelikes. 2014. Available online: https://www.engadget.com/2014/01/18/the-game-archaeologist-a-brief-history-of-roguelikes/ (accessed on 2 October 2017).
15. Lee-Urban, S. Procedural Content Generation. 2016. Available online: https://www.cc.gatech.edu/~surban6/2016-cs4731/lectures/2016_06_30-ProceduralContentGeneration_intro.pdf (accessed on 2 October 2017).
16. Peachey, D.R. Solid Texturing of Complex Surfaces. In Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques, San Francisco, CA, USA, 22–26 July 1985; ACM: New York, NY, USA, 1985; pp. 279–286.
17. Musgrave, F.K.; Kolb, C.E.; Mace, R.S. The Synthesis and Rendering of Eroded Fractal Terrains. In Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques, Boston, MA, USA, 31 July–4 August 1989; ACM: New York, NY, USA, 1989; Volume 23, pp. 41–50.
18. Oppenheimer, P.E. Real Time Design and Animation of Fractal Plants and Trees. In Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, Dallas, TX, USA, 18–22 August 1986; ACM: New York, NY, USA, 1986; Volume 20, pp. 55–64.
19. Ebert, D.; Musgrave, F.; Peachey, D.; Perlin, K.; Worley, S. *Texturing and Modeling: A Procedural Approach*; (The Morgan Kaufmann Series in Computer Graphics); Elsevier Science: Amsterdam, The Netherlands, 2002.
20. Westwood Pacific. *Command and Conquer: Red Alert 2*; Westwood Pacific: Las Vegas, NV, USA, 2000.
21. Blizzard North. *Diablo*; Blizzard North: San Mateo, CA, USA, 1996.

22. Bethesda Softworks. *The Elder Scrolls II: Daggerfall*; Bethesda Softworks: Bethesda, MD, USA, 1996.
23. Blender Foundation. Blender. 2017. Available online: https://www.blender.org/ (accessed on 7 October 2017).
24. Autodesk. Character Generator. 2014. Available online: https://charactergenerator.autodesk.com/ (accessed on 2 October 2017).
25. Mixamo. Fuse. 2014. Available online: https://www.mixamo.com/ (accessed on 7 October 2017).
26. MakeHuman team. MakeHuman. 2017. Available online: http://www.makehuman.org/ (accessed on 2 October 2017).
27. Farbrausch. *.kkrieger*. 2004. Available online: https://web.archive.org/web/20120204065621/http://www.theprodukkt.com/kkrieger (accessed on 28 October 2017).
28. Digitalekultur e.V. Die Demoszene—Neue Welten im Computer. 2003. Available online: https://www.digitalekultur.org/files/dk_wasistdiedemoszene.pdf (accessed on 2 October 2017).
29. Farbrausch. Werkkzeug3. 2011. Available online: https://github.com/farbrausch/fr_public (accessed on 27 October 2017).
30. Compton, K.; Osborn, J.C.; Mateas, M. Generative methods. In Proceedings of the Fourth Procedural Content Generation in Games Workshop, Chania, Greece, 14–17 May 2013.
31. Macri, D.; Pallister, K. Procedural 3D Content Generation. Available online: http://web.archive.org/web/20060719005301/www.intel.com/cd/ids/developer/asmo-na/eng/20247.htm (accessed on 28 October 2017).
32. Smith, G. An Analog History of Procedural Content Generation. 2015. Available online: http://sokath.com/main/files/1/smith-fdg15.pdf (accessed on 29 August 2017).
33. Pixologic. ZBrush. 1999. Available online: http://pixologic.com/ (accessed on 29 August 2017).
34. Epic Games. Unreal Engine 4. 2014. Available online: https://www.unrealengine.com (accessed on 7 October 2017).
35. Unity Technologies. Unity 2017. Available online: https://unity3d.com (accessed on 7 October 2017).
36. Crytec. Cry Engine V. 2016. Available online: https://www.cryengine.com/ (accessed on 7 October 2017).
37. Epic Games. Procedural Buildings. 2012. Available online: https://docs.unrealengine.com/udk/Three/ProceduralBuildings.html (accessed on 27 October 2017).
38. Side Effects Software. Houdini. 2017. Available online: https://www.sidefx.com/products/houdini-core/ (accessed on 7 October 2017).
39. Esri R&D Center Zurich. Esri City Engine. 2014. Available online: http://www.esri.com/software/cityengine (accessed on 7 October 2017).
40. Smelik, R.M.; Tutenel, T.; de Kraker, K.J.; Bidarra, R. A declarative approach to procedural modeling of virtual worlds. *Comput. Graph.* **2011**, *35*, 352–363.
41. Roden, T.; Parberry, I. From artistry to automation: A structured methodology for procedural content creation. In Proceedings of the International Conference on Entertainment Computing (ICEC 2004), Eindhoven, The Netherlands, 1–3 September 2004; pp. 301–304.
42. Togelius, J.; Kastbjerg, E.; Schedl, D.; Yannakakis, G.N. What is procedural content generation?: Mario on the borderline. In Proceedings of the 2nd International Workshop on Procedural Content Generation in Games, Bordeaux, France, 28 June 2011; ACM: New York, NY, USA, 2011.
43. Hendrikx, M.; Meijer, S.; Van Der Velden, J.; Iosup, A. Procedural Content Generation for Games: A Survey. *ACM Trans. Multimedia Comput. Commun. Appl.* **2013**, *9*, doi:10.1145/2422956.2422957.
44. Togelius, J.; Shaker, N.; Nelson, M.J. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research/J*; Togelius, J., Shaker, N., Nelson, M.J., Eds.; Springer: Berlin, Germany, 2014.
45. Prusinkiewicz, P.; Lindenmayer, A. *The Algorithmic Beauty of Plants*; Springer Science & Business Media: Berlin, Germany, 2012.
46. Finkenzeller, D. Modellierung Komplexer Gebäudefassaden in der Computergraphik. Ph.D. Thesis, Universität Karlsruhe (TH), Karlsruhe, Germany, 2008.
47. Pixar Animation Studios. RenderMan. 2014. Available online: https://renderman.pixar.com/ (accessed on 8 October 2017).
48. Bradbury, G.A.; Choi, I.; Amati, C.; Mitchell, K.; Weyrich, T. Frequency-based Controls for Terrain Editing. In Proceedings of the 11th European Conference on Visual Media Production, London, UK, 13–14 November 2014; ACM: New York, NY, USA, doi:10.1145/2668904.2668944.

49. Soto, J. Statistical testing of random number generators. In Proceedings of the 22nd National Information Systems Security Conference, Arlington, VA, USA, 18–21 October 1999; Volume 10, p. 12.
50. L'Ecuyer, P. Uniform random number generators: A review. In Proceedings of the 29th Conference on Winter simulation, Atlanta, GA, USA, 7–10 December 1997; IEEE Computer Society: Washington, DC, USA, 1997; pp. 127–134.
51. Gerhard, G.; Thomas, H.K.; Claus, N.; Karl-Heinz, H. OGC City Geography Markup Language (CityGML) En-coding Standard. 2012. Available online: https://portal.opengeospatial.org/files/?artifact_id=47842 (accessed on 2 October 2017).
52. Chomsky, N. Three models for the description of language. *IRE Trans. Inf. Theory* **1956**, *2*, 113–124.
53. Dapper, T. Practical Procedural Modeling of Plants. 2003. Available online: http://www.td-grafik.de/artic/talk20030122/overview.html (accessed on 2 October 2017).
54. Deussen, O.; Hanrahan, P.; Lintermann, B.; Měch, R.; Pharr, M.; Prusinkiewicz, P. Realistic Modeling and Rendering of Plant Ecosystems. In Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, Orlando, FL, USA, 19–24 July 1998; ACM: New York, NY, USA, 1998; pp. 275–286.
55. Prusinkiewicz, P.; Hanan, J.; Hammel, M.; Mech, R.; Room, P.; Remphrey, W. *Plants to Ecosystems: Advances in Computational Life Sciences*; CSIRO: Clayton, Australia, 1997; pp. 1–134.
56. Měch, R.; Prusinkiewicz, P. Visual models of plants interacting with their environment. In Proceedings of the 23rd Annual Conference on Computer Graphics And Interactive Techniques, New Orleans, LA, USA, 4–9 August 1996; ACM: New York, NY, USA, 1996; pp. 397–410.
57. Sorrensen-Cothern, K.A.; Ford, E.D.; Sprugel, D.G. A model of competition incorporating plasticity through modular foliage and crown development. *Ecol. Monogr.* **1993**, *63*, 277–304.
58. Chen, X.; Neubert, B.; Xu, Y.Q.; Deussen, O.; Kang, S.B. Sketch-based Tree Modeling Using Markov Random Field. *ACM Trans. Graph.* **2008**, *27*, doi:10.1145/1457515.1409062.
59. Okabe, M.; Owada, S.; Igarash, T. Interactive Design of Botanical Trees using Freehand Sketches and Example-based Editing. *Comput. Graph. Forum.* **2005**, *24*, 487–496.
60. Reche-Martinez, A.; Martin, I.; Drettakis, G. Volumetric reconstruction and interactive rendering of trees from photographs. In Proceedings of the ACM SIGGRAPH 2004, Los Angeles, CA, USA, 8–12 August 2004; ACM: New York, NY, USA, 2004; Volume 23, pp. 720–727.
61. Shlyakhter, I.; Rozenoer, M.; Dorsey, J.; Teller, S. Reconstructing 3D tree models from instrumented photographs. *IEEE Comput. Graph. Appl.* **2001**, *21*, 53–61.
62. Tan, P.; Zeng, G.; Wang, J.; Kang, S.B.; Quan, L. Image-based tree modeling. In Proceedings of the ACM SIGGRAPH 2007, San Diego, CA, USA, 5–9 August 2007; ACM: New York, NY, USA, 2007; Volume 26, p. 87.
63. Mandelbrot, B.B.; Pignoni, R. *The Fractal Geometry of Nature*; WH Freeman: New York, NY, USA, 1983; Volume 173.
64. Greenworks Organic Software. Xfrog. 2017. Available online: http://xfrog.com/ (accessed on 2 October 2017).
65. Interactive Data Visualization. Speed Tree. 2015. Available online: https://speedtree.com (accessed on 2 October 2017).
66. Kim, J. Modeling and optimization of a tree based on virtual reality for immersive virtual landscape generation. *Symmetry* **2016**, *8*, doi:10.3390/sym8090093.
67. Wong, S.K.; Chen, K.C. A Procedural Approach to Modelling Virtual Climbing Plants with Tendrils. *Comput. Graph. Forum.* **2016**, *35*, 5–18.
68. Stava, O.; Pirk, S.; Kratt, J.; Chen, B.; Mźch, R.; Deussen, O.; Benes, B. Inverse Procedural Modelling of Trees. *Comput. Graph. Forum* **2014**, *33*, 118–131.
69. Smelik, R.M.; De Kraker, K.J.; Tutenel, T.; Bidarra, R.; Groenewegen, S.A. A Survey of Procedural Methods for Terrain Modelling. Available online: http://cg.its.tudelft.nl/~rafa/myPapers/bidarra.3AMIGAS.RS.pdf (accessed on 2 October 2017).
70. Freiknecht, J. Terrain Tutorial Using Shade-C. 2011. Available online: http://www.jofre.de/Transport/Terrain%20Tutorial%20using%20ShadeC.pdf (accessed on 2 October 2017).
71. Matthews, E.A.; Malloy, B.A. Incorporating Coherent Terrain Types into Story-Driven Procedural Maps. Available online: http://meaningfulplay.msu.edu/proceedings2012/mp2012_submission_41.pdf (accessed on 2 October 2017).

72. Perlin, K. Improving noise. In Proceedings of the ACM SIGGRAPH 2002, San Antonio, TX, USA, 23–26 July 2002; ACM: New York, NY, USA, 2002; Volume 21, pp. 681–682.
73. Prusinkiewicz, P.; Hammel, M. A Fractal Model of Mountains and Rivers. Available online: http://algorithmicbotany.org/papers/mountains.gi93.pdf (accessed on 2 October 2017).
74. Belhadj, F.; Audibert, P. Modeling Landscapes with Ridges and Rivers: Bottom Up Approach. In Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia, Dunedin, New Zealand, 30 November–2 December 2005; ACM: New York, NY, USA, 2005; pp. 447–450.
75. Miller, G.S. The definition and rendering of terrain maps. In Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, Dallas, TX, USA, 18–22 August 1986; ACM: New York, NY, USA, 1986; Volume 20, pp. 39–48.
76. Olsen, J. Realtime Procedural Terrain Generation. 2004. Available online: http://web.mit.edu/cesium/Public/terrain.pdf (accessed on 2 October 2017).
77. Kahoun, M. Realtime Library for Procedural Generation and Rendering of Terrains. Ph.D. Thesis, Charles University, Prague, Czech Republic, 2013.
78. Duchaineau, M.; Wolinsky, M.; Sigeti, D.E.; Miller, M.C.; Aldrich, C.; Mineev-Weinstein, M.B. ROAMing terrain: Real-time Optimally Adapting Meshes. In Proceedings of the Visualization '97, Phoenix, AZ, USA, 18–24 October 1997; pp. 81–88.
79. Lee, J.; Jeong, K.; Kim, J. MAVE: Maze-based immersive virtual environment for new presence and experience. *Comput. Animat. Virtual Worlds* **2017**, *28*.
80. Benes, B.; Forsbach, R. Layered Data Representation for Visual Simulation of Terrain Erosion. In Proceedings of the 17th Spring Conference on Computer Graphics (SCCG '01), Budmerice, Slovakia, 25–28 April 2001; IEEE Computer Society: Washington, DC, USA, 2001; p. 80.
81. Santamaría-Ibirika, A.; Cantero, X.; Salazar, M.; Devesa, J.; Santos, I.; Huerta, S.; Bringas, P.G. Procedural approach to volumetric terrain generation. *Vis. Comput.* **2014**, *30*, 997–1007.
82. Cui, J.; Chow, Y.W.; Zhang, M. A Voxel-Based Octree Construction Approach for Procedural Cave Generation. 2011. Available online: http://ro.uow.edu.au/cgi/viewcontent.cgi?article=10948&context=infopapers (accessed on 2 October 2017).
83. Boggus, M.; Crawfis, R. Procedural Creation of 3D Solution Cave Models. 2009. Available online: ftp://ftp.cse.ohio-state.edu/pub/tech-report/2009/TR19.pdf (accessed on 2 October 2017).
84. Hammes, J. Modeling of Ecosystems as a Data Source for Real-Time Terrain Rendering. In Proceedings of the Digital Earth Moving: First International Symposium, DEM 2001, Manno, Switzerland, 5–7 September 2001; Westort, C.Y., Ed.; Springer: Berlin/Heidelberg, Germany, 2001; pp. 98–111.
85. Alsweis, M.; Deussen, O. Wang-tiles for the simulation and visualization of plant competition. In *Advances in Computer Graphics*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 1–11.
86. Berger, U.; Hildenbrandt, H.; Grimm, V. Towards a standard for the individual-based modeling of plant populations: self-thinning and the field-of-neighborhood approach. *Nat. Resour. Model.* **2002**, *15*, 39–54.
87. Wang, H. Proving theorems by pattern recognition I. *Commun. ACM.* **1960**, *3*, 220–234.
88. Deussen, O.; Lintermann, B. *Digital Design of Nature: Computer Generated Plants and Organics*, 1st ed.; Springer: Berlin/Heidelberg, Germany, 2010.
89. Huijser, R.; Dobbe, J.; Bronsvoort, W.F.; Bidarra, R. Procedural Natural Systems for Game Level Design. Available online: https://graphics.tudelft.nl/Publications-new/2010/HDBB10a/HDBB10a.pdf (accessed on 2 October 2017).
90. Derzapf, E.; Ganster, B.; Guthe, M.; Klein, R. River Networks for Instant Procedural Planets. *Comput. Graph. Forum* **2011**, *30*, 2031–2040.
91. Doran, J.; Parberry, I. Controlled procedural terrain generation using software agents. *IEEE Trans. Comput. Intell. AI Games* **2010**, *2*, 111–119.
92. Loopix. Mystymood. 2016. Available online: http://www.loopix-project.com/ (accessed on 27 October 2017).
93. Ilangovan, P.K. Procedural City Generator. Master's Thesis, Bournemouth University, Bournemouth, Dorset, England, 2009.
94. Felix Queißner, M.K.; Freiknecht, J. TUST Scripting Library. 2013. Available online: https://github.com/MasterQ32/TUST (accessed on 3 October 2017).

95. Banf, M.; Barth, M.; Schulze, H.; Koch, J.; Pritzkau, A.; Schmidt, M.; Daraban, A.; Meister, S.; Sandhöfer, R.; Sotke, V.; et al. On-demand creation of procedural cities. In *Game and Entertainment Technologies*; IADIS Press: Freiburg, Germany, 2010.
96. Kelly, G.; McCabe, H. Citygen: An Interactive System for Procedural City Generation. Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.688.7603&rep=rep1&type=pdf (accessed on 3 October 2017).
97. Parish, Y.I.; Müller, P. Procedural modeling of cities. In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, Los Angeles, CA, USA, 12–17 August 2001; ACM: New York, NY, USA, 2001; pp. 301–308.
98. Nadeo. *TrackMania Original*; Nadeo: Issy-les-Moulineaux, France, 2003.
99. Namco. *Ridge Racer 64*; Namco: Tokio, Japan, 1999.
100. Acclaim Entertainment. *Re-Volt*; Acclaim Entertainment: Glen Cove, NY, USA, 1999.
101. Taplin, J. Simulation Models of Traffic Flow. Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.1933&rep=rep1&type=pdf (accessed on 3 October 2017).
102. Fritzsche, H.T. A model for traffic simulation. *Traffic Eng. Control* **1994**, *35*, 317–321.
103. Sewall, J.; Wilkie, D.; Merrell, P.; Lin, M.C. Continuum Traffic Simulation. *Comput. Graph. Forum* **2010**, *27*, 439–448.
104. Brenner, C. Towards Fully Automatic Generation of City Models. 2000. Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.4549&rep=rep1&type=pdf (accessed on 3 October 2017).
105. Saldana, M.; Johanson, C. Procedural modeling for rapid-prototyping of multiple building phases. *Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci.* **2013**, *5*, doi:10.5194/isprsarchives-XL-5-W1-205-2013.
106. Birch, P.J.; Browne, S.P.; Jennings, V.J.; Day, A.M.; Arnold, D.B. Rapid Procedural-modelling of Architectural Structures. In Proceedings of the 2001 Conference on Virtual Reality, Archeology, and Cultural Heritage, Athens, Greece, 28–30 November 2001; ACM: New York, NY, USA, 2001; pp. 187–196.
107. Martin, J. Algorithmic Beauty of Buildings Methods for Procedural Building Generation. Available online: http://digitalcommons.trinity.edu/cgi/viewcontent.cgi?article=1003&context=compsci_honors (accessed on 3 October 2017).
108. Stocker, J. BuildR2. 2003. Available online: http://support.jasperstocker.com/buildr2/ (accessed on 3 October 2017).
109. Ibele, T. Building Generator. 2009. Available online: http://tysonibele.com/Main/BuildingGenerator/buildingGen.htm (accessed on 3 October 2017).
110. Podevyn, M. Developing an Organisational Framework for Sustaining Virtual City Models. Ph.D. Thesis, Northumbria University, Newcastle upon Tyne, England, 2013.
111. Merrell, P.; Schkufza, E.; Koltun, V. Computer-generated residential building layouts. In Proceedings of the ACM SIGGRAPH Asia 2010, Seoul, Korea, 15–18 December 2010; ACM: New York, NY, USA, 2010; Volume 29, p. 181.
112. Müller, P.; Wonka, P.; Haegler, S.; Ulmer, A.; Van Gool, L. Procedural modeling of buildings. In Proceedings of the ACM SIGGRAPH 2006, Boston, MA, USA, 30 July–3 August 2006; ACM: New York, NY, USA, 2006; Volume 25, pp. 614–623.
113. Bruls, M.; Huizing, K.; van Wijk, J.J. Squarified Treemaps. In *Data Visualization 2000, Proceedings of the Joint EUROGRAPHICS and IEEE TCVG Symposium on Visualization in Amsterdam, The Netherlands, 29–30 May 2000*; de Leeuw, W.C., van Liere, R., Eds.; Springer: Vienna, Australia, 2000; pp. 33–42.
114. Johnson, B.; Shneiderman, B. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In Proceedings of the 2nd Conference on Visualization'91, San Diego, CA, USA, 22–25 October 1991; pp. 284–291.
115. Mirahmadi, M.; Shami, A. A Novel Algorithm for Real-Time Procedural Generation of Building Floor Plans. *arXiv* **2012**, arXiv:1211.5842. Available online: https://arxiv.org/abs/1211.5842 (accessed on 3 October 2017).
116. Lopes, R.; Tutenel, T.; Smelik, R.M.; De Kraker, K.J.; Bidarra, R. A Constrained Growth Method for Procedural Floor Plan Generation. Available online: https://graphics.tudelft.nl/Publications-new/2010/LTSDB10a/LTSDB10a.pdf (accessed on 3 October 2017).
117. Merrell, P.; Schkufza, E.; Li, Z.; Agrawala, M.; Koltun, V. Interactive furniture layout using interior design guidelines. In Proceedings of the ACM SIGGRAPH 2011, Vancouver, BC, Canada, 7–11 August 2011; ACM: New York, NY, USA, 2011; Volume 30, p. 87.

118. Wonka, P.; Wimmer, M.; Sillion, F.; Ribarsky, W. *Instant Architecture*; ACM: New York, NY, USA, 2003; Volume 22.
119. Whitehead, J. Toward proccedural decorative ornamentation in games. In Proceedings of the 2010 Workshop on Procedural Content Generation in Games, Monterey, CA, USA, 19–21 June 2010; ACM: New York, NY, USA, 2010.
120. Barreto, N.; Roque, L. A Survey of Procedural Content Generation tools in Video Game Creature Design. In Proceedings of the Second Conference on Computation Communication Aesthetics and X, Porto, Portugal, 26–27 June 2014.
121. UMA Steering Group. UMA—Unity Multipurpose Avatar. 2017. Available online: https://www.assetstore.unity3d.com/en/#!/content/13930 (accessed on 3 October 2017).
122. Maxis. *Spore*; Maxis: Redwood Shores, CA, USA, 2008.
123. Hecker, C.; Raabe, B.; Enslow, R.W.; DeWeese, J.; Maynard, J.; van Prooijen, K. Real-time Motion Retargeting to Highly Varied User-created Morphologies. Available online: http://chrishecker.com/images/c/cb/Sporeanim-siggraph08.pdf (accessed on 3 October 2017).
124. Hudson, J. Creature Generation Using Genetic Algorithms and Auto-Rigging. Ph.D. Thesis, Bournemouth University, Poole, Dorset, UK, 2013.
125. Anderson, E.F. Playing Smart-Artificial Intelligence in Computer Games. 2003. Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.8170&rep=rep1&type=pdf (accessed on 3 October 2017).
126. Curtis, S.; Best, A.; Manocha, D. Menge: A modular framework for simulating crowd movement. *Collect. Dyn.* **2016**, *1*, 1–40.
127. Rockstar Games. *Grand Theft Auto III*; Rockstar Games: New York, NY, USA, 2001.
128. Szymanezyk, O.; Dickinson, P.; Duckett, T. From Individual Characters to Large Crowds: Augmenting the Believability of Open-World Games through Exploring Social Emotion in Pedestrian Groups. In Proceedings of the DiGRA 2011 Conference: Think Design Play, Hilversum, The Netherlands, 14–17 September 2011.
129. Lee, S. *The Effect of RPG Newness, Rating, and Character Evilness on The NPC Believability*; Michigan State University: East Lansing, MI, USA, 2009.
130. Horswill, I.D. Lightweight Procedural Animation With Believable Physical Interactions. *IEEE Trans. Comput. Intell. AI Games* **2009**, *1*, 39–49.
131. Karim, A.A.; Gaudin, T.; Meyer, A.; Buendia, A.; Bouakaz, S. Procedural locomotion of multilegged characters in dynamic environments. *Comput. Anim. Virtual Worlds* **2013**, *24*, 3–15.
132. Traum, D. Computational Approaches to Dialogue. In *The Routledge Handbook of Language and Dialogue*; Taylor & Francis: Abingdon, UK, 2017; p. 143.
133. Strong, C.R.; Mateas, M. Talking with NPCs: Towards Dynamic Generation of Discourse Structures. Available online: http://www.aaai.org/Papers/AIIDE/2008/AIIDE08-019.pdf (accessed on 3 October 2017).
134. MacNamee, B.; Cunningham, P. Creating socially interactive no-player characters: The μ-SIV system. *Int. J. Intell. Games Simul.* **2003**, *2*, 28–35.
135. Nevigo. Articy:draft. 2017. Available online: https://www.nevigo.com/en/articydraft/overview/ (accessed on 27 October 2017).
136. Brom, C.; Poch, T.; Sery, O. AI level of detail for really large worlds. *Game Programm. Gems* **2010**, *8*, 213–231.
137. Cassell, J.; Vilhjálmsson, H.H.; Bickmore, T. Beat: The behavior expression animation toolkit. In Proceedings of the 28th Annual Conference on Computer Graphics And Interactive Techniques, Los Angeles, CA, USA, 12–17 August 2001; ACM: New York, NY, USA, 2001; pp. 477–486.
138. Gratch, J.; Rickel, J.; André, E.; Cassell, J.; Petajan, E.; Badler, N. Creating interactive virtual humans: Some assembly required. *IEEE Intell. Syst.* **2002**, *17*, 54–63.
139. De Carolis, B.; Pelachaud, C.; Poggi, I.; Steedman, M. APML, a markup language for believable behavior generation. In *Life-Like Characters*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 65–85.
140. Bickmore, T.W.; Picard, R.W. Establishing and maintaining long-term human-computer relationships. *ACM Trans. Comput.-Hum. Interact. (TOCHI)* **2005**, *12*, 293–327.
141. Miyamoto, S. *The Legend of Zelda*; Nintendo: Kyoto, Japan, 1986.
142. Shaker, N.; Togelius, J.; Nelson, M.J. *Procedural Content Generation in Games*; Springer: Berlin/Heidelberg, Germany, 2016.

143. Mehm, F. Authoring of Adaptive Single-Player Educational Games. In *PIK-Praxis der Informationsverarbeitung und Kommunikation*; De Gruyter: Berlin, Germany, 2014; Volume 37, pp. 157–160.
144. Ashmore, C.; Nitsche, M. The Quest in a Generated World. Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.9841&rep=rep1&type=pdf (accessed on 3 October 2017).
145. Hartsook, K.; Zook, A.; Das, S.; Riedl, M.O. Toward supporting stories with procedurally generated game worlds. In Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games (CIG), Seoul, Korea, 31 August–3 September 2011; pp. 297–304.
146. Sullivan, A.; Mateas, M.; Wardrip-Fruin, N. Making quests playable: Choices, CRPGs, and the Grail framework. *Leonardo Electron. Almanac* **2012**, *17*.
147. Dormans, J.; Bakkes, S. Generating missions and spaces for adaptable play experiences. *IEEE Trans. Comput. Intell. AI Games* **2011**, *3*, 216–228.
148. Colton, S.; Wiggins, G.A. Computational creativity: The final frontier? In Proceedings of the 20th European Conference on Artificial Intelligence, Montpellier, France, 27–31 August 2012; pp. 21–26.
149. Liapis, A.; Yannakakis, G.N.; Togelius, J. Computational Game Creativity. Available online: http://julian.togelius.com/Liapis2014Computational.pdf (accessed on 3 October 2017).
150. Pednault, E.P. Formulating multiagent, dynamic-world problems in the classical planning framework. In *Reasoning about Actions and Plans*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1987; pp. 47–82.
151. Riedl, M.; Young, R. Character-focused narrative generation for execution in virtual worlds. In Proceedings of the International Conference on Virtual Storytelling. Using Virtual RealityTechnologies for Storytelling, Toulouse, France, 20–21 November 2003; pp. 47–56.
152. Riedl, M.O.; Young, R.M. Narrative planning: Balancing plot and character. *J. Artif. Intell. Res.* **2010**, *39*, 217–268.
153. Gervas, P. *The Living Handbook of Narratology, Chapter Story Generator Algorithms*; Hamburg University: Hamburg, Germany, 2013.
154. Beaudouin-Lafon, M.; Mackay, W.; Andersen, P.; Janecek, P.; Jensen, M.; Lassen, M.; Lund, K.; Mortensen, K.; Munck, S.; Ratzer, A.; et al. CPN/Tools: A post-WIMP interface for editing and simulating coloured Petri nets. In Proceedings of the International Conference on Applications and Theory of Petri Nets, Newcastle upon Tyne, UK, 25–29 June 2001; pp. 71–80.
155. Reuter, C. Authoring Collaborative Multiplayer Games-Game Design Patterns, Structural Verification, Collaborative Balancing and Rapid Prototyping. Ph.D. Thesis, Technische Universität Darmstadt, Darmstadt, Germany, 2016.
156. Lee, Y.S.; Cho, S.B. Context-aware petri net for dynamic procedural content generation in role-playing game. *IEEE Comput. Intell. Mag.* **2011**, *6*, 16–25.
157. BioWare. *Neverwinter Nights*; BioWare: Edmonton, AB, Canada, 2002.
158. Krajzewicz, D.; Hertkorn, G.; Rössel, C.; Wagner, P. SUMO (Simulation of Urban MObility)—An Open-Source Traffic Simulation. Available online: http://sumo.dlr.de/pdf/dkrajzew_MESM2002_SUMO.pdf (accessed on 3 October 2017).
159. Horni, A.; Nagel, K.; Axhausen, K.W. *The multi-agent transport simulation MATSim*; Ubiquity Press: London, UK, 2016.
160. Munty Engine Team. Munty Engine. 2014. Available online: https://sourceforge.net/projects/muntyengine/ (accessed on 12 October 2017).
161. Steven Savage. Seventh Sanctum. 2013. Available online: http://www.seventhsanctum.com (accessed on 2 October 2017).
162. Guard, D. OpenSimulator. 2007. Available online: http://opensimulator.org (accessed on 2 October 2017).
163. Gloor, C. PEDSIM. 2016. Available online: http://pedsim.silmaril.org (accessed on 2 October 2017).
164. Smelik, R.M.; Tutenel, T.; Bidarra, R.; Benes, B. A Survey on Procedural Modelling for Virtual Worlds. *Comput. Graph. Forum* **2014**, *33*, 31–50.
165. Ubisoft Montreal. *Watch Dogs*; Ubisoft: Montreal, QC, Canada, 2014.
166. Team Bondi. *L.A. Noire*; Rockstar Games: New York, NY, USA, 2011.

167. Markuz. IGN's "Making of": The Hidden Secrets. 2014. Available online: http://www.accesstheanimus.com/Making_of_hidden_secrets_part2.html (accessed on 3 October 2017).
168. Savage, R. VisitorVille. 2003. Available online: http://www.visitorville.com/ (accessed on 2 October 2017).