Procedural Content Generation for Games

Inauguraldissertation zur Erlangung
des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von
M.Sc. Jonas Freiknecht
aus Hannover

Mannheim, 2020

Dekan:	Dr. Bernd Lübcke, Universität Mannheim
Referent:	Prof. Dr. Wolfgang Effelsberg, Universität Mannheim
Korreferent:	Prof. Dr. Colin Atkinson, Universität Mannheim

Tag der mündlichen Prüfung: 12. Februar 2021

Danksagungen

Nach einer solchen Arbeit ist es nicht leicht, alle Menschen aufzuzählen, die mich direkt oder indirekt unterstützt haben. Ich versuche es dennoch. Allen voran möchte ich meinem Doktorvater Prof. Wolfgang Effelsberg danken, der mir - ohne mich vorher als Master-Studenten gekannt zu haben - die Promotion an seinem Lehrstuhl ermöglichte und mit Geduld, Empathie und nicht zuletzt einem mir unbegreiflichen Verständnis für meine verschiedenen Ausflüge in die Weiten der Informatik unterstützt hat. Sie werden mir nicht glauben, wie dankbar ich Ihnen bin.
Weiterhin möchte ich meinem damaligen Studiengangsleiter Herrn Prof. Heinz Jürgen Müller danken, der vor acht Jahren den Kontakt zur Universität Mannheim herstellte und mich überhaupt erst in die richtige Richtung wies, um mein Promotionsvorhaben anzugehen.
Auch Herr Prof. Peter Henning soll nicht ungenannt bleiben, der mich - auch wenn es ihm vielleicht gar nicht bewusst ist - davon überzeugt hat, dass die Erzeugung virtueller Welten ein lohnenswertes Promotionsthema ist.
Ganz besonderer Dank gilt meiner Frau Sarah und meinen beiden Kindern Justus und Elisa, die viele Abende und Wochenenden zugunsten dieser Arbeit auf meine Gesellschaft verzichten mussten. Jetzt ist es geschafft, das nächste Projekt ist dann wohl der Garten!
Ebenfalls gebührt meinen Eltern und meinen Geschwistern Dank. Meinem Vater Matthias, der mir 1999 das erste Game Development SDK kaufte, meiner Mutter Ulla, die meine Studien als Lektorin begleitete, meiner Schwester Judith, die mich mit unserem kleinen Wettstreit immer wieder zum Schreiben motivierte und meinen beiden Brüdern Ruben und David, die seit Jahren geduldig mit mir über Spiele fachsimpeln.
Nicht zu vergessen sind Benjamin Guthier und Philip Mildner, die mir beim Schreiben, Zitieren und Evaluieren mit Rat und Tat beiseite standen. Ich hätte gerne mehr Zeit mit euch am Lehrstuhl verbracht.

Abstract

Virtual worlds play an increasingly important role in game development today. Whether in the entertainment industry, education, collaboration or data visualization - virtual space offers a freely definable environment that can be adapted to any purpose. Nevertheless, the creation of complex worlds is time-consuming and cost-intensive. A classic example for the use of a virtual world is a driving simulator where learner drivers can test their skills.
The goal of the generation process is to model a realistic city that is large enough to move around for a long time without constantly passing places that have already been seen. Streets must be realistically modeled, have intersections, represent highways and country roads and create an image through buildings that create the greatest possible immersion in the virtual world. But there is still a lack of life. Pedestrians have to populate the streets in large numbers, other cars have to take part in the traffic, and a driving instructor has to sit next to the learner driver, commenting on the actions and chatting away on long journeys. In short, the effort to model such a world by hand would be immense. This thesis deals with different approaches to generate digital content for virtual worlds procedurally i.e., algorithmically.
In the first part of this thesis, virtual, three-dimensional road networks are generated using a pre-defined network graph. The nodes in the graph can be generated procedurally or randomly or can be imported from open data platforms, e.g., from OpenStreetMaps (OSM). The automatic detection of intersections makes the generation flexible. The textures used for roads and intersections are constructed from prefabricated sprites whenever possible, or, in the case of a very individual construction, are newly generated during generation. The ability to create multi-lane roads gives the virtual cities a higher degree of realism.
The interstices of the road network usually contain buildings, industrial areas, common areas or agricultural land. Once these so-called parcels have been identified, they can be populated with precisely these contents. In this dissertation we focus on accessible residential buildings.
The second part of this thesis discusses a novel method of building generation that allows to procedurally create walk-in, multi-storey buildings. The proceeding of simple mesh generation as shown in the road network generation is extended by rules and constraints that allow a flexible floor planning and guarantee a connection of all rooms by a common corridor per floor and a staircase. Since a cityscape is usually characterised by different building shapes, the generation can be parameterized with regard to texturing, roof design, number of floors, and window and door layout. In order to ensure performance

when rendering the city, each building is generated in three levels of detail. The lowest level only shows the outer walls, the highest level shows the interior rooms including stairs, doors and window frames.
Once the environment is created in a way that allows the player a certain immersion, the game world has to be filled with life. Thus, the third part of this thesis discusses the procedural creation of stories for games based on pre-trained language models. The focus here is on an interactive, controlled way of playing, in which the player can interact with the objects, persons and places of the story and influence the plot. Actions generated from the entities of the previous section of the story should give a feeling of a prepared story, but always ensure the greatest possible flexibility of course. The controlled use of places, people and objects in the player's inventory allows a porting to a three-dimensional game world as well as the gameplay in the form of a text adventure.
All methods for creating digital content presented in this thesis were fully implemented and evaluated with respect to usability and performance.

Zusammenfassung

Virtuelle Welten spielen in der heutigen Zeit der Spieleentwicklung eine immer wichtigere Rolle. Sei es in der Unterhaltungsbranche, in der Ausbildung, der Kollaboration oder der Datenvisualisierung - der virtuelle Raum bietet eine frei definierbaren Umgebung, die jedem Zweck beliebig angepasst werden kann. Dennoch ist die Erzeugung komplexer Welten zeitaufwendig und kostenintensiv. Ein klassisches Beispiel für die Verwendung einer virtuellen Welt ist ein Fahrsimulator, an dem Fahrschüler ihre Fähigkeiten testen können.
Dafür gilt es, eine realistische Stadt zu modellieren, die groß genug ist, um längere Zeit darin herumzufahren, ohne andauernd auf bereits gesehenen Orten zu stoßen. Straßenzüge müssen realistisch modelliert werden, Kreuzungen, Autobahnen und Landstraßen auf-weisen und durch Gebäude ein Bild vermitteln, das eine größtmögliche Immersion in die virtuelle Welt zulässt. Doch fehlt es nun immer noch an Leben. Passanten müssen in großer Zahl die Straßen bevölkern, andere Autos am Verkehr teilnehmen, und neben dem Fahrschüler muss ein Fahrlehrer sitzen, der das Fahrverhalten seines Schülers kommentiert und bei langen Strecken aus dem Nähkästchen plaudert. Kurzum, der Aufwand eine solche Welt händisch zu modellieren wäre immens. Um diesen Herausforderungen entgegen zu treten, werden in der vorliegenden Arbeit Methoden aus der prozeduralen (algorithmischen) Generierung virtueller Welten vorgestellt.
Im ersten Teil werden virtuelle, dreidimensionale Straßennetze anhand fester Knotenpunkte erzeugt. Diese Knotenpunkte können wiederum prozedural oder zufällig erzeugt oder aber aus bestehendem Kartenmaterial, z.B. aus OpenStreetMaps (OSM) importiert werden. Durch eine automatische Erkennung von Kreuzungen ist die Erzeugung flexibel auf beliebige vordefinierte Netze anwendbar. Die Texturen, die für Straßen und Kreuzungen verwendet werden, werden, sofern möglich, aus einfachen Sprites konstruiert oder im Falle eines sehr individuellen Konstrukts während der Erzeugung generiert. Die Möglichkeit, mehrspurige Straßen zu erzeugen, verleiht den virtuellen Städten einen erhöhten Grad an Realismus.
In den Zwischenräumen des Straßennetzes finden sich häufig Wohngebäude, Industriegebiete, Gemeinschaftsplätze oder Agrarland. Sind diese sogenannten Parzellen erst einmal identifiziert, können sie mit eben diesen Objekten versehen werden. In dieser Arbeit fokussieren wir uns auf die Erzeugung von Wohngebäuden. Im zweiten Teil der Arbeit werden begehbare, mehrstöckige Gebäude erzeugt, die sich in das Stadtbild einfügen. Neben dem flexiblen Grundriss wird insbesondere Wert auf eine möglichst glaubhafte Gestaltung der Innenräume gelegt, sowie auf eine Verbindung über einen gemeinsamen Flur oder ein Treppenhaus. Da ein Stadtbild in der Regel unterschiedliche Baufor-

men aufweist, kann die Generierung hinsichtlich Texturierung, Dachgestaltung, Höhe der Gebäude, sowie Fenster- und Türlayout parametrisiert werden. Um die Performance beim Rendering der Stadt zu gewährleisten, hat jedes Gebäude bei Fertigstellung drei Detailstufen. Die niedrigste Stufe bildet lediglich die Außenmauern ab, die höchste die Innenräume samt Treppen, Tür und Fensterrahmen.
Ist die Umgebung in einer Ausprägung geschaffen, die dem Spieler eine gewisse Immersion ermöglicht, gilt es, die Spielwelt mit Leben zu füllen. Im dritten Teil dieser Arbeit wird die prozedurale Erzeugung von Geschichten für Spiele auf Basis von vortrainierten Language Models besprochen. Dabei liegt der Fokus auf einer interaktiven, kontrollierten Spielweise, in der der Spieler mit den Gegenständen, Personen und Orten der Geschichte interagieren und auf die Handlung Einfluss nehmen kann. Aktionen, die unter Verwendung aus vorherigen Abschnitten extrahierten Entitäten generiert werden, sollen das Gefühl vermitteln, dass die Geschichte sorgfältig vorbereitet wurde, und doch die größtmögliche Flexibilität für den Handlungsverlauf ermöglichen. Die kontrollierte Verwendung von Orten, Personen und Gegenständen im Inventar des Spielers erlauben eine Portierung auf eine dreidimensionale Spielwelt sowie das Gameplay in Form eines Textadventures.
Alle Methoden zur Erzeugung digitaler Inhalte, die in dieser Arbeit vorgestellt werden, wurden in vollem Umfang implementiert und hinsichtlich Nutzbarkeit und Performance evaluiert.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Virtual worlds are becoming more and more important in the context of games and simulations. When taking a look at Bethesda's *Skyrim* [1], for example, we can see that the complexity and effort involved in creating a credible and realistic 3D world is a task of several months for a large team of professional designers. But not only the overall visual quality plays an important role in the design process; also, the sheer amount of different objects needed to avoid a repetitive look where objects such as furniture, plants or buildings reappear frequently [2]. As a result, game development studios face three challenges:

- to create a realistic and credible environment within a reasonable timeframe,
- to keep the project within budget in order to make the result affordable for the end consumer,
- to bring in creativity to give the game an individual and innovative appearance.

The renowned game designer Will Wright (*The Sims* [3], *Sim City* [4]) has proven that these challenges exist not only in theory, but also in practice. He called it *The Mountain of Content Problem* [5]. *Procedural generation* is one approach to address these challenges. For years, developers have been inventing methods and patterns to create textures, 3D models, or even entire game levels using algorithms to create a unique looking environment that requires no or very little adjustment. Another reason for the growing interest in procedural content generation (PCG) is to make a game worth playing again by changing levels or quests and offering new stories and impressions in each new session. The game mechanics remain unchanged, so that the player can use the acquired abilities and skills. Examples of games that take advantage of these opportunities are *Elite: Dangerous* [6] or *Minecraft* [7], which create new worlds at the beginning of each game if the player wishes.

The principle of PCG does not serve exclusively the domain of digital games. Board games like *The Settlers of Catan* [8] require players to create a world map by randomly distributing resource fields on the table [9], giving the player a completely new experience in each game. This approach has led to an enormous success of the game.
In addition, PCG has influenced the development of *serious games*, with the emphasis on learning in addition to entertainment as in classic games [10]. In the field of education, changing conditions have the effect that students not only repeat the knowledge or movements they have learned, but also apply their skills to new situations. These situations can be generated randomly with the help of procedural generation mechanisms. Using the example of a virtual driving school, generated road networks can teach players to orient themselves in the real world in an unknown area. Other scenarios for randomly generated environments can easily be derived for serious games in different industries such as transportation, health and marketing.
This work focuses on procedurally generating virtual worlds for games. It contains improvements to existing PCG approaches as well as novel methods with a focus on practical use. The areas of *Procedural Modelling* and *Procedural Storytelling* are particularly highlighted.

1.1 History of Procedural Content Generation

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 exciting sensation was the game *Rogue* [13], which also used algorithms to create levels for this very famous dungeon crawler [14]. The game was released in 1980. The name Rogue served as the eponym for the next generation of Rogue-like games that had the following features:

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

Contrary to expectations, the developers neither chose the generative approach for level creation—in the interest of making the game playable 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, the decision was made to generate the levels on the fly instead of writing them to disk [15].
From a research perspective, Darwin R. Peacheys' paper Solid Texturing, published in 1985, was one of the first publications to explicitly deal with the generation of procedural content [16]. Similar to today's *normal mapping*, Peachey proposed a technique that allowed two-dimensional textures to look three-dimensional. This early paper was followed by various other publications dealing, for example, with terrain generation [17] or the design and animation of plants using fractals [18], and thus the establishment of procedural content generation as a scientific field of research.
The resulting methods were not only used in the digital games industry, but also in animated film. Pixar's animation tool *RenderMan* [19], for instance, offered procedural

Figure 1.1: The game .kkrieger has a file size of 96 kb.

functions to define textures and materials algorithmically or to generate a multitude of simple primitives by handwritten subroutines. With increasing maturity in research and practice, generative techniques have been increasingly used in AAA games (AAA game is a widely used term for games with high development and promotion budgets) such as *Command and Conquer: Red Alert 2* (2000) [20], *Diablo* (1996) [21] or *The Elder Scrolls II: Daggerfall* (1996) [22]. In these games the developers deliberately implemented PCG as a concrete game element, not because of a resource bottleneck like in earlier times.

In 2000, EA Games (Redwood City, CA, USA) released *The Sims* [3], which not only introduced a completely new game concept, but also a highly innovative yet easy-to-use character editor. Players could create a character in the game in the form of a detailed 3D model consisting of customizable body parts, face, clothing, and personality. Using the editor did not mean moving individual nodes and adjusting a UV map (a UV map is the projection of the surface of a 3D model onto a 2D image), as is done in modeling tools such as *Blender* [23]; Maxis designed the editor to offer a set of parameters for editing individual human features, such as waist, size, or eye relief. Appropriate limits support players in designing their characters to ensure a credible and believable look of the result. Many 3D tools implemented a similar workflow to create humanoid models, such as Autodesks (San Rafael, CA, USA) *Character Generator* (formerly known as Autodesk Pinocchio) [24], *Fuse* [25], or the open source *Makehuman* [26] application. We consider this approach as procedural content generation with strong user interaction.

A year later, in 2004, the group .theprodukkt released a simple shooter called *.kkrieger* [27], which originated in the demo scene (see Figure 1.1). The aim of this scene is to create visually or acoustically impressive applications that are often referred to as digital art, because they show simple, moving scenes—similar to a painting [28].

A specific category focuses on demos with a maximum file size of 4, 8 or 64 kb. This small footprint does not allow you to use pre-built assets such as graphics, music or models. Instead, these assets are created procedurally at runtime, or before the demo starts. Considering that rogue games are also determined by memory, there are some parallels. Using *.kkrieger* as an example, the development team .theprodukkt generated animations, levels, textures, shaders and music based on a tool set with process methods. With great interest in the final demo (size 96 kb) the developers decided to publish their tools in the form of the editor *.werkkzeug* [29].

Compton, Osborn and Mateas identify computer graphics as the origin of procedural

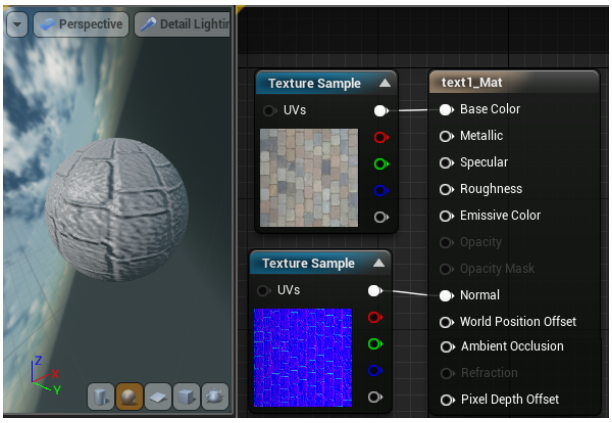


Figure 1.2: The Unreal Engine's (version 4, Epic Games, Inc., Cary, NA, USA) material editor allows for concatenating mathematical operations, textures and shader functions.

generation and explain that the discipline was associated with computer games by a publication of Intel (Santa Clara, CA, USA) [30]. In this particular contribution, generative techniques such as L-systems, Perlin noise or fractals were investigated [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]. Today, most of the utilities discussed have arrived in modern game development tools and facilitate the generation of procedural content. 3D models are constructed by deforming, cutting or merging primitives instead of putting them together triangle by triangle (e.g. in *ZBrush* [33]). This allows developers to focus on the procedural algorithms used to generate believable content rather than on the technical challenges of some lower layers. Examples include the *Unreal Engine 4* [34], *Unity 2017* [35] and *Cry Engine V* [36] game engines, which provide tools to easily manipulate terrain and *paint* vegetation such as trees or flowers on the ground (see Figure 1.3), dramatically speeding up the level creation workflow. This allows developers to focus on finding an algorithm to distribute vegetation objects over the level without worrying about colliding flowers or repeatedly formed trees.
A remarkable achievement of today's game development tools is the procedural generation of textures and materials. Similar to *.theprodukkt*, which was introduced in *.werkkzeug* a few years ago, the procedural generation of textures and materials is now used by most big players in the game engine market. In addition, instead of manually writing shaders (in GLSL (OpenGL Shading Language) or HLSL (High-Level Shader Language)), graphic designers were offered modular tools to construct materials by combining images, mathematical operations, or shader functions, for example to create a normal mapping effect, as shown in Figure 1.2.
The next generation of game engines has begun to implement functions to create even more complex objects. *Epic*, for example, included a simple tool for generating 3D buildings [1].

[1] https://udn.epicgames.com/Three/ProceduralBuildings.html

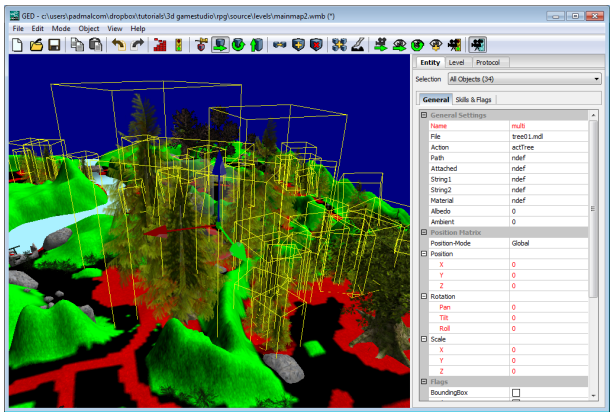


Figure 1.3: Modern editors offer functions to place objects by using brushes or color maps. These methods allow an easy integration of procedural methods.

Some very specific applications like Side Effects Software's (Toronto, ON, Canada) *Houdini* [37] or the *Esri CityEngine* [38] have an even longer experience in building generation, but in the end they still create non-random worlds without a living simulated population.

1.2 Structure of this Thesis

This thesis is structured as follows. After the introduction and the history of PCG, a survey in Chapter 2 summarizes the current state of research in the different sub-areas of procedural content generation and compares them to the production-ready generators available on the game development tool market.
After the survey, Chapter 3 introduces a method for the generation of three-dimensional road networks. It discusses the creation of meshes and the corresponding texturing, as well as detection and modeling of intersections. Chapter 4 picks up the idea of city generation from Chapter 3 and adds procedurally generated buildings to the road network. The method presented here allows creating not only facades and roofs but also accessible rooms, which are equipped with doors and windows and can be reached through a corridor. Different floors are connected by staircases.
Chapter 5 deals with the procedural creation of interactive, non-linear stories to bring the virtual world to life. Language models form the basis of the text generation process and with the help of Natural Language Processing (NLP) methods branched stories are generated, which the player can continue by performing different, pre-calculated actions.
The thesis ends with a conclusion and an outlook on further research topics in Chapter 6.

Chapter 2

Definitions and Related Work

2.1 Definitions

Ruben M. Smelik et al. define the generation of procedural content as "any kind of automatically generated asset based on a limited set of user-defined input parameters" [39]. They also refer to Roden and Parberry [40], who call these types of algorithms *amplification algorithms*, taking a small set of input parameters to transform them into a larger set of output data. Togelius formulates a definition by antithesis, saying that procedurally generated content does not match content generated by users, even if they use procedural algorithms, because they must be parameterized manually [41]. Hendrikx et al. see procedural generation as an alternative to manual design, but emphasize the need for possible parameterization so that designers can influence the generated object [42]. Shaker et al. are more concrete and define PCG using examples of what PCG is (e.g., a software tool for creating random dungeons without user input) and what not (a map editor that allows users to place elements) [43].
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 interest in other communities that arises from interdisciplinarity, and affects both natural sciences and biology [44]. This strong interest in other research areas is an indicator for the presence of the topic. The harmonization of all these disciplines such as biology, architecture, urban studies, psychology, etc. and the search for the right formalisms and data structures is, however, a huge effort. Finkenzeller [45] narrows the affected areas of computer science to:

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

He also points out that programming is the most flexible, yet error-prone method for automatically generating procedural content. Hendrikx et al. introduce the abbreviation PCG-G (procedural content generation for games) [42] to distinguish PCG for games from other areas such as simulations or (animated) movies. This shows that PCG-based methods, algorithms and tools can be applied to a variety of areas such as urban planning (e.g., the Esri CityEngine) or the (animation) movie industry. Pixar is a company that uses procedural content generation in *RenderMan* [46]; furthermore, Disney Research (`https://www.disneyresearch.com/`) provides publications that mention procedural techniques, e.g., for virtual terrain editing [47]. That shows that the benefits of automated content generation are well known in the film industry.

2.1.1 Theoretical Considerations

Computer science often measures the efficiency and maturity of software or algorithms to ensure their quality and applicability. A widely used metric is a simple subjective estimate of the extent to which the generated content looks *realistic*; this is not the case if it can be easily identified by a human observer as automatically generated. We suggest balancing between performance and fidelity in PCG: If an algorithm returns an accurate result (e.g., a natural-looking forest), the algorithm requires more processing power, more memory or more storage to produce more variations of trees, textures in higher resolution, more detailed meshes or a denser planting. Depending on the desired result, the user must choose between performance and realism to achieve the optimum for the given system and the requirements of the virtual world.
Often, the generated objects can either be categorized as handmade or as made by nature. A central question is if one could tell if one or the other category can be created with lesser effort than the other. As an example, the complexity of the tasks to create 3D trees can be compared with the complexity of the tasks to create 3D buildings. If much of the visible content is produced automatically, the order of its creation becomes relevant. Therefore, we compare a highly simplified projection of the earth's natural formation (see Figure 2.1b) with the procedural generation of a virtual world (see Figure 2.1a). 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. When vegetation or mountains are in the way, mankind tends to remove them to build roads or buildings.
Mapping PCG of a virtual world to the simplified nature-like creation of the earth leads to some additional iterations, where forests are cut down to build roads, a terrain is flattened

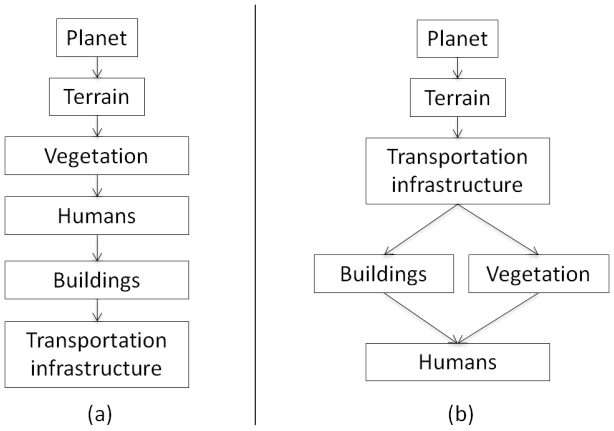


Figure 2.1: The construction of a virtual world can either happen in a simplified natural order (**a**) or in an optimized procedural one (**b**).

to place cities on it, or rivers are redirected to grow a city in the desired direction. As an alternative, we suggest using the order shown in Figure 2.1b to create the terrain, followed by a transportation infrastructure, followed by buildings and vegetation. This procedure will most likely not be in line with the idea of natural growth, but it may facilitate the computer-aided generation of virtual worlds.

2.1.2 Random Number Generators

Random number generators (RNG) can be either a hardware device or software [48]. They generate a deterministic and periodic sequence of (pseudo-)random numbers [49]. Their existence and functionality is assumed in many publications on PCG [32]. Not all researchers agree with their omnipresence in PCG. Some point out that pure random generation would lead to chaos [41]. We believe that the use of randomness depends on the context of each generation process. Using the example of the road network, the placement of roads and intersections does not have to follow a reasonable pattern, but the algorithm for generating road network meshes and textures must be adequately implemented in advance.

2.2 Related Work

We now provide a classification for the types of objects that are most frequently the subject of procedural content generation (see Figure 2.2). 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 [50]. In the remaining chapter, we will present vegetation, water, road networks, buildings, creatures, humans and stories as typical examples for these classes. The work described here was published in [51].

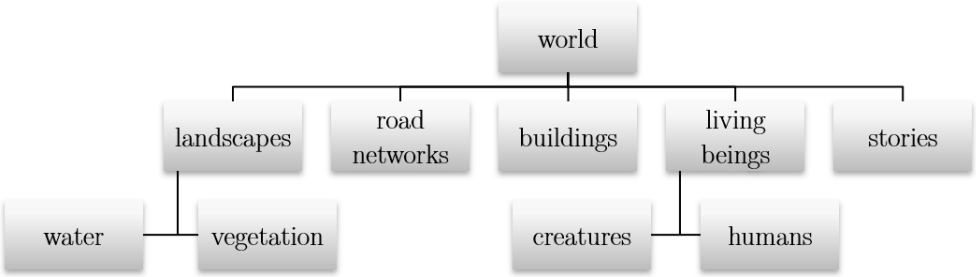


Figure 2.2: Classes of procedurally creatable objects discussed in this chapter.

2.2.1 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.

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 2.3.
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:

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

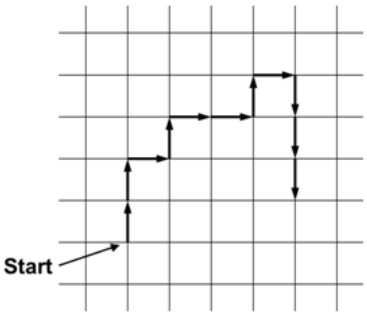


Figure 2.3: A structure produced with the turtle model, position (1,1) is the starting point, initial direction upwards and $\delta = 90°$. The structure corresponds to the word FF-F+F-FF+F-F-FFF.

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:

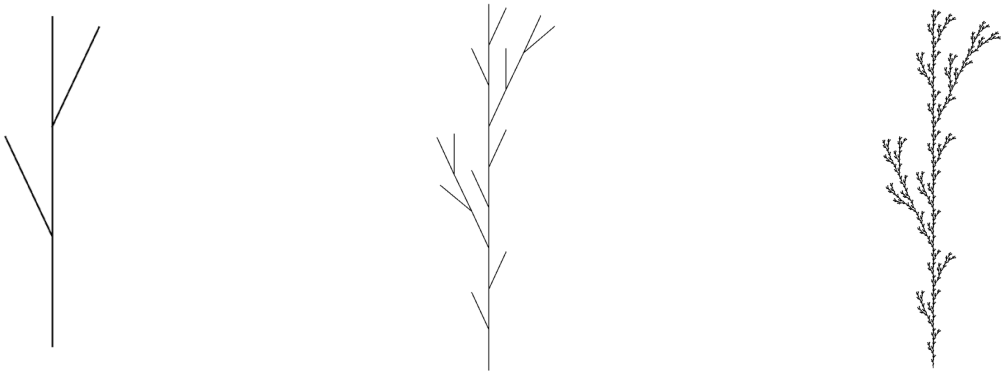
a
ab
aba
abaab
abaababa.
...

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):

n = 1, $\delta = 25°$
F
F→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 2.4a.
We now apply the re-writing rule twice, i.e., we change our production system as follows:

n = 2, $\delta = 25°$
F
F→F[+F]F[-F]F.



(a) (b) (c)

Figure 2.4: Simple plants generated with an L-System with brackets and different rewriting rule iterations. (**a**) one iteration; (**b**) two iterations; (**c**) five iterations.

We then get the plant shown in Figure 2.4b. If we use $n = 5$, we get the realistic plant shown in Figure 2.4c.
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:

$$\begin{gathered} \mathrm{n} = 5,\ \delta = 25^\circ \\ \mathrm{F} \\ \mathrm{p}_1 = 0.33\text{: } \mathrm{F}{\rightarrow}\mathrm{F[+F]F[-F]F} \\ \mathrm{p}_2 = 0.33\text{: } \mathrm{F}{\rightarrow}\mathrm{F[+F]F} \\ \mathrm{p}_3 = 0.34\text{: } \mathrm{F}{\rightarrow}\mathrm{F[-F]F}. \end{gathered}$$

The $\mathrm{p_i}$ are the probabilities for the productions; they add up to 1. This stochastic L-System might produce the plants shown in Figure 2.5. Note that they all seem to be of the same species, just variations at different levels of growth.

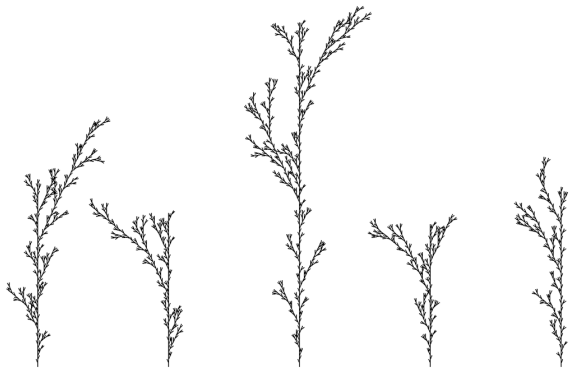


Figure 2.5: A number of plants generated with the stochastic L-System above.

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., [53, 54, 44, 55, 56, 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, 61, 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 2.6). When taking 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.

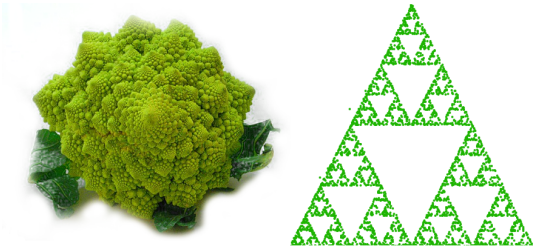


Figure 2.6: Structural similarities between a Romanesco and the Sierpinski triangle (From [66])

In [67], 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, 68, 69] confirms the high maturity level of this research area.

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

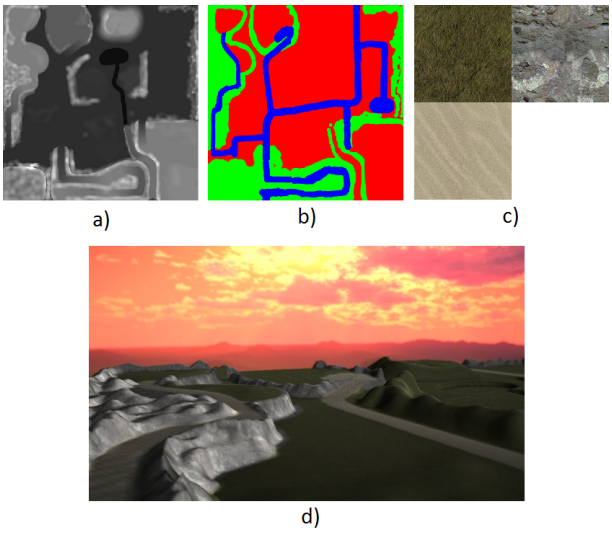


Figure 2.7: Creation of a landscape (**d**) using a height map (**a**), a color map (**b**) and textures (**c**).

of the bitmap's pixel (see Figure 2.7a). The calculated height is then projected to a flat 3D mesh [70]. Usually, the brighter 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 2.7b) that is used in the game or simulation to map a set of textures (see Figure 2.7c) to specific colors (in general by making use of a shader) is a common practice and returns visually impressive results (see Figure 2.7d) but is still a manual process.
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 [71], 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 such as humidity and temperature [72]. Biomes are regions of land, and, depending on

their characteristics, the terrain is formed according to 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 [73],
- midpoint displacement (mostly used to generate 2D landscapes) [74, 75],
- the diamond-square algorithm (adapts midpoint displacement to generate 3D terrains) [76].

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's shape, in contrast to noise algorithms [77]. 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 [78]. 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 [79], 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 [80].
A limitation to common algorithms for terrain generation is the creation of caves [70] or overhangs, which can either be addressed by layered terrains [81] or by voxel terrains [82]. 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 [83]. Boggus and Crawfis make use of 3D models to generate pattern based caves using prefabricated pattern images [84].

Placing Vegetation in a Landscape

Once a set of plants and trees has been created, the question of their 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 [85] 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 [71]. An interesting alternative was presented by Alsweis and Deussen in 2006 [86]. 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 [87]), 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* [88]), 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 an outer radius in which no other distribution point can be placed [89].

Water

Although water as an element is always the same, the creation of rivers, oceans, lakes and waterfalls differs in many ways. Whereas 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 [39]. Another option is to refer to the sea level and assume the presence of water everywhere in the virtual world beneath this predefined height [85]. In contrast, Kahoun proposes a procedure of natural growth by the spreading of the flow of water [78]. This flow then iteratively forms the river courses. Ebert refers to the use of *dilation symmetry* to achieve realistic-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 [39], and it focuses mainly on the shape and course of riverbeds [90]. 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 [91] 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 [92] 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 [93].

2.2.2 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, these two topics are excluded from our discussion.
The generation of road network models is subject to the automated generation of 3d meshes, this will be briefly mentioned here. Automated mesh generation for complex shapes is still seen as a bottleneck in the area of simulation processes of the Finite Element Method [94]. Shewchuk specifies the complexity of mesh generation by naming three main requirements which have to be met when constructing triangular meshes out of arbitrary shapes [95]. These meshes must:

1. conform to the shape of the simulated object
2. be composed of elements (triangles or tetrahedra) of the right size and shape, while avoiding overlapping shapes
3. allow acute angles.

Havemann gives a good overview over the basics of generating meshes and the diverse data types that form a mesh, as well as other possibilities to manipulate an existing mesh [96].

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 [97]. 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 2.8 shows an implementation from the game development framework TUST [98] in which road networks are created by placing streets in a 3D environment.

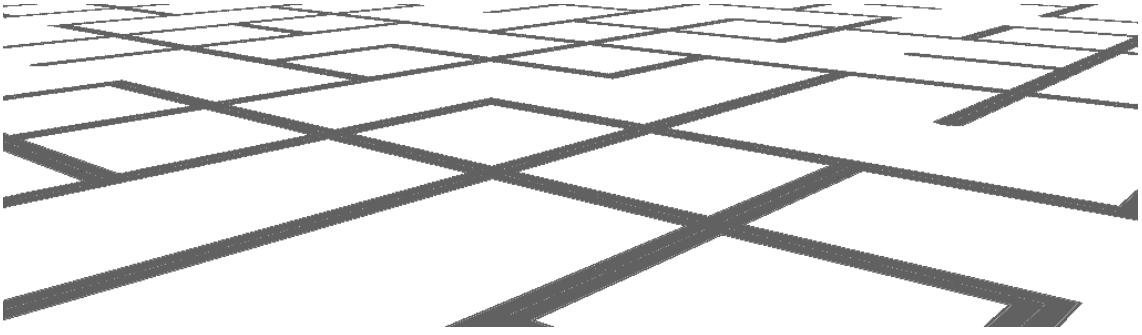


Figure 2.8: Road network based on junction points.

Figure 2.8 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

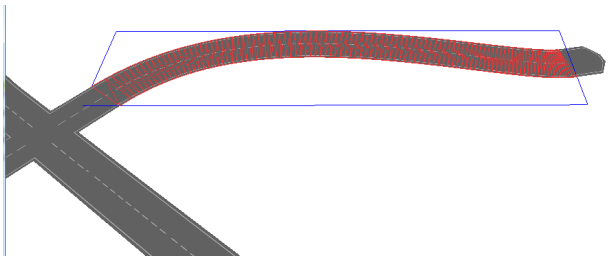


Figure 2.9: Spline-based road mesh generation.

nearby 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 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, a street mesh is generated by expanding a simple spline to both sides, as can be seen in Figure 2.9.
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 [99]. Figure 2.10 shows a single sidewalk adjacent to a street.

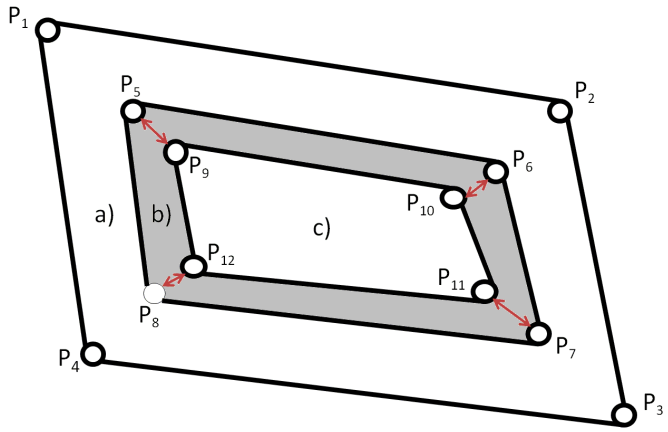


Figure 2.10: Creation of a smaller polygon within a street parcel results in a sidewalk (**a**) street; (**b**) sidewalk (grey area); (**c**) parcel of land.

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 2.11). All these structures span a network that resembles a road network and can be generated algorithmically with little effort. Keeping in mind that a game is not always about creating the most realistic picture possible, but also about creating creative, unpredictable and unexpected worlds, the value of being able to use different patterns for their generation becomes obvious.

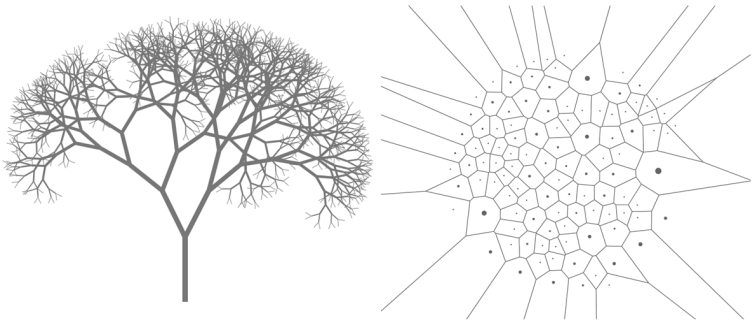


Figure 2.11: Binary tree (**left**) and *Voronoi diagram* (**right**) resemble a road network in their structure.

Furthermore, an explicit search for dedicated street patterns is performed, as shown in Figure 2.12. Parish and Müller [100] propose a procedural system based on L-Systems [101].

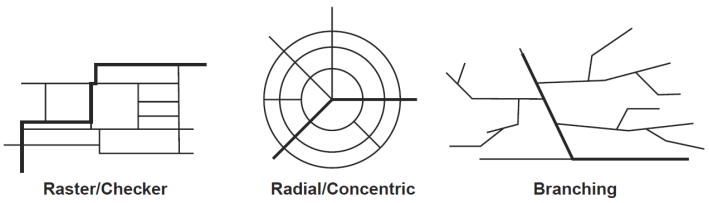


Figure 2.12: Road network pattern (From [100])

One single street grows a complete network which delivers a high quality result but lacks user control and generates networks which need manual adaption. Chen et al. propose an improvement to this limitation based on tensor fields which serve as basis for the generation of one major and several minor street graphs [90]. Making the tensor field as well as the resulting street graph editable, their solution is much more adaptable (see Figure 2.13).

Figure 2.13: Tensor field (left) translated to a road network (right) with a major graph (yellow) and several minor graphs (white) (From [90])

Groenewegen et al. propose a system to generate a city layout which includes the partitioning into diverse districts like residential districts (high class, middle class, working class), industry districts, green spaces or commercial districts. Their eight step process involves terrain generation (1), setting city limits (2), creating highways (3), random candidate location placement (4), district location determination (5), voronoi graph generation (6), noise (7) and street generation (8) [70].

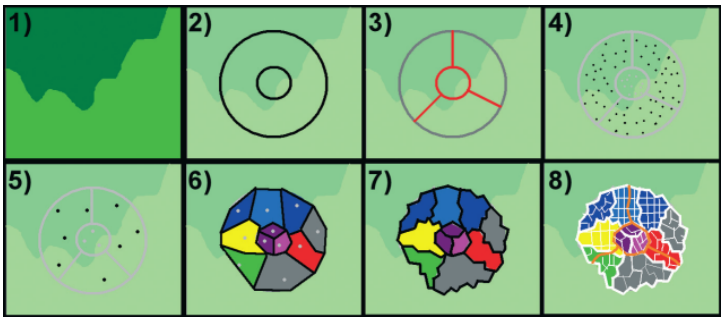


Figure 2.14: Creation of a city by districts (From [70])

As it can be seen in Figure 2.14 the street layout differs from district type to district type and is arranged in two layers - one for highways and one for inner city roads.
Kelly and McCabe introduce a real-time editing solution called Citygen in which a road network is constructed out of a primary road network and a secondary road network [102]. The primary road network is adapted to the surrounding environment which means that the course of each street is calculated depending on e.g. the height of the underlying terrain. Three sampling strategies (minimum elevation, minimum elevation difference, even elevation difference) determine the course from a road within the primary road network. In a second step the cells within the primary road network are determined and used as borders to create a secondary road network within. Kelly and McCabe offer different road network pattern to create this road network within each cell (see Figure 2.15).
Another approach is to break the road network generation down to a few reoccurring shapes like grids or radials [102]. 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 [100] are

Figure 2.15: Raster, industrial and organic road network patterns (From [102]))

a hierarchical and adaptable method that allows for modifying the L-System modules during the road generation process.

The authors of [100] point out that their system is not only applicable to road network generation but also to e.g., buildings. A tile-based system, as shown in Figure 2.16, 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* [103], *Ridge Racer* [104], or *Re-Volt* [105]. Furthermore, more complex street forms such as highways exits, non-standard crossroads and interchange roads are not covered in PCG-related publications.

Cura, Perret and Paparoditis describe an implementation called *StreetGen* which is able to generate road networks based on Geographic Information System (GIS) data. Since GIS data does not contain the shape of intersections, they introduce a method to calculate the streets' borders by rounding the edges, using a circle point at each of the intersections' corners [106].

Galin et al. propose a method for automated road generation on an arbitrary terrain, including mountains, valleys, and rivers. In contrast to Cura, Perret, and Paparoditis, their algorithm merely requires a start point and an end point on the terrain, while the road is generated using a shortest path algorithm. The resulting road may contain turns, tunnels, and bridges to achieve a natural look [107].

Ostadabbas et al. talk about the generation of roads for driving simulations, introducing the Layered Semi-Markov Model. In their work, they give a good overview over the characteristics of a road network, such as road types, lanes, intersections, and buildings. These have been taken into consideration for our implementation [101].

While Huijser et al. write about the generation of rivers in a natural environment, their work is relevant since their approach to generate riverbeds resembles our method to generate roads. Creating a so called *fat curve mesh* along a base curve serves as an area marker to manipulate the terrain below and place vegetation on the meshes borders [90].

In contrast to existing work, we propose a method in Chapter 3 that generates splines for a given set of points that belong to a road. Along these splines a flat mesh is generated and reduced in length at the street's start and end points, so that an intersection can be modeled to connect all of the roads at the intersection point.

Traffic simulation is frequently discussed in the literature [108, 109, 110] but is beyond the scope of this thesis.

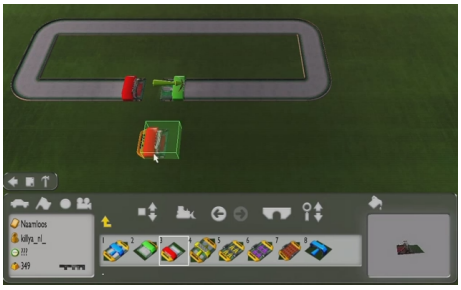


Figure 2.16: Modular road building system

2.2.3 Buildings

The procedural modeling of buildings, as one of the best developed PCG areas [111], is still very present in research and ranges from shape grammars to data-driven, self-learning approaches, or reconstruction of models from photographs and videos of real architecture [112, 113, 114, 115]. Despite rich research, only a few concrete implementations exist [116, 117].
Roughly, most papers can be distinguished by the creation of buildings with and without interior. Non-exclusive sub-research areas are:

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

The combination 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. Hence, we adress it in Chapter 4.
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).

We first shortly present related work on buildings without interior and then address buildings with interior.

Buildings without Interior

Müller et al. present the shape grammar *CGA shape* to create building hulls with an arbitrary level of detail. Their rule-based approach allows consistent mass modeling by a simple adaption of the corresponding grammar [118]. Schwarz and Müller create the successor of *CGA* called *CGA++* that provides more context information about other shapes during the generation and allows editing entire shape trees or even creating new ones [119].
Kelly and Wonka use a procedural extrusion to model high-detail buildings, accepting the building footprint and profile as input. They achive impressive results when it comes to the modeling of roofs [120].
Marvie, Perret and Bouatouch extend an L-System by replacing symbols with functions that can be executed at any time during the generation process. Since these functions can be parameterized, the user sees the influence of each change instantly in the resulting building architecture [121].

Spacial Allocation and Floor Planning

Guo and Li characterize building layout as a major task in architectural design and stress the challenge in optimally shaping, positioning, and scaling rooms in a complex topology with many intermediate relationships [122]. They present an agent-based, topology finding system, working on a regular grid and supporting an evolutionary optimization as proposed by Dillenburger, Braach and Hovestadt [123].

Similar to Guo and Li, the authors Merrel, Schkufza and Koltun classify the layout of architectural spaces as a *spatial allocation problem*, with the purpose of arranging rectangles on a plane or grid, and date its origin as an assisting method for conceptual architectural design to the early 1970s [124].

First attempts to address the *spatial allocation problem* evaluated all possible rectangular layouts (see [125]) with the result of an exponentially growing number of patterns. Even if this first approach was considered infeasible, the grid-based strategies remained the focus of researchers featuring genetic algorithms [126], procedural modelling [127] or integer programming [128].

Lopes et al. even address multi-story buildings involving staircases, and stress the importance to place a staircase room at the same position in each story. In contrast to our work, the automated placement of outer doors and windows is missing, and non-rectangular rooms are not supported.

Mirahmadi and Shami make use of the *Squarified Treemap Algorithm* [129] to equally distribute rectangular rooms over a rectangular floor plan. They rely on the *Squarified Treemap's* advantage compared to the original *Treemap Algorithm* [130] to optimize the rooms' aspect ratios, avoiding long, narrow rooms. Furthermore, they propose to optimize the algorithm to detect corridors that connect all rooms on a floor [131].

As one can see in Figure 2.17, 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 and 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 2.18, 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

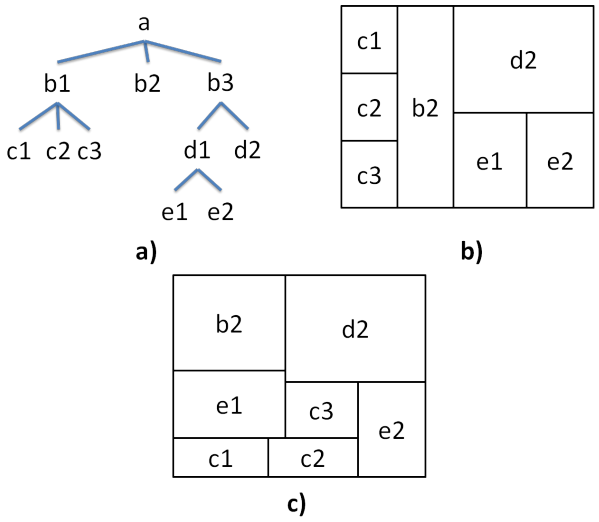


Figure 2.17: Tree structure (**a**); treemap (**b**) and squarified treemap (**c**)

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 [131]. 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 overlapping lines with the rooms to allow the placement of doors.

Another known limitation of the treemap algorithm is that it can only handle rectangular areas. This does not hold for another approach, which can grow a room on any initial area [127]. 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 2.19, 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 2.19, 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, the more accurate is the result for non-rectangular areas.

Melin and Bengtsson, inspired by Lopes et al., evaluate their work to see if a combination of the subdivision-growth method is a viable approach to generate varied and believable tile-based floor plans [132].

Huang and Korf [133] present a method to pack n rectangles to minimize their bounding box, while wasting a minimum of space in the bounding box's inner. Their algorithm calculates all inner rectangles' x-positions first before assigning the y-positions accord-

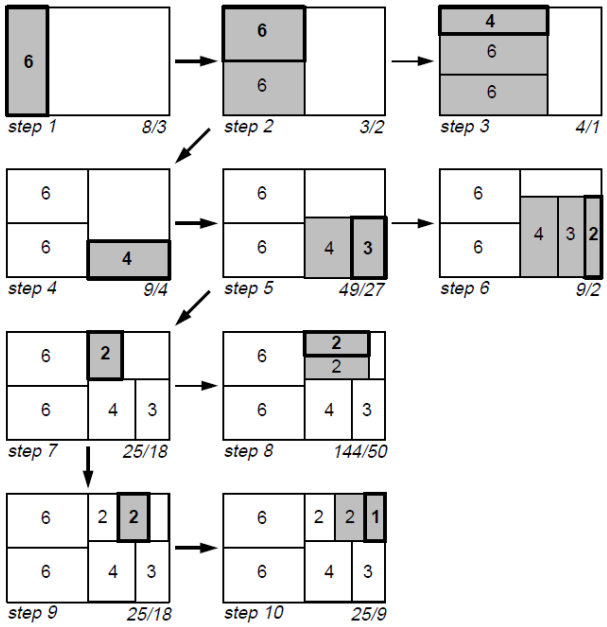


Figure 2.18: The squarified treemap algorithm [129].

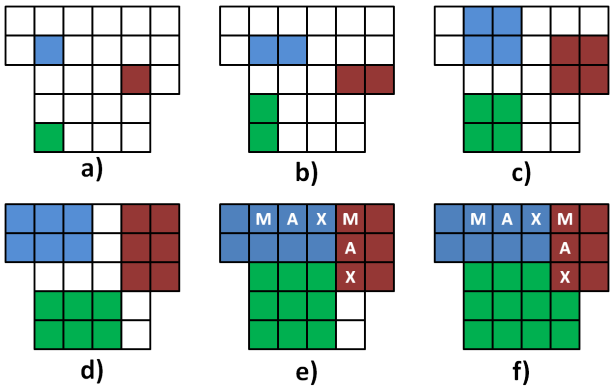


Figure 2.19: The growth algorithm of [127] is capable of populating even non-rectangular areas with rooms in different stages: (**a**) Initial room start cell placement, (**b-e**) growth of rooms, (**f**) connection of remaining cells to adjacent rooms.

ingly. By doing so, they reduce the dimension to one and apply one of Korf's earlier wasted-space pruning algorithms [134].

Although *Rectangle Packing* is rarely mentioned when it comes to floor plan generation, we name it here since we will refer to it later when placing rooms in a given floor plan (see Section 4.3.3). Chen and Chang discuss various structures for packing floor plans such as *O-tree* or *B*-tree*, and compare their capability to represent a floor plan's topology [135].

Related to *Rectangle Packing*, *Dense packing* is reviewed and applied by Koenig and Knecht [136], and finally compared to common subdivision algorithms in terms of speed and reliability. According to their work, the subdivision layout solver has a performance

advantage, whereas dense packing shows advantages when a user interaction is involved.
A possible next step 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 [137], defining the placement of furniture depending on accessibility, symmetry or adaptability to the room's structure.

Grammars

Wonka et al. present a *split grammar* for automated modeling of architecture. Their system supports a hierarchical definition of rules to manipulate a given surface. They can hence create buildings with a high or a low level of detail [138].
Talton et al. claim that procedural methods are difficult to control and present an approach to control grammar-based procedural models. Their algorithm optimizes the product of the grammar, based on a specification provided with the grammar itself [139].
In order to facilitate grammar-based procedural modeling, Patow proposes to visualize grammar rules in a *directed acyclic graph* (DAG) to see all dependencies along the rule chain, and consequently have a more user-friendly representation of the grammar [140].
Edelsbrunner et al. go into detail concerning modeling round geometry and employ different coordinate systems to facilitate structures, such as domes or towers [141].
Regarding the generation of believable facades, Finkenzeller gives a comprehensive overview of the diverse techniques used to generate facades via *L-Systems* or *shape grammars* [45]. He not only explains the creation of doors and windows, but also presents a semantic model for ornaments and wall structures.

Procedural Content Generation via Machine Learning (PCGML)

Summerville et al. address the momentum machine learning gained again in the last years. However, they make it clear that PCGML still requires a lot of work and research until item, character, rule, or 3d level generation are fully explored [142].
The application of machine learning and stochastic optimization to generate accessible residential buildings is examined by Merrel et al. [124]. Using *Bayesian Networks*, the authors train an algorithm using real architectural data. The algorithm fits a given set of user-defined variables (e.g., building size and layout) to a resulting topology of rooms, using their individual size and room adjacencies. By using stochastic optimization, a floor plan is generated and optimized in an arbitrary number of iterations.
Martinovic and Van Gool apply *Bayesian Model Merging*, a technique from Natural Language Processing to labled training facades. The resulting model is used to sample context-free grammars to imitate an existing building style [143].
Chaillou trained three models leveraging Generative Adversarial Neural Networks (GANs) to design, partition and furnish floor plans. Based on pix2pix [144] and a generator / discriminator architecture, the author achieves good results for a wide variety of shapes and claims that neural networks will play a role in the future of building generation [145].

Evaluation of Procedural Generators

Beneš et al. conducted a user study to investigate the realism of procedurally generated buildings and found that the quality of textures, presence of irregularity, and structure plausibility have a positive influence of the perception of building models [146].
Shaker, Smith, and Yannakakis provide a general overview of the evaluation of content generators, with a focus on visualizing the range of generation results, and using questionnaires to learn about the perception of generators for the players [147].
Smith and Whitehead explore the expressive range of a procedural level generator by visualizing its generative space. This visualization allows us to analyze the impact of the diverse parameters of the algorithm [148].
All papers we reviewed presented innovative ideas and contributed to the big picture of methods to procedurally generate virtual building models. However, they do not go into detail when it comes to the actual generation of three-dimensional meshes, and do not name the pitfalls one has to avoid after reviewing the generated architechture such as obstructions, optimal texture choice and placement, and the challenge of generating different levels of detail. We will address those topics in Chapter 4.

Facades

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 [45]. 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 [138]. 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, the extensive use of ornaments and decoration is recommended [149].

2.2.4 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 [150]. We focus here on the creation of static meshes and skins for humans and creatures.

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.

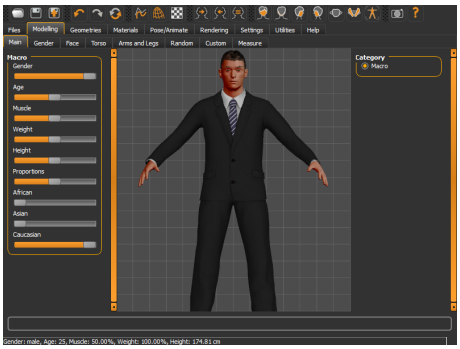


Figure 2.20: MakeHuman (version 1.1.1) allows for creating individual characters by parameterization of a base model.

There exist tools to create and customize a generic character model, like Autodesk's *Character Generator* [24], *MakeHuman* [26] (see Figure 2.20) or *Unity Multipurpose Avatar (UMA)* [151]. The use of a framework acting in that 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).

Creatures

As already mentioned, research on the generation of creatures is few and far between. An outstanding game called *Spore* [152] 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 in which generalized motion data could be applied during runtime to achieve an unexpected but realistic-looking animation [153]. Hudson [154] 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.

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 the 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 [155] 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 [156].
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* [157] 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 [158]. 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 [156]:

- 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 [159]. 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 [160]. 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 [161]. 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.

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 [162]. One of the easier and vivid ways to address it is by using Finite State Machines (FSM) [163], known from classical computer science. Figure 2.21 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.

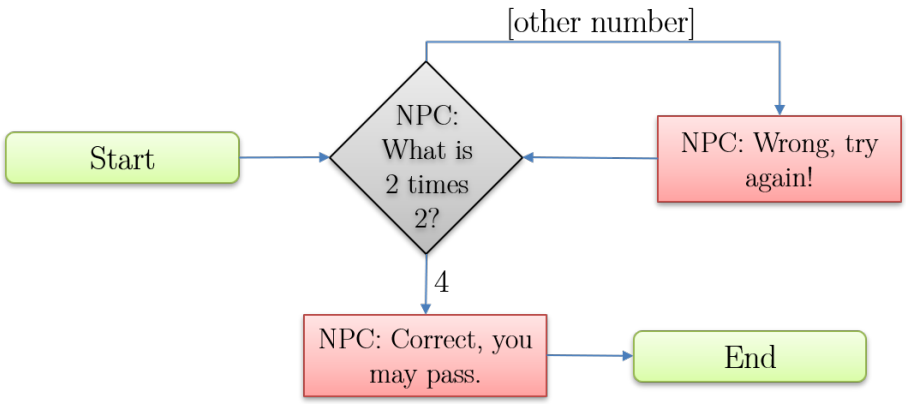


Figure 2.21: Flow chart illustrating a simple dialog.

This idea can be enhanced by the aspects of personality, relations and moods [164]. 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 [1]. 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 2.22 for the projection of moods on an NPC's model).

Figure 2.22: 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 approriate. 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 [165]. In this 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

[1] https://www.nevigo.com/en/articydraft/overview/

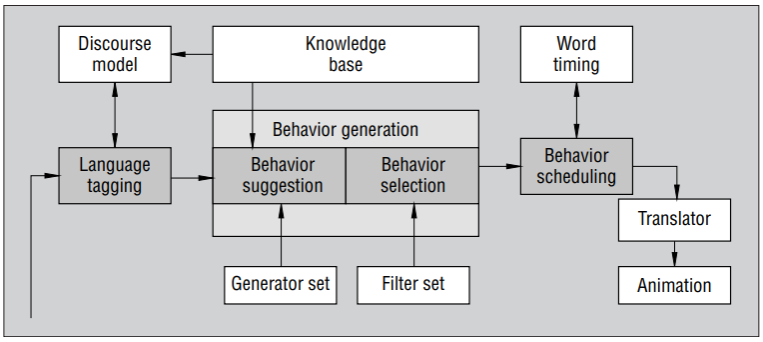


Figure 2.23: Behavior Expression Animation Toolkit (BEAT) converts text to nonverbal behavior [167].

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 [166] (see Figure 2.23).
In [167], Gratch et al. go more into detail and show how BEAT annotates text with hand gestures, eye and eyebrow movement and intonation, and they schedule 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 [168]. 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) [169]. 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.

2.2.5 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 players. This is especially true for roleplaying games. The players typically have 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 [170].
The procedural generation of stories has many advantages [147, 171, 172]. 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 [173]. Second, when new story variations are created at runtime, this can keep players motivated when replaying the game [172, 174]. Third, it becomes possible

to generate stories that automatically adapt to the players' skills and preferences [175]. Yao et al. describe the story generation process as a two-step process, planning a sequence of events (story planning) and formulating the actual text that the player sees (surface realization) [176]. In fact, the complexity of both steps is divided into many sub-processes, especially with adaptive approaches. *Façade*, as an example, is a system that builds dramatic tension by concatenating events and by reacting carefully on the player's actions - both in a very fine-grained way (*beats* as the autors call it) [177].

Grammars

One of the most obvious approaches is to use prefabricated grammars, often in combination with manually generated building blocks (templates). A famous representative is *Tracery* by Compton et al. [178] (see Listing 2.1) Its core element is a list of production rules, which create sentences by substitution of symbols. The concatenation of those sentences results in a story. The advantage of this approach is that the computer has maximum control over the text to be generated. The disadvantage is also obvious: Creating templates and complex rules is time-consuming. Furthermore, simulated creativity can result in a creativity that differs from what a human would expect [179].

Listing 2.1: Example *tracery* grammar that generates sentences such as "Darcy traveled with her pet unicorn. Darcy was never impassioned, for the unicorn was always too indignant."

```
{
"name": ["Arjun","Yuuma","Darcy"],
"animal": ["unicorn","raven","sparrow"],
"mood": ["vexed","indignant","impassioned"],
"story": ["#hero# traveled with her pet #pet#. #hero# was never #
    mood#, for the #pet# was always too #mood#."],
"origin": ["#[hero:#name#][pet:#animal#]story#"]
}
```

Planning Algorithms

Grammars can (but do not have to) go hand in hand with planning algorithms. Those generate stories under the assumption that stories consist of a sequence of actions that work towards a pre-defined goal (see Chapter 7 in [147]). 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* [180]. 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 state of the story.

Let us 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) ∧ location(bobsShack) ∧ location(royalPalace) ∧ At(bob, bobsShack),

Goal State Representation:
At(bob, royalPalace),

Action Representation:

1. Action(walk(p:person, from:location, to:location)
   Precondition: At(p, from) ∧ random = 1
   Effect: ¬At(p, from) ∧ At(p, to)
   ),
2. Action(fly(p:person, from:location, to:location)
   Precondition: At(p, from) ∧ random = 2
   Effect: ¬At(p, from) ∧ At(p, to)
   ),
3. Action(telePort(p:person, from:location, to:location)
   Precondition: At(p, from) ∧ random = 3
   Effect: ¬At(p, from) ∧ 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 [180]. A disadvantage of using planners (often based on the Planning Domain Definition Language (PDDL) standard) is that a planner works strictly towards the given goal. An interactive story, on the other hand, can also be worth playing if it contains detours and setbacks or if the protagonist has to completely reorient herself. All these events speak against the nature of a planner, originally designed for optimization. On the other hand, a planner can easily evaluate whether the goal of a story can be reached under the given conditions.

Petri Nets

Petri nets are a well known extension 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

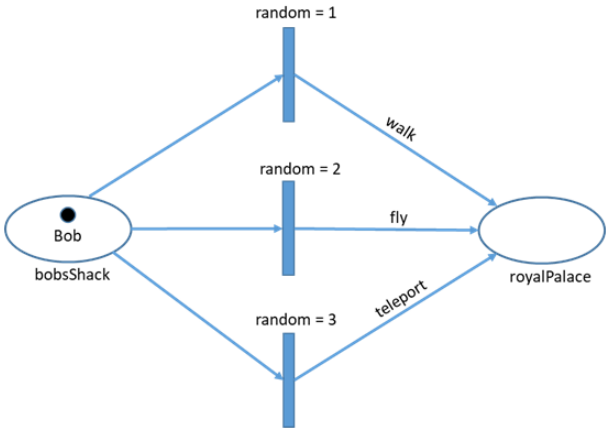


Figure 2.24: Petri net for the shack-to-palace example.

- 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 M_0. 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 [181], 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 [182]. The Petri net describing our example from above might look as in Figure 2.24. 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 [182], 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 2.24 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.
A different use of Petri nets is described in [183]. 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* [184]). A major drawback of their approach is that they lose the power of formal verification for standard Petri nets because they introduce new constructs.

StoryTec

A less formal model is the basis of StoryTec, a system for the specification of story-based games [171]. 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 System

A powerful StoryTec editor supports easy graphical editing of both the structure model and the logic model. Figure 2.25 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 of the screen 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 of the screen 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

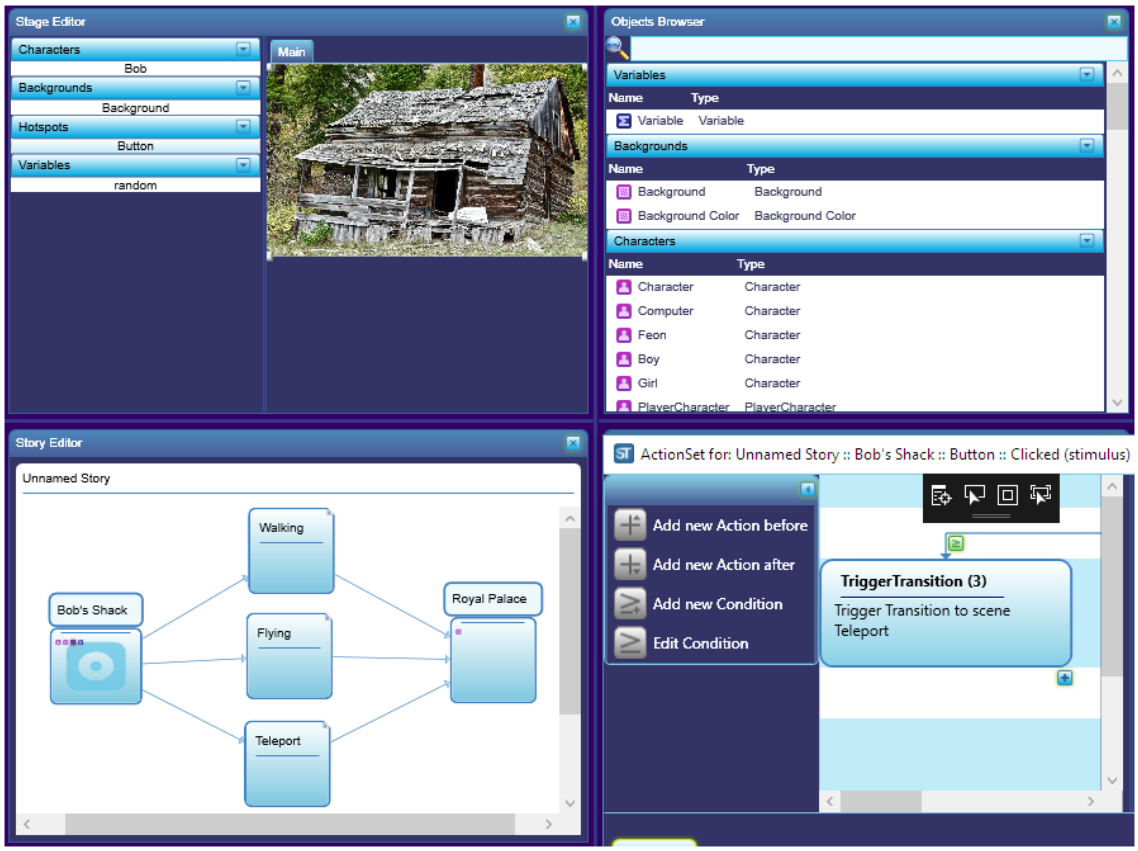


Figure 2.25: The StoryTec editor with our example.

ICML file (INSCAPE Communication Markup Language) file, similar to an XML file. Listing 2.2 shows an extract from this file for our example. 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 2.26.

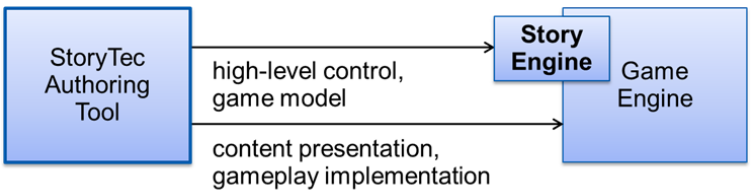


Figure 2.26: The story engine, the runtime of StoryTec (adapted from [171]).

Listing 2.2: 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>
```

Neural Networks

Neural network based story generators were frequently discussed in the last years and also operate in the domains *story planning* and *surface realization*. L. J. Martin et al. trained two neural networks, *event2event* and *event2sentence*, to generate chains of events maintaining the semantic meaning of the story and translating the event chain to readable and understandable texts [185].
Yao et al. propose a hierarchical generation framework that is not limited to any domain. After generating the storyline first, it extends this storyline in a second step to a story. The interesting part is that they compare a full storyline planning and a subsequent story generation to a partial storyline planning with an interlacing story generation [176]. Their comparison shows that the full sotryline planner produces more coherent

and relevant stories.
Fan, Lewis, and Dauphin show a four-step coarse-to-fine model which returns a structured action plan to an arbitrary story prompt. The action plan containing placeholders for entities is assembled to a story, and placeholders are replaced by specific references [186].
A holistic approach to narrative continuation is presented by Roemmele. Besides traditional NLP techniques, she proposes to use neural networks to solve various tasks including story ending prediction, and she presents an automated evaluation of stories based on specific linguistic metrics [187].
Cychosz et al. created DINE (Data-driven Interactive Narrative Engine), an authoring platform for interactive fiction primarily addressing the authors' tasks, instead of the players'. Their approach resembles our method but instead of generating new passages, DINE classifies the user input and maps it to professionally, pre-authored texts [188].
Ammanabrolu et al. use language models and Markov chains to create quests in text adventure games and compare both approaches. By taking the cooking domain as example, they used an ingredient knowledge graph to train models to select and mix ingredients. They conclude that Markov chains generate more suprising quests whereas the language model requires less domain knowledge and can therefore be applied to other quest domains [189].
Angela Fan et al. explore the generation of game environments by machine learning using crowd-sourced data from the multiplayer text adventure game environment *LIGHT* [190]. They show how to construct cohesive arrangements of locations, characters, and objects and emphasize the diversity and variety of their results [191].
Hybrid approaches of predefined and dynamic content have been explored as well. Mendonça and Ziviani propose an approach that generates stories based on a pre-defined event network and a dynamically generated *social graph* [192].
In Chapter 5 we will address possible shortcomings regarding the creativity in grammars and planning algorithms by statistical language models. Such models provide information about the probability of the correlations of word sequences. In recent years language models based on neural networks have appeared, such as Google's *Bidirectional Encoder Representations from Transformers* (BERT), Facebook's *Cross-lingual Language Model Pretraining* (XLM), or OpenAI's *Generative Pretrained Transformer* (GPT and GPT-2).

Based on *GPT-2*, created by Radford et al. [193] and trained and published by Hugging Face [194], Whitmore recently released a text adventure using GPT-2's strength to predict the next token(s) in a sentence to continue a story [195]. The fact that previously played passages form the input text for the following paragraphs creates a continuing context, which is a key element in creating procedural stories [196]. We base our work on *GPT-2 Adventure* and extend it as it will be discussed in Chapter 5. Tambwekar et al. note the lack of player guidance when using language models for story generation and present a reward-shaping system which re-trains a language model in order to reach a given goal [197].
Language models calculate the probability of word or character sequences $p(x)=p(x_1,x_2,...,x_n)$, frequently in an autoregressive manner

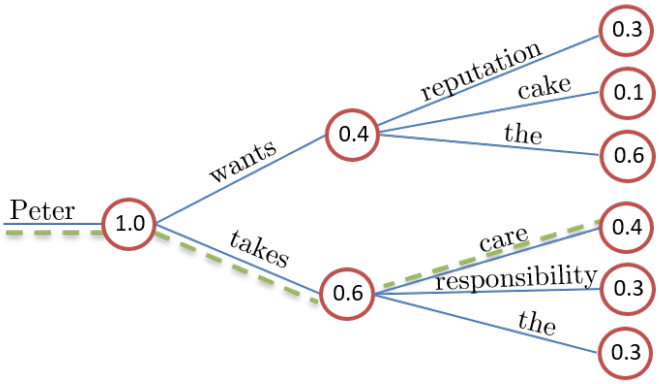


Figure 2.27: Decoding methods (here Greedy Search) of neural language models allow the unlimited generation of text based on the probability distribution of word sequences.

The creation of story generation systems is a long-standing field of procedural content generation (PCG). A story generation system is designed to generate coherent, credible, and dramatically meaningful narratives. A

Input symbols $s_1,\ldots,s_{n-k-1}$

"Story" system uses storytelling elements designed to drive a particular purpose, and is the most popular type of PCG. There are also "Story Templates" which use procedural story generation techniques to achieve a particular goal. It is recommended to use a story generation system for all your PCG content. The system used by our story generation system is a system called the K-Pole, which is described in detail on its page on the wiki. The system can be defined by the type of elements used in narrative, or the type of gameplay elements used.

Sampled output symbols $s_{n-k},\ldots,s_n$

Figure 2.28: The introduction of this paper as generated by a GPT-2 model.

$p(x)=p(x_1)*p(x_2|x_1)*p(x_3|x_2,x_1)$. The training process is unsupervised and allows the use of very large amounts of data.Thus the model can easily calculate the probability $p(output|input)$ of a following sequence under the assumption of a certain input sequence (see Figure 2.27). The application of models trained in this way helps with various tasks, such as text translation, summaries, question answering, and text generation (see Figure 2.28). Chapter 5 will show how a hybrid approach of templating, language

models and related NLP techniques can lead to new, interesting interactive narratives. It will also discuss how the weaknesses of the respective approaches (predictability, limited domain and uncontrollability) can be circumvented.

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 [173]. 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 [173] and in [147]. The following chapter goes into detail regarding the procedural generation of road networks and introduces a method to generate three-dimensional meshes for streets and intersections and the corresponding textures.

Chapter 3

Procedural Road Network Generation

Many of today's video games take place in a virtual world consisting of cities, which are crisscrossed by road networks. Video games are not the only medium to rely heavily on those virtual urban structures, but also simulations such as digital driving schools, or applications for city planning. To generate realistic road networks, we present a method that stresses a robust implementation, capable of creating arbitrary constellations of streets and intersections. The algorithm takes a list of streets as arguments, generates the corresponding meshes for the streets and intersections, and places them into a level instance. To achieve a realistic look we make use of splines to generate curvy streets. The intersection generation algorithm is based on a backtracking approach that allows the generation of an intersection to take up the minimum amount of space. The implementation of a simple square function algorithm helps to generate rounded edges for the intersections. We include a simple algorithm to calculate a street texture, with markers on the borders of the generated intersections to make them appear more realistic. With our results, we enrich research results from various authors dealing with the description of algorithms to generate road network patterns (Manhattan, Voronoi, Organic, Radial, etc.) [97, 102, 100]. Our approach allows the translation of those patterns into flat meshes. Keeping in mind that network structures can not only be found in a man-made transport infrastructure, our results can also be slightly modified to visualize, for example, electric circuits, data flows, or graph databases [198].

3.1 Mesh Generation and Texturing Techniques

The basic requirements of dynamic mesh generation and texturing are explained in this section to support the understanding of our road network generation algorithm and the following building generation approach in Chapter 4. These requirements can be split into the four topics

- mesh generation,
- UV mapping,
- cubic splines,
- and triangulation.

Since intersections and roads are visualized as textured meshes we begin with an explanation of mesh generation techniques.

3.1.1 Mesh Generation

Polygon meshes - defined by a list of connected surfaces - are responsible for shape, surface and detection of collisions of objects in a virtual world. Meshes traditionally consist of triangles since triangles can be used to create any shape (see Figure 3.1), and commercial graphics-acceleration hardware is designed for triangle rasterization [94].

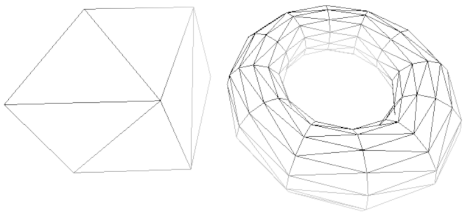


Figure 3.1: Two triangle-based meshes in the shape of a cube and a donut.

McShaffry et al. explain that triangles are made of three vertices which are points in 3d space. Those triangles do not have a height so that they are either visible from the front or from the back. As consequence, ordering the three vertices clockwise or counterclockwise defines the side that is visible. Since some points may occur more than once one has the opportunity to additionally use indexes on the list of vertices as a reference [95] (see Figure 3.2). Storing a referencing integer is cheaper than storing a new vertex which normally consists of three floats (e.g., x=0.0f, y=0.0f, z=0.0f).

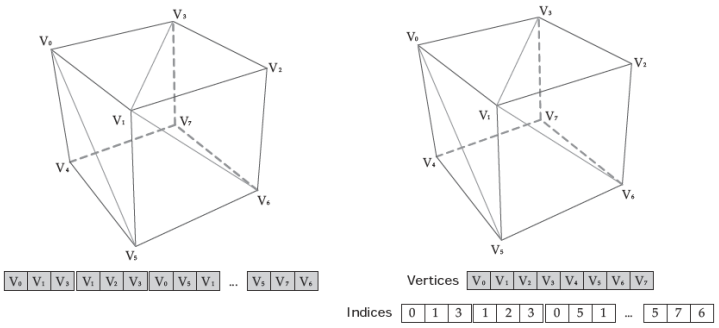


Figure 3.2: Vertex list and indexed vertex list (from [94])

3.1.2 UV Mapping

Once a mesh is created by defining vertices and indices it comes to applying textures to it. A common challenge is to find a way to map a 2d texture to a 3d object. Here comes a technique into play that is called UV mapping. UV is not an abbreviation but stands for a pair of coordinates, u and v. They are commonly represented by two floats (e.g., u=0.0f, v=1.0f) where 0.0f, 0.0f represents the top left corner of a texture bitmap and 1.0f, 1.0f the bottom right corner. A two dimensional UV coordinate is assigned to each vertex in the list of vertices [95] and hence define which part of a texture is mapped to which area of a 3d mesh. A common practice to explain UV mapping is to imagine a cube with six sides. By unfolding this cube we receive a flat object consisting of six squares. Each vertex of the cube can then be enriched with UV coordinates that fit the flat texture to the cube (see Figure 3.3).

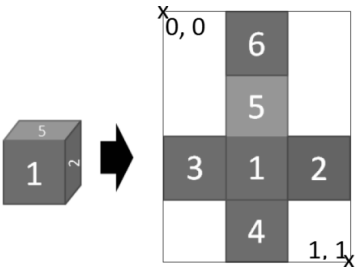


Figure 3.3: UV mapping explained by unfolding a cube and mapping its faces to a texture.

Rectangle #6 would e.g. be defined by the UV coordinates (in clockwise order starting with the top left corner):

- (1/3, 0)
- (2/3, 0)
- (1/3, 1/4)
- (2/3, 1/4)

Areas on a texture can be used multiple times as we will show when generating roads.

3.1.3 Splines

Spline algorithms are frequently used in computer graphics to interpolate between n points with the goal to calculate a curvy line. This is achieved by connecting (or approximating) the given points by *n - 1* polynomial functions blending seamlessly.
There are various forms of splines. One of the most public ones is the *cubic spline* which can be calculated for an arbitrary dimension [106] (see Figure 3.4). The name *cubic* addresses the fact that each of the splines' polynomial segments have a degree of 3. As an extension to the *cubic spline* the *cubic hermite spline* is also defined by a third-degree polynomial between two following points n_k and n_{k+1}. Each segment of the *cubic hermite*

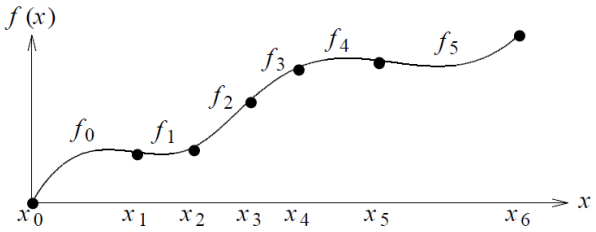


Figure 3.4: A cubic spline along n=7 data points (From [96]).

spline ends where the next begins, and tangent vectors are the same for the current and the following segment at that point. *Bezier splines* are splines which are created out of n - 1 Bezier curves. Those curves use so-called control points that do not necessarily touch the calculated curves. Bezier splines were created between 1960 and 1970 by Paul de Casteljau and Pierre Bézier to be used in the automotive sector [106]. Changing one control point in a Bezier spline has an effect on the entire curve which is often seen as a disadvantage of this method [101]. So-called *B(asis)-splines* are splines which allow an arbitrary degree for the polynomial segments (hence a cubic spline can also be a *B-spline*). Furthermore, changing one control point in a *B-spline* only has a local effect on the touching segment(s) and does not change the course of the entire curves; this allows an enhanced handling and modeling [90]. The difference between a *B-spline* and a *Bezier spline* can be seen in Figure 3.5.

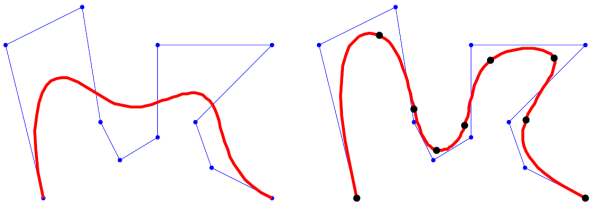


Figure 3.5: Comparison of a Bezier spline (left) and a B-spline (right), which consists of several joined Bezier splines. Their sections are represented by the black dots.

In computer visualization splines support the developers to implement a smooth movement of objects (e.g., a camera tracking shot) along a given path [106] or bending models like plants or pipes in a given form. Some game engines like the *Unreal Engine 4* or *Acknex A8* support developers with prefabricated spline components [101, 107]. In computer graphics spline-based interpolation is used, for example, for image magnification. Its application returns - compared to other common methods - the best trade-off between accuracy and computation effort [107].

3.1.4 Triangulation

Triangulation algorithms as the fourth part of this section are responsible for generating a triangular mesh out of a polygon whose boundary is a closed loop of straight edges [199]. In contrast to tessellation, which is a frequently mentioned technique when it comes to

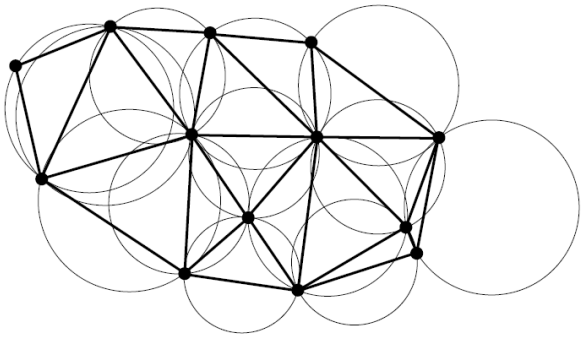


Figure 3.6: No point P is inside any triangle's circumcircle (From [199]).

add detail to an existing 3d mesh in real time on a GPU [200], triangulation is meant to only produce triangles whereas tessellation is meant to subdivide an existing surface into several smaller parts of an arbitrary form (e.g., squares or hexagons) without gaps [200]. Already in the year 1934 Delaunay introduced his work on what became later the famous Delaunay triangulation [201]. Such a Delaunay triangulation is a set of points $P = \{p_1, \ldots, p_n\}, N \geq 3$ which are neither colinear, nor are four points cocircular, and all triangles of the triangulation have an empty circumcircle [199] (see Figure 3.6).

Delaunay could not have had a clue on the impact of his research to the modern computer graphic and all the algorithms that followed his definition. Su and Drysdale found five different categories for algorithm, to generate a Delaunay triangulation:

- divide-and-conquer [202],
- sweep line [203],
- incremental [204],
- growing a triangle at a time in a manner similar to gift wrapping algorithms for convex hulls [205],
- lifting the sites into three dimensions and computing their convex hull [206].

We will make use of one of the most famous libraries *poly2tri* implemented by Wu Liang which is based on sweep line [207].

3.2 Data Model

Before we begin to describe our actual algorithms we present the data model behind the road network generation to provide a better understanding of the objects used during the generation process (see Figure 3.7).

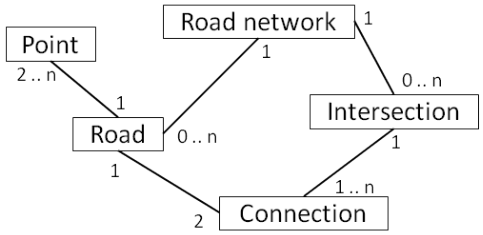
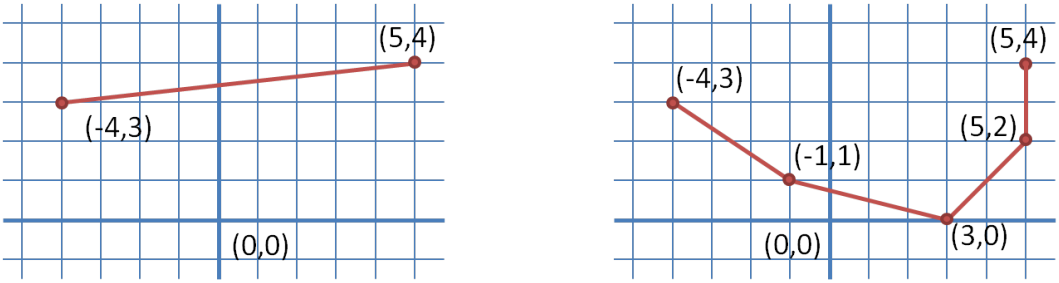


Figure 3.7: Data model of the road network generation algorithm

A road network typically consists of several roads and intersections. A road is defined by a start point and an end point, and it may contain n points in between to shape its course. Each street has exactly two intersections, one at its beginning and one at its end; we handle dead-ends as intersections to be able to generate extra features, such as turning circles. For each new intersection that is established in the road network (see algorithm 1), we add a connection to this particular intersection. A connection is an abstract, non-visible type that defines the spot (and angle) where the road connects to the intersection. Hence, each intersection has at least one connection.

3.3 Placing Streets

We designed our procedure to be as dynamic, simple and adaptive as possible. Thus, the placement of a single road can be done by setting a start and an end coordinate in 2d space, as seen in Figure 3.8a.



(a) (b)

Figure 3.8: Definition of a simple street starting at (-4,3) and ending at (5,4) (**a**) and definition of a street with multiple points (**b**)

To enable users to define serpentines among a mountain's slopes, or other winding courses along rivers or valleys, streets can contain 0 .. n points between the start point and the end point (see Figure 3.8b).

We refer to each line as a segment. After giving an explanation of how intersections are defined based on a streets' points, we follow up with the generation of winding roads.

3.4 Defining Intersections

Intersections are detected by comparing the points of all streets in a road network for similarity. The comparison is made before a spline is created out of the non-detailed street points, as described in Section 3.3 above, for reasons of performance. In a two-dimensional road network there are two possibilities to determine if two roads intersect (see Figure 3.9).

- *Full Segment Scan*: Calculate the intersection of each road segment, as defined in Section 3.3.
- *Endpoints Only Scan*: Check if either the end or start points of two roads match.

Both options are not exclusively applicable, but can be combined. Although a combination may achieve the best result (meaning the most complete result), the application of a *Full Segment Scan* prevents the generation of 3d road networks, or at least makes their generation more complex. This is because two crossing streets in 3d space do not explicitly form an intersection; they may have a different height, so that one street is bridging the other. If the *Endpoint Only Scan* is applied to the list of streets, it is the responsibility of the user to construct the intersections by letting a street start and end at a certain point. Streets that are accidentally crossing are not recognized as such (see Figure 3.9).

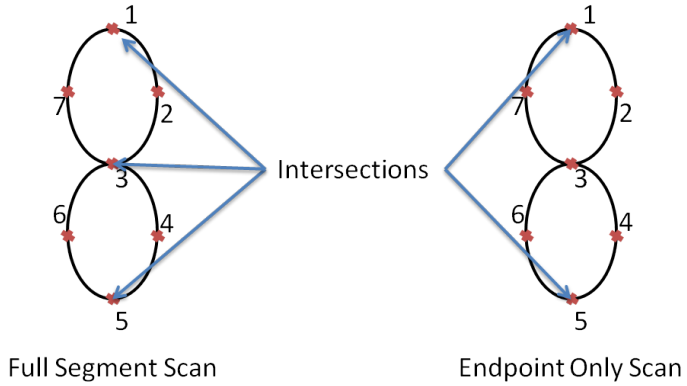


Figure 3.9: Two approaches to define intersections in a road network

We want to leave the intersection placement method entirely to the user, and therefore we use the *Endpoint Only Scan*. The algorithm for the scan looks as follows.
To begin, an empty list of intersections is created which will store all intersections found by the algorithm. The first and last point of each street is then compared to each intersection's position. If the positions of the current street's start is not equal to the position of any intersection in the list, a new intersection is created and initialized with the street's start point. Furthermore, the intersection is enriched with an object that describes the connection properties (we call it a *connection* (see Figure 3.7)). This object stores the following information:

Algorithm 1: Endpoint Only Scan

```
intersections ← empty list;
foreach street s1 in streets do
    start ← s1.getFirstPoint;
    end ← s1.getLastPoint;
    foundStart ← false;
    foundEnd ← false;
    foreach intersection i1 in intersections do
        if i1.position==start then
            i1.connectedStreets.add(s1, START, s1.directionStart);
            foundStart ← true;
        end
        if i1.position==end then
            i1.connectedStreets.add(s1, END, s1.directionEnd);
            foundEnd ← true;
        end
    end
    if foundStart==false then
        intersections.add(s1,START,s1.directionStart);
    end
    if foundEnd==false then
        intersections.add(s1,END,s1.directionEnd);
    end
end
```

- a flag to show if the connection is bound to the street's start or end,
- the street's direction at both ends (see Section 3.5, Figure 3.10),
- the position of the left and right vertex at the street's start or end.

If the position of the street's start in the list of intersections is found, a new *connection* containing the street's information is attached to the corresponding intersection. The same check is then carried out for the street's end. When all intersections have been found, we propose to compare the distance of the intersections to each other. Intersections that are closer to each other than n units should be merged to one single intersection. This procedure helps to avoid intersections from overlapping in the final road network.

3.5 Spline-Based Road Generation

The goal of the procedure to generate a single road is to calculate indices, vertices, and UV coordinates to construct a mesh. Normals are defined by the indices' order forming a triangle, as it has become common practice. The basis for the mesh generation is a

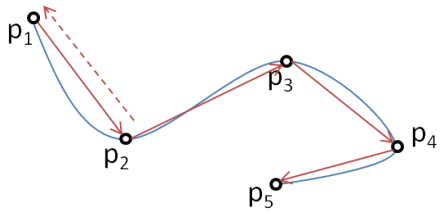


Figure 3.10: Calculating directions from each point to the next point on a spline

spline, which is calculated stepwise along the street points. In addition to the position on the spline, we calculate the direction from the current point to the next point. This is achieved by subtracting p_n - p_{n-1}. If the point is the first one on the spline, the successive point is taken to determine the direction by subtracting p_1 - p_2 and inverting the resulting vector (see the dotted line in Figure 3.10).
When the points and directions are set, an offset is generated to the left and right of each point.

Algorithm 2: Offset generation for left and right street borders

```
ang ← direction as angle;
offset ← offset vector;
w ← width of a lane;
l ← number of lanes;
x ← 0.0;
y ← 0.5 * w * l;
foreach point p in points do
    ang ← atan(p.dir.x / p.dir.y);
    offset.x ← x * cos(ang) - y * sin(ang);
    offset.y ← x * sin(ang) + y * cos(ang);
    p.offsetLeft ← p + offset;
    p.offsetRight ← p - offset;
end
```

Algorithm 2 traverses all points on the spline. By setting x and y, we initially define where the offset should be placed. Since the offset will later on be used to define the street border, we have to take the street width w and the number of lanes l into account. In a loop, x and y are rotated towards the look direction of the current point, and are respectively added or subtracted as an offset to the current point, resulting in a left and right vector (see Figure 3.11).
Beginning with point p_2, we can now take the offset of the current point and the previous offsets p_{n-1} to create a pair of triangles. For the first two points, those triangles are:

- triangle 1: (p_1o_2, p_2o_1, p_2o_2)
- triangle 2: (p_1o_1, p_2o_1, p_1o_2)

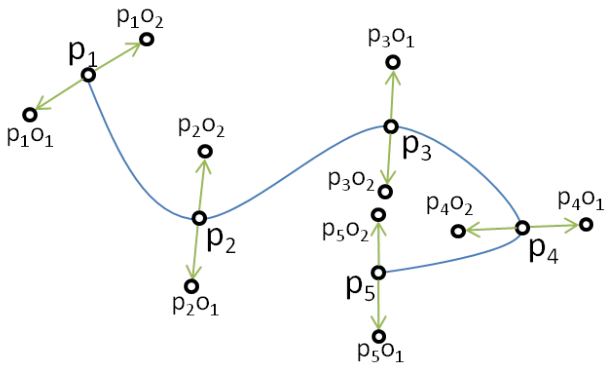


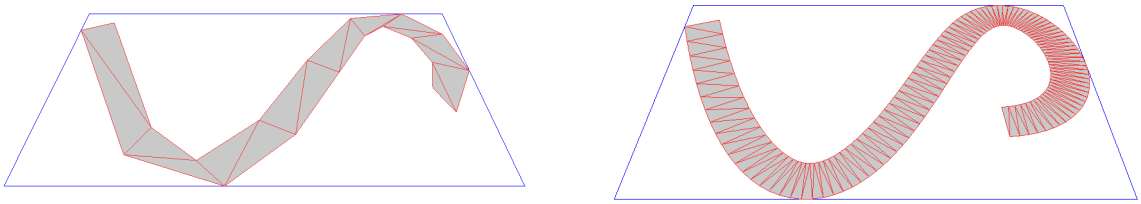
Figure 3.11: Spline with offsets for each point

Table 3.1: Vertices, indices, and triangles created in each processing step

	step 1	step 2	step3
vertex	p_1o_1, p_1o_2	p_2o_1,p_2o_2	p_3o_1,p_3o_2
index	0, 1	2, 3	4, 5
triangle	-	[0,1,2], [1,3,2]	[2,3,4], [3,5,4]

Both offset vectors are pushed to the vertex buffer. The indices 0, 1 and 2 and 1, 3 and 2 are added to the index buffer to generate two triangles between p2 and p1. In the next step, addressing p_2 and p_3, 2 is added to each vertex index so that the next triangles are generated by adding the indices 2, 3 and 4 and 3, 5 and 4 (see Table 3.1).

The UV coordinates for both new vertices are (0, d) for the left vertex and (l, d) for the right vertex, where d is the distance on the spline (we assume a value between 0.0 and 1.0), and l is the number of lanes. For our exemplary spline from Figure 3.10, this proceeding will result in the mesh shown in Figure 3.12a.



(a) (b)

Figure 3.12: Street mesh generated out of a spline with a step length of 0.1 units (**a**) and street mesh generated out of a spline with a step length of 0.01 (**b**)

Lowering the step length (in this case to 0.01 units) will significantly increase the mesh's quality (see Figure 3.12b).

3.5.1 Avoiding Overlapping Triangles

For very sharp turns, it might happen that triangles overlap if the street is wide enough.

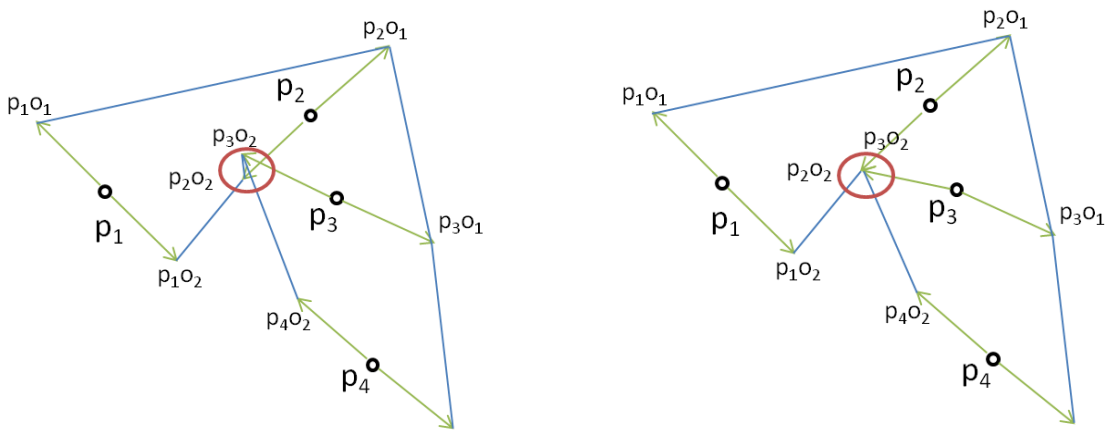


Figure 3.13: Overlapping triangles due to a sharp turn in combination with a high street width (**a**) and a fixed overlap (**b**).

In Figure 3.13, one can see that the triangles (p_1o_2, p_2o_1, p_2o_2) and (p_2o_2, p_2o_1, p_3o_1) would overlap, which leads to artifacts during the rendering of the street. To find these overlaps, a simple check for a line-to-line intersection is performed for the previous point (p_2), both of its offsets (p_2o_2), and the current point (p_3) and its offset (p_3o_2). If such an intersection is detected, p_3o_2 is set to the position of p_2o_2, eliminating the intersection (see Figure 3.14).
Our algorithm is now able to generate streets with an arbitrary lane count and an arbitrary set of points in different LODs. Furthermore, we have shown how to enhance the algorithm to avoid overlapping mesh triangles for sharp turns. Applying a texture

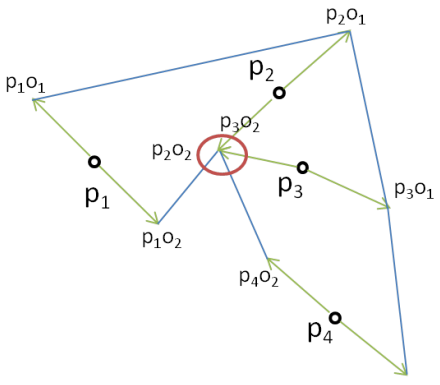


Figure 3.14: In case of an intersection of the offsets we reposition the current offset to the previous offset's position

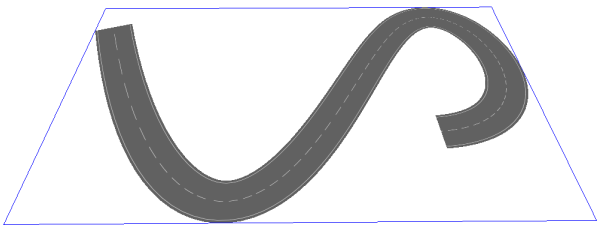


Figure 3.15: Textured street model with a sharp turn

to the mesh will result in a usable street model as can be seen in Figure 3.15.

3.6 Dynamic Intersection Generation

We continue by connecting the street meshes to a road network and creating intersection models in the right places.

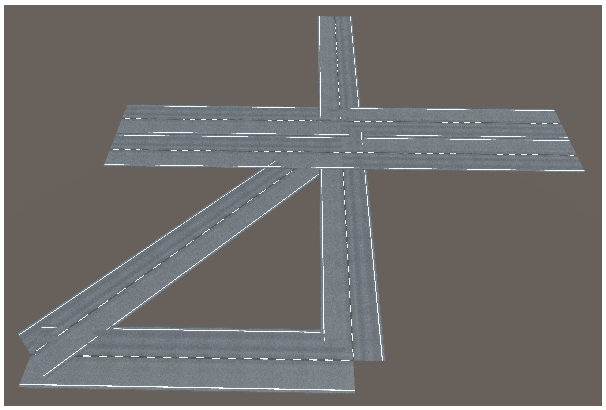
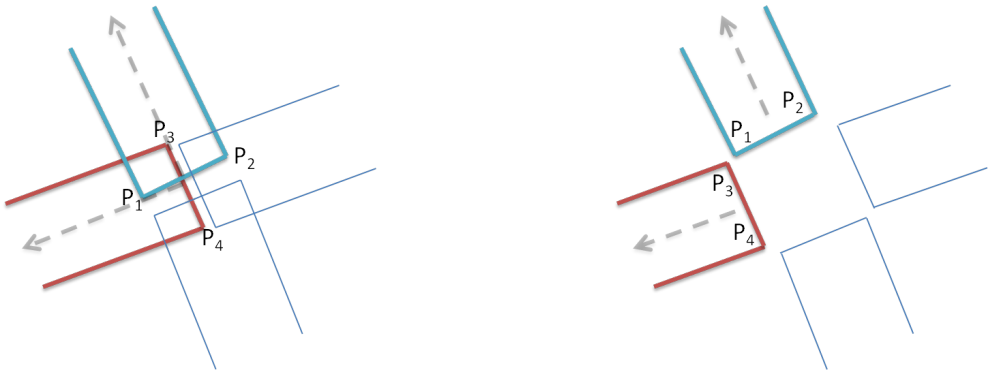


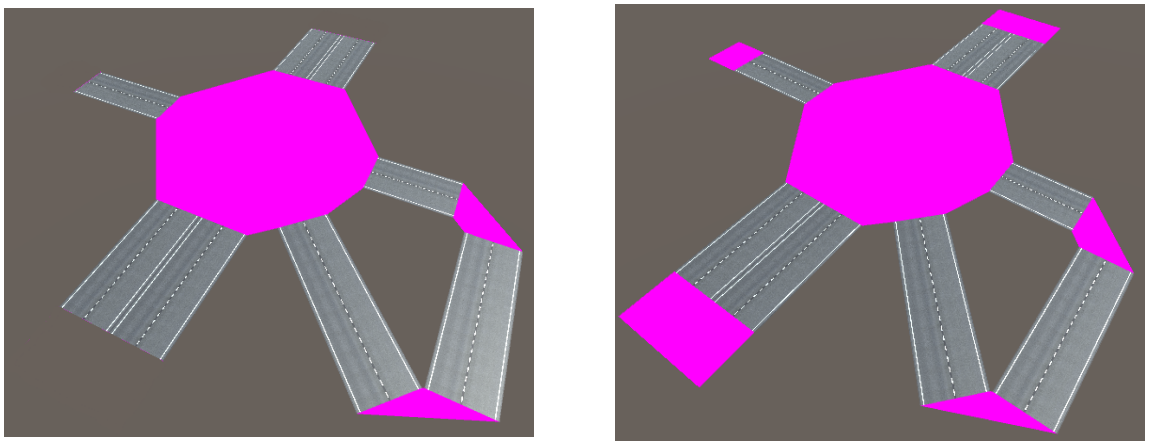
Figure 3.16: Road network without intersections

Figure 3.16 shows an exemplary road network without intersections. At this point, the streets' previously calculated directional vectors (see Figure 3.10) come into play. For each intersection found in the road network, the corresponding directional vectors are inverted, and the streets' end points are shifted unit-wise in the given direction until there are no more line-to-line intersections of the streets' end segments.
In Figure 3.17a the lines p1 $\rightarrow$ p2 and p3 $\rightarrow$ p4 are checked for intersections. If these lines intersect, all of the streets' endpoints are reduced in length at the corresponding end, by n units. This step is repeated until there are no remaining end line intersections at the specific intersection (see Figure 3.17b).
At this point it would be simple to connect all end vertices (p_1, p_2 ... p_4, p_3) and use a triangulation algorithm to receive an adequate intersection mesh (see Figure 3.18); however, that would look unrealistic.
Instead of doing so, the edges of each intersection that are not connected to a street will be rounded to



(a) (b)

Figure 3.17: Reduce the streets' lengths until there are no remaining line-to-line intersections of the streets' end segments (**a**) and reduced end segments do not intersect after a reduction of their length (**b**)



(a) (b)

Figure 3.18: Simple intersections with connected end segments (**a**) and dead-ends at each road end that are not connected to any other street (**b**)

1. provide enough space for turning left or right (see both triangle-shaped intersections that connect two streets).
2. limit the space on a large intersection to provide a clear structure of the intersection (see the large intersection connecting five streets).

Beforehand, a dead-end will be built at all streets that are not connected to any other street at both ends. We allow intersections with only two connecting streets as a special case; the reason is that this makes a change in the number of lanes easier to handle. Figure 3.18a shows two examples.
The algorithm for the dead-end generation takes the left and right end vertices as origins, and extends them in the street's end direction by the defined street width. In the following, the dead-end's two triangles are generated by connecting all four vertices (see

Figure 3.18b).

3.6.1 Rounding Intersection Meshes

The following algorithm will take care of the intersection edges that lie between two connected streets (the edges that are rounded in Figure 3.20). The vertices of such a non-occupied edge are used as left and right markers to generate the points in between. The rounding edge generation algorithm makes use of the angle α (see Figure 3.19) between the two inverses of the intersections' directions ($\overrightarrow{d1}$ and $\overrightarrow{d2}$) providing these left and right markers. This angle is calculated by:

$$\alpha = cos^{-1} * \left(\frac{\overrightarrow{d1} \cdot \overrightarrow{d2}}{|\overrightarrow{d1}| * |\overrightarrow{d2}|} \right) \tag{3.1}$$

Since the resulting angle is determined by the co-domain of $\cos^{-1}$, which is 0 to 180 degrees of the cross product, both 2d vectors have to be calculated additionally to map the angle to a range of 0 to 360 degrees.

Algorithm 3: Calculating the cross product determines the rotation direction of the angle

$\alpha \leftarrow$ calculated angle;
$\overrightarrow{d1} \leftarrow$ inverse direction of first street end;
$\overrightarrow{d2} \leftarrow$ inverse direction of second street end;
crossZ $\leftarrow \overrightarrow{d1}.x * \overrightarrow{d2}.y - \overrightarrow{d1}.y * \overrightarrow{d2}.x$;
if *crossZ > 0.0* **then**
  $\alpha \leftarrow$ 360.0 - α;
end

Using an algorithm for the cross product in 2d space means calculating the cross product in 3d space with both z components set to zero. Instead of returning a rotation axis to rotate the first vector to the second, the formula returns a value whose sign tells if the second vector is on the left or on the right side of the first vector. It has to be specified by the user if the edge has to be concave or convex. Taking the cross product into account, we can now say that an angle between the adjacent streets smaller than 170 degrees is concave, and an angle larger than 190 degrees is convex. All edges in between will not be curved since they are almost straight. Finally, the vertices of the curvy edge have to be calculated. Here we make use of a simple square function, which is solved for a and b with two known xy value pairs:

$$f(x) = ax^2 + bx + c \tag{3.2}$$

The first consists of the distance between both points (p_1 and p_2) that are connected by the curve, divided by two (x value), with zero as its y value. The second consists of zero

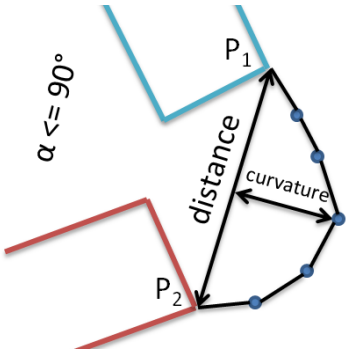


Figure 3.19: Curve of an intersection calculated by a square function

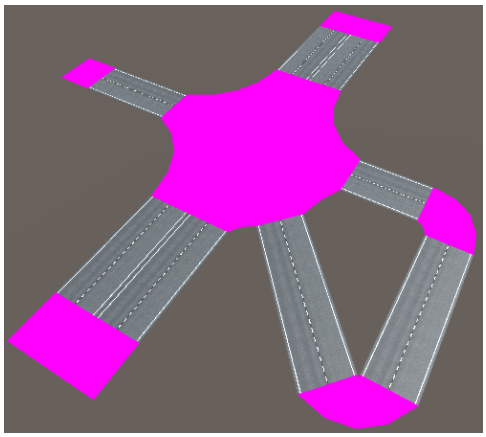


Figure 3.20: Intersections with rounded edges

as its x value, with the maximum height of the curve as the y value. This curvature is set depending on the angle between both streets. An angle larger than 90 degrees will result in an eight times higher y value than an angle below 90 degrees. According to the given x and y values, a and b are calculated for a specified number of vertices between both streets' end vertices (there are five in Figure 3.19).
Raising the number of generated vertices will increase the level of detail of the generated mesh. Figure 3.20 shows the application of the algorithm, creating rounded edges at each non-occupied edge for each intersection. One can see the difference between concave and convex turnings, which are calculated by the angle of both connected streets.

3.6.2 Texturing Intersections

The final step of our approach automatically generates textures and UV maps for an intersection mesh; each intersection gets an individual texture fitting its shape. The creation is split up into two steps. The first step is designed to create a tiled map of a prefabricated grey street texture for the size of the intersection. In a second step, white markers are drawn onto the tiled map. We decide between three kinds of edges:

1. no streets connected,

2. connected normal/priority street,
3. connected secondary street.

To draw the markers, all outer vertices of the intersection are traversed.

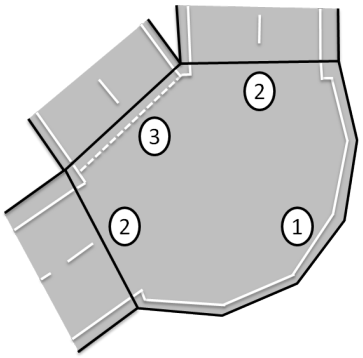


Figure 3.21: Three kinds of street markers are drawn on the intersection texture

If there are no streets connected to the edge formed by the current and the last vertex, a white line is drawn within a few pixels of the intersection texture. In case there is a street connected to the current edge, a white line is not drawn between the two vertices, however, there are two short lined markers painted on the street texture's border where the incoming street connects. The last case is a connected priority street, where a dotted line is drawn between the two vertices. The result is realistic and visually pleasing as shown in Figure 3.22.

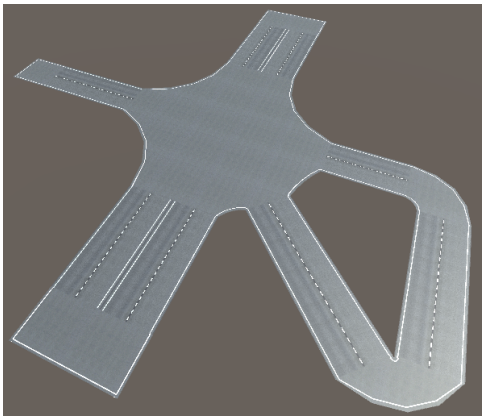


Figure 3.22: Textured intersections with marking

3.7 Evaluation

To find out whether the method can be used to create three-dimensional road networks, a plugin for the editor of the game engine Unity was written (see Figure 3.23). The implementation includes a generator for a *manhattan-like* road network and a generator

that creates real road networks from an *openstreetmaps.org* export file. Users can specify whether pedestrian routes and public spaces should also be generated.

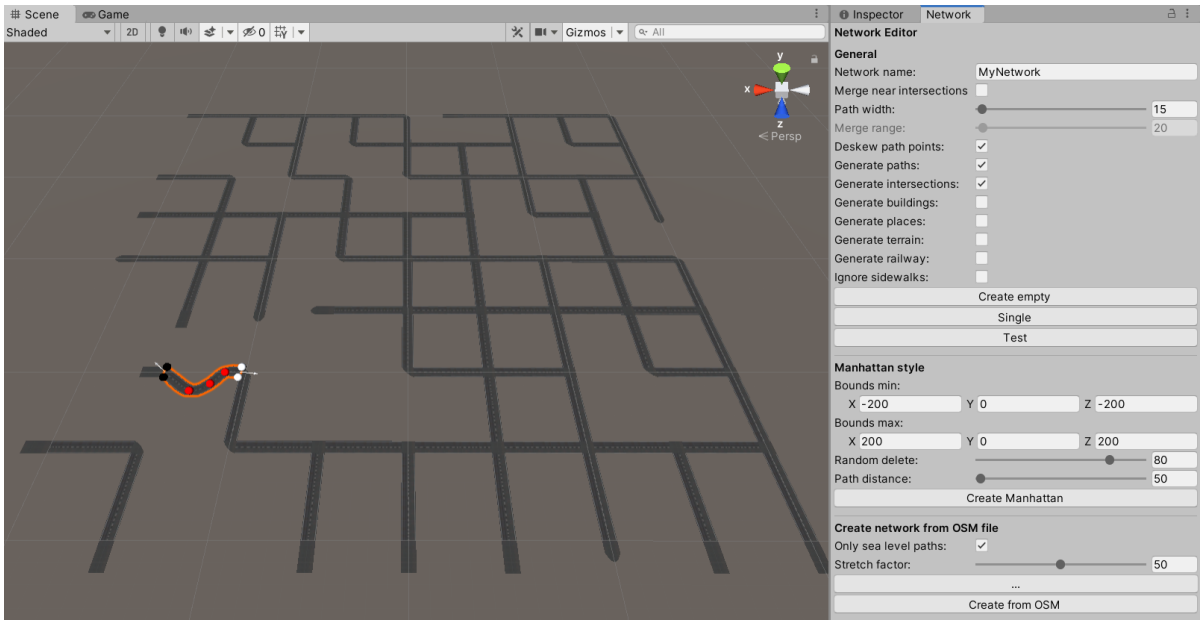


Figure 3.23: The proposed method has been implemented as Unity plugin for the evaluation which was used here to generate a Manhattan-like road network with randomly deleted road segments. One street segment (red, black, and white dots) is selected and can be edited.

The plugin was evaluated by a group of 8 male testers who all have several years of experience in game design. Their age was between 24 and 40 years. The questioning accompanying the evaluation referred to usability, realism, and variety of the results.

The entire group of testerns mentioned that the generation of known patterns requires little effort and was intuitive. All could roughly predict the final shape of the network and provide an accurate estimation of the number of streets and intersections. Some testers expressed the wish to be able to reproduce other styles besides the *manhattan* pattern. Besides the possibility to edit road segments after the generation, the ability to remove random segments to create an individual look of the road network was appreciated.

Using real map data as the basis for the generation process surprised the majority of the testers with the complexity of the network. Especially the number of smaller side streets was underestimated, as well as the total number of streets in famous inner city areas.

In practice, the implementation has proven to be applicable for procedurally generated road networks and for existing road network data from *openstreetmaps.org*. Our road network for the inner city of Mannheim, Germany, with 236 streets and 205 intersections, takes up to 7 minutes to be generated on an Intel i7 processor running at 3.4 GHz.

3.8 Conclusion

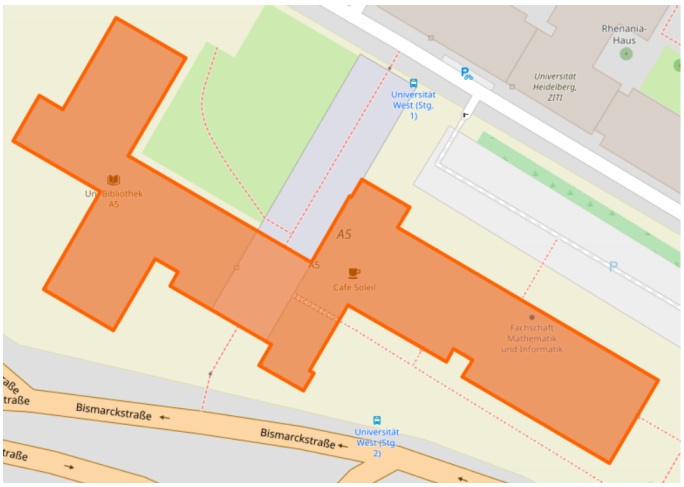
Our approach shows how textured meshes for arbitrary road networks can be generated without manual intervention. The results look pleasant, and they can be used instantly not only in games but also in any 3d application. The algorithms presented can be enhanced to generate sidewalks next to streets, or to introduce a layer system to allow bridges or underpasses to be created. We recommend to merge intersections that are close to each other to one single intersection to avoid overlays of meshes. Furthermore, the customization of textures for each intersection has been found to be very memory consuming. Using a standard texture and a shader that highlights the intersection markers would be a better alternative.

Chapter 4

Procedural Building Generation

Many of today's AAA games take place in an open world that the player can freely explore with only limited restrictions. In series like *Grand Theft Auto*, *The Witcher*, *Fallout*, *Assassin's Creed*, or *Watch Dogs* one or more huge cities serve as a central setting for the plot. The freedom associated with open world games is perceived more intensely when the player encounters fewer barriers that prevent him from exploring surroundings. One of the most common types of barriers in urban settings are certainly buildings that the player cannot enter and that are only represented by facades. The limiting factor not to model the interior of every building is often time and money. To name some figures: Ubisoft employs 400 - 600 people for the development of an AAA open world game [208], whereas the development of Grand Theft Auto IV took over 1000 people and three years to complete [209].
Since many communities are not satisfied with the lack of accessibility, the need of a more accessible interior is discussed controversely, and some fans even create modifications to open locked doors, create missing interiors, or even add whole new districts to existing game environments [210, 211, 212].
Consequently, the question arises what game developers could achieve if the construction of buildings with a credible interior was simpler, faster and cheaper. Two possible effects could be a higher immersion due to more realism and fewer barriers, or the increase of replayability because new areas could be generated over and over again - even during gameplay.
In this chapter, we present a workflow to procedurally generate 3d building models with a focus on usability and realism. Our main contribution is the creation of accessible multi-story buildings with different types of stairways, and a random, yet reasonable texturing of the buildings' surfaces. To furnish the buildings after the generation process, we introduce a system to keep track of free spaces in the rooms and along the room walls.
The procedural component of this work is expressed by an algorithm that creates a building from a given ground plan, which can be a polygon either with or without holes.

The main advantage of having the ability to create buildings for a given polygon is that the algorithm can be applied to GIS data, which is a common way to store the floor plans of a building (see Figure 4.1). This allows game designers to create virtual maps from real geographic locations in an instant, which has become increasingly popular in games lately [213]. The work described here was published in [214].




```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" copyright="OpenStreetMap and contributors"
    attribution="http://www.openstreetmap.org/copyright"
    license="http://opendatacommons.org/licenses/odbl/1-0/">
 <way id="35154744" visible="true" version="14"
    changeset="27641696" timestamp="2014-12-23T00:29:06Z"
    user="whb" uid="202726">
  <nd ref="412350337"/>
  <nd ref="412350338"/>
  <!-- ... -->
  <nd ref="3241380146"/>
  <nd ref="412350349"/>
  <nd ref="412350350"/>
  <nd ref="1019984303"/>
  <nd ref="1019984284"/>
  <nd ref="412350337"/>
  <tag k="addr:city" v="Mannheim"/>
  <tag k="addr:country" v="DE"/>
  <tag k="addr:housenumber" v="6"/>
  <tag k="addr:place" v="A5"/>
  <tag k="addr:postcode" v="68159"/>
  <tag k="addr:street" v="A5"/>
  <tag k="building" v="university"/>
  <tag k="name" v="A5"/>
  <tag k="source" v="HiRes aerial imagery"/>
  <tag k="wheelchair" v="limited"/>
 </way>
</osm>
```

Figure 4.1: University of Mannheim - Institute of Computer Science and Mathematics in Open Street Map and the corresponding XML OSM representation.

We present the building's data model with its topology in Section 4.1 and give a brief introduction to the two-step building generation process in Section 4.2. The procedural aspect of this work is presented in Section 4.3; here we explain how a building can be constructed with a minimum of manual work and parametrization. Chapter 4.4 explains how the building data from Section 4.3 is translated into a textured building mesh. We terminate with an experimental evaluation, conclusion, and outlook to further work on our method in Sections 4.5 and 4.6.

4.1 Data Model and Topology

We now define the data model for our algorithms (see Figure 4.2). A building contains several rooms, several stairs, and at least one roof. Textures are defined by a *room texture set*. At least three walls are required to shape a room. A room can contain several *stair areas* and *occupied areas*. Both define a space that blocks the placement of other objects. A room's wall has zero to n doors and windows.
The data model in Figure 4.2 does not represent an exhaustive geospatial topology; for example, it does not contain the information for whether or not two rooms are adjacent to each other, or if they cover each other in two superimposed stories. This topology is derived implicitly during the generation process to validate spatial relations when required. The decision not to track the building's topology within the data model is based on the simplicity to identify dependencies from the data model (e.g., if two rooms

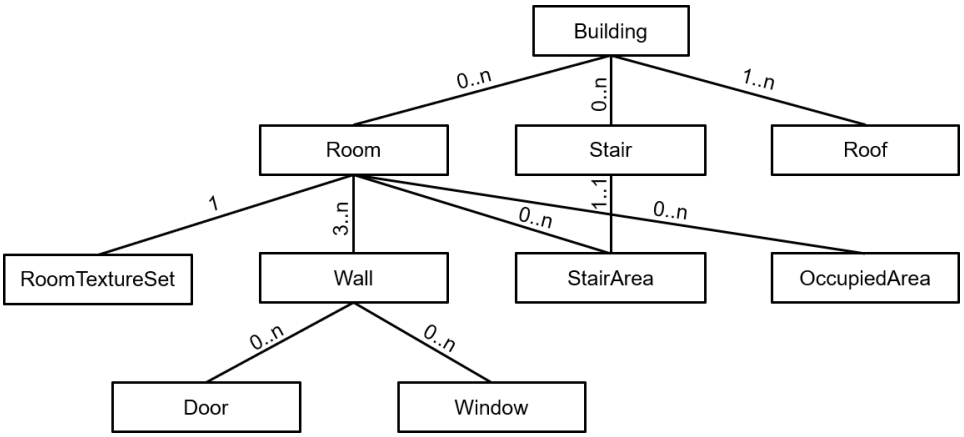


Figure 4.2: A building's data model.

are adjacent, or if they are connected by a door). This reduces the complexity and error-proneness of monitoring the validity, state, and constraints of the topology during each change in the building's architecture.

4.2 Procedural Building Generation

The building generation process can be split into *object model instantiation* and *3d model generation*. The intention of separating both is to exclude all random operations from the 3d model generation, and hence to guarantee reproducability for each procedurally generated building. Furthermore, the procedural part can be replaced by another one, and the mesh generator can still be used without any changes.

1. *Object model instantiation*: The data model described in 4.1 is calculated using all of the building dimensions, layouts, and constraints. The user can influence the model by parameterisation. Since the deterministic instantiation contains random factors, it is not reproducible, and it can succeed after it failed before with identical user-defined parameters.
2. *3d model generation*: The object model is translated into a 3d model under the assumption that it has been validated. This process is completely automatic and can only be influenced by changing the underlying *object model*.

The focus of our research in the domain of PCG lies in the object model instantiation and the corresponding validation and calculation. Even though we consider translating the validated coordinates into a 3d model as trivial, we discuss some of the model generation's sub-areas like Level of Detail, texturing, and polygon manipulation in order to point out the advantages and pitfalls.

4.3 Object Model Instantiation

Instantiating the object model is a twelve-step sequential process (see Figure 4.3). Our algorithm has five mandatory input parameters[1]:

- building width (float),
- building length (float),
- building shape (closed polygon),
- room count (integer),
- floor count (integer).

The parameters should be self-explanatory. In an application where many buildings are generated, it is possible to calculate the number of rooms and floors based on the buildings' dimensions to minimize the required user interaction.

4.3.1 Building Shape Selection

When thinking about residential houses there are three frequent building shapes: rectangular, T-shape, and L-shape. Using parameterizable width and length allows us to generate a variation of those three shapes, for a slight diversity. Nevertheless, without anticipating the evaluation, we found that even though this parametrization of the basic shapes was possible, a partially random building shape will result in a more interesting, inviting, and explorable building. In addition to these most common floor plan shapes, the method presented here supports any conceivable closed shapes.

4.3.2 Stairwell Selection and Placement

Given that the building has several floors, a staircase type is selected. For our prototype, three types have been implemented: L-shape, U-shape, and straight stairs.
The positioning of stairwells in a multi-story building is a challenging task. It involves optimal placement simultaneously on all stories, as well as the generation of the individual stairwell mesh. Sorting the floors by story helps in validating that they are truly ascending, and that there is no missing level in between. The intersection of the upper floor's polygon with the polygon below identifies shared areas that come into question as a possible placement area for the stairs. This intersection is successively calculated from the top story to the lowest story until n areas remain that exist on each story (see the red area in Figure 4.4 as an example).
In a next step, the required area for the stairwell is calculated. Its size depends on the stair type, the room height, and the number of steps. To avoid a blocking wall at the stair's entry and exit, an upper and a lower landing is added.
The placement of the stairwell polygon into a floor's polygon is achieved by superimposing the stairwell's and the floors's outline. The stairwell polygon is rotated until

[1]Figure 4.25 shows many more parameters but all others are initialized with default values.

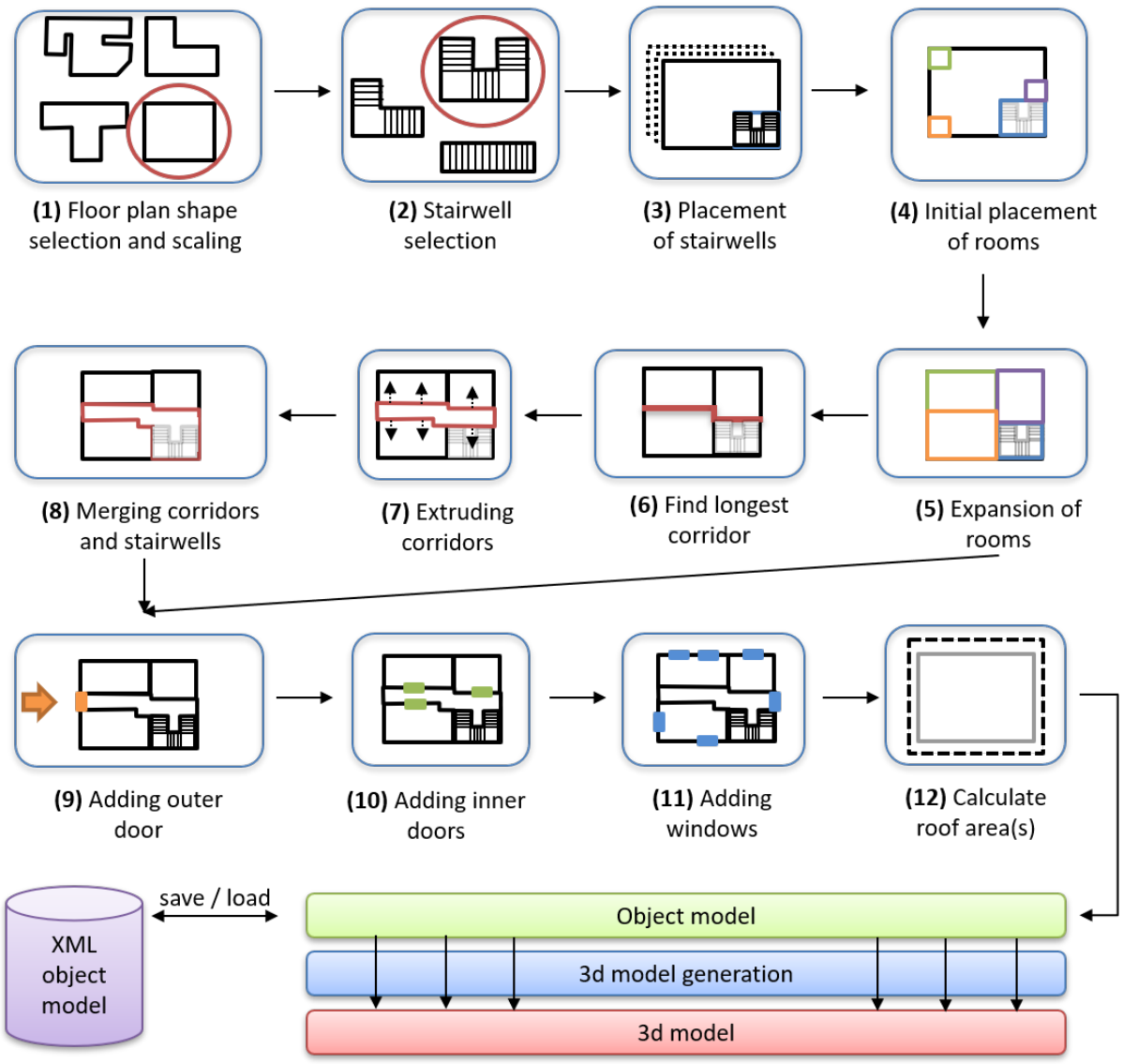


Figure 4.3: Instantiating the object model creates a validated description of a building that is later on translated into a 3d model.

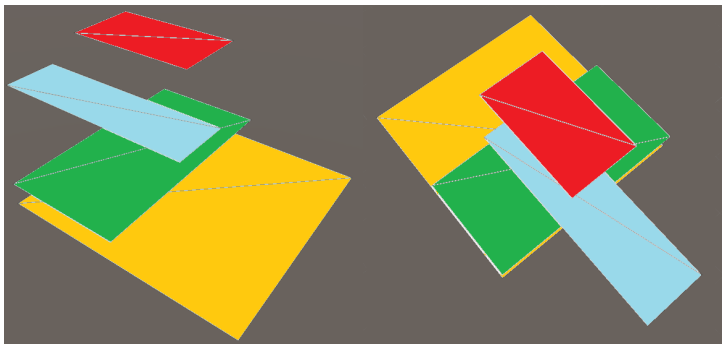


Figure 4.4: Calculation of the intersecting area (red polygon on top) of three polygons (yellow, green, blue).

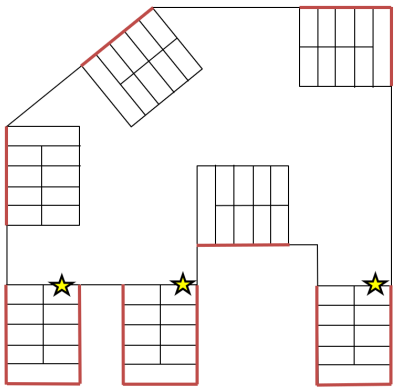


Figure 4.5: Optimal placement of U-shaped stairs in a room's polygon. The best-rated positions (highest edge overlay) are marked with a star.

the entire polygon lies within the floors's polygon (or on its borders, to be precise). To measure the quality of the found position, the length of the edges that the stairwell and room polygon share are summed up. A good placement results in a high overlay value. We call this method *Best Fit* (see Algorithm 4).
Figure 4.5 describes the exemplary placement of rectangular stairways in a polygonal room. The algorithm evaluates many other rotations and positions of these stairwells but for the sake of simplicity we only include the seven shown here.
The three stairway positions on the bottom of the room will be rated high, since the overlay value of both the room and the stairwell edges are the highest (see the red lines). After calculating the stair's shape, hand-railing edges, entry and exit segments, and optimal position, the stairs are registered for each superimposed room.

4.3.3 Initial Placement of Rooms

After placing the stairwells, we use a bucketing approach to distribute all of the rooms over all of the stories, to have the most equal number of rooms per story possible. Our algorithm is based on region growing. Creating an exemplary building with three stories

Algorithm 4: The *Best Fit* algorithm finds the optimal space to place a polygon into another

```
Input: Polygon p2 to be fit in polygon p1
Output: Polygon p such that p1 contains p2 and od is maximal, overlay
        distance od, edges that overlay eo
if p1.isClockwise ≠ p2.isClockwise then
    p1.invertEdges;
end
if p1.area < p2.area then
    return null;
end
res ← empty polygon;
foreach edge e1 in p1 do
    outerEdgeDir ← (e1.p1 - e1.p2).normalize;
    foreach edge e2 in p2 do
        innerEdgeDir ← (e2.p1 - e2.p2).normalize;
        tmp ← copy of p2;
        tmp.move(e1.p1 - e2.p1);
        rotationAngle ← angle between outerEdgeDir and innerEdgeDir;
        rotationPoint ← e1.p1;
        tmp.rotate(rotationAngle, rotationPoint);
        if not p1.contains(tmp) then
            continue;
        end
        tmpOverlayDistance ← calcOverlayDistance(p1, tmp);
        tmpOverlayEdges ← cntOverlayLines(p1, tmp);
        if tmpOverlayDistance > od then
            od ← tmpOverlayDistance;
            eo ← tmpOverlayEdges;
            res ← tmp;
        end
    end
end
return res;
```

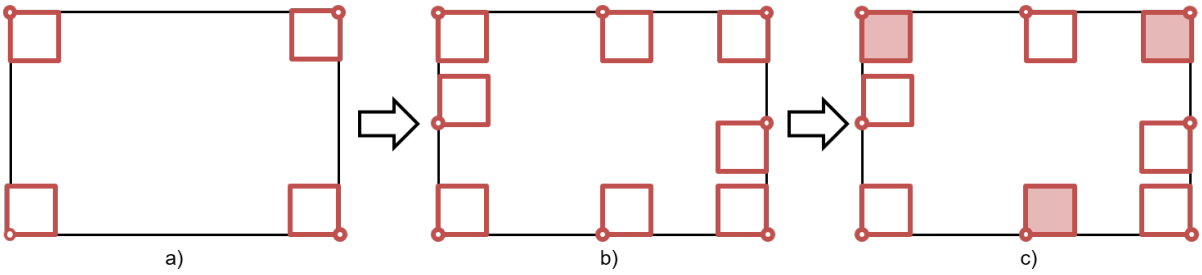


Figure 4.6: Adding more vertices to a polygon without changing its shape allows placing more inner polygons using *Best Fit*. Here, four vertices (**a**) are transformed into eight by splitting each edge once (**b**). From all valid polygon positions that are returned by the *Best Fit* algorithms pick three which are the farthest away from each other (**c**).

and fourteen rooms results in a distribution of 5, 5 and 4 rooms, respectively. Those rooms are scaled to a small initial size of 1x1 units and distributed equally on each floor using *Best Fit*. Here, we face two challenges of the corresponding algorithm:

1) How can n polygons of shape A be placed in another polygon B if B has fewer vertexes than n?
2) How can we assure that not all n polygons of shape A are placed in one dense area of polygon B?

Point 1) is addressed by splitting the outer polygon's edges into several sub-edges, so that there are always more, or as many, vertices as the number of rooms. In the example of Figure 4.6a, we have eight rooms to be placed in a polygon with four vertices. Hence, we subdivide the polygon, adding one additional edge per existing edge, so that the total number of vertices for that polygon equals eight (see red dots in Figure 4.6b). The subdivision is a simple, stepwise interpolation between the start and end point of the edges.
Point 2) is addressed in one single solution that we call *Best Fit by Highest Distance*. As known from the basic *Best Fit* method, all possible room placements are stored in a list. This allows us to see all of the possible placement areas of the rooms in the building shape. In the next step we take the room's midpoint and determine n room positions from all possible positions with the highest distance to each other.
Finding the highest distance is addressed in research as the *Farthest Neighbors Problem*. Due to its complexity, we have chosen a sufficient approximation algorithm from Le Bourdais [215], which is fast and works well.

4.3.4 Expansion of Rooms

When each of the rooms holds an initial part of the floor plan, we expand them using a round robin approach along the sequence A→B→C→A→... The growth algorithm expands the polygon's vertices in each direction until they intersect with an outer boundary (the building shape), or with another polygon (another room). When a vertex hits the

outer boundary, the polygon can still be expanded in all other directions. In contrast, an intersection with another polygon, which is simultaneously expanded within the same shape, terminates the algorithm to avoid bad aspect ratios, and force more irregular shapes by merging the remaining artefacts (see the following paragraph). An exemplary expansion is shown in Figure 4.7a, b and c.

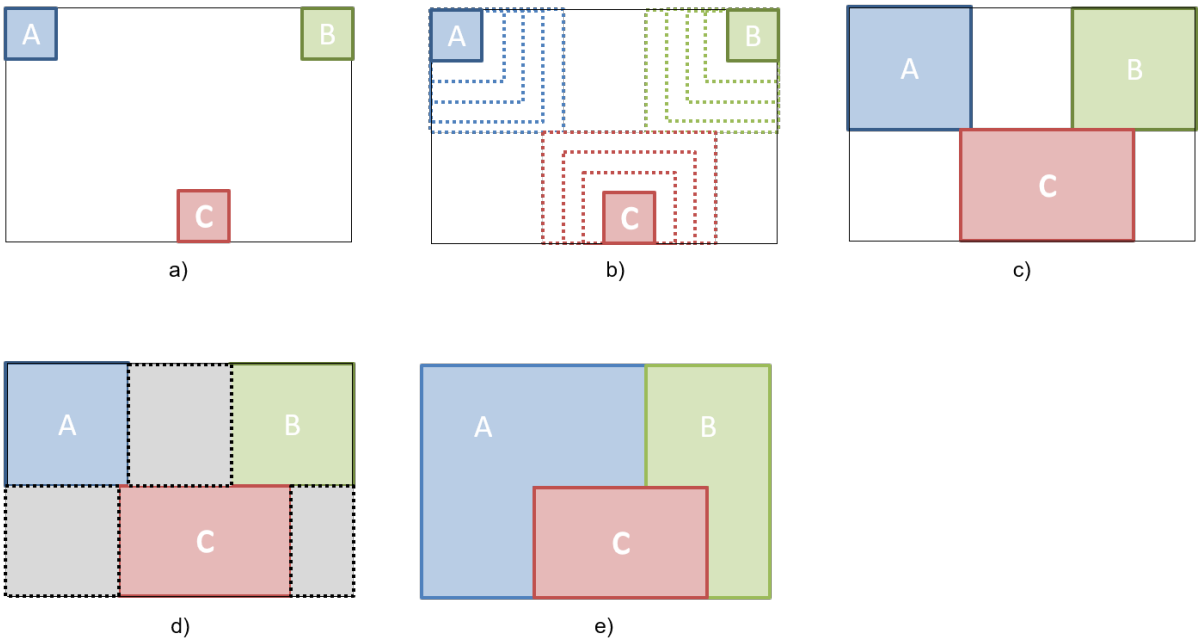
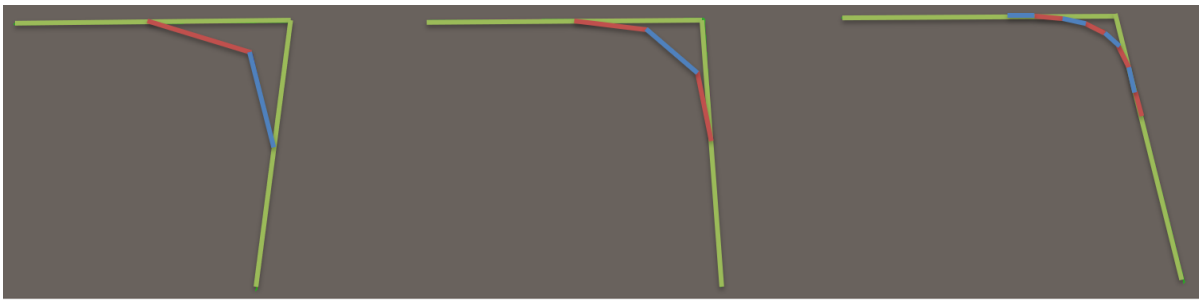


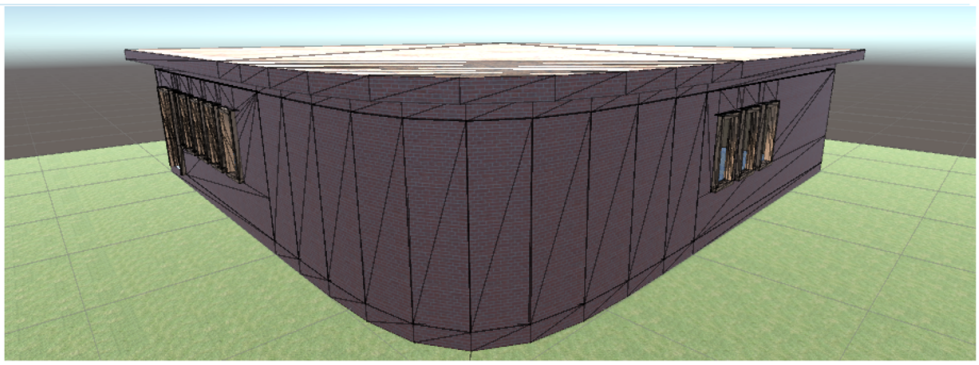
Figure 4.7: Polygons A, B and C are expanded stepwise (dotted lines) until they intersect with other polygons. The remaining areas are merged with rooms A, B and C.

If none of the rooms can be expanded anymore, the growth algorithm has reached its final state. Optimally, the width and height of the building are a multiple of the step size, so that the algorithm can use the entire space within the outer polygon. Otherwise, there might be small, unused areas in the outer polygon that lead to undesired artefacts in the generated mesh. After the expansion, unused areas remain on the floor, and can be seen as grey rectangles in Figure 4.7d. Those areas can be extracted by subtracting all room shapes from the floor plan. The resulting shapes are then joined with the first adjacent room in the list of rooms on the story. This approach is similar to the one presented by Lopes et al. [127], which uses a grid and expands the rooms cell by cell. By using vertices as starting points for the growth of each room, we diversify the procedure and the result. Furthermore, since our approach is based on polygons and Boolean operations, we can handle non-rectangular building shapes.
The resulting room polygons are provided with a story index and saved as polygons in a room list. To make the shape of a room more interesting, its corners can be arched. An interpolation between the mid points of both walls that form the corner allows for the definition of a parameterized number of walls that represent the arch. To create such a curve, we create a list of edges drawn between the mid point of their two neighbour edges. The detail and smoothness of the arch raises with the number of edges. In the

end, the algorithm traverses all of the edge's mid points and creates the new wall edges from the current and the last mid point pair. Figure 4.8 exemplarily illustrates how a corner of two edges can be rounded and replaced by a list of new edges. The segments of an arch can again be treated as simple walls and equipped with windows and doors.



a) b) c)



d)

Figure 4.8: Three forms of archs with (**a**) two iterations, (**b**) three iterations and (**c**) ten iterations. The green lines are the original edges that are rounded by the algorithm. (**d**) shows an exemplary building with an arched corner.

4.3.5 Texturing

To apply textures to a room's surfaces, we introduce the *room texture set* (see Figure 4.10). This data structure contains eight different textures, one for each of the room surfaces, as shown in Figure 4.9.
The *stair cut area* and the *window cut area* reference a texture for those areas that are created when a cutting block subtracts parts of the room's mesh for a door, window, or stairway. Each texture in a *room texture set* can be explicitly defined, but to achieve a diverse look for the generated building, we introduce another texturing method. This method supports three granularity levels, allowing the addition of a randomization factor to the choice of textures. Each texture is indexed by a six-digit number (see an example in Figure 4.10) and a name that relates to the actual texture file. The first two digits of its number relate to the surface type, for example *ceiling*. The second two digits identify

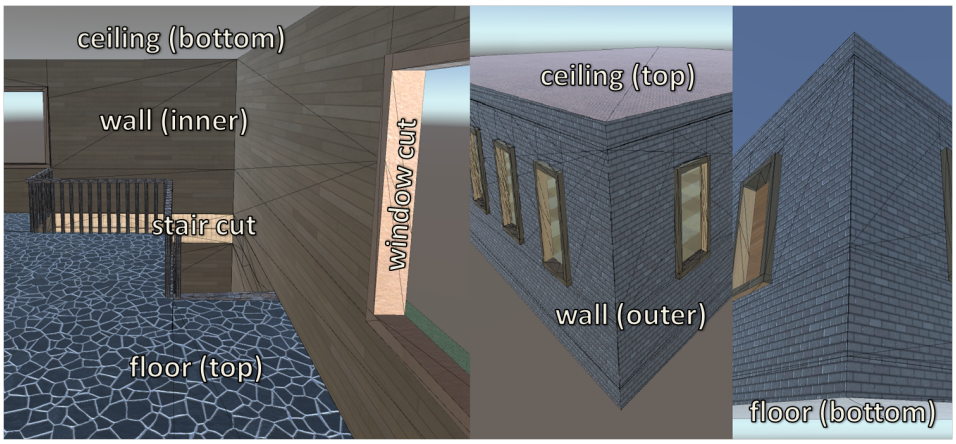


Figure 4.9: Surfaces of a *room texture set* mapped to the faces of a mesh.

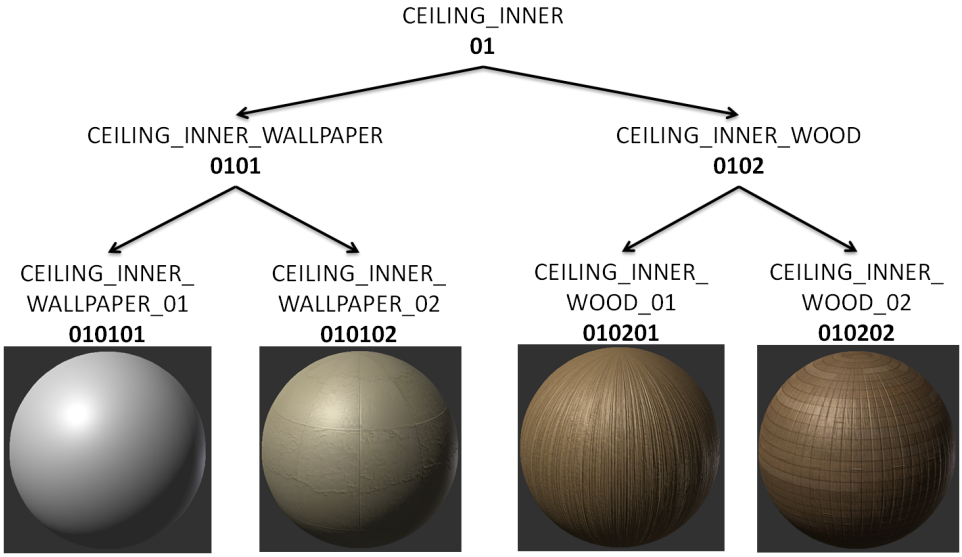


Figure 4.10: The granularity of the surface type adds a random factor or selects a concrete texture.

the second level of detail, such as *inner ceiling* and *outer ceiling*. The final two digits specify the exact texture. When digits are not specified, the algorithm selects a random texture within the given scope (such as *ceiling* or *inner ceiling*). Therefore, initializing a *room texture set* with four digit identifiers (see the second tier in Figure 4.10) will result in rooms with different textures per iteration.

4.3.6 Finding the Longest Corridor

In this section we make use of the research by Mirahmadi and Shami [131] and extend it to be compatible with the generation of stairwells. Before corridors are generated, the staircase is added to the list of existing rooms. We apply a *Connected component* algorithm to all inner walls' edges of the current floor to find all available paths that

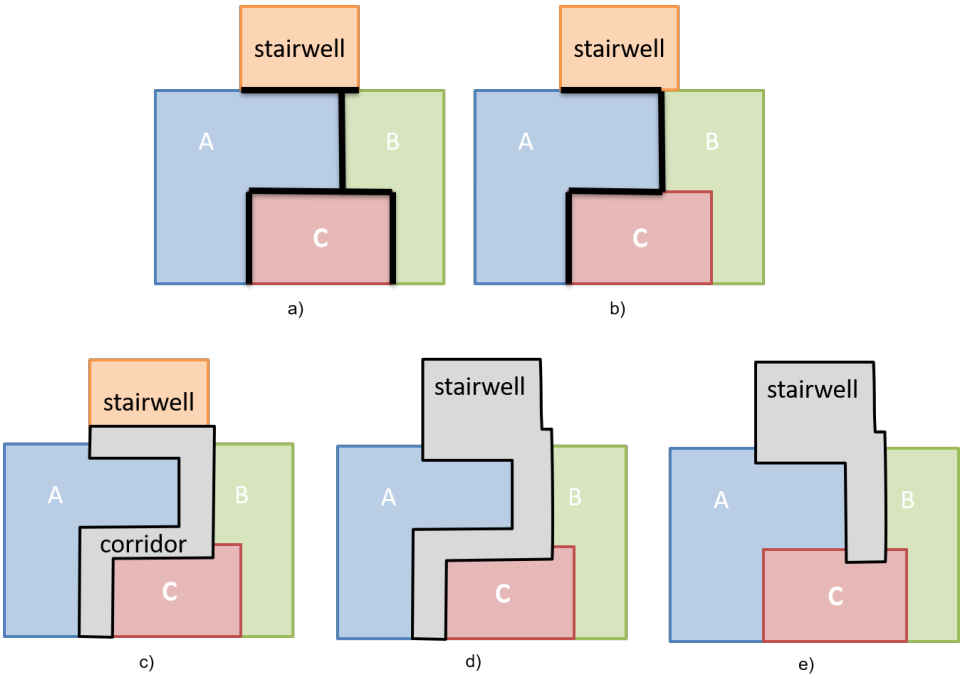


Figure 4.11: Connected component analysis (**a**) in combination with Dijkstra allows the detection of a path passing as many rooms as possible (**b**). Expanding the longest path in width results in a polygon that is subtracted from all rooms (**c**), merged with the stairwell (**d**), and successively shortened edge by edge as long as all rooms are adjacent to at least one of the corridor's edges (**e**).

connect rooms (see Figure 4.11a). By iterating over all of the connected graphs, we determine the longest path that has a direct connection to the stairwell. To do so, we apply a *Shortest Path* algorithm (e.g., *Dijkstra*) using the edges' negated lengths as weights. We use each coordinate of the walls' edges as a starting point to make sure we have actually found the longest path.

In all stories where the corridor does not require a direct connection to the outer wall (e.g., for an entrance door), we can successively eliminate the last edge to maximize the building's effective area (step e). The final state is reached when the next deleted edge would decrease the number of rooms adjacent to the corridor.

4.3.7 Extruding Corridor and Merging with Stairwell

The longest path is shown in Figure 4.11b as a black line. In the following it is expanded in width, and the resulting polygon is subtracted from all existing rooms (see Figure 4.11c); the stairwell room and corridor are merged (step d) by merging the polygons. To maximize the room size, the corridor is shortened edge by edge as long as all rooms in the story can be reached (step e).

4.3.8 Adding Doors and Windows

Adding windows and doors follows a strict order. External doors are prioritized, followed by internal doors and windows.
Focusing on a European architecture standard, the algorithm seeks to place the main entrance door in a corridor. By adding a simple rule to allow the main entrance in the living room, we can easily change the layout to that of an American style building.
In real-life buildings, the positioning of outer doors depends on many factors, e.g., the terrain around the building, the neighborhood, or the type of rooms. Since this is not in our scope, we only create one door on the first floor where the corridor touches an outer wall. If there is no such wall, the entrance is created in the stairwell, or as second option, in the largest room of the building, which is generally the living room.
The placement of inner doors is achieved by recursively tracking if a room is directly or indirectly (via n rooms) connected to a corridor or to a room with an outer door. In case it is not, the algorithm connects this room with an adjacent room that has no connection to this room yet. The algorithm then continues with the next rooms until all rooms are accessible.
The number of windows is calculated based on the size of the room, using a fixed number per square meter. The placement of doors and windows in walls depends on the topology of the building, respectively on the position, size, and adjacency of its rooms. In general, there should be a distinction between outer doors and outer windows, respectively inner doors and inner windows (e.g., a service hatch between the kitchen and the dining room). Each of the latter connects two rooms, and consequently it has to be validated that both rooms share a common wall and that the wall is not blocked by an obstacle (e.g., stairs). Outer rooms and windows only require validation if there is an existing window or door on the target wall.
To offer a dynamic door and window positioning we introduce the placement rules *First*, *Last*, *Best*, and *Random*. These rules are applied when the placement algorithm selects free, shared areas of the edges of two adjacent rooms that receive a common door or window.

- *First* - First free area that the algorithm determines.
- *Last* - Last free area that the algorithm determines.
- *Best* - The area with the best fitting size (minimal surplus).
- *Random* - A random area.

The algorithm traverses all the focused walls and collects the free segments with a length larger than the window's and/or door's width. Using strategies that imply an order (*First* and *Last*) requires a strict order of the walls to traverse. This order is given by the sequential wall index (0, 1, 2...) and the requirement that walls are always defined clockwise. Thus, the *First* rule will return the first free area for the first common wall, directed from the wall's start point to the wall's end point, e.g., wall 0 of room 1 and wall 1 of room 2 in Figure 4.12. Accordingly, the *Last* rule will return the last free area on the last common wall (e.g., wall 3 of room 1 and wall 2 of room 2).

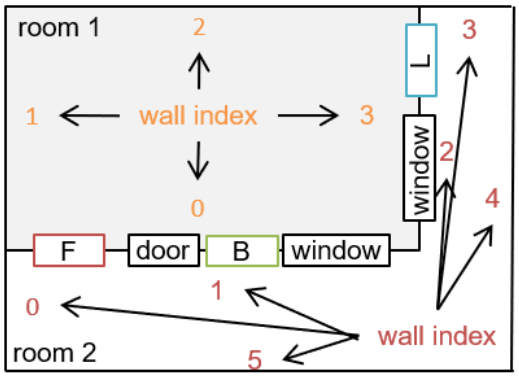


Figure 4.12: Placing a new door or window with the *First* (F), *Best* (B), and *Last* (L) rule. *Random* can return any of the three exemplarily placed objects.

Since the width of the segment can be larger than the object to be placed, we allow aligning the object within the segment either on the left, in the middle, or on the right.

The mechanism used to verify if two rooms share walls is to compare each wall edge of each room with one another and calculate their mutual overlay. The result of the shared wall function is a list of edges specifying the shared and free areas of room 1 and 2.

Since windows and doors are stored as rectangular shapes, it is a straight forward matter to allow the parameterization of width, height, and vertical offsets.

Keeping in mind that the position of windows and doors not only influences the building's structure but also the interior, we keep track of the spaces in front and behind each object using a so-called *occupied area*, which is memorized at the room level (see highlighted objects in Figure 4.13). These areas are actively used when it comes to placing staircase objects in the rooms, or when the building is furnished by a following procedural algorithm (as discussed in the work of Merrell et al. [137]). An *occupied area* is represented by a two-dimensional rectangle that allows it to adapt to the width of a window or door.

If the *occupied area* of a newly created object intersects with an existing *occupied area* in this or an adjacent room, it will lead to a placement invalidation. As an example, *occupied areas* avoid doors placed in sharp angle walls or doors covered by stairwells.

We now discuss the alignment of windows and doors on a building's hull. As Bao, Schwarz and Wonka point out, there is a significant difference between the perception of windows aligned along all stories, and a random distribution [216]. In addition to the free positioning of windows, we introduce a *facade cloning* approach. Once created on the first story, windows can be cloned to all rooms in the above floors, unifying the look and structuring the building's appearance. Figure 4.14 shows a comparison of the random window positioning and *facade cloning*.

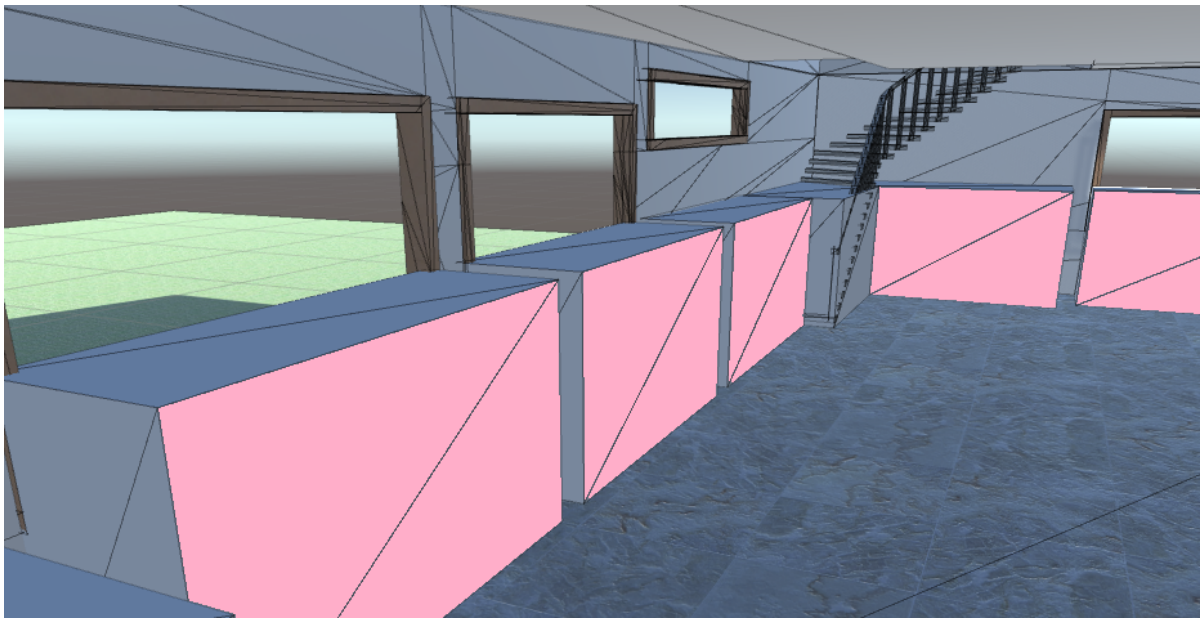


Figure 4.13: *Occupied areas* signify spaces in which no other object might be placed.

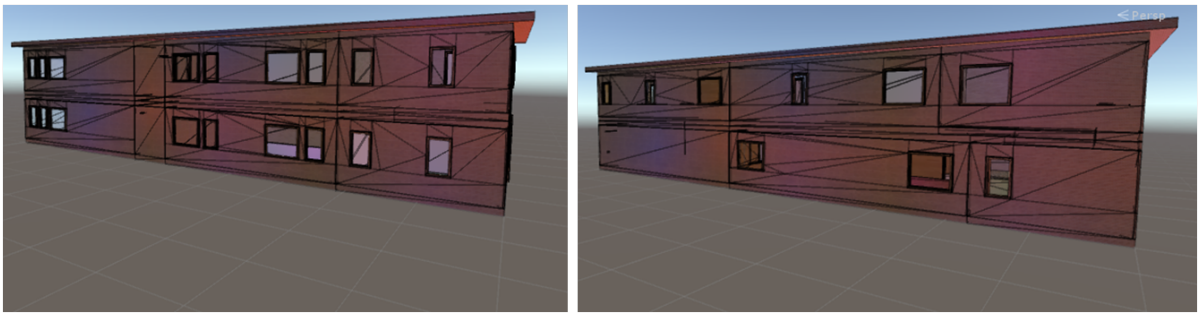


Figure 4.14: Cloned (left) and random facades (right)

Here, the lower room's window positions are stored and validated against the rooms above. If an adequate placement can be guaranteed, the windows are created in the rooms above with the same placement, including the width and vertical offset. This is only possible under the premise that the cloning of the windows successfully passes the entire validation process.

4.3.9 Calculating Roof Areas

Our contribution to the generation of roofs is the detection of roof areas and the creation of hip roofs. Laycock and Day have presented algorithms to generate roofs efficiently, such as gable or hip roofs [217]. In favor of completeness, we have also implemented their method to validate its practicability. Consequently, our algorithm allows the generation of four roof shapes, where the type can be specified for each individual floor level. Figure 4.15 shows the most common roof shapes.

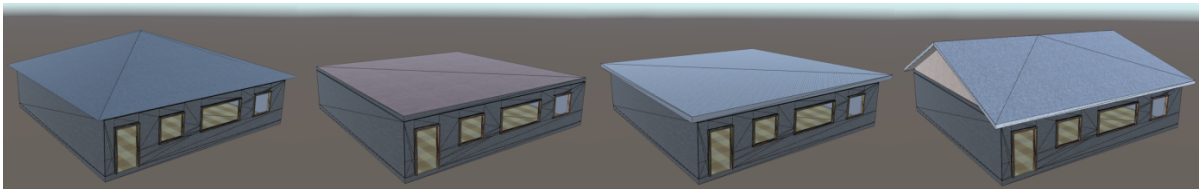


Figure 4.15: Roof shapes from left to right: hip, none, flat and gable.

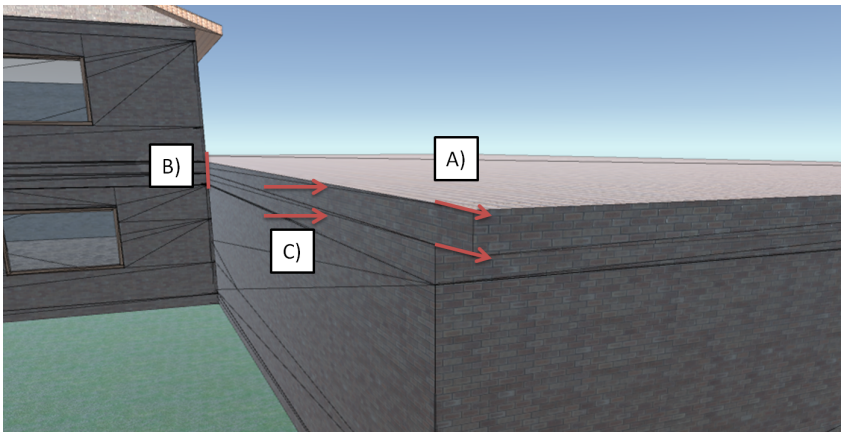


Figure 4.16: The flat roof is extended by a few units to create an overlap (**A**). In case the overlap would intersect with another room (**B**) it is reset to its original position (**C**).

To calculate all roof surfaces, we join each floor's room surfaces beginning at the top. We create such a join for each floor and subtract the above surface from the lower one so that it is guaranteed that each area of the entire building shape is covered by exactly one roof, regardless of the level. Existing covered areas are not covered again. The result is a list of 0..n polygons per story. To generate a volumetric, three-dimensional structure of a roof, it is necessary to offset each roof polygon to create an inner and an outer surface.

Furthermore, the surface calculation requires an intersection check with all rooms in the story above, so that no roof mesh intrudes upon an adjacent room. Where such an intrusion would occur, the corresponding edge is set back to the original edge position (see Figure 4.16).

The generation of flat roofs is trivial and requires no explanation. Creating a structure for a hip roof can be achieved by the usage of the *straight skeleton* algorithm; This method returns a list of triangular or rectangular surfaces that can easily be triangulated and rendered.

Laycock and Day [217] state as well that the transition to a gable roof can easily be accomplished by moving triangular roof surfaces with two connections, to a connected roof's polygon edge. Similar to the *room texture sets*, we assign two *surface types* to the roof surfaces and the roof wall surfaces, which affect all vertical walls in the roof area - so that each part can be textured separately.

During the creation of a building object, there is a roof type defined for each story. The idea to store one roof type per story, whether a particular story exposes an area requiring

a roof or not, is based on the requirement to keep the algorithm as generic and simple as possible; it does not know upfront if a room is covered by a roof. In case the list of roof shapes is smaller than the number of stories, the type list is filled up by replicating the last roof shape until the expected count is reached.
A per-room definition of a roof might be more flexible, but is rarely required in a real-life scenario. Managing roofs on a story scope facilitates the management of roof meshes in terms of surface detection and mesh generation.

4.4 3d Mesh Generation

The object model now contains a list of rooms together with their floor area, number and position of windows and doors, and staircases and their position. The dimensions of the roof areas are also known. All this information is now translated into a 3D model.
To manage textures during the procedural generation process without creating complex UV maps, we modified the common mesh structure which generally consists of indices and vertices. The modification is a simple second index for a mesh group which is added to the existing vertex index, so that we have a *tuple<int,int>* structure (see Figure 4.17c) for each index instead of a simple *int*.

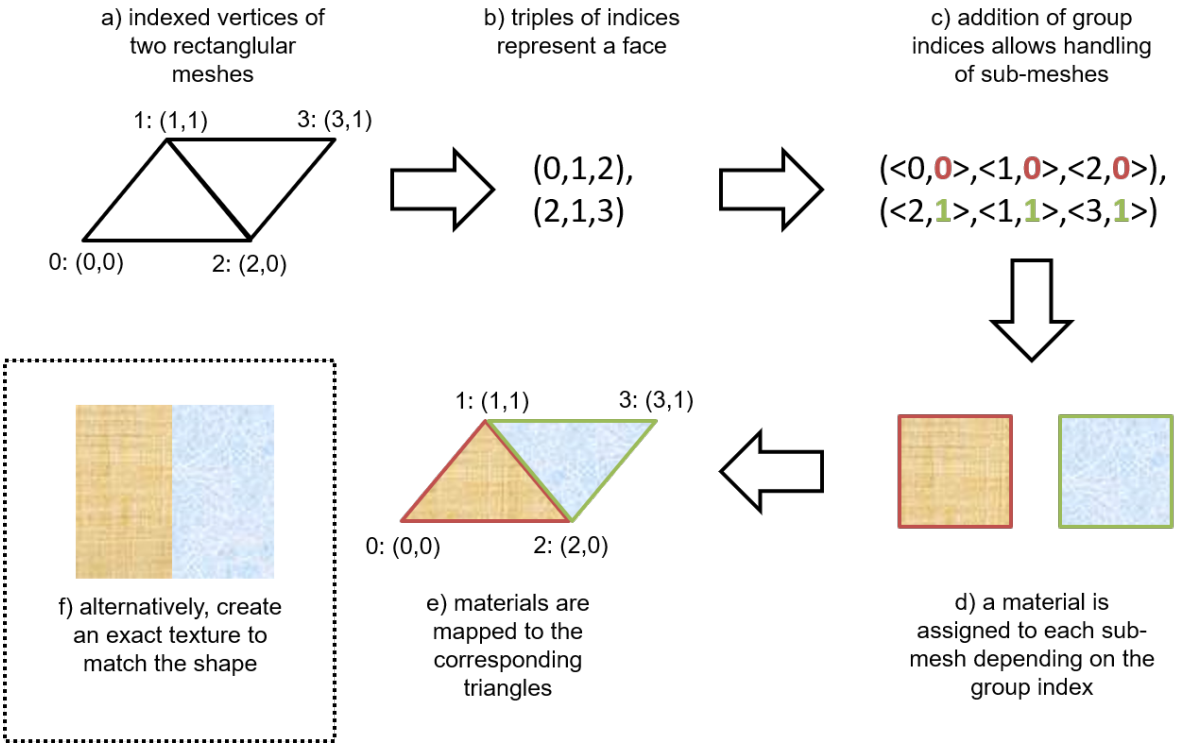


Figure 4.17: The introduction of index groups facilitates algorithmic texturing of multi-texture meshes.

During generation, we collect all faces (groups of three indices) belonging to one mesh group and assign a dedicated material to them, e.g., a floor or wall surface. This method

allows texturing surfaces individually, without unwrapping the entire architecture's mesh. As a drawback, the number of draw calls in the rendering process will increase with each new mesh group, since each sub-mesh with a distinct texture will trigger a draw call.

4.4.1 Rooms

A room is managed as a non-self-intersecting, closed polygon. This polygon is triangulated into a flat, single-textured mesh with its faces pointing downwards. The floor's height offset is calculated using the floor height (FH), i.e., the thickness of the foundation, and the room height (RH):

$$z\_offset = FH + floor * (2 * FH + RH) \tag{4.1}$$

We then extrude the resulting mesh in the upwards direction for FH units to receive a base plate (see Figure 4.18 left).

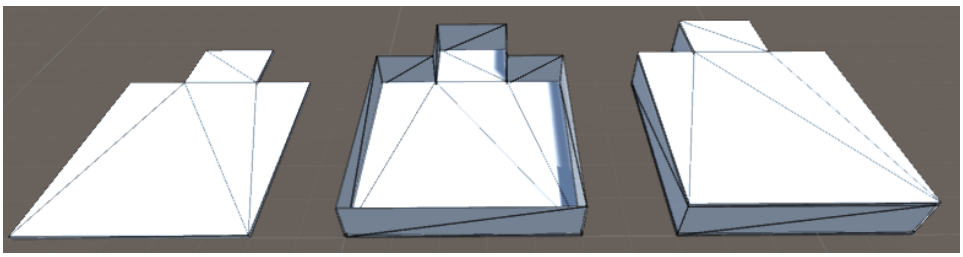


Figure 4.18: The creation of floor, walls and ceiling meshes contribute to the generation of an arbitrarily shaped room.

Walls are created along the outer room edges with a given thickness (see Figure 4.18 middle). The ceiling is a copy of the base plate.
The extrusion algorithm detects the boundary edges of the polygon mesh, creates identical copies of the indexed vertices at their exact position, and copies them to a certain height above the original. Creating these two copies is the only way to assign the individual index groups, as presented above, to the extrusions' side faces. The mesh is finally closed by an identical copy of the original input mesh, with an inverted index order for each face, so that the new surface points in the opposite direction (see Figure 4.18 right). Calculating the vertices' normal vectors is trivial and not further discussed.
In the following step, we subtract the stairs' areas of each room from the floor or ceiling mesh. A subtraction mesh is a mesh used to cut a hole in another mesh using Constructive Solid Geometry (CSG), as described by Requicha and Voelker [218]. CSG allows the application of binary operations to solid geometry, and it is frequently used in procedural modeling [219, 220, 221, 222]. Depending on the stair's start and end story, the hole is cut in the floor or in the ceiling. When the CSG operation succeeded, the subtraction mesh is discarded.
To construct the walls (see Figure 4.18 middle), the room's outer polygon is offset by the negative wall thickness, so that there is an inner and an outer polygon. Keeping track of

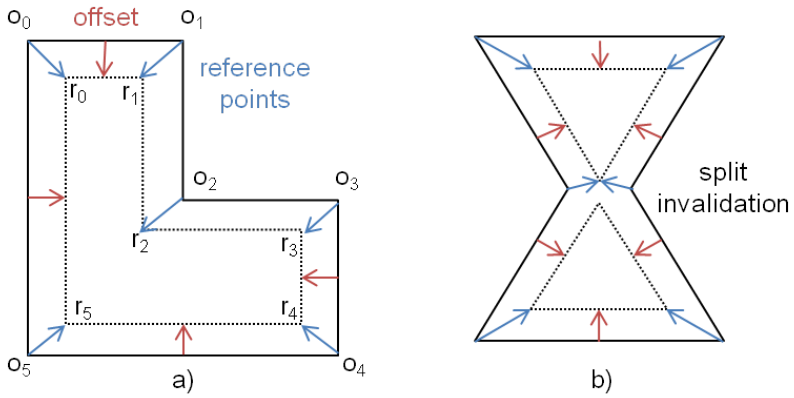


Figure 4.19: (**a**) Valid polygon offset of an exemplary room shape, (**b**) invalid offsetting results in two new inner polygons.

each reference of the outer and inner polygon vertices (see o_i and r_i pairs in Figure 4.19a) allows handling walls individually, as all four corner marks are known (two vertices of the inner polygon and two of the outer, e.g., o_0, o_1, r_0, r_1 for the top wall in Figure 4.19). Having access to each individual wall mesh enables the framework to remove walls from a room (or better hinder its rendering), as it might be done, e.g., for open plan kitchens. Creating such a reference is not trivial, as offsetting a polygon can result in multiple (see Figure 4.19b) or corrupt polygons. To detect inner reference points, the offsetting algorithm shoots rays from each outer polygon vertex m to each inner polygon vertex n and creates a reference $m \leftrightarrow n$ for the minimal distance pair of m and n as long as the $m \leftrightarrow n$ does not intersect with any outer or inner polygon edge. In case that there are n>1 inner polygons, the algorithm declares the room shape as invalid and omits its mesh generation.
Now that the offsetting algorithm detected quadruples of wall marks, mesh quadrilaterals are generated and extruded to room height. The result is a hollow polygon.

4.4.2 Stairs

A stair mesh consists of railings and steps. The step count can be calculated by subtracting the vertical offset of the bottom floor from the vertical offset of the top floor (see Formula 4.1), and dividing the result by the step height. Since the railing edges on the left and the right of the staircase are calculated and stored in the object model, the algorithm simply interpolates between the left and the right railing to get the step boundaries.
Figure 4.20 shows that the result differs a little from most real stairs, since steps of a stair's straight segments are in general rectangular but the visual difference is, in our eyes, hardly noticeable and justified by the reliable and robust algorithm. When it comes to the generation of railings (see Figure 4.21), the left and right vector pairs of each step are stored in an array, so that the dimensions of each step can be accessed once the steps are generated. Iterating over these edges allows the placement of piles at the edges' mid

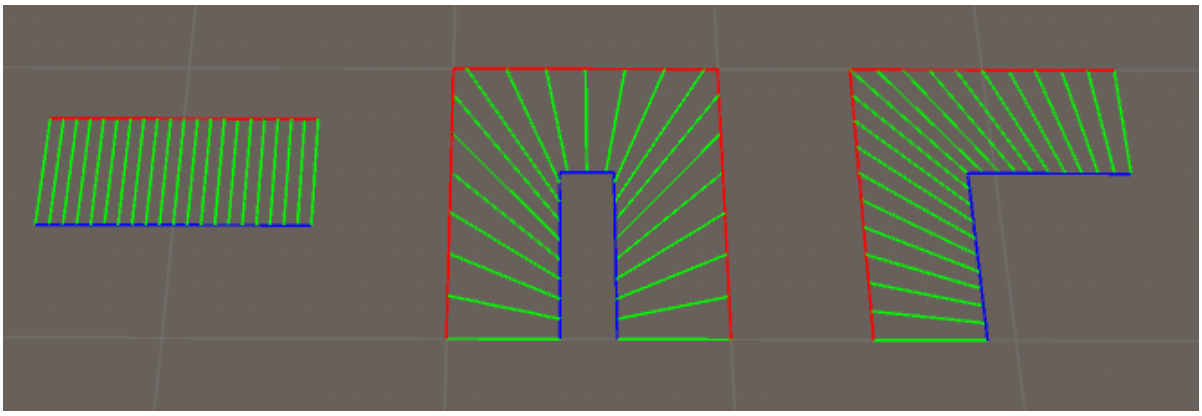


Figure 4.20: Interpolating between stair railing edges (red and blue lines) allows the creation of stairs with arbitrary shapes (e.g., straight, U-shape and L-shape).

positions, or the creation of a solid border by offsetting the two-dimensional edges to a three-dimensional mesh, for example.

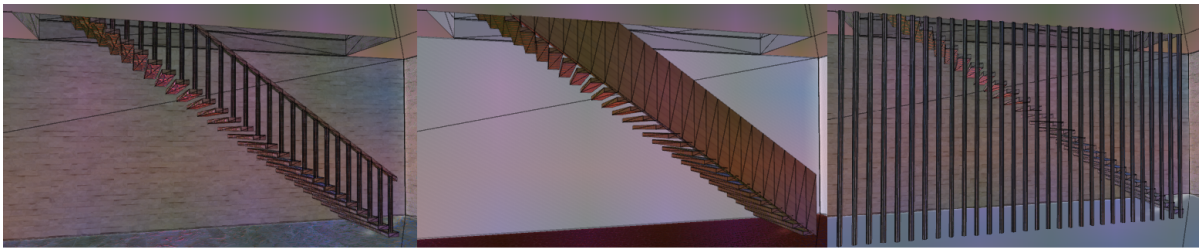


Figure 4.21: The three railing types a) piles b) solid c) wall height piles

Railings are not generated if the railing edge overlaps with any wall edge of the surrounding room, so that there are only railings on open edges where someone could actually fall. The railing edges are flagged correspondingly while executing the *Best Fit* algorithm.

4.4.3 Doors and Windows

After the staircases, the window and door cut-outs follow along the assigned wall and the start and end point of the respective aperture. The two-dimensional lines are extruded into a three-dimensional subtraction mesh where the extrusion depth is defined by the wall thickness and the width by the width of the window. Hence, these two-dimensional segments have to be extruded to a three-dimensional shape by giving a 2D line. This is achieved by extruding the segment, firstly in a 90° angle to the left and a −90° angle to the right (Figure 4.22b). This construct is then extruded to the top to produce a rectangular mesh which is used as a window frame; it can be subtracted from the room's mesh (Figure 4.22c). A second rectangular mesh with a larger vertical offset and a smaller height and width is then subtracted from the window frame mesh (Figure 4.22d).

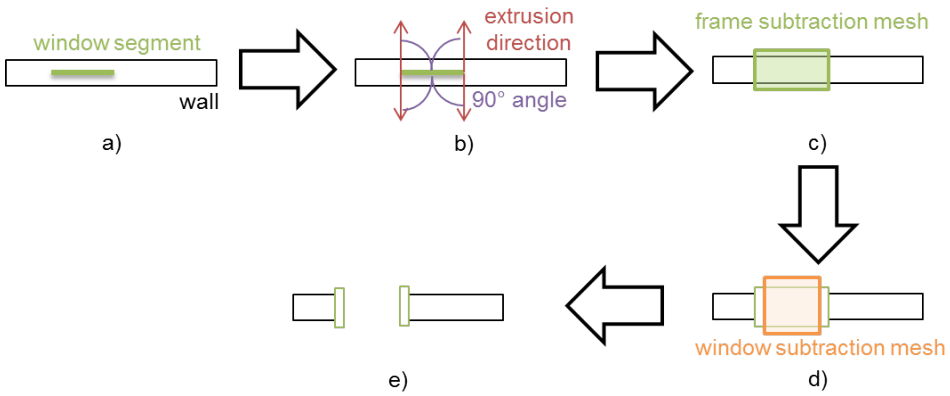


Figure 4.22: Two-dimensional creation process of a window subtraction including window frame.

The algorithm for doors is identical, except that the door's threshold is removed. As one can see, CSG drastically facilitates procedural modeling: Instead of modeling each wall segment separately, the corresponding manipulations can be applied precisely to an existing wall, and repeated easily.

4.4.4 Roofs

The roof outlines were calculated in Section 4.3.9 and consist of n surfaces on m floors.

For flat roofs, such surfaces are triangulated and given a height by extrusion. The offsetting process in Section 4.3.9 allows adding a minor overlap of the roof surfaces in relation to the building's outer walls (see Figure 4.23a).

The generation of hip roof surfaces by applying a straight skeleton algorithm to the roof area has been introduced by Laycock and Day [217]. There are different approaches to create a straight skeleton, but most rely on the idea to shrink the polygon step by step and collect all the intersection points which, once connected, form the inner skeleton. Aichholzer et al. [223] introduced the name *Angular Bisector Network*, relating to the idea of using the polygon angle's bisectors to construct the skeleton. Our application makes use of the implementation by Felkel and Obdrzalek [224]. Even though it was proven wrong in some corner cases (see [225, 226]), the simplicity, speed, and robustness of their algorithm, going hand in hand with the ability to handle holes in polygons, are convincing to us (see Figure 4.23b). The algorithm returns n lists of lists of vertices of each roof surface.

The vertices are marked with a Boolean *true* if the vertex is one of the original polygon's vertices. This flag is used when calculating the vertical offset of each roof part. Each list of vertices is then triangulated, and all vertices lying inside of the original polygon are raised (Figure 4.23c).

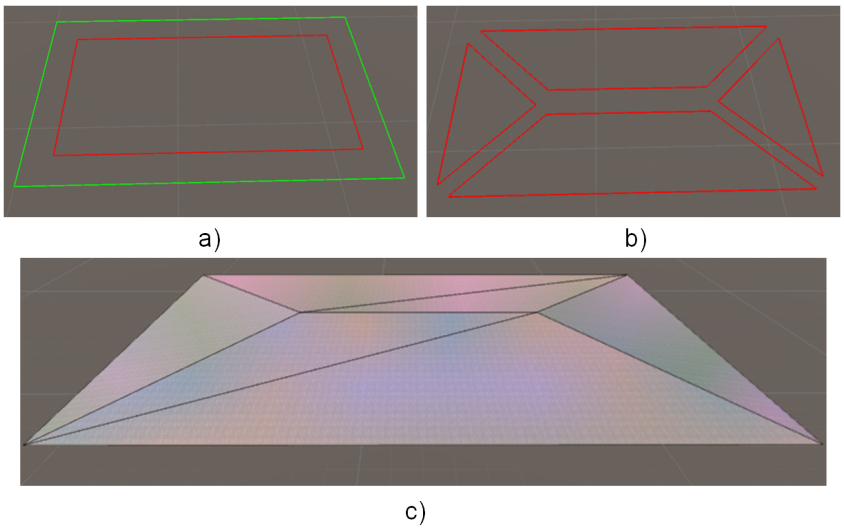


Figure 4.23: The outer and inner polygons mark the roof outline (**a**). The straight skeleton for the inner polygon is calculated (**b**), and each surface is triangulated and merged to a mesh (**c**).

4.4.5 Level Of Detail

The LOD concept involves multiple quality stages of a three-dimensional model, with the intention of optimizing memory usage during the rendering process [227]. There are several factors influencing LODs, such as the object's distance to the viewer, its movement speed, and its importance in the scene. The most influentcial factor contributing to the difference in quality is the vertex count of the mesh, but there are other factors, such as the resolution of the textures and the shader complexity. Biljecki et al. [228] stress the usefulness of LODs in three-dimensional city modelling and provide a detailled LOD categorization within the markup language CityGML [229]. Based on this definition, we introduce three LOD levels in our implementation to control the quality of the generated buildings, which roughly relate to CityGML's LODs 2 to 4 [50]:

1) LOD 0 - building with interior, floors and stairways,
2) LOD 1 - building hull with outer details (door and windows),
3) LOD 2 - building hull with roof structures and simple textures. Wall and window meshes and textures are omitted.

The building mesh generation procedure is responsible for the generation of each individual level, and includes an implementation to handle every LOD separately (see Figure 4.24).

4.5 Evaluation

We apply the proposed methods by Shaker, Smith and Yannakakis to evaluate our generator qualitatively, including a top-down evaluation via expressivity measures and a

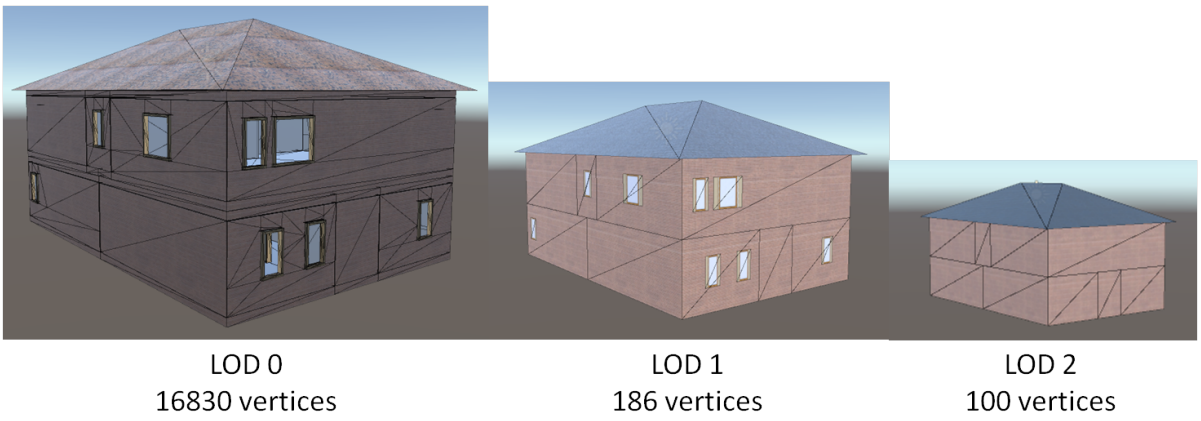


Figure 4.24: The same building is generated in three different LODs.

bottom-up evaluation by a user questionnaire (see Chapter 12 in [147]).

4.5.1 Questionnaire

The questionnaire was filled in by 32 persons, of which 7 were full-time game developers, 12 were hobby game developers, 9 were non-game developers, and 4 had no professional connection to IT. Their ages ranged from 22 to 40 years. 4 were female, 28 were male.
The 33 questions (see Table 4.1) were divided into the categories *general*, *building generation*, *usage*, *personal*, and a free comment field. Filling all fields except the comments was mandatory.
In order to test our method, the procedural part of this work was implemented as a Unity plugin (see Figure 4.25) and as a WebGL application and distributed to the participants so that they could experiment without time constraints[2]. Both UIs allow visual parametrization of building dimensions, room count, floor count, roofs, stairwells, LODs, and the window placement strategy.
A good usability of the tool was confirmed by all participants regarding installation, parameterization and use. Only the experimental generation of larger areas consisting of many individual buildings was criticized in the comment field by three testers in terms of the lack of a placement strategy and building diversity with similar parameters.
Seven participants requested a feature to post-edit a previously generated building. This wish confirmed our strategy to first generate an object procedurally, which could then be edited by an artist before the actual mesh generation.
The survey also revealed that a realistic appearance was almost achieved. The most frequent points of criticism were the sometimes suboptimal placement of the stairs, the length of the corridors (comments from two participants), as well as the proportions and aspect ratios of the rooms (three participants). All testers agreed that the diversity of the generated buildings was good, including the diversity of the textures. This confirmed

[2]Since the WebGL implementation with the same code base was many times slower than the native Unity editor plugin, we only use the plugin for performance measurements, e.g., in Figure 4.28.

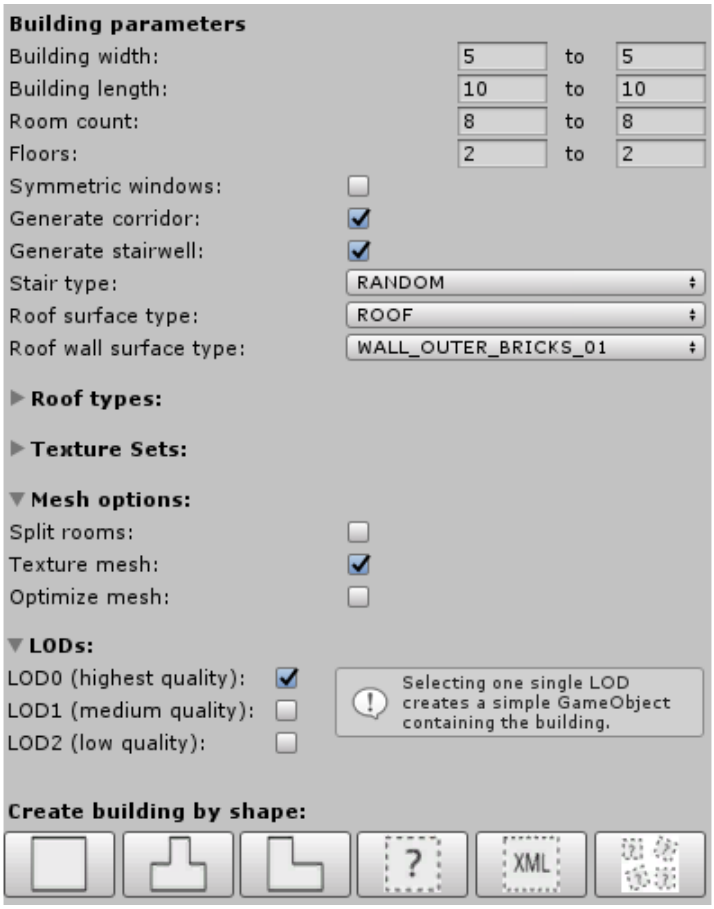


Figure 4.25: A plugin for the game engine Unity encapsulates our procedural building generator.

that the distribution algorithm worked well, even with a set of only 160 textures for all surfaces.
The applicability of the tool in game development was taken for granted. Only a few alternatives to our implementation are known on the market. Three participants suggested working out additional levels of detail between the individual LOD steps, two others wanted automatic lighting of the buildings for practical use. Furthermore, one game developer wanted more details in the buildings (thresholds, downpipes, chimneys, etc.).

4.5.2 Expressivity Measures

The authors of Chapter 12 in [147] proposed to avoid metrics that are related to the input parameters, hence we focused on values that express the quality and usability of buildings in games. The key figures we chose were

1. accessibility of rooms,
2. average room aspect ratio,

3. corridor area in % of the total floor size,
4. doors per room,
5. generation duration,
6. and successful builds.

As test data, 6270 buildings with random ground plans were generated on a maximum area of 10x10 units with 1 to 32 rooms and 1 to 19 floors. Only buildings with corridors and staircases were generated, as these two features were an elementary part of our work (see Figure 4.26 for a room layout of an exemplary test building).

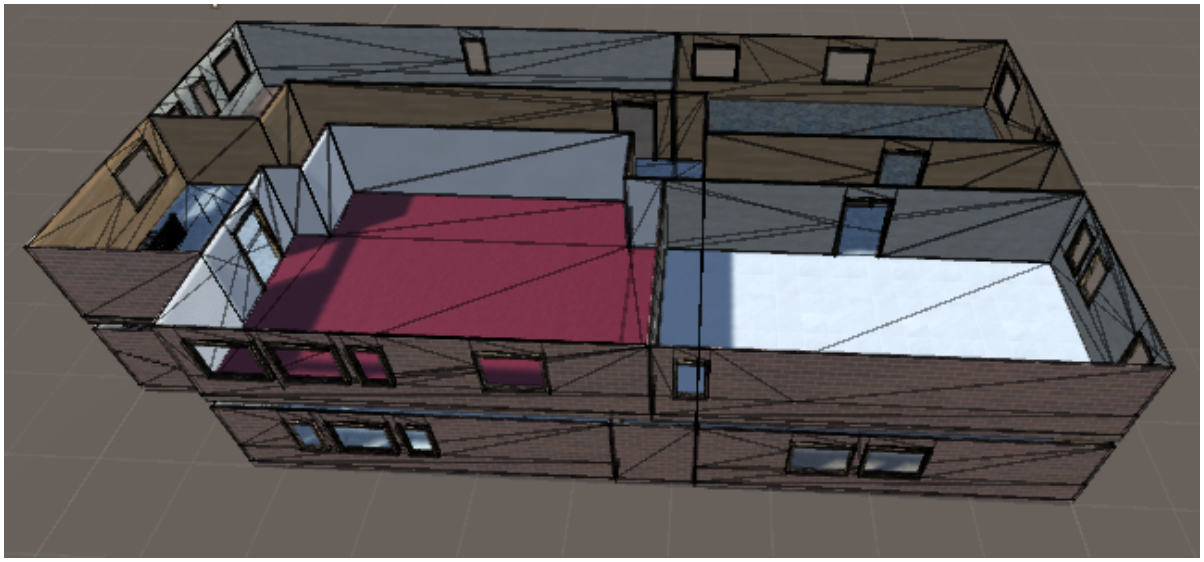


Figure 4.26: Exemplary output of the test building generator without roof.

Let us first look at the two technical indicators *successful builds* and *generation duration*. Figure 4.27 visualizes a *success rate* of 87,89%, a *success with warnings rate* of 3,55% and an *error rate* of 8,56% in a detailed heatmap. *Success* is defined by the successful creation of the object model and the resulting 3D mesh. These numbers prove a high stability and robustness with our method. The most common errors in generation are missing space to place all initial rooms in the given floor area, and too small rooms to form a hollow corpus for a room. Those two errors are characterized by the staircase artifact in the upper right quadrant (the generator only allows buildings whose floors have at least one room) and the bottom left where not all rooms can fit in an area of 10x10 units.
Figure 4.28 illustrates the duration of the generation process. This consideration is interesting with regard to the applicability of our algorithm in realtime scenarios in which, for example, a generated city grows during game time if the player moves into an area that has not yet existed.

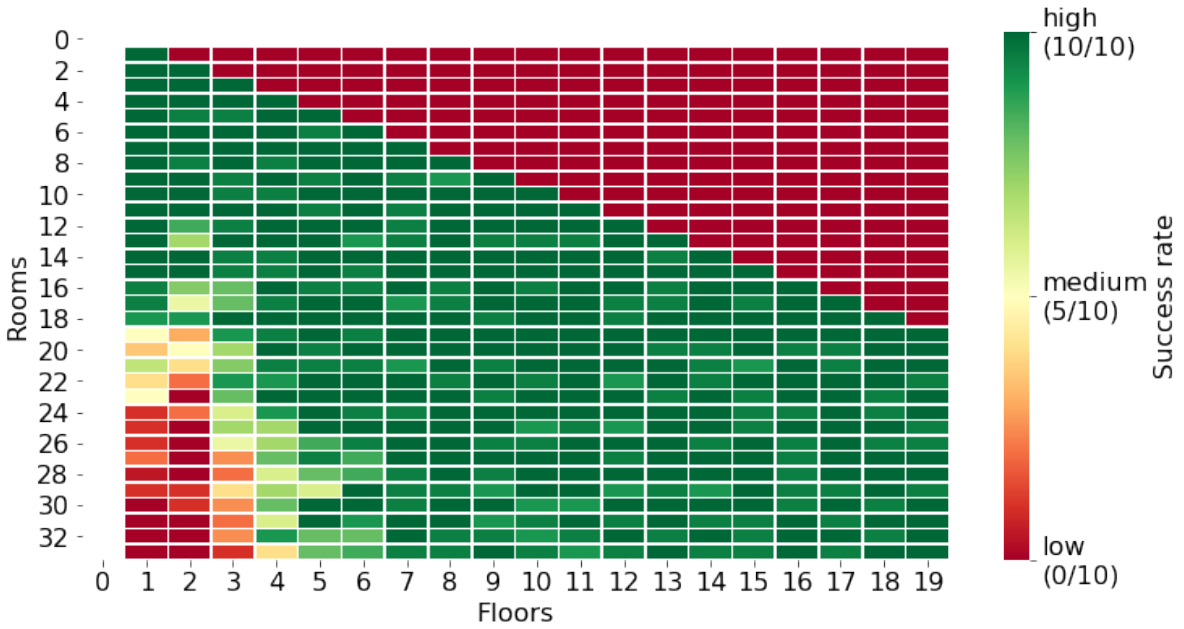


Figure 4.27: Each box represents 10 generator runs for the respective combination of rooms and floors. The upper right area shows only failures, because no building can be generated that has more floors than rooms. The success rate in the lower left area decreases when many rooms need to be packed in a small space.

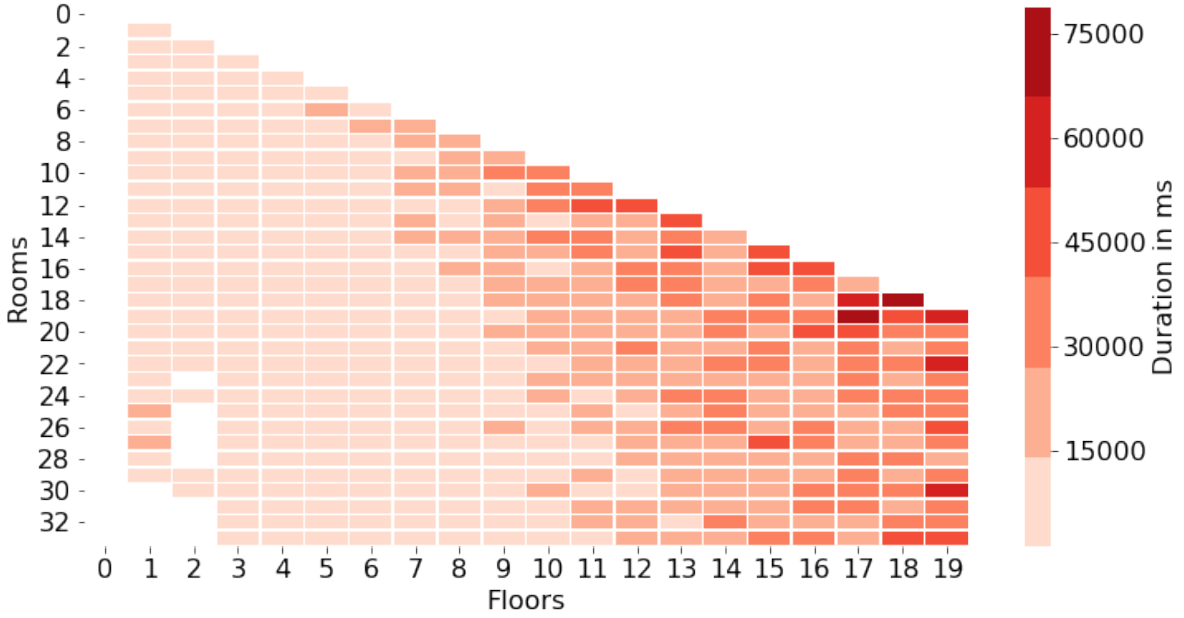


Figure 4.28: Generation of buildings with fewer floors can be used for realtime applications, whereas skyscrapers should be generated at design time.

The results show that our method allows for the real-time generation for houses with a

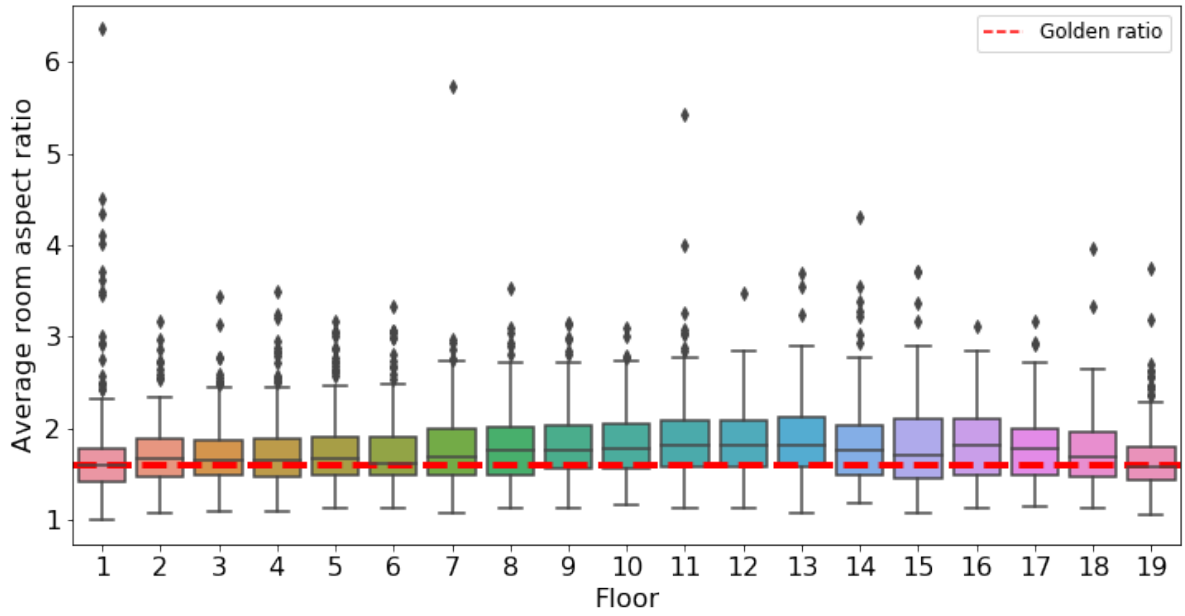


Figure 4.29: The median of the aspect ratio, regardless of the number of floors, for all rooms of the generated buildings varies close to the golden ratio (1.6), independent of the number of floors.

small number of floors (approximately 6-7) whereas it is not suitable for a large number of walk-in skyscrapers at runtime. Measurements of the individual generation steps show that the CSG operations take the most time. As the number of window openings increases, the duration of the mesh generation also increases. As a consequence, CSG operations are to be eliminated in favour of acceleration.

In the next measurement, we compared the aspect ratio of the individual rooms with the *golden ratio* which frequently serves as a reference when designing architecture [230]. Figure 4.29 shows that the aspect ratio is independent of the number of rooms, floors, or the random floor plan. The value varies around 1.6 which we consider good.

Backtracking made it easy for us to show that every room in a building is accessible. In combination with the *Doors per Room* indicator, which ranges between one and four for all the generated buildings, it is shown that an unrealistic image of rooms with too many doors is avoided.

Next, we consider the area occupied by the corridor compared to the total area of a floor. Since the approach of the longest path is designed to touch as many rooms as possible, it is logical that as the number of rooms on a floor increases, the size of the corridor areas increases as well (see Figure 4.30 for illustration).

Finally, we evaluate the distribution of the corridor polygons of 18 exemplary single-story buildings in order to demonstrate the diversity of our method. All floor plans in Figure 4.31 are based on random polygons whose dimension is determined by black framing. Although we do not make any form specifications, random shapes can still result to a T- or L-shape, which we consider as realistic. The corridors, marked in

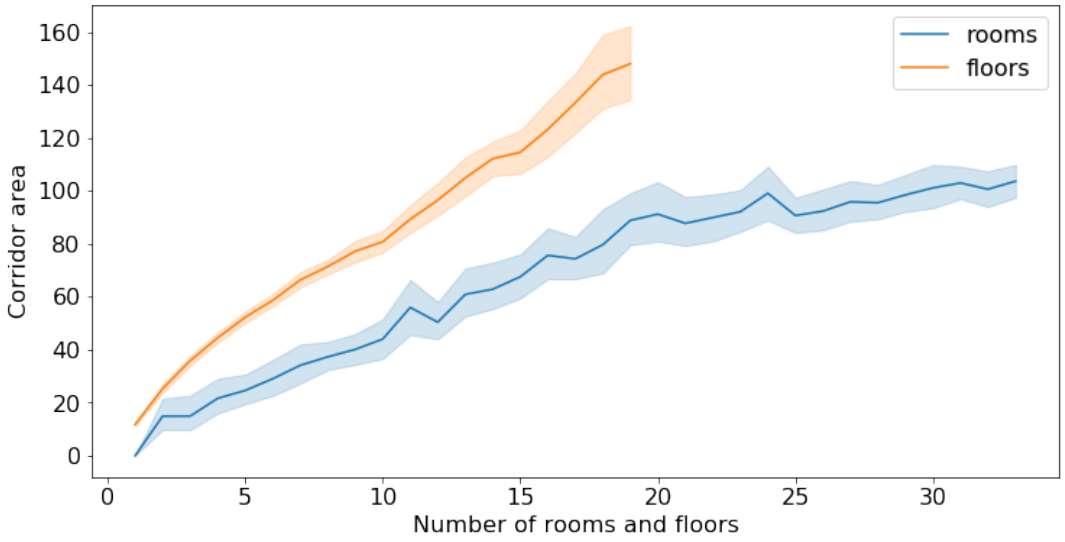


Figure 4.30: The orange bar shows the corridor area growing with the number of floors. The same applies to the blue line and the number of rooms. The pale area around the two curves shows the minimum and maximum of the corridor area, since different measured values were taken for each of the same number of floors and rooms.

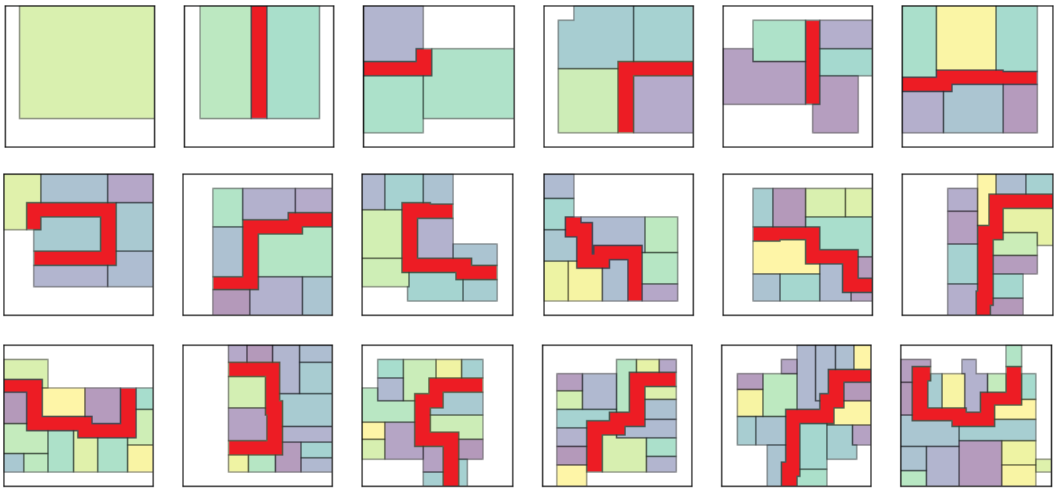


Figure 4.31: Exemplary floor plans of generated buildings with 1 to 18 rooms (top left to bottom right). The corridor is highlighted in red.

red, reliably connect all rooms on floors with up to six rooms. Here, the importance of the downstream step to force room-to-room connections is shown if corridor-to-room connections are not sufficient to reach all rooms.

We conclude that our evaluation shows that we can generate multiple homes with a believable and diverse layout in real time, as long as they do not have too many floors. Thus, our generator is best suited for rural residential areas. Inner cities with higher buildings should be generated at design time.

4.6 Conclusion and Outlook

In our work we have seen how a building structure can be defined, instantiated (while fulfilling a list of placement and validation conditions), and transformed into a three-dimensional, textured model which can then be used in any authoring tool or game engine. Whereas the definition of a data structure is fairly trivial, we have pointed out that valid object placement can be complex; it relies on many characteristics of floor plans and room shapes. Boolean operations on polygons and the *Best Fit* algorithm have proven to be invaluable tools for our building generation process. In the construction of the three-dimensional mesh, we strongly rely on the introduction of index/mesh group tuples, which make it easier to keep track of textures in a procedural design process. We can recommend this proceeding for algorithmic mesh generation, since the common techniques to generate densely packaged UV maps include tasks which require manual re-editing. Furthermore, we recommend to address ornamentation, furniture, lightning, room types, and room semantics in future research. Ornamentation (e.g., stucco or skirting boards) will improve the individual look of a building, and the introduction of furniture would make the rooms appear more realistic. Furthermore, lighting should be adressed in terms of optimizing illumination by placing glass elements or electric lights where needed. As a final improvement, we would like to continue our research to allow the generation of rooms in the attic (below the slopes of the roof), and the corresponding window and door placement.

Table 4.1: Results of the conducted survey on the perception of the implementation of our method (mandatory questions, average ratings rounded).

Question	Rating
General	
The building generator is easy to install.	4.4 / 5
The tool is easy and intuitively to use.	4.6 / 5
It is a good idea to use the tool as editor plugin for Unity.	4.8 / 5
It is a good idea to use the tool as WebGL application.	2.4 / 5
There are already enough tools out there serving the same purpose.	1.4 / 5
It is easy to set up the parameters for the buildings.	4.6 / 5
Building Generation	
The generated buildings are realistic.	4.0 / 5
The tool would ease the generation of settlements with many buildings.	3.6 / 5
The time it takes to generate a building is acceptable (Plugin).	4.6 / 5
The time it takes to generate a building is acceptable (WebGL).	1.4 / 5
The stairways are properly placed.	3.4 / 5
The corridors are properly placed.	4.0 / 5
The appearance of the generated roofs is good.	3.6 / 5
All rooms in the buildings can be reached.	5.0 / 5
The diversity of the buildings is satisfying.	4.6 / 5
The diversity of the textures is satisfying.	4.2 / 5
The quality of the generated mesh is acceptable.	4.4 / 5
The relation between successful and failed generations is good.	4.2 / 5
I see an added value to the creation of Serious Games by using the plugin.	3.8 / 5
I see an added value to the creation of games by using the plugin.	4.2 / 5
The three LOD stages are believable and useful.	4.2 / 5
The user can take enough influence on the generation process.	3.4 / 5
Usage	
It is easy to embed the buildings into a game during design time.	4.6 / 5
It will be easy to embed the buildings into a game during runtime.	4.0 / 5
The generated buildings are easy to manipulate manually afterwards.	3.0 / 5
There is a need to manually improve the generated buildings afterwards.	2.3 / 5
Personal	
What is your age?	avg. 29
Would a procedural building generator ease your job?	2.2 / 5
What are your level design skills?	1.8 / 5
Are you familiar with architecture?	2.0 / 5

Chapter 5

Procedural Generation of Interactive Stories using Language Models

The creation of story generating systems is a long-standing field in the domain of procedural content generation for games (PCG-G). A story generating system is designed to generate coherent, credible, and dramatically meaningful narratives [231]. This is a task that even human authors fail to accomplish from time to time, often due to a limited budget and a heavy workload. These last two stumbling blocks are a common reason to explore procedural storytelling, even if the aspect of time pressure is always under discussion [232]. It is not only useful for the main plot of a game, but also for subplots. In this way the authors retain sovereignty over the main storyline so that the course of play remains known for the entire development team.
Another good argument to deal with procedural storytelling is the creativity that an algorithm generates. People often write about their personal experience or modify stories that they have read themselves. In contrast, an algorithm based on the vast set of available literature and language models draws from a nearly infinite pool of rules, possibilities, phrasing and connections. So, instead of reflecting the opinions, prejudices, tastes or moods of a single author, the algorithm returns a mixture of its training data. However, even these can be biased, and they might violate ethical principles.
The use of stories in the domain of games, may they be hand-written or generated, is manifold: Some games only aim to have a basic story to convey mood and setting [233], so that the player can put herself in the position of the protagonist. Others live from diverse, branched stories that take the decisions of the players into account and might even have different endings. In total, we identified five different classes of games, characterized by the nature of their story:

- games without story (e.g., *Tetris*, *Rollercoaster Tycoon*),
- games with a simple background story to convey mood and setting (e.g., *Need for Speed*, *Street Fighter II*),
- games with complex but fixed stories (e.g., *Final Fantasy VII*, *Legend of Zelda*),
- games with interactive stories that convey the feeling that the player can influence the course of the game (e.g., *Borderlands*, *Mass Effect*),
- games with interactive stories whose course and ending can be actively influenced by the player (e.g., *Life is Strange*, *Detroit: Become Human*).

This work focuses on the last category, namely on the generation of interactive stories that the player can influence during gameplay. Our motivation is that text adventures are currently experiencing a revival through the generation via language models. Similar to Nick Walton's *AI Dungeon 2* [234] and Nathan Witmore's *GPT-2 Adventure* [195] we make use of statistical language models to generate a unique story with each new run of the generator.
We distinguish our approach from the approaches mentioned above by providing a reasonable set of actions from which the player can choose to continue the story, similar to the famous *verbs* in traditional LucasArts adventure games (e.g., *Day of the Tentacle* [235] (see Figure 5.1)). Presenting three methods of action generation, we compare the results in regard of creativity, controllability, coherence, mood, and variety.
In addition, we evaluate two processes of story generation in terms of feasibility and performance: one in which the generator reacts spontaneously to the previously executed action, and the other in which all actions and their consequences are precalculated. We examine how to control the development of the stories by managing the state, progress and inventory outside the generator and by placing prefabricated text modules in the generated paragraphs to keep objects, places and people in the storyline. We will not

Figure 5.1: Actions (verbs) in the adventure game *Day of the Tentacle* by LucasArts.

focus on retraining the language model beneath the generator which has been done by Walton and Whitmore and has proven itself valuable for adventure-like free text input such as *take sword* or *open door*. The work described here was published in [236].

5.1 A Hybrid Approach of Language Models and State Management

We use the creative possibilities of language models in our approach and add controls to allow the player to interact with the story generator. In the following sections we will look at the three phases of initialization, runtime, and ending, as outlined in Figure 5.2.

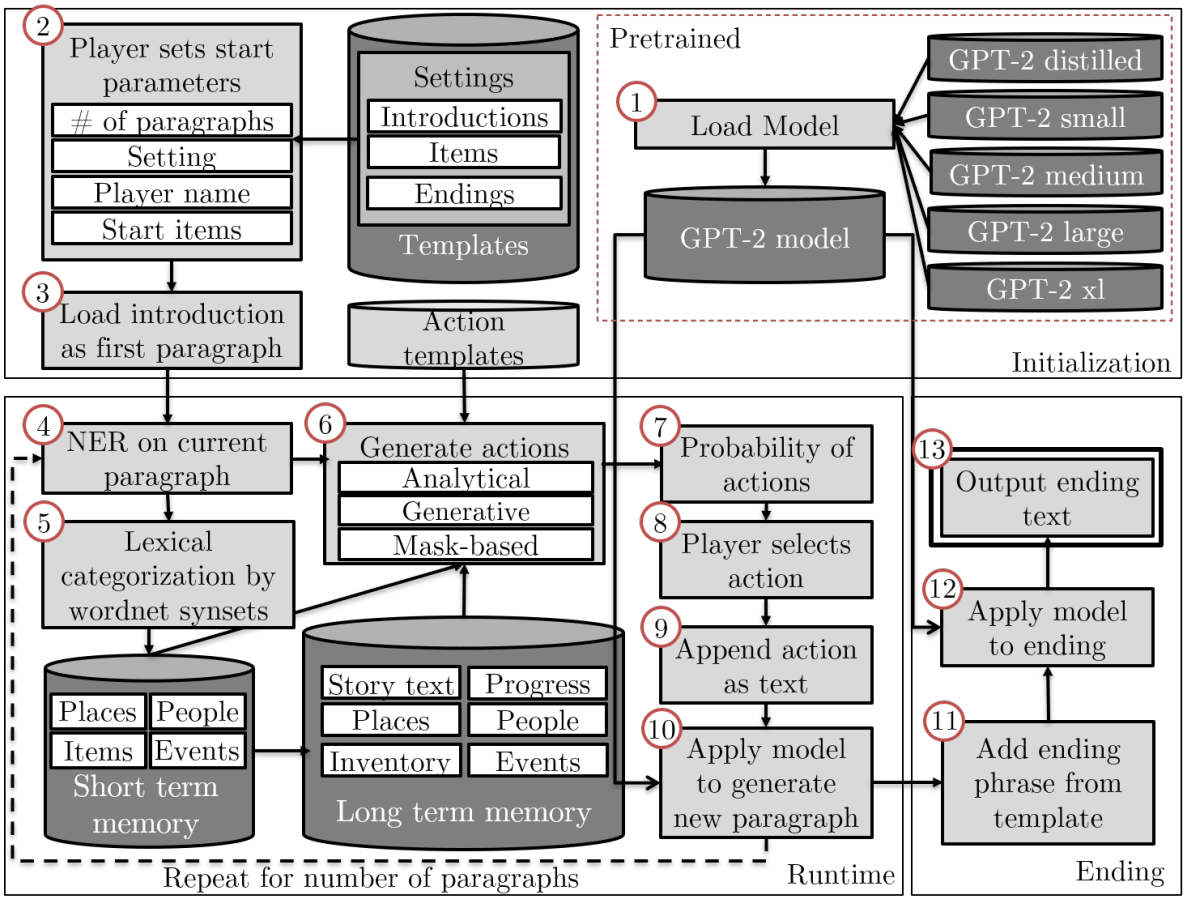


Figure 5.2: The hybrid story generation approach is split up into the three steps initialization, runtime, and ending.

Subsequently, we discuss the challenges of coherence, action generation and performance of the story generator.

5.1.1 Initializing Language Model and Templates

The initialization phase is a two-step process. The model is loaded before the generator is started (step 1 in Figure 5.2). The player can choose between different model variants (see Table 5.1), which are mainly determined by their size which in turn depends on the number of layers and parameters of the neural network. The models will be examined again later with regard to diversity and speed of application. We did not retrain a model as done by Whitmore [195].

Table 5.1: GPT-2 pretrained models.

Model	Layers	Hidden layers	Parameters (million)
DistilGPT-2	6	768	81,9
GPT-2 small	12	768	117
GPT-2 medium	24	1024	345
GPT-2 large	36	1280	774
GPT-2 xl	48	1600	1558

In a second step of the initialization phase (step 2 in Figure 5.2) the player selects various parameters such as the number of paragraphs the narrative should have, a player name, optional party members, and a set of items for the game. Afterwards, one of multiple possible introductions is loaded from the settings object and presented to the player (step 3 in Figure 5.2).
Since the introduction is the basis of the story, it should contain as much information about characters, locations and items in the story as possible. Only with a sufficiently large amount of information can the language model establish a context and later credibly continue the story. Here we introduce two party members of the main character, whose name the player can freely choose and who will accompany her during the journey through recurring mentions. In addition to printing out the introduction as a first paragraph, the items of the game are added to a visible inventory so that the player knows what she is carrying around.

5.1.2 Course of the Plot

The Runtime phase is a repetitive process that continues until the player has completed a number of paragraphs of her choice. It starts with a named entity recognition (NER) on the last paragraph - which is the introduction in the first run. Entities are not only recognized but also categorized with *WordNet Synsets* in order to select the group of nouns that the player can physically interact with [237] (step 4 and 5). For example, those categories can be

- noun.animal (nouns denoting animals),
- noun.artifact (nouns denoting man-made objects),
- noun.food (nouns denoting foods and drinks),

- noun.plant (nouns denoting plants), and
- noun.object (nouns denoting natural objects (not man-made)).

This avoids later actions like *Take love* or *Talk to faith* in the action generation. Such actions may seem creative in some places, but more often they disturb the flow of the game because they are confusing or inappropriate.
Those entities are stored in a *short-term memory* with a life-cycle spanning the current paragraph. They are also transferred to a *long-term memory* to be referenced later. This allows the generator to relate to past events or past conversations and hence add control and consistency to the stories. Distinguishing between a short-term and a long-term memory allows the game mechanism to forget objects of lesser importance over time.

Redundancy Avoidance of Entities

Entities can occur more than once with different names in a paragraph. The NER process will detect the expressions *Elisabeth* and *Queen* from the sentence *Elizabeth II is Queen of the United Kingdom*. For a human being it is clear that both terms refer to the same person but for a NER component it is not. Thus, the story generator will treat *Elisabeth* and *Queen* as separate entities and might thus simultaneously generate the actions *Talk to Queen* and *Talk to Elisabeth*. This leads to an inconsistency in the story and disrupts the flow of reading, so expressions that refer to the same entity must be reduced to a single expression. We use *coreference resolution* to detect such expressions and keep the first expression found while discarding all others. In first evaluations, we tried the Spacy *KnowledgeBase*[1] for *coreference resolution*, which worked well for simple and plausible texts. But since our story generator often creates scenarios that do not always have these two characteristics, either because unknown names appear or coreferences are not always given, the knowledge base was only reliable in a few cases. In the end, we used the implementation by Hugging Face as presented by Wolf in [238]; it is based on word vectors and neural networks and detects coreferences more reliably.

Action Generation

As mentioned in the introduction we implemented three methods to generate actions that the player can use to continue the story (see Figure 5.3 and step 6 in Figure 5.2).
Such an action is not only a verb and an object but has multiple attributes as shown in Listing 5.1. The *type* informs about the named entity type, the *action* about the action type, the *sentence* about the full sentence based on a template as it is added to the story if the player picks this action, the *simple sentence* contains a simplified version of the template sentence, and the *probability* is the sentence probability calculated using the underlying language model.

[1] https://spacy.io/api/kb

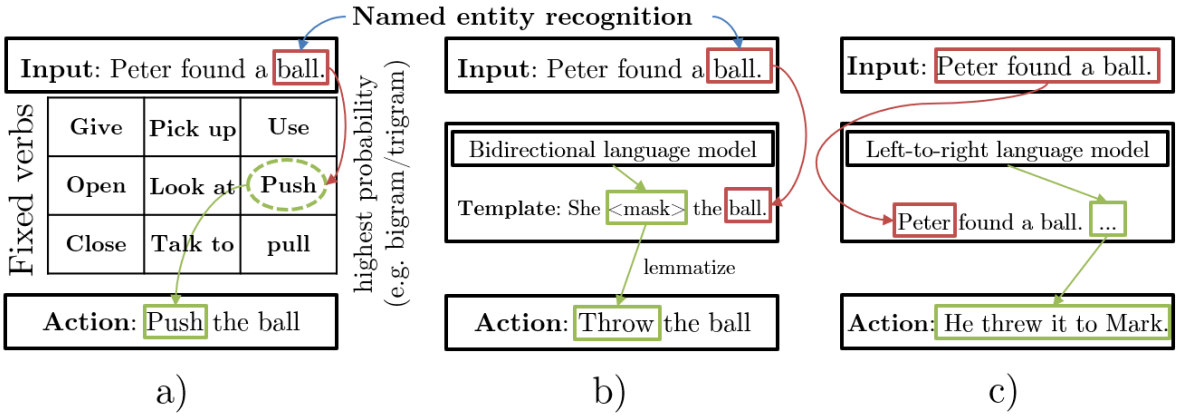


Figure 5.3: The three different ways to offer fixed actions to continue a story: The analytical (**a**), the mask-based (**b**) and the generative approach (**c**).

Listing 5.1: The action object potion with its individual properties

```
1 {"name":"potion", "type":"item", "action":"use", "sentence":"Peter
    bravely used the potion without hesitation.", "simple": "Peter
    used the potion.", "probability":0.76}
```

Approach a) in Figure 5.3 uses NLP to analyse the previous story paragraph and offer senseful actions such as *take*, *pick up*, *visit*, *use* or *push* from a static list in combination with an item, person, or place.
Since the actions are derived from the recognized entities, a paragraph with many entities can lead to an equally high number of actions. Therefore, prioritizing the actions is useful. We do this in step 7 by checking the actions for plausibility. This check computes the probability of occurrence of a combination of the action and the object in question. Several approaches were considered. First, we evaluated ngrams (bi- and trigrams) to calculate the probability p of an action, e.g., *Peter used a potion* would be much more likely than *Peter talks to a potion*. The trained ngram model was based on *Reuters corpus* containing 10.788 news documents with 1.3 million words [239]. In many cases, however, even simple combinations such as *takes an apple* could not be found in our generated list of trigrams. Hence, we evaluated semantic similarities in the second place using word vectors to calculate probabilities. Here, the result was unsatisfactory, too, since the similarity often could not give any information about the connection between verbs and nouns. For example, the combination of *push* and *love* was rated higher than *take* and *apple*. The probability calculation based on the GPT-2 model finally brought convincing results. Calculating the loss as an error of the model for the given sentence helps to figure out which sentence is more likely to be found in the training corpus of GPT-2. We have observed that the use of a simple sentence reduced to subject, predicate and object, and thus linguistically normalized, provides a better comparable probability than the embellished sentences based on the templates.
The approach b) is also based on the extracted entities of the previous paragraph. Here, a simple sentence is formed from the combination of a personal pronoun and the object,

Table 5.2: Results of the masked sentence "He <mask>the ball." and "She <mask>the ball." including the probabilities and the lemmatized verb.

Gender	Sequence	Score	Lemmatized verb
male	He catches the ball.	0.049	catch
male	He dropped the ball.	0.033	drop
male	He threw the ball.	0.026	throw
male	He throws the ball.	0.024	throw
male	He touched the ball.	0.021	touch
female	She dropped the ball.	0.051	drop
female	She throws the ball.	0.039	throw
female	She threw the ball.	0.035	throw
female	She catches the ball.	0.030	catch
female	She kicked the ball.	0.028	kick

in which the position of the verb is masked. Using a bidirectional language model (such as BERT) allows to replace this placeholder by the statistically most reasonable verb. Table 5.2 shows sequences that fill the masked field in "He <mask>the ball." and "She <mask>the ball." Although it cannot be explicitly requested that a verb is used for the mask, linguistically there is rarely another option for the model. Two particularities stand out: First, the chosen personal pronoun plays a role in the choice of the verb. Thus, the probabilities differ for *he* and *she* as the first word in the masked phrase. Secondly, it can happen that the same verb is used again in a different tense. Thus lemmatizing the verbs is recommended. Here one can clearly see the gender bias, which is based on the language model's training data. Bordia and Bowman state that this bias can even be amplified in the models and propose methods to measure and minimize this bias [240]. The probability used to decide which verb fits best comes with resolving the masked word so that is does not have to be calculated manually as it is done in approach a). Control over items while using the actions *use*, *take*, and *combine* can be maintained by using *WordNet Synsets* and checking the chosen verb for synonyms. If the verb is *take* or a synonym, the object can be added to the player's inventory. Usage of an item is processed analogously. Combining items is a special case; it can once again be achieved by masking the target object in a sentence. A combination is always preceded by two objects in the inventory. Masking is used to determine the statistically most probable result of the combination using BERT.
Let us look at an example.
Input: He combined iron and hammer to receive a <mask>.

- **Output 1:** blade
- **Output 2:** handle
- **Output 3:** shield

The items *iron* and *hammer* are removed from the inventory and the masked object, determined by the BERT model (blade, handle, or shield) is added in return.

Approach c) uses the same language model that is deployed for the paragraph generation and to create the subsequent sentence using the previous paragraph as input (see c) in Figure 5.3). The result is a freely formulated sentence which can not be fit to the attributes of an action object (see Listing 5.1) without a huge effort, because the generator does not generate sentences in the pattern of *subject, verb, object*. As a consequence, control over using, taking and combining items in the player's inventory is lost. Furthermore, the following example shows that actual actions in the form of a command, as it is typical for a text adventure, are generated only in rare occasions by this approach.

Input: The conference was over and Diana was on her way home. She...

- **Output 1:** was in charge of organizing the conference, taking the oath of office.
- **Output 2:** walked onto the street and then the entire group of people had heard about the situation.
- **Output 3:** turned around and asked me if I had any reservations.

Generating new Paragraphs

If the player has chosen an action to perform in step 8, its full clause is appended to the text of the story (step 9). This new text forms the basis for the next generation process. When generating texts using GPT-2 in step 10, the input of a source text is crucial. Too short input does not give the model enough context to generate a credible continuation. An input that is too long increases the memory consumption when calculating the subsequent text and also the duration of the generation process. In this evaluation the previous 300 characters were used.

Steps 4 to 10 are now repeated until the initially set number of paragraphs is reached.

5.1.3 Coming to an End

If this is the case, a template for introducing the **Ending** phase is initialized. In step 11, an analysis determines the sentiment of the last n sentences. Based on this, a template of the respective mood (positive or negative) is loaded, and placeholders for the protagonist are filled (step 12). In a last step the generator creates a final paragraph based on the template before the game ends. We have found that an ending is more believable if there are only one or two paragraphs following the template-based ending. Several paragraphs would give the impression that the story is to be continued after the end, and the effect of the prepared paragraph heading for an end fades away. Mostafazadeh et al. contribute the *Story Cloze Test* which is able to determine the end to a four-sentence story based on the *ROCStories* corpus, a collection of 50.000 commonsense stories [241]. This test allows a precise evaluation of the quality of the generated ends.

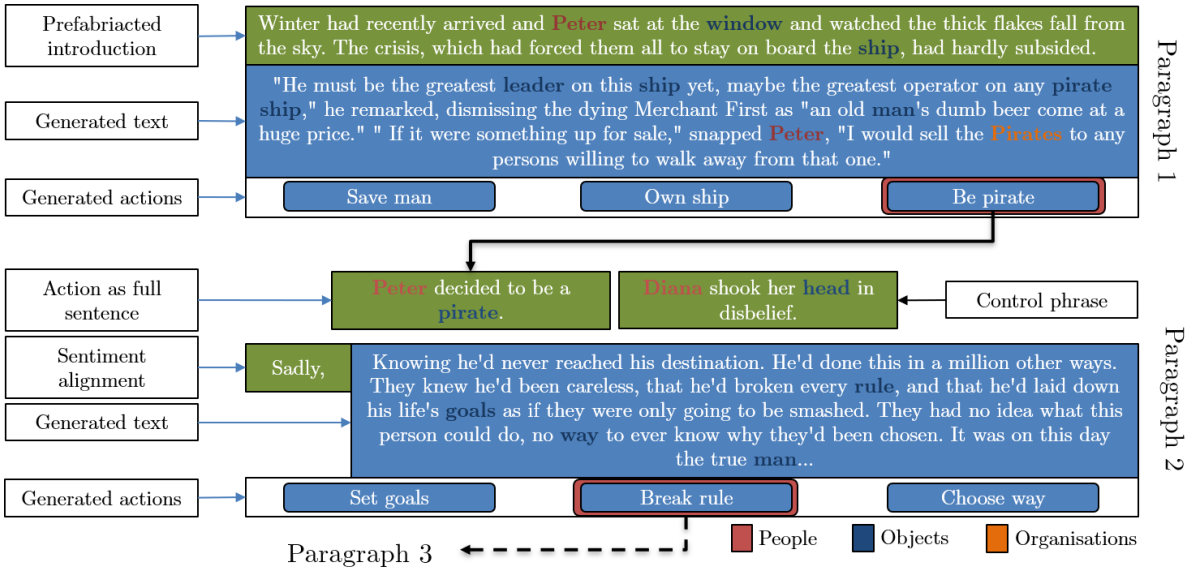


Figure 5.4: First two paragraphs of a story with all their components. Paragraphs have been shortened for reasons of space. The green components are based on text templates, the blue boxes are generated using language models. Named entities are highlighted according to the legend in the bottom right corner.

5.2 Implementation, Observations and Enhancements

This section focuses on the observations we made while implementing and testing our approach. We discuss the technical realisation, performance, character diversity, coherence, and the alignment of sentiments. The resulting improvement potentials are shown, and their implementation is discussed. The exemplary output of a story generation after all enhancements discussed in this section will also be presented (see Figure 5.4).

5.2.1 Technical Realization and Performance

We did two separate implementations to generate a story based on language models. On the one hand, Twine was used, a tool that allows to write interactive, non-linear stories and make them available to the players on a website. When using Twine, it is essential that all branches of a story are precalculated since the game is compiled and therefore cannot be updated while playing. The advantage is obvious: The entire calculation can be done on a suitable system, and the generated story can be played on any device. If desired the game can be passed on, and the limitation that language models usually do not provide a reproducible output can be avoided. The disadvantage is that all paths of a story have to be generated, no matter whether the player takes them or not. If only one action is offered to the player, a total of num_paragraphs = story_depth paragraphs have to be calculated. If more than one action is offered, num_paragraphs equals to:

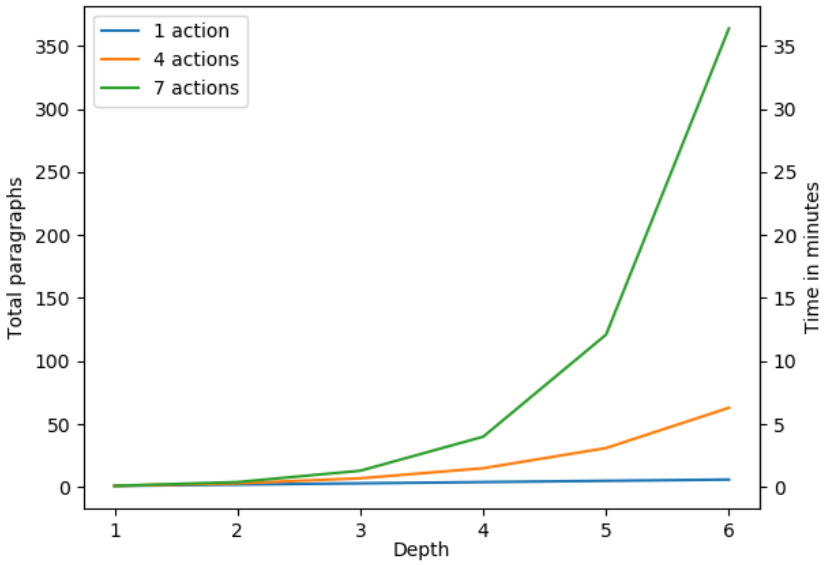


Figure 5.5: The number of paragraphs for one, four and seven actions per story. The depth growths exponentially while the generation of a linear story (one action) remains constant.

$$\text{num\_paragraphs} = \frac{\text{num\_actions}^{\text{story\_depth}} - 1}{\text{num\_actions} - 1} \tag{5.1}$$

The variable story_depth corresponds to the number of paragraphs a player sees to get to the end of the story. Equation 5.1 shows the exponential character of the calculation when increasing the possible actions per paragraph. When measured on an Intel i7 processor and a GeForce GTX 1080 TI with 11 GB of RAM the generation took approximately 10 seconds for one single paragraph (including NLP preprocessing, applying templates, and generating new texts; based on the GPT-2 small model). Figure 5.5 illustrates the growth of the number of paragraphs and the corresponding generation time (see y axis on the right).

The second implementation reacts to the user input and calculates the subsequent paragraphs and actions on the fly, during the game. Since there is no precalculation of possible subsequent paragraphs in this approach, the depth of the story corresponds to the number of paragraphs to be calculated. This eliminates the exponential growth of precalculation. The disadvantage is that the device on which the story is played must have the capability to generate the text in real-time using a neural network-based language model. Even with suitable hardware, the generation of a new paragraph takes some time, so that the flow of gameplay may be disturbed. Furthermore, this requirement prevents the possibility to play the game on mobile devices, unless there is a constant connection to a corresponding cloud service. For these two reasons we refer our further considerations to the precalculated variant.

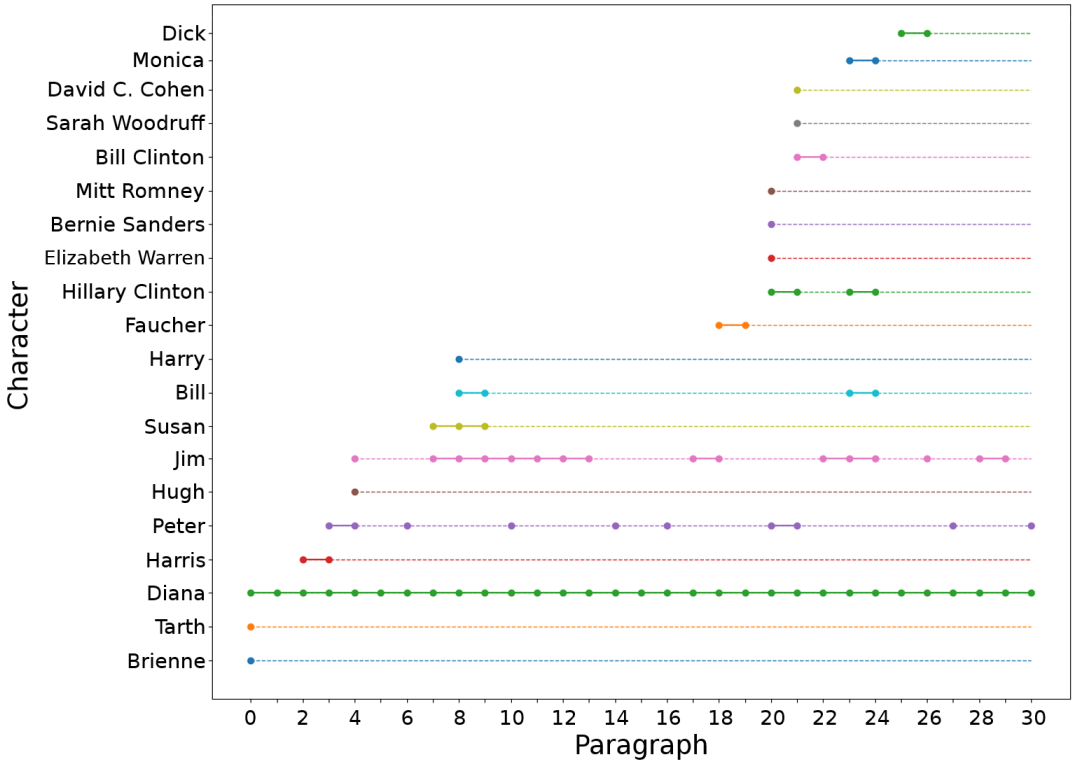


Figure 5.6: Nearly each new paragraph introduces a new character except the main character and the two party members.

5.2.2 Character Diversity

We have observed that the number of characters increases with the number of paragraphs played. Recurring characters are extremely rare, which has a negative influence on the story's context. Thus, there is no long-term relationship between protagonist and antagonist, lasting love affairs or friendships. In our example, except for the main character named in each paragraph and the two optionally pre-defined party members, no relationship can be built up with another character, as these usually have a life span of only a few paragraphs. The fact that people are forgotten during the course of the game is due to the fact that the language model never takes the whole story as input, but only the last n sentences. Of course, this limited input does not include all persons that have appeared so far.

It quickly became apparent that the number of characters involved had to be controlled in order to keep the course of the story within a boundary. For this purpose, a simple replacement of new names in the text with already known names was done, excluding those of the main character and his two companions. Care was taken to replace them with the same gender, so that personal pronouns continue to match the gender of the person. The gender detection is done by the *gender-guesser* library which relies on the dictionary of first names by of Jörg Michael [242]. It has also been observed that when a story begins with an excerpt from well-known literature, characters from the book

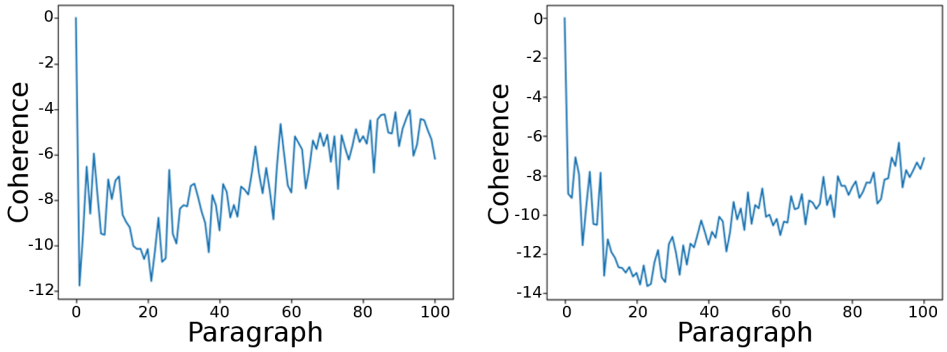


Figure 5.7: A story with 100 paragraphs with (left) and without (right) elements supporting coherence.

appear. On one hand, this results in an interesting story, on the other hand, it does not prolong the time during which a character appears in the story.

5.2.3 Coherence

One of the biggest challenges in generating text adventures automatically is to make the context of the story clearly visible and maintain it across all sections. Although the results of the generated texts using transformer-based language models are very credible in themselves, there is a lack of a red thread when generating longer texts. Through the measures mentioned in Section 5.1, such as the regular naming of companions or items that the player carries in his inventory, a positive effect on coherence was observed. The following list shows some examples for our self-written control sentences:

- [person] rolled [hisher] eyes.
- [person] cleared [hisher] throat.
- [person] took a deep breath.

We have measured the coherence by applying the four-stage topic coherence pipeline presented by Roeder, Both and Hinneburg. They split a word set into pairs of words. Based on a reference corpus the word probabilities are calculated. Then the entire agreement of all pairs is calculated using Normalized Pointwise Mutual Information (NPMI). All these scores are aggregated (arithmetic mean) to a final *uMass* coherence value with a range of $-14 < x < 14$ [243]. We use a word set of ten topics extracted from the respective texts on the basis of Latent Dirichlet allocation (LDA) [244]. In order to determine a reasonable number of topics, we have calculated the coherence of a prepared text for different numbers of topics. The highest coherence and the most meaningful topics was found to be at a number of ten topics. Figure 5.7 shows a comparison of a story with (left) and without (right) naming companions and inventory items. It shows that the coherence can be slightly improved by interspersing existing information on persons and inventory. This can be explained by the fact that the LDA extracts persons and objects as topics and recognizes them later in subsequent paragraphs.

On a perceptual level, it was also observed that coherence is not always the most important factor when it comes to the entertainment value of the game. Often it is also the game mechanics or a humorous element that causes a positive reaction of the players. Nevertheless, measures that increase the coherence of the story are important to increase the general acceptance of a procedurally generated story.

5.2.4 Aligning Sentiments

We have also observed the relationship between positive and negative moods in individual paragraphs by means of a sentiment analysis, and we have noticed that mood does not remain the same over a certain number of paragraphs, but changes indiscriminately[2]. Thus, we introduced control words like *luckily* or *unfortunately* as further input for the text generator.

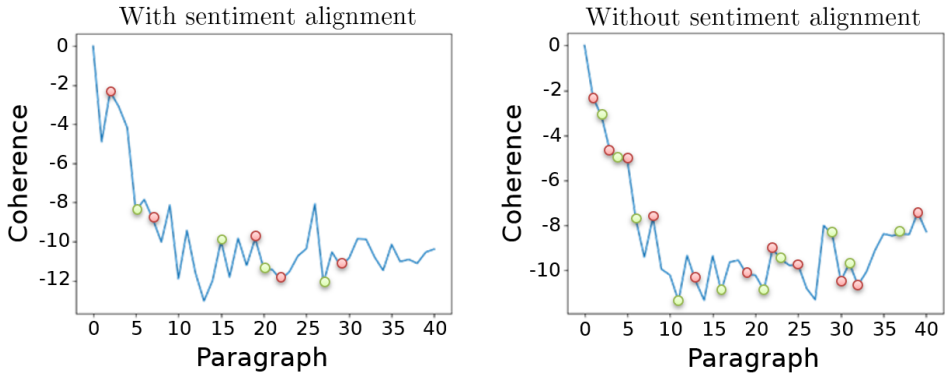


Figure 5.8: A story with 40 paragraphs with and without sentiment alignment. The red dots signify a negative mood change, the green dots signify a positive mood change.

Figure 5.8 shows that we were able to influence the mood of some paragraphs through the control words. In our example, the mood did not change 31 times (left), but 19 times (right). On the other hand, it also shows that this instrument does not seem to have any influence on the coherence since the two upper graphs are similar.

5.2.5 Actions

We have evaluated three different approaches constructing actions that the player can select to continue the story. The *analytical* and the *mask-based* approach both aim at generating concrete actions. The *generative* approach produces the most appropriate sentence that continues the story without necessarily containing an activity. Table 5.3 gives an overview of the observed characteristics of each method. The analytical approach is certainly the one that can and must be controlled most. It allows a more targeted interaction with people and objects. Partially, the actions get implausible because interactions with non-material objects can occur, such as *take love*. Also, actions

[2]We measure mood by applying the *bert-base-multilingual-uncased-sentiment* model by NLP Town [245]

Table 5.3: Three approaches to dynamic action generation.

	Analytical	**Mask-based**	**Generative**
Controllability	high	medium	low
Diversity	low	medium	high
Interactivity	high	high	low
Complexity	high	low	low
Authenticity	medium	high	high

are often repeated, which is why a list of already generated actions was kept in the implementation, allowing to hide already offered actions. Repeated actions have not been negatively noticed with the mask-based and generative approaches.
In our tests, the *mask-based* approach offered a good balance between controllability and variety. The generative approach tends to create a narrative out of the text adventure that is more a novel and less a game.
Actions in which objects could be picked up were very rare in our tests. This results in an even lower probability of being able to combine two objects. As mentioned in Section 5.1.2 synonyms for the verb *take* were used to increase the frequency of such an action. The use of the wordnet corpus [246] turned out to be difficult, because the synonyms for *take* can have other meanings than taking objects. For this reason, a curated, static list of matching synonyms was created and used.

5.2.6 Different Language Models

We have analyzed the various language models from Table 5.1 for their coherence using the Topic Model Coherence described above. In each case, stories with one possible action and a length of one hundred paragraphs were considered. Figure 5.9 describes, similar to the previous figures, the context of the topics extracted from the paragraphs via LDA. The consideration is done holistically and not sequentially from two consecutive paragraphs. It shows that GPT-2 small achieves the highest average coherence. The GPT-2 models medium, large and xl are quite close to each other, and the model DistilGPT-2 achieves a visibly lower coherence in the generated example stories. This can be explained by the fact that GPT-2 small is able to establish a logical connection between the paragraphs (which is apparently difficult for DistilGPT-2), but due to its limited domain it deals more often with the same topics. The variety of contents of medium, large and xl seems to have the effect that the models tend to change subjects frequently.

5.3 Evaluation

To determine whether our method can effectively tell more credible, coherent stories, sample Twine stories with a length of ten paragraphs were generated for play testing. 10 players with an age between 24 and 65 years (2 female, 8 male) and no experience in game

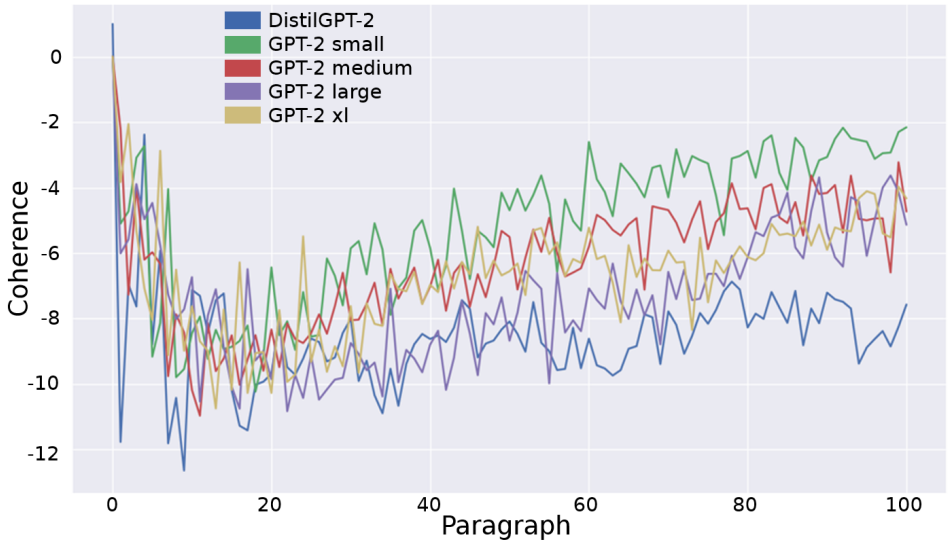


Figure 5.9: Coherence of the different language models in stories with a length of 100 paragraphs.

design played 2 of 4 generated stories with 8 paragraphs and 3 actions. They were told to pay explicit attention to the coherence and credibility of actions after each paragraph. 2 games contained neither a unification of characters nor control sentences. The other half did. 100% of all players found the games with activated coherence control were more credible and realistic. 40% noted, however, that the character behavior is implausible from time to time.

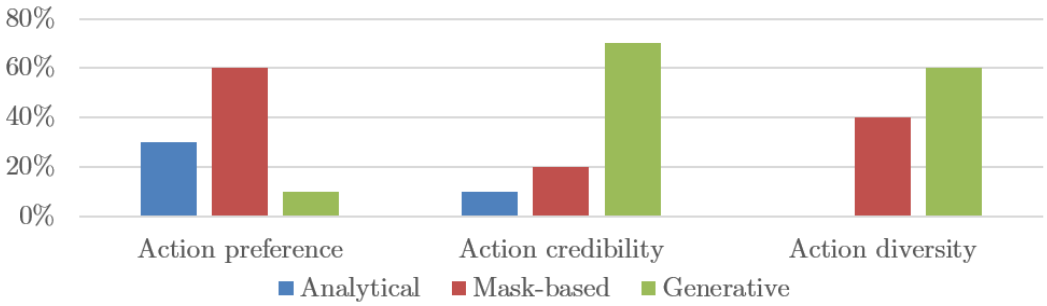


Figure 5.10: Evaluation of action generation methods

80% of the test players pointed out that especially the interaction with objects and inventory created a feeling of immersion. It also emphasized the character of a game as opposed to a simple narrative. The evaluation of the action generation methods show that the formulation of the mask-based approach was preferred (see Figure 5.10). In contrast, the credibility and diversity of the generative actions were perceived most distinctly. However, the players stated in the freeform feedback that the game drifted more into the literary realm by using actions entirely generated by a language model. Furthermore, it was mentioned several times by the test players that a story can be

specifically controlled if the initial template (the "introduction") comes from a well-known story, such as *The Lord of the Rings* or *Harry Potter*.

5.4 Conclusion

We have shown a practical approach to creating procedural, interactive stories for games based on five different language models: DistilGPT-2, GPT-2 small, medium, large, and xl. Three approaches to a creative and credible action generation were evaluated in terms of their diversity and controllability. It could be shown that an analysis of the previous texts via a classical NER, *WordNet Synsets* and the subsequent application of a mask-based approach to find a suiting verb leads to the most credible and yet controllable actions that can be offered to the player to continue the story of the game. By limiting the number of different characters and using control sets, we were able to improve the coherence of the stories. We have measured this coherence based on LDA and Topic Model Coherence. We believe that the use of language models trained via neural networks will give a boost to the procedural generation of stories for text-based games.

Chapter 6

Conclusions and Outlook

We have presented different new techniques in the domain of procedural content generation to facilitate the creation of virtual worlds. Procedural content generation has been introduced in Chapter 2 as follows:

> 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.

Specifically, we contributed to the fields of road network generation, the generation of accessible 3d buildings, and the generation of interactive stories for games. These contributions support the game design process in two ways: On one hand, they save development time by automatically generating large amounts of artifacts. On the other hand, they broaden the creative spectrum of the developers by generating unbiased content that goes beyond the imagination of a human being.
Our approach to generate arbitrary road networks, as it can be seen in Chapter 3 shows that mesh generation without manual intervention is possible and can be used in practice. The result looks pleasant and can be extended in multiplay ways, e.g., by integrating a generator for sidewalks, parcels including houses or industrial buildings, or bridges. The interface of the proposed generator can be used to generate purely random road networks, road networks from one of the various methods shown in Section 2.2.2, i.e., raster, radial, manhattan, branching, or even from real GIS data exported from *openstreetmaps.org* or *Google Maps*. The experimental generation of some streets shows that a generator does not make a quantity estimate of the generated contents if it is not explicitly programmed to do so. The runtime for 236 given roads and 205 derived intersections was about 7 minutes on an i7 processor with 3.4 GHz. Therefore, we recommend reducing the amount of input data, for example by merging nearby intersections into one or by removing smaller road segments. An alternative to generating road networks during design time is a streaming or runtime generation in which only those parts of the road network are

generated that are in the player's view. We were also able to identify the texturing of roads as a further field of research. A road, for example, not only has a texture with two lanes, but can theoretically have any number of lanes, a side strip or various markings. In order not to have to keep a texture for each type, the generation with a shader is obvious.

Chapter 4 shows our approach to generate three-dimensional, textured building models. The presented method ensures that each room is accessible via a corridor, and each floor can be reached via a unique staircase. Compared to the road network generation, it extends the logic of the mere generation of a mesh by various constraints that ensure the validity of the generated building. These constraints come from the domain of floor planning and the placement of objects in a given space. Boolean operators for meshes as well as for polygons have proven to be particularly useful, since the rules for creating a building can be formalized similar to a mathematical equation. Thus, a door section corresponds to a simple subtraction of two meshes instead of a manual calculation of a set of coordinates. The generation of UV maps is a major challenge, which we have solved by introducing the index/mesh group tuples. Those values helped to keep track of textures in a procedural generation process.

In Chapter 5 we presented an approach that generates interactive stories by using methods from classical and neural network based NLP. An initial introduction provided by the user is analysed, and entities such as objects, places, and characters are extracted and classified using *WordNet Synsets*. We then offer three approaches to generate selectable actions for the player to write a sentence that continues the story. Including the extracted items, places, and characters from the current passage, the actions strongly relate to the story. The differences between the three approchaes to generate those actions, namely credibility, variety and controllability, were pointed out. The best results were achieved by constructing actions from a subject, predicate, object construct in which the predicate is masked and predicted by a language model. We implemented a coherence measure using LDA and Topic Model Coherence so that the current coherence of the entire story could be observed in each new passage. We were able to increase the coherence of the story by using long- and short-term memories that keep characters and items in a narrative loop so that the player could create a long-term relation to objects, places, and characters.

To confirm the functionality of our approaches, each method was fully implemented and evaluated. It was shown that today's game development tools make it easier than ever to experiment with generative ideas from different domains. Modern game engines, for instance, support the embedding of generators into the workflow of human designers and allow for a quick evaluation, helping to decide if a generator is used for the development process or not.

Furthermore, Chapters 3 to 5 have shown that adding control mechanisms and handling all special cases is expensive in terms of developing the generator, as well as in testing its functionality. Taking the building generator as an example, there were test runs in which a building with ten rooms was supposed to fit into a $10m^2$ area. Implementing plausibility checks for a generator is time consuming and might limit creativity (who knows if the generator finds a way to map so many rooms to such a small area?).

And even if the generator does not deliver the desired result, it is also conceivable that it will at least serve as a booster for human creativity. A building's surfaces, for example, may not fit perfectly into the style of a game, but textures can be adapted afterwards to create a harmonious overall picture. Or a generated story is a bit absurd, but can be used as inspiration for a professional writer.

6.1 Future Work

During literature research and the implementation of the individual contributions it became clear that PCG is such a large field that it can be extended into any domain at any depth.
In the context of procedural building generation, for example, we see the automated placement of furniture, the semantic arrangement of rooms and the more precise illumination by glass elements or lamps as interesting fields in order to achieve an attractive appearance of a fully-equipped building. Although the buildings generated by our approach are diverse, additional decorations, ornamentation or details could be used in future to create an even more individual look.
The challenge here is not only to implement all systems, but also to harmonize and connect them so that they create an entire environment procedurally. An example would be a 2D Jump 'n Run with generated levels, generated character models including animations and procedural music that supports the game in a context-sensitive auditory way.
In the context of a three-domensional virtual world as it has been discussed in this thesis, the building blocks would be a terrain, a road network, a parcel detector and a building generator; enhanced with the accomodation of residents and an adequate simulation of their daily life. To ensure this for as many genres and game mechanics as possible, a formal language for generators is indispensable, not only for trivial requirements, such as adjusting the proportions of 3D models, but also to harmonize the intellectual claim, the setting or the art style. Designing such a formal language would be challenging in several ways:

- defining the interfaces to connect the individual generators,
- unambiguous classification of a generator and defining its clear task,
- guarantee an extensibility of the language and
- and ensuring the correct sequence of execution of the individual generators when creating a virtual world.

Such a formal definition of a generation process would have the positive effect that many developers can work on a procedural world generator at the same time, evaluate their ideas using already proven methods from others and extend the overall system when they could validate their new contribution. The impact of such a technology could be enormous if it practically complements the existing design flow in game development.

Any work that machines do must relieve the human, not replace it. However, the outcome of the introduction of a new technology is difficult to predict and may create the fear of humans of being replaced. Fortunately, the previous development of procedural generators did not show any negative effects. On the contrary, generators have been positively received, as the increasing use of e.g. Houdini shows. Nevertheless, many tools still require the involvement of designers to achieve the desired result.
Procedural Content Generation via Machine Learning (PCGML) is another field that is currently experiencing great enthusiasm for research and will certainly continue to do so in the future.
Learning from existing data has already arrived in the games industry, as demonstrated by the startup *modl.ai*[1], which, in addition to using machine learning to generate digital content, also uses AI methods to test games or detect the use of illegal cheat software.
The advantage of traditional rule-based generators is that they can generate a large number of assets, which in turn form the basis for training a machine learning model. During the implementation of the building generator, we generated about 30 gigabytes of 3D models that could be used for such a training. An interesting question here is now what should be generated: The floor plans, which the generator then translates into 3D models? Or the direct structure of an FBX file? Facebook recently published *PyTorch3D*[2], a framework that is able to apply deep learning to 3d meshes, e.g., for deformation or rendering tasks.
When applying any form of machine learning to procedural generation a fundamental question is whether a PCGML generator can be parameterized in the same way as a rule-based generator, how varied the output and how high the validity of the results would be. We have pointed out in Chapter 5 that applying machine learning may involve high resource costs. A further question is whether PCGML will take place in the future on a local platform, or decentralized in a cloud.
With regard to the topics discussed in this thesis, the next step is clearly to merge road network, building and story generation into one single workflow and close the gaps in between, such as terrain generation, parcel detection and character generation. We hope to be able to pursue PCG for virtual worlds in the future and to continue, together with the growing PCG community, to bring most of the PCG tools to production.

[1]https://modl.ai/
[2]https://github.com/facebookresearch/pytorch3d

References

[1] Bethesda Game Studios. The elder scrolls v: Skyrim. video game, 2011.

[2] Joel Burgess. Modular level design for skyrim. `http://blog.joelburgess.com/2013/04/skyrims-modular-level-design-gdc-2013.html`, 2013. Accessed on 2017-10-02.

[3] Maxis. The sims. video game, 2000.

[4] Maxis. Sim city. video game, 1989.

[5] Esben Bach and Andreas Madsen. *Procedural Character Generation: Implementing Reference Fitting and Principal Components Analysis*. Aalborg University. Department of Computer Science, 2007.

[6] Frontier Developments. Elite: Dangerous. video game, 2014.

[7] Mojang. Minecraft. video game, 2009.

[8] Klaus Teuber. The settlers of catan. board game, 1995.

[9] Miao Wang. Java settlers intelligente agentenbasierte spielsysteme für intuitive multi-touch-umgebungen.

[10] Ralf Dörner, Stefan Göbel, Wolfgang Effelsberg, and Josef Wiemeyer. *Serious Games: Foundations, Concepts and Practice*. Springer, 2016.

[11] Don Worth. Beneath apple manor. video game, 1978.

[12] D. Carreker. *The Game Developer's Dictionary: A Multidisciplinary Lexicon for Professionals and Students*. Course Technology, 2012. ISBN 9781435460812. URL `https://books.google.de/books?id=nwSDZwEACAAJ`.

[13] Glenn Wichman Michael Toy. Rogue. video game, 1980.

[14] Justin Olivetti. The game archaeologist: A brief history of roguelikes. `https://www.engadget.com/2014/01/18/the-game-archaeologist-a-brief-history-of-roguelikes/`, 2014. Accessed on 2017-10-02.

[15] Stephen Lee-Urban. Procedural content generation. https://www.cc.gatech.edu/~surban6/2016-cs4731/lectures/2016_06_30-ProceduralContentGeneration_intro.pdf, 2016. Accessed on 2017-10-02.

[16] Darwyn R. Peachey. Solid texturing of complex surfaces. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, pages 279–286, New York, NY, USA, 1985. ACM. ISBN 0-89791-166-0. doi: 10.1145/325334.325246. URL http://doi.acm.org/10.1145/325334.325246.

[17] F Kenton Musgrave, Craig E Kolb, and Robert S Mace. The synthesis and rendering of eroded fractal terrains. In *ACM Siggraph Computer Graphics*, volume 23, pages 41–50. ACM, 1989.

[18] Peter E Oppenheimer. Real time design and animation of fractal plants and trees. In *ACM SiGGRAPH Computer Graphics*, volume 20, pages 55–64. ACM, 1986.

[19] D.S. Ebert, F.K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. The Morgan Kaufmann Series in Computer Graphics. Elsevier Science, 2002. ISBN 9780080518756. URL https://books.google.de/books?id=fXp5UsEWNX8C.

[20] Westwood Pacific. Command and conquer: Red alert 2. video game, 2000.

[21] Blizzard North. Diablo. video game, 1996.

[22] Bethesda Softworks. The elder scrolls ii: Daggerfall. video game, 1996.

[23] Blender Foundation. Blender. https://www.blender.org/, 2017. Accessed on 2017-10-07.

[24] Autodesk. Character generator. https://charactergenerator.autodesk.com/, 2014. Accessed on 2017-10-02.

[25] Mixamo. Fuse. https://www.mixamo.com/, 2014. Accessed on 2017-10-07.

[26] MakeHuman team. Makehuman. http://www.makehuman.org/, 2017. Accessed on 2017-10-02.

[27] Farbrausch. .kkrieger. video game, 2004.

[28] digitalekultur e.V. Die demoszene - neue welten im computer. https://www.digitalekultur.org/files/dk_wasistdiedemoszene.pdf, 2003. Accessed on 2017-10-02.

[29] Farbrausch. Farbrausch werkkzeug3. https://github.com/farbrausch/fr_public, 2011. Accessed on 2017-10-27.

[30] Kate Compton, Joseph C Osborn, and Michael Mateas. Generative methods. In *The Fourth Procedural Content Generation in Games workshop, PCG*, volume 1, 2013.

[31] Dean Macri and Kim Pallister. Procedural 3d content generation. *Retrieved April*, 26:2004, 2000.

[32] Gillian Smith. An analog history of procedural content generation. In *FDG*, 2015.

[33] Pixologic. Zbrush. `http://pixologic.com/`, 1999. Accessed on 2017-10-07.

[34] Epic Games. Unreal engine 4. `https://www.unrealengine.com`, 2014. Accessed on 2017-10-07.

[35] Unity Technologies. Unity 2017. `https://unity3d.com`, 2017. Accessed on 2017-10-07.

[36] Crytec. Cry engine v. `https://www.cryengine.com/`, 2016. Accessed on 2017-10-07.

[37] Side Effects Software. Houdini. `https://www.sidefx.com/products/houdini-core/`, 2017. Accessed on 2017-10-07.

[38] Esri R&D Center Zurich. Esri city engine. `http://www.esri.com/software/cityengine`, 2014. Accessed on 2017-10-07.

[39] Ruben Michaël Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics*, 35(2):352–363, 2011.

[40] Timothy Roden and Ian Parberry. From artistry to automation: A structured methodology for procedural content creation. *Entertainment Computing–ICEC 2004*, pages 301–304, 2004.

[41] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N Yannakakis. What is procedural content generation?: Mario on the borderline. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, page 3. ACM, 2011.

[42] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, February 2013. ISSN 1551-6857. doi: 10.1145/2422956.2422957. URL `http://doi.acm.org/10.1145/2422956.2422957`.

[43] Julian Togelius, Noor Shaker, and Mark J Nelson. Procedural content generation in games: A textbook and an overview of current research/j. *Togelius, N. Shaker, M. Nelson—Berlin: Springer*, 2014.

[44] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

[45] Dieter Finkenzeller. *Modellierung komplexer Gebäudefassaden in der Computergraphik*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008.

[46] Pixar Animation Studios. Renderman. `https://renderman.pixar.com/`, 2014. Accessed on 2017-10-08.

[47] Gwyneth A. Bradbury, Il Choi, Cristina Amati, Kenny Mitchell, and Tim Weyrich. Frequency-based controls for terrain editing. In *Proceedings of the 11th European Conference on Visual Media Production*, CVMP '14, pages 15:1–15:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3185-2. doi: 10.1145/2668904.2668944. URL `http://doi.acm.org/10.1145/2668904.2668944`.

[48] Juan Soto. Statistical testing of random number generators. In *Proceedings of the 22nd National Information Systems Security Conference*, volume 10, page 12. NIST Gaithersburg, MD, 1999.

[49] Pierre L'Ecuyer. Uniform random number generators: a review. In *Proceedings of the 29th conference on Winter simulation*, pages 127–134. IEEE Computer Society, 1997.

[50] Claus Nagel Karl-Heinz Häfele Gerhard Gröger, Thomas H. Kolbe. Ogc city geography markup language (citygml) en-coding standard. `https://portal.opengeospatial.org/files/?artifact_id=47842`, 2012. Accessed on 2017-10-02.

[51] Jonas Freiknecht and Wolfgang Effelsberg. A survey on the procedural generation of virtual worlds. *Multimodal Technologies and Interaction*, 1(4):27, 2017.

[52] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, September 1956. ISSN 0096-1000. doi: 10.1109/TIT.1956.1056813.

[53] Timm Dapper. Practical procedural modeling of plants. `http://www.td-grafik.de/artic/talk20030122/overview.html`, 2003. Accessed on 2017-10-02.

[54] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 275–286, New York, NY, USA, 1998. ACM. ISBN 0-89791-999-8. doi: 10.1145/280814.280898. URL `http://doi.acm.org/10.1145/280814.280898`.

[55] P Prusinkiewicz, J Hanan, M Hammel, R Mech, PM Room, WR Remphrey, et al. Plants to ecosystems: Advances in computational life sciences. *Colingwood (Australia): CSIRO*, pages 1–134, 1997.

[56] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410. ACM, 1996.

[57] Kristin A Sorrensen-Cothern, E David Ford, and Douglas G Sprugel. A model of competition incorporating plasticity through modular foliage and crown development. *Ecological Monographs*, 63(3):277–304, 1993.

[58] Xuejin Chen, Boris Neubert, Ying-Qing Xu, Oliver Deussen, and Sing Bing Kang. Sketch-based tree modeling using markov random field. *ACM Trans. Graph.*, 27 (5):109:1–109:9, December 2008. ISSN 0730-0301. doi: 10.1145/1409060.1409062. URL `http://doi.acm.org/10.1145/1409060.1409062`.

[59] Makoto Okabe, Shigeru Owada, and Takeo Igarash. Interactive design of botanical trees using freehand sketches and example-based editing. In *Computer Graphics Forum*, volume 24, pages 487–496. Wiley Online Library, 2005.

[60] Alex Reche-Martinez, Ignacio Martin, and George Drettakis. Volumetric reconstruction and interactive rendering of trees from photographs. In *ACM transactions on graphics (ToG)*, volume 23, pages 720–727. ACM, 2004.

[61] Ilya Shlyakhter, Max Rozenoer, Julie Dorsey, and Seth Teller. Reconstructing 3d tree models from instrumented photographs. *IEEE Computer Graphics and Applications*, 21(3):53–61, 2001.

[62] Ping Tan, Gang Zeng, Jingdong Wang, Sing Bing Kang, and Long Quan. Image-based tree modeling. In *ACM Transactions on Graphics (TOG)*, volume 26, page 87. ACM, 2007.

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

[64] Greenworks Organic Software. Xfrog. `http://xfrog.com/`, 2017. Accessed on 2017-10-12.

[65] Interactive Data Visualization. Speed tree. `https://speedtree.com`, 2015. Accessed on 2017-10-02.

[66] John Walker. Fractal food - self-similarity on the supermarket shelf, 2005. URL `https://www.fourmilab.ch/images/Romanesco/`.

[67] Jinmo Kim. Modeling and optimization of a tree based on virtual reality for immersive virtual landscape generation. *Symmetry*, 8(9):93, 2016.

[68] Sai-Keung Wong and Kai-Chun Chen. A procedural approach to modelling virtual climbing plants with tendrils. In *Computer Graphics Forum*, volume 35, pages 5–18. Wiley Online Library, 2016.

[69] O. Stava, S. Pirk, J. Kratt, B. Chen, R. Mźch, O. Deussen, and B. Benes. Inverse procedural modelling of trees. *Comput. Graph. Forum*, 33(6):118–131, September 2014. ISSN 0167-7055. doi: 10.1111/cgf.12282. URL `https://doi.org/10.1111/cgf.12282`.

[70] Ruben M Smelik, Klaas Jan De Kraker, Tim Tutenel, Rafael Bidarra, and Saskia A Groenewegen. A survey of procedural methods for terrain modelling. In *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, pages 25–34, 2009.

[71] Jonas Freiknecht. Terrain tutorial using shade-c. `http://www.jofre.de/Transport/Terrain%20Tutorial%20using%20ShadeC.pdf`, 2011. Accessed on 2017-10-02.

[72] Elizabeth A Matthews and Brian A Malloy. Incorporating coherent terrain types into story-driven procedural maps.

[73] Ken Perlin. Improving noise. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 681–682. ACM, 2002.

[74] Przemyslaw Prusinkiewicz and Mark Hammel. A fractal model of mountains and rivers. In *Graphics Interface*, volume 93, pages 174–180. Canadian Information Processing Society, 1993.

[75] Farès Belhadj and Pierre Audibert. 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*, GRAPHITE '05, pages 447–450, New York, NY, USA, 2005. ACM. ISBN 1-59593-201-1. doi: 10.1145/1101389.1101479. URL `http://doi.acm.org/10.1145/1101389.1101479`.

[76] Gavin SP Miller. The definition and rendering of terrain maps. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 39–48. ACM, 1986.

[77] Jacob Olsen. Realtime procedural terrain generation. 2004.

[78] Martin Kahoun. *Realtime library for procedural generation and rendering of terrains*. PhD thesis, Charles University, 2013.

[79] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. Roaming terrain: Real-time optimally adapting meshes. In *Proc. Visualization '97*, pages 81–88, Oct 1997. doi: 10.1109/VISUAL.1997.663860.

[80] Jiwon Lee, Kisung Jeong, and Jinmo Kim. Mave: Maze-based immersive virtual environment for new presence and experience. *Computer Animation and Virtual Worlds*, 28(3-4), 2017.

[81] Bedrich Benes and Rafael Forsbach. Layered data representation for visual simulation of terrain erosion. In *Proceedings of the 17th Spring Conference on Computer Graphics*, SCCG '01, pages 80–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1215-1. URL http://dl.acm.org/citation.cfm?id=882484.883939.

[82] Aitor Santamaría-Ibirika, Xabier Cantero, Mikel Salazar, Jaime Devesa, Igor Santos, Sergio Huerta, and Pablo G. Bringas. Procedural approach to volumetric terrain generation. *The Visual Computer*, 30(9):997–1007, Sep 2014. ISSN 1432-2315. doi: 10.1007/s00371-013-0909-y. URL https://doi.org/10.1007/s00371-013-0909-y.

[83] Juncheng Cui, Yang-Wai Chow, and Minjie Zhang. A voxel-based octree construction approach for procedural cave generation. 2011.

[84] Matt Boggus and Roger Crawfis. Procedural creation of 3d solution cave models. In *Proceedings of the 20th IASTED International Conference on Modelling and Simulation*, pages 180–186, 2009.

[85] Johan Hammes. *Modeling of Ecosystems as a Data Source for Real-Time Terrain Rendering*, pages 98–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-44818-1. doi: 10.1007/3-540-44818-7_14. URL https://doi.org/10.1007/3-540-44818-7_14.

[86] Monssef Alsweis and Oliver Deussen. Wang-tiles for the simulation and visualization of plant competition. In *Advances in Computer Graphics*, pages 1–11. Springer, 2006.

[87] Uta Berger, Hanno Hildenbrandt, and Volker Grimm. Towards a standard for the individual-based modeling of plant populations: self-thinning and the field-of-neighborhood approach. *Natural resource modeling*, 15(1):39–54, 2002.

[88] Hao Wang. Proving theorems by pattern recognition i. *Communications of the ACM*, 3(4):220–234, 1960.

[89] Oliver Deussen and Bernd Lintermann. *Digital Design of Nature: Computer Generated Plants and Organics*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 3642073638, 9783642073632.

[90] Remco Huijser, Jeroen Dobbe, Willem F. Bronsvoort, and Rafael Bidarra. Procedural natural systems for game level design. In *Proceedings of the 2010 Brazilian Symposium on Games and Digital Entertainment*, SBGAMES '10, pages 189–198, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4359-8. doi: 10.1109/SBGAMES.2010.31. URL http://dx.doi.org/10.1109/SBGAMES.2010.31.

[91] E. Derzapf, B. Ganster, M. Guthe, and R. Klein. River networks for instant procedural planets. *Computer Graphics Forum*, 30(7):2031–2040, 2011. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2011.02052.x. URL http://dx.doi.org/10.1111/j.1467-8659.2011.02052.x.

[92] Jonathon Doran and Ian Parberry. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):111–119, 2010.

[93] Loopix. Mystymood, 2016. URL http://www.loopix-project.com/. accessed on 27 October 2017.

[94] J Schoen. Robust, Guaranteed-Quality Anisotropic Mesh Generation. Master's thesis, University of California at Berkeley, 2008.

[95] J Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. URL http://www.cs.cmu.edu/~jrs/jrsresearch.html.

[96] S Havemann. *Generative Mesh Modeling*. PhD thesis, Institute of Computer Graphics, Braunschweig Technical University, 2005.

[97] P K Ilangovan. Procedural city generator. Master's thesis, Bournemouth University, 2009. URL https://nccastaff.bournemouth.ac.uk/jmacey/MastersProjects/MSc09/Ilangovan/Thesis_i7834000.pdf.

[98] Magomet Kochkarov Felix Queißner and Jonas Freiknecht. Tust scripting library. https://github.com/MasterQ32/TUST, 2013. Accessed on 2017-10-03.

[99] M Banf, M Barth, H Schulze, J Koch, A Pritzkau, M Schmidt, A Daraban, S Meister, R Sandhöfer, V Sotke, et al. On-demand creation of procedural cities. *Game and Entertainment Technologies. Freiburg: IADIS Press*, 2010.

[100] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM, 2001.

[101] Sarah Ostadabbas, Masoud Farshbaf, Mehrdad Nourani, and Sateesh Addepalli. Layered semi-markov road generation for driving simulations. In *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 365–370. IEEE, 2011.

[102] George Kelly and Hugh McCabe. Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, pages 8–16, 2007.

[103] Nadeo. Trackmania original. video game, 2003.

[104] Namco. Ridge racer 64. video game, 1999.

[105] Acclaim Entertainment. Re-volt. video game, 1999.

[106] Rémi Cura, Julien Perret, and Nicolas Paparoditis. Streetgen: In-base procedural-based road generation. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2:409, 2015.

[107] E. Galin, A. Peytavie, N. Maréchal, and E. Guérin. Procedural generation of roads. *Computer Graphics Forum*, 29(2):429–438, 2010. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2009.01612.x. URL `http://dx.doi.org/10.1111/j.1467-8659.2009.01612.x`.

[108] John Taplin. Simulation models of traffic flow. In *de The 34th Annual Conference of the Operational Research Society of New Zealand, New Zealand*, 1999.

[109] Hans-Thomas Fritzsche. A model for traffic simulation. *Traffic Engineering+ Control*, 35(5):317–21, 1994.

[110] Jason Sewall, David Wilkie, Paul Merrell, and Ming C Lin. Continuum traffic simulation. In *Computer Graphics Forum*, volume 29, pages 439–448. Wiley Online Library, 2010.

[111] Ruben M Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. A survey on procedural modelling for virtual worlds. In *Computer Graphics Forum*, volume 33, pages 31–50. Wiley Online Library, 2014.

[112] Claus Brenner. Towards fully automatic generation of city models. In *In: IAPRS*, pages 85–92, 2000.

[113] Marie Saldana and Christopher Johanson. Procedural modeling for rapid-prototyping of multiple building phases. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 5:W1, 2013.

[114] P. J. Birch, S. P. Browne, V. J. Jennings, A. M. Day, and D. B. Arnold. Rapid procedural-modelling of architectural structures. In *Proceedings of the 2001 Conference on Virtual Reality, Archeology, and Cultural Heritage*, VAST '01, pages 187–196, New York, NY, USA, 2001. ACM. ISBN 1-58113-447-9. doi: 10.1145/584993.585023. URL `http://doi.acm.org/10.1145/584993.585023`.

[115] Jess Martin. Algorithmic beauty of buildings methods for procedural building generation. *Computer Science Honors Theses*, page 4, 2005.

[116] Jasper Stocker. Buildr2. `http://support.jasperstocker.com/buildr2/`, 2003. Accessed on 2017-10-03.

[117] Tyson Ibele. Building generator. `http://tysonibele.com/Main/BuildingGenerator/buildingGen.htm`, 2009. Accessed on 2017-10-03.

[118] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *Acm Transactions On Graphics (Tog)*, volume 25, pages 614–623. ACM, 2006.

[119] Michael Schwarz and Pascal Müller. Advanced procedural modeling of architecture. *ACM Transactions on Graphics (TOG)*, 34(4):107, 2015.

[120] Tom Kelly and Peter Wonka. Interactive architectural modeling with procedural extrusions. *ACM Trans. Graph.*, 30(2):14:1–14:15, April 2011. ISSN 0730-0301. doi: 10.1145/1944846.1944854. URL `http://doi.acm.org/10.1145/1944846.1944854`.

[121] Jean-Eudes Marvie, Julien Perret, and Kadi Bouatouch. The fl-system: a functional l-system for procedural geometric modeling. *The Visual Computer*, 21(5): 329–339, 2005.

[122] Zifeng Guo and Biao Li. Evolutionary approach for spatial architecture layout design enhanced by an agent-based topology finding system. *Frontiers of Architectural Research*, 6(1):53 – 62, 2017. ISSN 2095-2635. doi: https://doi.org/10.1016/j.foar.2016.11.003. URL `http://www.sciencedirect.com/science/article/pii/S2095263516300565`.

[123] Benjamin Dillenburger, Markus Braach, and Ludger Hovestadt. Building design as an individual compromise between qualities and costs. In *Proceedings of the 13Th International CAADFutures Conference, CAADFutures*, pages 458–471. Les Presses de l'Université de Montréal, 2009.

[124] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. In *ACM Transactions on Graphics (TOG)*, volume 29, page 181. ACM, 2010.

[125] Per Galle. An algorithm for exhaustive generation of building floor plans. *Communications of the ACM*, 24(12):813–825, 1981.

[126] MA Rosenman. The generation of form using an evolutionary approach. In *Evolutionary Algorithms in Engineering Applications*, pages 69–85. Springer, 1997.

[127] Ricardo Lopes, Tim Tutenel, Ruben M Smelik, Klaas Jan De Kraker, and Rafael Bidarra. A constrained growth method for procedural floor plan generation. In *Proc. 11th Int. Conf. Intell. Games Simul*, pages 13–20, 2010.

[128] Chi-Han Peng, Yong-Liang Yang, and Peter Wonka. Computing layouts with deformable templates. *ACM Transactions on Graphics (TOG)*, 33(4):99, 2014.

[129] Mark Bruls, Kees Huizing, and Jarke J Van Wijk. Squarified treemaps. In *Data visualization 2000*, pages 33–42. Springer, 2000.

[130] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2nd conference on Visualization'91*, pages 284–291. IEEE Computer Society Press, 1991.

[131] Maysam Mirahmadi and Abdallah Shami. A novel algorithm for real-time procedural generation of building floor plans. *arXiv preprint arXiv:1211.5842*, 2012.

[132] Daniel Bengtsson and Johan Melin. Constrained procedural floor plan generation for game environments. Master's thesis, Department of Creative Technologies, 2016.

[133] Eric Huang and Richard E Korf. New improvements in optimal rectangle packing. In *IJCAI*, volume 9, pages 511–516, 2009.

[134] Richard E Korf. Optimal rectangle packing: Initial results. In *ICAPS*, pages 287–295, 2003.

[135] Tung-Chieh Chen and Yao-Wen Chang. Packing floorplan representations. In *Handbook of Algorithms for Physical Design Automation*, 2008.

[136] Reinhard Koenig and Katja Knecht. Comparing two evolutionary algorithm based methods for layout generation: Dense packing versus subdivision. *AI EDAM*, 28 (3):285–299, 2014.

[137] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. Interactive furniture layout using interior design guidelines. In *ACM Transactions on Graphics (TOG)*, volume 30, page 87. ACM, 2011.

[138] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. *Instant architecture*, volume 22. ACM, 2003.

[139] Jerry O Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. Metropolis procedural modeling. *ACM Transactions on Graphics (TOG)*, 30(2):11, 2011.

[140] Gustavo Patow. User-friendly graph editing for procedural modeling of buildings. *IEEE Computer Graphics and Applications*, 32(2):66–75, 2012.

[141] Johannes Edelsbrunner, Sven Havemann, Alexei Sourin, and Dieter W Fellner. Procedural modeling of architecture with round geometry. *Computers & graphics*, 64:14–25, 2017.

[142] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018.

[143] Andelo Martinovic and Luc Van Gool. Bayesian grammar learning for inverse procedural modeling. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 201–208, 2013.

[144] Christopher Hesse. Image-to-image translation in tensorflow, 2017. URL `https://affinelayer.com/pix2pix/`.

[145] Stanislas Chaillou. Ai architecture towards a new approach, 2019. URL `https://www.academia.edu/39599650/AI_Architecture_Towards_a_New_Approach`.

[146] J Beneš, Tom Kelly, F Děchtěrenko, J Křivánek, and Pascal Müller. On realism of architectural procedural models. In *Computer Graphics Forum*, volume 36, pages 225–234. Wiley Online Library, 2017.

[147] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural Content Generation in Games*. Springer, 2016.

[148] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 4. ACM, 2010.

[149] Jim Whitehead. Toward proccedural decorative ornamentation in games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 9. ACM, 2010.

[150] N.V. Barreto and Licinio Roque. A survey of procedural content generation tools in video game creature design. In *Second Conference on Computation Communication Aesthetics and X*, n/a, 2014.

[151] UMA Steering Group. Uma - unity multipurpose avatar. `https://www.assetstore.unity3d.com/en/#!/content/13930`, 2017. Accessed on 2017-10-03.

[152] Maxis. Spore. video game, 2008.

[153] Chris Hecker, Bernd Raabe, Ryan W. Enslow, John DeWeese, Jordan Maynard, and Kees van Prooijen. Real-time motion retargeting to highly varied user-created morphologies. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 27:1–27:11, New York, NY, USA, 2008. ACM. ISBN 978-1-4503-0112-1. doi: 10.1145/1399504.1360626. URL `http://doi.acm.org/10.1145/1399504.1360626`.

[154] Jon Hudson. *Creature Generation using Genetic Algorithms and Auto-Rigging*. PhD thesis, National Center of Computer Animation - Bournemouth University, 2013.

[155] Eike F Anderson. Playing smart-artificial intelligence in computer games. 2003.

[156] Sean Curtis, Andrew Best, and Dinesh Manocha. Menge: A modular framework for simulating crowd movement. *Collective Dynamics*, 1:1–40, 2016.

[157] Rockstar Games. Grand theft auto iii. video game, 2001.

[158] Oliver Szymanezyk, Patrick Dickinson, and Tom Duckett. From individual characters to large crowds: Augmenting the believability of open-world games through exploring social emotion in pedestrian groups. In *Proceedings of DiGRA 2011 Conference: Think Design Play*, 09 2011.

[159] Sangyeob Lee. *The effect of RPG newness, rating, and character evilness on the NPC believability*. Michigan State University, 2009.

[160] I. D. Horswill. Lightweight procedural animation with believable physical interactions. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1): 39–49, March 2009. ISSN 1943-068X. doi: 10.1109/TCIAIG.2009.2019631.

[161] Ahmad Abdul Karim, Thibaut Gaudin, Alexandre Meyer, Axel Buendia, and Saida Bouakaz. Procedural locomotion of multilegged characters in dynamic environments. *Computer Animation and Virtual Worlds*, 24(1):3–15, 2013.

[162] David Traum. Computational approaches to dialogue. *The Routledge Handbook of Language and Dialogue*, page 143, 2017.

[163] Christina R Strong and Michael Mateas. Talking with npcs: Towards dynamic generation of discourse structures. In *AIIDE*, 2008.

[164] Brian MacNamee and Padraig Cunningham. Creating socially interactive no-player characters: The μ-siv system. *Int. J. Intell. Games & Simulation*, 2(1):28–35, 2003.

[165] Cyril Brom, Tomáš Poch, and O Sery. Ai level of detail for really large worlds. *Game Programming Gems*, 8:213–231, 2010.

[166] Justine Cassell, Hannes Högni Vilhjálmsson, and Timothy Bickmore. Beat: the behavior expression animation toolkit. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 477–486. ACM, 2001.

[167] Jonathan Gratch, Jeff Rickel, Elisabeth André, Justine Cassell, Eric Petajan, and Norman Badler. Creating interactive virtual humans: Some assembly required. *IEEE Intelligent systems*, 17(4):54–63, 2002.

[168] Berardina De Carolis, Catherine Pelachaud, Isabella Poggi, and Mark Steedman. Apml, a markup language for believable behavior generation. In *Life-like characters*, pages 65–85. Springer, 2004.

[169] Timothy W Bickmore and Rosalind W Picard. Establishing and maintaining long-term human-computer relationships. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 12(2):293–327, 2005.

[170] Shigeru Miyamoto. The legend of zelda, 1986.

[171] Florian Mehm. Authoring of adaptive single-player educational games. *PIK-Praxis der Informationsverarbeitung und Kommunikation*, 37(2):157–160, 2014.

[172] Calvin Ashmore and Michael Nitsche. The quest in a generated world. In *DiGRA Conference*, 2007.

[173] Ken Hartsook, Alexander Zook, Sauvik Das, and Mark O Riedl. Toward supporting stories with procedurally generated game worlds. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 297–304. IEEE, 2011.

[174] Anne Sullivan, Michael Mateas, and Noah Wardrip-Fruin. Making quests playable: Choices, crpgs, and the grail framework. *Leonardo Electronic Almanac*, 17(2), 2012.

[175] Joris Dormans and Sander Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):216–228, 2011.

[176] Lili Yao, Nanyun Peng, Ralph Weischedel, Kevin Knight, Dongyan Zhao, and Rui Yan. Plan-and-write: Towards better automatic storytelling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7378–7385, 2019.

[177] Michael Mateas and Andrew Stern. Façade: An experiment in building a fully-realized interactive drama. In *Game developers conference*, volume 2, pages 4–8, 2003.

[178] Kate Compton, Ben Kybartas, and Michael Mateas. Tracery: An author-focused generative text tool. In Henrik Schoenau-Fog, Luis Emilio Bruni, Sandy Louchart, and Sarune Baceviciute, editors, *Interactive Storytelling*, pages 154–161, Cham, 2015. Springer International Publishing. ISBN 978-3-319-27036-4.

[179] Simon Colton and Geraint A Wiggins. Computational creativity: The final frontier? In *Proceedings of the 20th European conference on artificial intelligence*, pages 21–26. IOS Press, 2012.

[180] Edwin PD Pednault. Formulating multiagent, dynamic-world problems in the classical planning framework. *Reasoning about actions and plans*, pages 47–82, 1986.

[181] Michel Beaudouin-Lafon, Wendy Mackay, Peter Andersen, Paul Janecek, Mads Jensen, Michael Lassen, Kasper Lund, Kjeld Mortensen, Stephanie Munck, Anne Ratzer, et al. Cpn/tools: A post-wimp interface for editing and simulating coloured petri nets. *Applications and theory of Petri nets 2001*, pages 71–80, 2001.

[182] Christian Reuter. *Authoring Collaborative Multiplayer Games-Game Design Patterns, Structural Verification, Collaborative Balancing and Rapid Prototyping*. PhD thesis, Technische Universität Darmstadt, 2016.

[183] Young-Seol Lee and Sung-Bae Cho. Context-aware petri net for dynamic procedural content generation in role-playing game. *IEEE Computational Intelligence Magazine*, 6(2):16–25, 2011.

[184] BioWare. Neverwinter nights. video game, 2002.

[185] Lara J Martin, Prithviraj Ammanabrolu, Xinyu Wang, William Hancock, Shruti Singh, Brent Harrison, and Mark O Riedl. Event representations for automated story generation with deep neural nets. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[186] Angela Fan, Mike Lewis, and Yann Dauphin. Strategies for structuring story generation. *arXiv preprint arXiv:1902.01109*, 2019.

[187] Melissa Roemmele. *Neural Networks for Narrative Continuation*. PhD thesis, University of Southern California, 2018.

[188] Margaret Cychosz, Andrew S Gordon, Obiageli Odimegwu, Olivia Connolly, Jenna Bellassai, and Melissa Roemmele. Effective scenario designs for free-text interactive fiction. In *International Conference on Interactive Digital Storytelling*, pages 12–23. Springer, 2017.

[189] Prithviraj Ammanabrolu, William Broniec, Alex Mueller, Jeremy Paul, and Mark O Riedl. Toward automated quest generation in text-adventure games. *arXiv preprint arXiv:1909.06283*, 2019.

[190] Jack Urbanek, Angela Fan, Siddharth Karamcheti, Saachi Jain, Samuel Humeau, Emily Dinan, Tim Rocktäschel, Douwe Kiela, Arthur Szlam, and Jason Weston. Learning to speak and act in a fantasy text adventure game. *arXiv preprint arXiv:1903.03094*, 2019.

[191] Angela Fan, Jack Urbanek, Pratik Ringshia, Emily Dinan, Emma Qian, Siddharth Karamcheti, Shrimai Prabhumoye, Douwe Kiela, Tim Rocktäschel, Arthur Szlam, et al. Generating interactive worlds with text. In *AAAI*, pages 1693–1700, 2020.

[192] Matheus RF Mendonça and Artur Ziviani. Network-based procedural story generation. *Computers in Entertainment (CIE)*, 16(3):1–18, 2018.

[193] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1 (8):9, 2019.

[194] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771, 2019.

[195] Nathan Whitmore. Gpt adventure, 2019. URL https://quicktotheratcave.tumblr.com/post/187432425523/shall-we-play-a-game-a-gpt-2-text-adventure.

[196] Darius Kazemi. Keeping procedural generation simple. *Procedural Storytelling in Game Design*, pages 17–22, 2019.

[197] Pradyumna Tambwekar, Murtaza Dhuliawala, Lara J Martin, Animesh Mehta, Brent Harrison, and Mark O Riedl. Controllable neural story plot generation via reinforcement learning. *arXiv preprint arXiv:1809.10736*, 2018.

[198] J. Freiknecht. *Big Data in der Praxis : Lösungen mit Hadoop, HBase und Hive ; Daten speichern, aufbereiten, visualisieren*. Hanser eLibrary. Hanser, 2014. ISBN 9783446439597. URL https://books.google.de/books?id=AtjtoQEACAAJ.

[199] Jiwen Zhang. C-bézier curves and surfaces. *Graphical Models and Image Processing*, 61(1):2–15, 1999.

[200] Bender Michael and Brill Manfred. Computergrafik. ein anwendungsorientiertes lehrbuch, 2006.

[201] Steven M Drucker, Tinsley A Galyean, and David Zeltzer. Cinema: A system for procedural camera movements. In *Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 67–70, 1992.

[202] Erik HW Meijering. Spline interpolation in medical imaging: comparison with other convolution-based approaches. In *2000 10th European Signal Processing Conference*, pages 1–8. IEEE, 2000.

[203] Jonathan Richard Shewchuk. Lecture notes on delaunay mesh generation. 1999.

[204] Matthias Nießner, Benjamin Keinert, Matthew Fisher, Marc Stamminger, Charles Loop, and Henry Schäfer. Real-time rendering techniques with hardware tessellation. In *Computer Graphics Forum*, volume 35, pages 113–137. Wiley Online Library, 2016.

[205] Branko Grünbaum and Geoffrey Colin Shephard. *Tilings and patterns*. Courier Dover Publications, 1987.

[206] Boris Delaunay et al. Sur la sphere vide. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7(793-800):1–2, 1934.

[207] Der-Tsai Lee and Bruce J Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3): 219–242, 1980.

[208] Rachel Weber. On reflections: First interview with the ubisoft studio's new md, 2013. URL https://www.gamesindustry.biz/articles/2014-02-26-on-reflections-first-interview-with-the-ubisoft-studios-new-md.

[209] Mark Androvich. Gta iv: Most expensive game ever developed?, 2008. URL https://www.gamesindustry.biz/articles/gta-iv-most-expensive-game-ever-developed.

[210] NewTheft. Open all interiors, 2015. URL https://www.gta5-mods.com/scripts/open-all-interiors.

[211] Beyond Skyrim. Beyond skyrim, 2016. URL https://beyondskyrim.org/.

[212] The Silver. Living city, 2018. URL https://www.moddb.com/mods/living-city.

[213] Danny Saya. 10 best real life cities in video games, 20014. URL https://arcadesushi.com/best-real-life-cities-in-video-games/.

[214] J. Freiknecht and W. Effelsberg. Procedural generation of multistory buildings with interior. *IEEE Transactions on Games*, 12(3):323–336, 2020.

[215] Florian Le Bourdais. The farthest neighbors algorithm, 2015. URL https://flothesof.github.io/farthest-neighbors.html.

[216] Fan Bao, Michael Schwarz, and Peter Wonka. Procedural facade variations from a single layout. *ACM Transactions on Graphics (TOG)*, 32(1):8, 2013.

[217] R. G. Laycock and A. M. Day. Automatically generating roof models from building footprints. In *The 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG'03)*, pages 81–84, February 2003. URL https://ueaeprints.uea.ac.uk/23202/.

[218] A.A.G. Requicha, H.B. Voelker, University of Rochester. Production Automation Project, National Science Foundation (U.S.). Research Applied to National Needs Program, and United States. National Technical Information Service. *Constructive Solid Geometry, TM-25*. Technical memorandum. Production Automation Project, University of Rochester, 1977. URL https://books.google.de/books?id=tniuHAAACAAJ.

[219] Barbara Cutler, Julie Dorsey, Leonard McMillan, Matthias Müller, and Robert Jagnow. A procedural approach to authoring solid models. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 302–311. ACM, 2002.

[220] Björn Ganster and Reinhard Klein. An integrated framework for procedural modeling. In *Proceedings of the 23rd Spring Conference on Computer Graphics*, pages 123–130. ACM, 2007.

[221] Stephen F May, Wayne Carlson, Flip Phillips, and Ferdi Scheepers. Al: A language for procedural modeling and animation. Technical report, Tech. Rep. OSU-ACCAD-12/96-TR5, ACCAD, The Ohio State University, 1996.

[222] Luc Leblanc, Jocelyn Houle, and Pierre Poulin. Component-based modeling of complete buildings. In *Proceedings of Graphics Interface 2011*, pages 87–94. Canadian Human-Computer Communications Society, 2011.

[223] Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gärtner. A novel type of skeleton for polygons. In *J. UCS The Journal of Universal Computer Science*, pages 752–761. Springer, 1996.

[224] Petr Felkel and Stepan Obdrzalek. Straight skeleton implementation. In *Proceedings of Spring Conference on Computer Graphics*, pages 210–218, 1998.

[225] Gill Barequet and Evgeny Yakersberg. Morphing between shapes by using their straight skeletons. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*, SCG '03, pages 378–379, New York, NY, USA, 2003. ACM. ISBN 1-58113-663-3. doi: 10.1145/777792.777852. URL `http://doi.acm.org/10.1145/777792.777852`.

[226] Stefan Huber. *Computing straight skeletons and motorcycle graphs: theory and practice*. Shaker, 2011.

[227] J. Freiknecht. *Spiele entwickeln mit Gamestudio: Virtuelle 3D-Welten mit Gamestudio A8 und Lite-C*. Carl Hanser Verlag GmbH & Company KG, 2012. ISBN 9783446432673. URL `https://books.google.de/books?id=16dPAgAAQBAJ`.

[228] Filip Biljecki, Hugo Ledoux, Jantien Stoter, and Junqiao Zhao. Formalisation of the level of detail in 3d city modelling. *Computers, Environment and Urban Systems*, 48:1–15, 2014.

[229] Filip Biljecki, Hugo Ledoux, and Jantien Stoter. An improved lod specification for 3d building models. *Computers, Environment and Urban Systems*, 59:25–37, 2016.

[230] Krishnendra Shekhawat. Why golden rectangle is used so often by architects: A mathematical approach. *Alexandria Engineering Journal*, 54(2):213–222, 2015.

[231] Ernest Adams. *Fundamentals of game design*. Pearson Education, 2014.

[232] Tanya Short and Tarn Adams. *Procedural generation in game design*. CRC Press, 2017.

[233] Yun-Gyung Cheong, Mark O Riedl, Byung-Chull Bae, and Mark J Nelson. Planning with applications to quests and story. In *Procedural Content Generation in Games*, pages 123–141. Springer, 2016.

[234] Nick Walton. Ai dungeon, 2019. URL `https://aidungeon.io/`.

[235] LucasArts Entertainment. Day of the tentacle manual, 1993. URL `https://www.mocagh.org/lucasfilm/maniac2wl-manual.pdf`.

[236] Jonas Freiknecht and Wolfgang Effelsberg. Procedural generation of interactive stories using language models. In *International Conference on the Foundations of Digital Games*, FDG '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388078. doi: 10.1145/3402942.3409599. URL https://doi.org/10.1145/3402942.3409599.

[237] Christiane Fellbaum. A semantic network of english: the mother of all wordnets. In *EuroWordNet: A multilingual database with lexical semantic networks*, pages 137–148. Springer, 1998.

[238] Thomas Wolf. State-of-the-art neural coreference resolution for chatbots, 2017. URL https://medium.com/huggingface/state-of-the-art-neural-coreference-resolution-for-chatbots-3302365dcf30.

[239] David D Lewis, Yiming Yang, Tony G Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *Journal of machine learning research*, 5(Apr):361–397, 2004.

[240] Shikha Bordia and Samuel R Bowman. Identifying and reducing gender bias in word-level language models. *arXiv preprint arXiv:1904.03035*, 2019.

[241] Nasrin Mostafazadeh, Nathanael Chambers, Xiaodong He, Devi Parikh, Dhruv Batra, Lucy Vanderwende, Pushmeet Kohli, and James Allen. A corpus and evaluation framework for deeper understanding of commonsense stories. *arXiv preprint arXiv:1604.01696*, 2016.

[242] Jörg Martin. 40.000 names, 2017. URL https://www.heise.de/ct/ftp/07/17/182/.

[243] Michael Röder, Andreas Both, and Alexander Hinneburg. Exploring the space of topic coherence measures. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, page 399–408, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333177. doi: 10.1145/2684822.2685324. URL https://doi.org/10.1145/2684822.2685324.

[244] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3(null):993–1022, March 2003. ISSN 1532-4435.

[245] Yves Peirsman and Johan Leys. Nlp town, 2020. URL https://www.nlp.town.

[246] George A Miller. *WordNet: An electronic lexical database*. MIT press, 1998.