Modular PCG

An Architecture for Procedural Content Generation

Master's Thesis

By

Mikael Peter Olsen

Supervised by

Paolo Burelli

Aalborg University Copenhagen

3rd of February 2014 – 28th of May 2014

AAU PAGE

AAU PAGE BACK

PREFACE

Dear Reader

This report has been written in the early months of 2014, and many nights has been spend reading and writing. I am therefore truly grateful that you have taken the time to read my work.

I would like to thank my girlfriend for her patience during the stressful times, my supervisor Paolo Burelli for constructive criticism, my fellow students at AAU-CPH for support and finally Julian Togelius for initial inspiration and for providing initial material to get started.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

This chapter will explain the initial motivation behind this project, and by reading it, it should be clear to the reader why this project has be written and what the main purpose behind the project is. First, a general motivation will describe the research area, thereafter the initial problem area will be established based on the motivation and lastly section 1.3 will give a brief overview of the entire report.

1.1 MOTIVATION

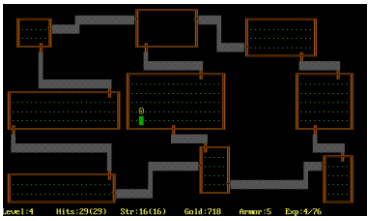
Computer games are a major part of our lives; many play computer games regularly, and during the last decade, the use of computer games has increased [1] [2]. Because of this huge industry, games are becoming more and more advanced in multiple fields – from general sound and graphics to the realism of environments and the believability of characters. In addition, the quantity of game content has increased and this increase in quantity and quality puts a challenge on the gaming industry to match the demand from the gaming community [3]. We, as players, expect the computer games to present us with new and engaging content, and while this demand

is increasing the manual content production is already expensive and unscalable [1]. This is a challenge that potentially could be aided by Procedural Content Generation (PCG), which, in short is the *"application of computers to generate game content, distinguish interesting instances among the ones generated, and select entertaining instances on behalf of the players"* [1]. PCG also refers to *"the creation of game content automatically, through algorithmic means"* [4]. These definitions will have to be investigated further and PCG will have to be specified for the purpose of this report. For now, however, the general definitions above serves as an initial understanding of PCG.

PCG offers an alternative to costly manual content creation, and can be integrated in the development process and help generate complex game worlds in a limited amount of time [1] [2] [3] [5]. This can help keep the expenses of game development down and allow designers and programmers some additional freedom, which might be the primary argument for using PCG. Another reason, which was more dominant in the past, is that PCG can keep the memory consumption of a computer game down by applying it as a method for decompressing data [5]. This method was used in the space trading game Elite to store hundreds of planets in a few tens of kilobytes. Likewise, PCG was used to generate dungeons at runtime for the game Rogue [5]. Rogue offered endless replayability and the game has formed its own sub-genre, referred to as Rogue-like, where among others the Diablo game series belongs.

PCG is an interesting field because it can not only support game creation, but also provide new techniques, facilitate new games and new ways of creating games [5] [2]. An example of this is the LUDI system by Browne & Maire [6], which was designed to invent board games autonomously. The system had to ensure that the game produced was not only playable but also that it met the requirements of being fun and engaging. The LUDI system invented a game it named Yavalath, which, in October 2011, was ranked in the top 100 abstract board games ever invented on the BoardGameGeek database [7]. This is one part of PCG, in which the algorithms can surprise the creator and create something unique, which can be very fascinating. On the other hand, PCG can be designed to support the human designer or programmer, and the collaboration between human and algorithms can prove fruitful in many cases. As mentioned before this could potentially help meet the demands for manual content production in computer games, but could also aid human creativity and

Rogue



A dungeon crawling game developed by Michael Toy and Glenn Wichman around 1980. All content is represented by letters and symbols. The layout and the placement of objects are randomly generated.

Elite

A space trading game, published in 1984 by Acornsoft. One of the first home computer games to use wire-frame 3D graphics.

Diablo

Series of action role-playing hack and slash games developed by Blizzard, released in 1996, 2000 and 2012.

Left4Dead

A cooperative first-person shooter arcade-style game set during the aftermath of an apocalyptic pandemic. Released by Valve Corporation in 2008.

enable the creation of adaptive games, i.e. games which gameplay and/or design adapted at runtime to its players [5]. An example of the latter is the game Left4Dead where enemy encounters are created *"based on the computer-analyzed stress level of the players"* [1].

PCG is a relatively young research field and previously the literature was divided across multiple disciplines (computer graphics, image processing, artificial intelligence, computer-human interfaces, psychology, linguistics, social sciences, ludology, etc.) [1]. In 2009 however, the first workshop devoted solely to PCG was held[1] and [8] state that the first paper regarding what they call search-based PCG, a special branch of PCG (see section 2.1.1), was published in 2006 [9].

Through the years, PCG has been used to create a variety of content, ranging from complete cities [10] to terrains [11] to detailed vegetation [12] to textures and materials [13]. Apart from that, PCG has been used to generate levels for 2D platform games [14] [15], creating personalized content [16] and generating levels [17] [18] for Super Mario Bros, generate infinite 2D cave-maps [19], evolving units [20] and generating maps for strategy games [21] [22], and levels for 3D games [23].

1.2 Initial Problem Area

As described, the gaming industry is challenged by the high demands for content, and by the cost of manual content creation. PCG, which is the automatic generation of content by the use of algorithms, can help overcome this challenge, and although it is a young research field, it offers great potential for further research in many different areas.

This project will therefore investigate the advantages of PCG in relation to game development and determine the how PCG can facilitate game creation. The purpose of this investigation is to advance the state of the art of PCG, and through findings contribute to the general research field.

[1] The PCG workshop is co-located with the Foundation of Digital Games Conference. The autumn 2011 issue of IEEE Transactions on Computational Intelligence and AI in Games was entirely devoted to PCG.

1.3 Report Overview

The purpose of this chapter is to give a structural overview of the project and report, describing the overall flow, allowing other researcher to understand and follow the different steps.

Firstly, an initial investigation will analyse a few definitions of PCG and establish how it should be understood in context of this project. Thereafter some existing research topics suggested by dominant researchers within the PCG community will briefly be investigated. The purpose of this investigation is to direct the research, and it is suspected that by directing PCG research in the direction of topics suggested by other dominant researchers, the outcome of this project will help advance the state of the art of PCG to the greatest extent.

By combining the initial focus on game development with one or more of the suggested research topics, a more concise research problem will be established and a concrete problem statement formulated. The report will thereafter investigate how complete game generation can be made accessible to human designers and how it can be integrated within the development pipeline.

To investigate complete game generation, the analysis will describe a few games and research projects that utilises complex procedural techniques, and investigate how these facilitate designer interaction. Because the examples provide very limited interaction, an alternative way of considering PCG in relation to game development will be proposed. This alternative is called Modular PCG and it describes a new way of designing PCG algorithms. Modular PCG facilitates the creation of individual PCG modules that applies procedural techniques to generate game content. The modules integrates directly into the virtual environment, which means that designers can apply different modules without considering existing content and other modules. For easy and rapid development, the necessary tools for authoring content are included in the modules themselves and work out of the box.

To explain and validate the concept, an initial architecture will be created based on initial ideas. This will later be dissected and each element will be analysed and discussed separately. From this analysis the elements is recombined into a final architecture describing how Modular PCG should be applied and understood in relation to game development. Lastly, Modular PCG will be evaluated by creating a theoretical game using theoretical modules in a theoretical implementation in CryEngine3. In this project, Modular PCG will not be tested and proven practical in a real game development scenario, however the theoretical evaluation of the concept will illustrate its application in game development and that it can successfully make procedural techniques accessible to designers and developers.

CryEngine

Game engine designed by the German developer Crytek. It has many high-end features, and is free for non-commercial use.

CHAPTER 2
INITIAL INVESTIGATION

This chapter will start by analysing the a few definitions of PCG to give the reader a better understanding of the concept and determine how PCG should be understood in context of this project. In relation to PCG, the contemporary taxonomy will be described to establish basis for later discussions and analysis, and the sub-sections 2.1.1 and 2.1.2 will describe two research branches of PCG, namely Search-Based PCG and Experience Driven PCG.

The purpose of this project is to investigate what research is needed to advance the state of the art of PCG in general, and how PCG can be applied in game development. To determine the current focus of research within the PCG community, section 2.2 will investigate contemporary research topics suggested by other researchers. With focus on game development, this project aims to contribute to the general research field by building on top of what is suggested.

2.1 DEFINITION OF PCG

This section is meant as a clarification of the previous definitions of PCG mentioned in the motivation (section 1.1). In the motivation, PCG was defined as the *"application of computers to generate game content, distinguish interesting instances among the ones generated, and select entertaining instances on behalf of the players"* [1] and *"the creation of game content automatically, through algorithmic means"* [4]. There are, however, some issues with these definitions and the following will elaborate on this and form a clearer definition of PCG.

The definition by Hendrix et al. might be too specific, because it relates to what [8] calls Search-Based PCG which represents one specific area of PCG. In Search-Based PCG the generated content is evaluated and assigned values based on this evaluation. It is often linked with evolutional algorithms, where the algorithm selects the best candidates (highest values) and generates new content based on those. This is a more advanced version of the generate-to-test method of PCG, which normally only tests the generated content according to some criteria, but does not necessarily feature a ranking of the generated content.

The second definition might not be suitable either. It might be too wide since it also captures content generated directly by a player/creator in an editor or as part of gameplay, with assistance from algorithms. It can also be seen as too narrow since the word "automatically" implies that there are possible way for humans or other algorithms to interact with the process[2]. This makes this definition very ambiguous. Like vice [5] defines PCG as having *"limited or no human contribution"*, however from a game design standpoint, a PCG system designed to have no human interaction seems impractical. In some special cases it could be desirable to have algorithms designed to be interacted with by other PCG systems, and therefore not by humans, but there are almost no practical reasons for having a PCG systems without any interaction. A completely autonomous PCG system would be more or less useless, however, one should not dismiss the thought of having a PCG system with no interaction, since such a system could spawn some interesting areas of research, and could be useable in very specific cases.

The definitions talks about "content", which can be defined as many things, and different fields might not agree on what content is. In relation to PCG used in computer games content is widely defined as dynamics, weapons, camera viewpoint, rulesets, characters, quests, dialogue, stories, levels, maps, terrain; in fact most game content besides the game engine and the behaviour of the NPCs [4] [8] [5] [2]. Even the game engine could potentially be procedurally generated and one could imagine a game where everything was generated from scratch, which is said to be one of the grand goals of PCG [5]. However, this might be too comprehensive for this project and

[2] For clarity, human interaction in this connection is seen as applying a PCG system in specific context and/or starting the generation process.

thus, in the context of this project, game content will refer to everything besides the game engine and NPC behaviours.

For practical reasons and applications the definition of PCG will, in this report, follow the definition presented by [24] stating that PCG is *"the algorithmic creation of game content with limited or indirect user input"* [24]. This definition does not allow direct or full human control over the generation and one might expect that this would be desirable, as the goal for this project is to make the generation process accessible to human designers. The reason why this is not desirable is that it will remove the system from the domain of PCG since a PCG system is required to have some form of automation. Without this, a PCG system will become an editorial tool. An assessable PCG system for complete game generation has to be autonomously enough to generate content sufficiently, while being flexible and transparent enough to give a human user agency and empowerment [25].

PCG can be used in different ways, which can require the generation process to be done either online, i.e. during runtime, or offline, during development. As an example, the interior of a building might be generated *online* when the player enters the building, or *offline* and edited by a human designer before the game is shipped [4]. A combination of the two is also possible. The generated content can be said to be *necessary*, i.e. necessary for progression, or *optional* meaning that the player can choose to avoid it.

Concerning the actual algorithms at use, they can be based either on *random seeds* or on *parameter values*. This has to do with the amount of control over the algorithm, if an algorithm is based on a random seed there is little control and if the algorithm takes a multidimensional vector as input a human designer can be allowed almost full control over the generated output by adjust the specific properties. The latter could be desirable regarding multi-level multi-content generators where a human designer needs to affect the generation. Note that random seed does not imply that the output of the algorithm is random. The algorithm can be either *stochastic,* meaning that it will create a new output every time, or *deterministic*, resulting in the same output every time [4]. Generally, algorithms can be said to be either *constructive* or *generate-to-test*. A constructive algorithm will generate the content once, which means that it has to create something that is correct, since it will not correct the generated content after it has been generated. A problem with constructive algorithms is that they often include some randomness, which leads to the lack of controllability [2]. Opposed to this, a generate-to-test algorithm includes a test mechanism that tests the generated content in accordance with some criteria and regenerate the content if this validation test fails. This refers to Search-Based PCG (see section 2.1.1) which ranks the tested content and selects the best for further generation. The difference between Search-Based PCG, constructive algorithms and generate-to-test algorithms can be seen in Figure 1.

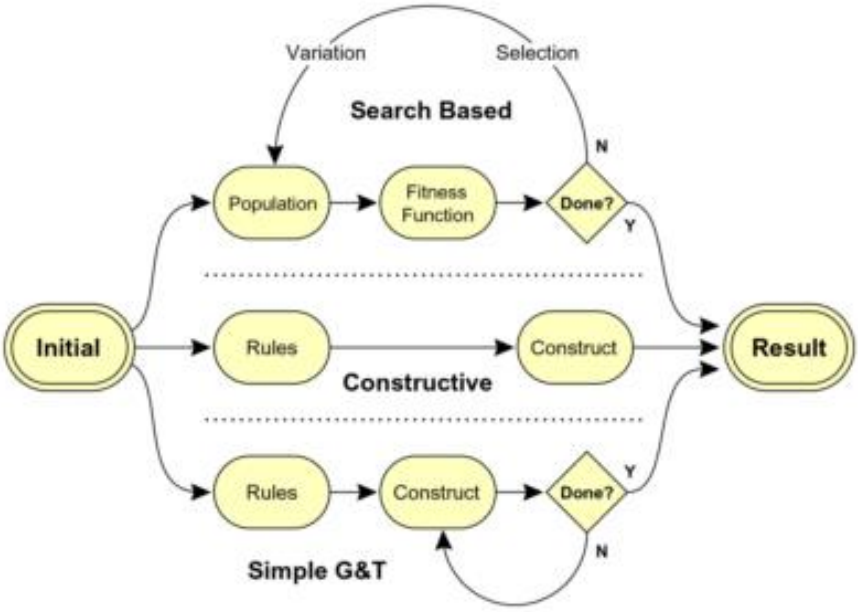


Figure 1: Overview of different approaches to PCG: Search-Based PCG, constructive algorithms and generate-to-test algorithms [8].

2.1.1 SEARCH-BASED PCG

The term Search-Based PCG was proposed by Togelius, et al. [4] and elaborated in [8]. As mentioned in section 2.1 (Definition of PCG) Search-Based PCG is a special case of a generate-to-test algorithm with two main differences. First, the test function grades the generated content; this function is often referred to as the *fitness function* and the grade is thus called the *fitness* of the content [4]. This function determines how well the generated content fits or matches the requirements of the generation. Secondly, new content is based on the content with the highest fitness and the algorithm aims to generate content with higher fitness [4]. For some cases of Search-Based PCG the main generation is based on evolutionary computation (EC), however, this is not necessarily the case. When describing a Search-Based PCG algorithm one talk about its *genotypes*, i.e. the data handled by the evolutionary algorithm[3], and its *phenotype*, i.e. the data handled by the fitness function [4]. Data can be encoded, or represented, from the genotype to the phenotype through either *direct encoding*, where genotype and phenotype is proportional in size, and though *indirect encoding*, where the mapping is nonlinear (see [4] and [8] for further exemplification). The main concern with the encoding is the "curse of dimensionality" that describes the paradox of representing data simple enough for a search algorithm to search though the data quickly and representing it with enough detail for the search algorithm to be

[3] In the case that the generation is based on evolutionary computation.

able to search though it precisely enough [4]. *Locality* is another principal that relates to content representation, and means that a small change in genotype should result in a small change in phenotype and vice versa [4].

A fitness function can be designed to rate content according to many different factors, such as how "fun" a racetrack is [9]. Three types of fitness functions are described in the literature [4]. First, the *direct fitness function*, which extract some specific features from the content and maps this directly to the fitness. The function can be either *theory-driven*, guided by designer intuition or qualitative theory, or *data-driven*, guided by collected data such as questionnaires or physiological measurements. Secondly, the *simulation-based fitness function*, which simulates gameplay with an artificial agent and extracts values from the observed gameplay. The agent can be either *static* or *dynamic*, depending on its ability to change behaviour during gameplay. A changing agent has some learnability, which the fitness function must be able to incorporate. Lastly the *interactive fitness function* is described, which collects data from the player during gameplay, either *explicitly*, e.g. though questionnaires, or *implicitly*, e.g. though measurements in the game.

One problem with Search-Based PCG, as suggested by [4], is that it might be best suited for offline generation since the time it take to generate the optimal content can vary a lot and one can never be sure how long the generation will take. One could incorporate a maximum time or maximum evolutions to compensate for this and keep the generation time down, however this might result in the creation of some less optimal content. Another issue with Search-Based PCG is that designers cannot be sure exactly how the content will manifest itself, but only explicitly specify some desirable properties of the content. This can be said to be the biggest flaw with Search-Based PCG. Even though the content is generated according to a fitness function making sure the content is valid and follow some design specifications, human designers has no say in the actual generation and are not able to adjust specific elements of the generation without generating the content again. This removes the design agency, which as stated before is an unwanted effect [25].

Search-Based PCG can been seen as a high-level content generation method, which is why it is important to consider human designer interaction, since designers normally are tasked with planning games on a higher level. Yannakakis & Togelius [2] suggests using constructive algorithms, such as L-Systems (see Appendix I), alongside with Search-Based PCG as a genotype-to-phenotype mapping. Such algorithms could also be used to support human designers, thus allowing them time to be creative and not preoccupied with time-consuming tasks.

2.1.2 EXPERIENCE DRIVEN PCG

Experience Driven PCG was proposed by Yannakakis & Togelius [2] to describe, "*a generic and effective approach for the optimization of user (player) experience*" [2]. They state that game content can be seen as indirect building blocks for player experience and it therefore is possible to change the experience by changing the content. The generation process of Experience Driven PCG is divided into four parts as illustrated in Figure 2.

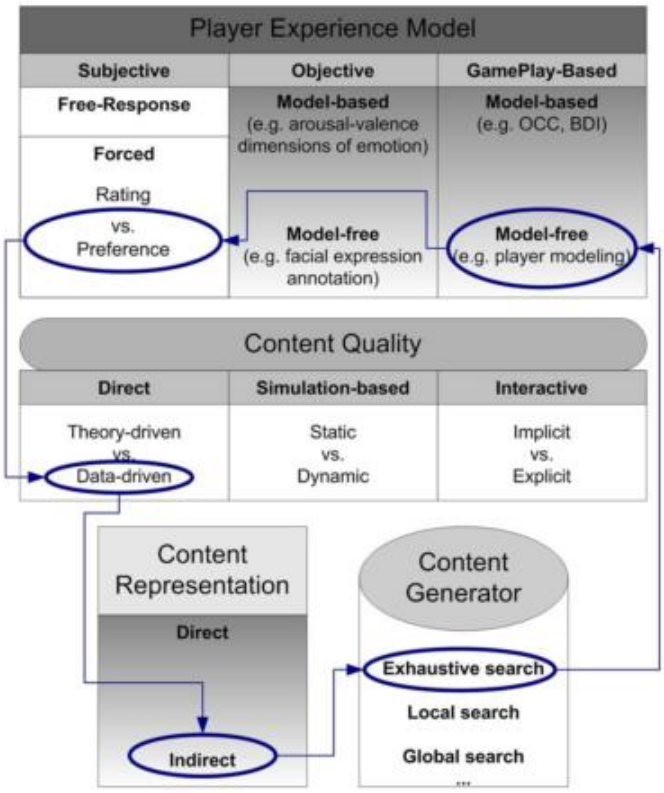


Figure 2: Framework of Experience Driven PCG [2],

The first part, the player experience model, is built based on collected data from the player(s). It can either be *subjective*, i.e. expressed by the players themselves, *objective*, i.e. gathered from the player through alternative means, and finally *gameplay-based*, i.e. gathered through an interaction between game and players [2]. Subjective player experience can be based on *free-response*, giving richer but more complex information, or *forced* data, giving answers to more specific questions. Objective experience modelling usually requires access to different modalities to determine the affective state of the player during gameplay. These modalities can be analyses though different means, for instance through electrocardiography (ECG), galvanic skin response (GSR), respiration, electroencephalography (EEG), motion tracking, facial expressions and gaze. The modelling can be either *model-based*, meaning experience models are formed based on theories for e.g. arousal, and *model-free*, meaning that new models are constructed and mapped to different modalities of player input [2]. Gameplay-based player experience modelling is based on the assumption that player experience is linked to player actions, and any player interaction with a game can form basis for this modelling. Like the objective approach, gameplay-based modelling can be *model-based* or *model-free* or a hybrid of the two. The advantage of this method is that it is the least intrusive and very computational efficient, even though it results in a low-resolution model and are often based on assumptions [2].

Experience Driven PCG relates to Search-Based PCG [4] in the sense that the acquired player model are used to validate the fitness of the generated content. Both methods tries to create the best suitable content, and in the case of Experience Driven PCG the content must be optimised for player experience. The evaluation process, the second part of Experience Driven PCG

assessing of content quality, follows the same taxonomy as Search-Based PCG and can be either *direct*, *simulation*-based or *interactive* [2] (see section 2.1.1).

The third part, the content representation, are again related to and uses the same taxonomy as Search-Based PCG. Content is represented as genotypes and phenotypes and different encodings is used to translate genotypes to phenotypes.

The final part of Experience Driven PCG, the content generation process, goes through the search space created by the player experience model, evaluation and representation and generates the final game content. The generation should be able to recognise "*if, how much, and how often content should be generated for a particular player*" [2], and identify the likes and dislikes of the player and adjust the content accordingly.

2.2 Research Topics

This section will give an idea of the current research within PCG, and which topics that could be interesting to pursue to advance the state of the art. As Togelius, et al. [5] states: *"PCG is a rich and fertile soil for research and experimentation into new techniques, with obvious benefits both for industry and for the science of game design"* [5]. By "*fertile soil*" Togelius, et al. refers to the youth of PCG and the many new and relatively uninvestigated areas that arise. They suggest pursuing three grand goals for PCG representing the most important topics, which should guide the overall direction of the research field [5]. The three goals cover multi-level multi-content generation, PCG-based game design and lastly the generation of complete games.

The second goal, about PCG-based games, i.e. games that are built around PCG and could not exist without it, is interesting because it would facilitate a completely new genre of games where PCG would be the central mechanic. In most of the games, that utilizes PCG, the generation is an add-on or replacement of human design, and the game could very well exist without it. PCG-based games would require innovative ways of using PCG and would prove an interesting area of research.

Accomplishing the first and third goals, creating multi-level multi-content generators and complete game generators, could be an amazing achievement, however, it might not be desirable as such. It could have the side effect of alienating the human designers from the game development process. The problem at hand is that PCG often is designed to work autonomously and offers very little to no human interaction. This can create an unwanted distance between the users, i.e. designers and developers and the PCG system. In some cases, only the creator of the system knows the functionalities. This proved a real issue in [25], where the designers felt a loss of agency as a PCG system was made responsible for parts of the design. Khaled, et al. [25] points

out that designers might be uncomfortable with relying on an automated system and they have to be comfortable with the system and know its capabilities. When creating a PCG system one has to consider how it integrates within other game technologies and how designers interface with it and in general how it fits within the development pipeline [25]. The system should be easily applied and it should be clear to designers, enabling them to evaluate if the system is applicable to their needs. It should not be the goal to replace human designers, but to facilitate and support their work and ease the development [3].

CHAPTER 3
PROBLEM STATEMENT

In the initial investigation, the concept of PCG was analysed and discussed together with the basic taxonomy and the two research branches Search-Based PCG and Experience Driven PCG was described. Different definitions of PCG was analysed, and in the context of this project it was decided to use the definition by Togelius, et al. stating that it is *"the algorithmic creation of game content with limited or indirect user input"* [24].

To direct the research of this project, section 2.2 mentioned a few contemporary research areas, referred to as the grand goals for PCG. Pursuing any of these should help advance the state of the art. Among the grand goals was the research in multi-level multi-content generators and complete game generators, and it could be interesting to investigate how this type of procedural generation can be integrated within a normal game development process and how it can be used to support human designers and developers.

This project will thus investigate how to make the procedural creation of complete games a practical possibility and how this will integrate with other game technologies and how it could be integrated into the development pipeline of human designers. In short, this project will try to answer the following problem statement:

How can a PCG system designed for complete game generation be made accessible to human designers and how can it be integrated within the development pipeline?

CHAPTER 4
ANALYSIS

This chapter will first analyse previous attempts to create multi-level multi-content generators to generate complete games. This analysis will investigate the relationship between the complete game generation systems and human designers, and determine how multi-level multi-content generators for complete game generation best facilitates human interaction. Through the analysis, it will become clear that the existing attempts provides very limited controllability, which fosters a gap between the PCG algorithms and the designers and developers. To close this gap the concept Modular PCG is proposed to describe a system that combines the strengths of PCG and the controllability of manual content creation.

4.1 Complete Game Generation

This section will review the literature and game industry for previous uses of multi-level multi-content generators and attempts to generate complete games. By analysing previous examples from the literature, this section will investigate how this type of generation can facilitate human designer interaction.

Dwarf Fortress

This single-player fantasy game published in 2006 is set in a randomly generated persistent world presented purely with ASCII graphics.

Cube World

An adventure game where players explore an endless procedurally generated world. The game was published in alpha in 2013.

.kkrieger

A first-person shooter, created by a German demogroup. It won first place in the 96k game competition at Breakpoint in April 2004.

Minecraft

A sandbox indie game originally created by the Swedish programmer Markus "Notch" Persson in 2009, and later published by Mojang.

Even though there are not many games using PCG in this extreme, there are a few examples worth mentioning, namely Dwarf Fortress, .kkrieger, Minecraft and Cube World. Among these, Dwarf Fortress and .kkrieger might be the best examples since all elements of these games are procedurally generated. In Dwarf Fortress, the world is generated completely from scratch including characters, civilization structures and ecosystems, which are able to react to their surroundings. The world history and historical events and figures are also procedural and documented as game lore. In .kkrieger PCG is used as data compression making this 3D shooter including textures and sounds uses only 95 kilobytes of data, which can be estimated to be approximately 0.1% of what a game of equal quality would use[4]. The other examples also relies heavily on PCG, however it is mostly for level generation purposes enabling near endless levels and huge variation.

As with games, there are only a few examples of multi-level multi-content generators and attempts of complete game generation within the research community [5]. Examples of

[4] As a comparison, the game Quake by id Software from 1996 requires 80 megabytes of disk space.

complete game generation include [6] [26] [27], while [28] [29] (and [30]) are examples of multi-level multi-content generation.

Browne & Maire [6] invented a system, the LUDI system, for procedurally generating board games through evolutionary techniques as described in the motivation (section 1.1). Togelius & Schmidhuber [26] and Cook & Colton [27] has tried generating arcade style games, resembling for instance Pac-Man, from scratch through evolutionary techniques. [26] uses Search-Based PCG, and both systems generates games with three main components, namely a map in the form of a 2D grid, a layout describing placement of players and NPCs on the map, and finally a ruleset describing the rules, e.g. movement, collision, time, etc., for the games. The games and rules produced are, however, still very simple, but one could argue that game rules in their basic form are rather simple.

Pac-Man

An arcade game first released in 1980, developed by Namco. Through the years several remakes has been made and the game has become cult.

The main concern with these approaches is that they are designed to have no human interaction. [26] is meant as a proof-of-concept demonstrating complete game generation, and how computational intelligence can be used to generate simple games. The main critic of [26] is that the generated games, according to themselves, does not represent good game design and are not particularly fun. It can be argued that automated complete game generation should only be used if the system was able to design games with the same quality as skilled human game designers. Togelius & Schmidhuber [26] argue that their system can be used to generate prototypes of new game ideas, where a human designer specifies the game engine and the axioms that define the rule space. Another possible use of automatic game design could be in the post-production stage to fine-tune the design of a level or to adjust the difficulty [26]. These two suggestions moves complete game generation towards a more supportive role, where the algorithms support human design. This would be a step in the right direction; however, the generation process is still not designed for human interaction, which, in my view, is required before complete game generation can be said to facilitate human design.

The ANGELINA system presented by Cook & Colton [27] has the same capabilities as the one presented in [26], with the addition ability of taking a human designed level and authoring rules specifically for that. This is again a step in the right direction, but as with the previous example, it is on its own an automatic enclosed system.

Unfortunately, none of the examples of complete game generation gives any solid solution on how such systems can be made accessible to human designers. Complete game generation might be too complex, since it implies incorporating all elements of game creation into one algorithmic bundle that often are very autonomous and closed. Multi-purpose multi-level generators might provide a more concrete solution, and thus the last part of this section will briefly discuss two

examples of multi-level multi-content generation [28] [29] (and [30]), to further investigate how human designers could be included in a more complex PCG context.

In [28] the story and map structure is generated using a waterfall model where story is generated before the map, making the map suit the story structure. Because the map structure is generated after, and in accordance with, the story structure, the system has to understand the story and context. To enable this the story is written as a list of plot points, which are high-level specification of time with a semantic and recognisable meaning. Each plot point include NPCs and locations (not information about spatial layout) and a reactive script that control NPCs and modifies the game world according to the plot point [28]. The system can procedurally generate these plot points, but the real benefit is that the system allows a human designer to author these plot points. To see an example of a story written as plot points, see Table 1. This method seems to be very practical since it allows an easy overview of the story and allows a human designer to author the main event, characters and locations, while procedural techniques can be tasked with the job of authoring the links between the plot points.

1. **Take** (paladin, water-bucket, palace)
2. **Kill** (paladin, baba-yaga, water-bucket, graveyard1)
3. **Drop** (baba-yaga, ruby-slippers, graveyard1)
4. **Take** (paladin, shoes, graveyard1)
5. **Gain-Trust** (paladin, king-alfred, shoes, palace)
6. **Tell-About** (king-alfred, treasure, treasure-cave, paladin)
7. **Take** (paladin, treasure, treasure-cave)
8. **Trap-Closes** (paladin, treasure-cave)
9. **Solve-Puzzle** (paladin, treasure-cave)
10. **Trap-Opens** (paladin, treasure-cave)
Hero (paladin), NPC (baba-yaga), NPC (king-alfred), Place (palace), Place (graveyard1), Place (treasure-cave), Thing (water-bucket), Thing (treasure), Thing (ruby-slippers), Type (baba-yaga, witch), Type (king-alfred, king), Type (palace, castle), Type (graveyard1, graveyard), Type (treasure-cave, cave), Type (water-bucket, bucket), Type (ruby-slippers, shoes), Type (treasure, gold), Evil (baba-yaga)

Table 1: An example of a simple story represented as a list of plot points (top) and an initial state (bottom) [28]

Beside plot point authoring, the system presented by Hartsook, et al. [28] enables a human designer to adjust the distribution maps (bitmap images) generated to locate object and scenery in the game. The techniques presented enables human interaction and helps close the gap between PCG and designers, and enables collaboration between the two. For further reading, [28] is also discussed in relation to quest generation in section 7.5.

Smelik, et al. [29] [30] criticises traditional procedural methods and gives three reasons why PCG has not been able to switch the content creation process of game development from manual to (semi-)automatic. They state that procedural methods often are complex and unintuitive to use, has little controllability and are difficult to integrate within an already existing virtual world. To solve this issue they presents a declarative modelling approach, which enables designers to

create virtual worlds fast and efficiently. Their approach aim to combine the strengths of PCG and the controllability of manual content creation. They have implemented this in the application Sketchaworld, which utilizes two novel techniques, namely *interactive procedural sketching* and *virtual world consistency maintenance*, letting designers sketch the world layout in rough details. *"Procedural sketching provides a fast and more intuitive way to model virtual worlds, by letting designers interactively sketch their virtual world using high-level terrain features […]. Consistency maintenance guarantees that the semantics of all terrain features is preserved throughout the modeling process"* [30]. The idea behind this technique is that designers will have enough control to specify what they want, and by controlling high-level terrain features, through *interactive procedural sketching*, and will be able to create a large virtual landscape quickly and efficiently [29]. The high-level features will in turn control different procedural methods, which will add details to the world. The second and more automated part of the framework is the *virtual world consistency maintenance*, which allows designers to freedom to change features that might affect others without redesigning each to solve potential conflicts. The Sketchaworld framework is illustrated in Figure 3, and described in more details in [30].

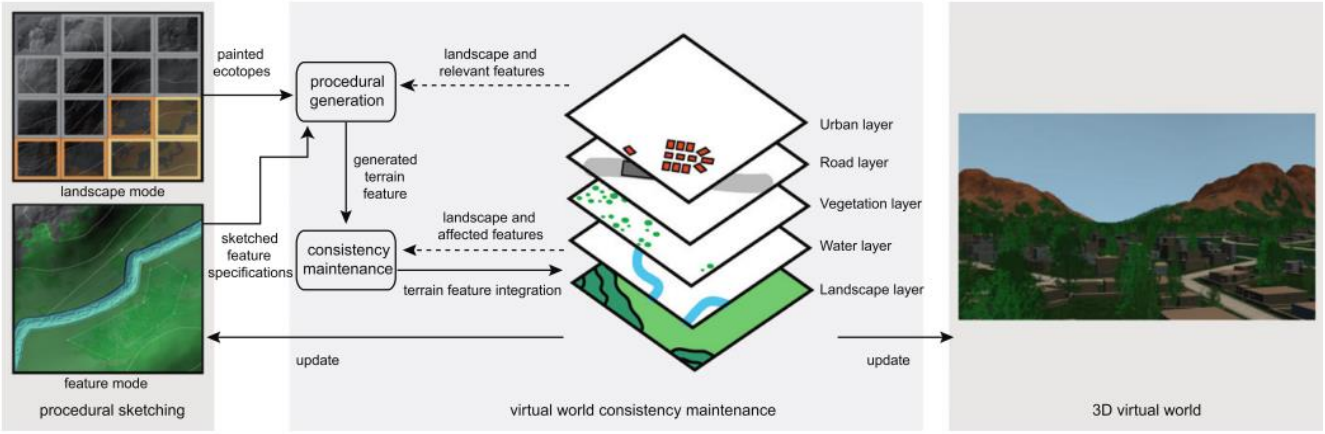


Figure 3: Overview of the Sketchaworld framework.

What is interesting about their framework is the focus on accessibility. They have created editorial tools with the designer in mind and designed them such that they resemble familiar tool from classical image editing software, thus making them more relatable. They have incorporated a feedback loop between designer actions and the visual output to allow near real-time interaction with the virtual world [30].

They have validated their approach through different user sessions where professionals and non-professionals have tested Sketchaworld. The users found it easy to create virtual worlds matching their intent, even with no 3D modelling experience [30]. Sketchaworld has proven a powerful tool; however, some designers requested more design freedom and controllability over individual models. Even though designers can adjust the consistency maintenance settings in

different ways, Smelik, et al. believe that designers should be provided with an even more fine-grained editing option [30].

They have designed Sketchaworld to facilitate replacement of the currently used content, for instance replacing the textures with high-quality textures. This makes their approach very flexible and it can be adapted to fulfil many design needs. Because of the structure, it is technically also possible to design new procedural methods within the same framework. New high-level generators could potentially be implemented to generate elements like railways, lakes, etc. [30]. In theory, new methods only have to collaborate with other methods and the framework on a semantic level, because the generation can be independent from the feature interaction. The new methods has to be made compatible with procedural sketching and be aware of their surroundings, meaning that rules should be designed to solve feature interaction and they should be able to cope with loosing *claims* (i.e. when a feature requests a terrain area).

The two examples of multi-level multi-content generators [28] [30] illustrates how PCG can support human design, and how techniques can be made accessible to designers. Both examples talk about content generation on a semantic level, where high-level content is authored by a human designer and low-level content is authored by algorithms reacting to the high-level content. This facilitates a collaboration between PCG and human designers. The declarative modelling approach presented by Smelik, et al. [30] might be the best suggestion on how game designers can create a complete world fast and effectively using PCG, while keeping the artistic control. An important aspect is the editorial option, procedural sketching, incorporated into Sketchaworld, which uses the same metaphors as normal image editing software, thus making it more relatable.

In [30] Smelik, et al. states that other researchers are able to expand the capabilities of Sketchaworld by creating additionally functions that can generate other types of content. This should be possible as long as new elements incorporate semantic rules that are compatible with the existing framework and an option for procedural sketching is designed. However, from the articles [29] [30] it is not clear how to design such generators and adapt them to the Sketchaworld application. Furthermore, there seems to be too many considerations regarding their interaction with other features in the virtual world, which makes the design rather complicated. A better and more designer friendly approach would be to establish some concrete design guidelines and/or templates which designers could base their implementation on. These guidelines and templates should allow designers to implement PCG algorithms that are able to interact with other content generators in the environment without considering the specific application of each generator. New generators will thus fit within the architecture of the existing ones and designers will have the freedom to create as many generators as they need and use generators designed by other developers.

If this was possible, one could imagine an application, much like Sketchaworld, which instead of one large system consisted of many individual content generators integrated with one another in a common framework that allowed designers and developers to add and remove different generators to achieve a desired result. Depending on the design, such generators will give designers the possibility to control high-level features for fast and efficient development, and individual elements could be added and removed with ease. Different types of content and functions could be implemented facilitating a variety of applications and games without the difficulty of adapting generators and content to each other and the virtual environment. Such an architecture could be used to design and implement complex games without too much hassle.

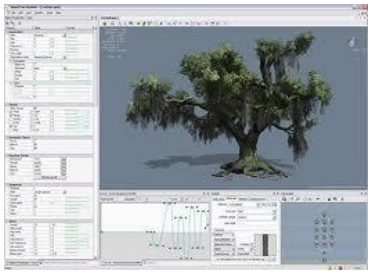
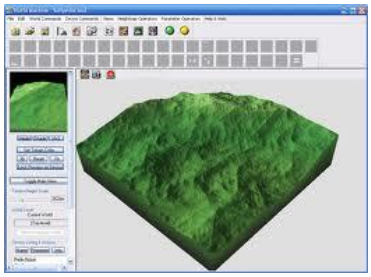
It can be theorised that such a system would close the gap between designers and PCG algorithms and make PCG accessible to game designers and make it a more integrated part of game development. The creation of such a system could be interesting both for the research community and for the game industry. To elaborate on this concept, section 4.2 will focus on how such a system can be made a reality and how it can facilitate game creation in collaboration with game designers and developers.

4.2 Introducing Modular PCG

In the motivation (section 1.1) the games Elite and Rogue were mentioned as two of the earliest examples of games that utilised PCG. Despite the long history of the technology, however, PCG is still not widely used and Yannakakis & Togelius [2] mentions two reasons why. One reason might be that not all types of game content can be generated with the desired reliability, variability and quality. Secondly, PCG techniques are not controllable enough, meaning that a designer or algorithm cannot shape the outcome [2]. This is an issue also mentioned by Smelik, et al. [30] (see section 4.1), and this controllability issue is something, which Modular PCG should aim to solve.

Another issue with PCG is that most generators are designed for a specific purpose for a specific implementation, which often means they cannot be reused in other application and, as stated before, offers very little interaction. Togelius, et al. [5] mentions the lack reusable content generators as a problem. For other types of game content, plug-and-play middleware are available, but within PCG only SpeedTree, and a few landscaping tools, such as World Machine and CityEngine [31], can be mentioned as widely used software, and they only cover a limited space of content. It would be very interesting to have an array of plug-and-play content generators that could be applied across different games and different genres. Theoretically, this will increase the use of PCG in commercial games and could help meet the players demand for

content, as discussed in the motivation (section 1.1), and allow designers more time and freedom.

SpeedTree	CityEngine	World Machine
		
Toolkit used to create 3D animated plants and trees for games, animations, visual effects shots, and architectural renderings.	A 3D modeling software developed by Esri R&D Center Zurich. Specialized in the generation of 3D urban environments.	Used for procedural terrain creation, simulations of nature, and interactive editing to produce realistic looking terrain quickly and easily.

4.2.1 Definition of Modular PCG

As described throughout this report the main concern with PCG is its limited accessibility, which is the main suspect as to why PCG is not widely used in game development. This section will define Modular PCG, which aim is to close the gap between PCG algorithms and game designers and developers and thus easing the development and help the adaptation of PCG into the game industry.

If PCG should adapted into the workflow of designers it should be easy to interface with and control, as emphasised by several researchers [25] [29] [30] [3] [32]. This would require the algorithms to be more transparent and relatable, as opposed to one large PCG system that generates all parts of game autonomously, similar to the systems described by [6] [26] [27] (see section 4.1). Smaller PCG systems, or modules, should also allow designers to intervene and adjust the outcome of any of them, thus shaping the generation and getting back their design agency. Therefore, it can be theorised that smaller relatable and controllable modules might help integrating PCG into the workflow of human designers and developers. This need for controllability would also be the case if algorithms should be controlled by, or interact with, other PCG algorithms [1]. This interaction is also mentioned as a possible research topic by Togelius, et al. [5], which suggest either using a waterfall approach, where each type of content is generated after the other and where one puts constraints on the following, or an interaction, where constraints are posted in global space and all generators react to these constraints (see Figure 4).

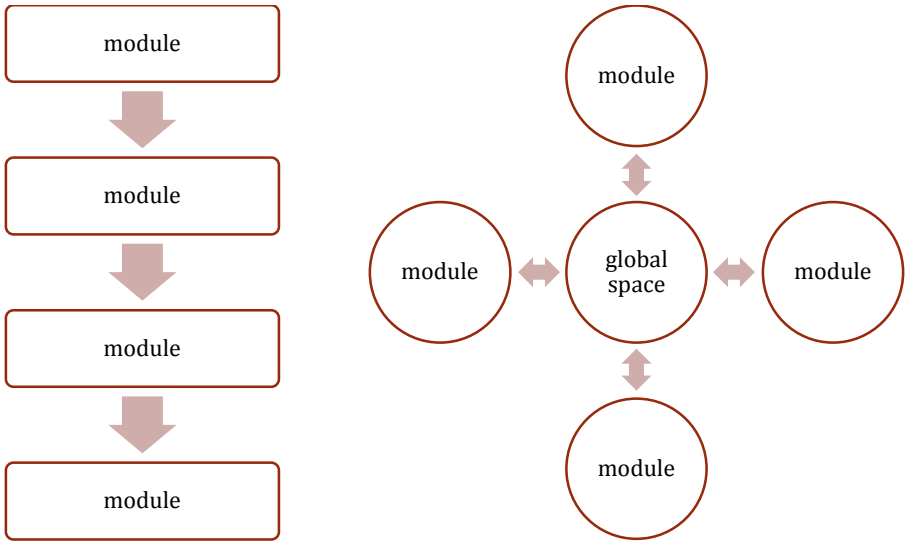


Figure 4: Left: Illustration of the waterfall approach. Right: Illustration of the interactive approach.

As said these smaller PCG systems can be seen as modules, which is why I propose the term Modular PCG to describe a system of multiple individual PCG algorithms, or modules, that acts on their own, which combined facilitates easy and relatable game development. Modular PCG can be described as a system, but the terms structure and architecture will also be used to describe it throughout the report, and the terms will be used interchangeably.

It can be theorised that Modular PCG will be a better approach than developing one large PCG system for two main reasons. First, by having a modular setup, each module will be more specific and thus easier to relate to and human designers will better understand the capabilities of each module. Secondly, different modules with different capability will enable designers to choose only the ones they need for the implementation they are working on. It can therefore be theorised that research in this area will prove beneficial for both the game industry and general PCG research, and suspect that this will help integrate PCG into commercial game development, which is a necessary step for the success of PCG. A Modular PCG system could be used to create PCG-based games, complete games and the system would generate multi-level multi-content, and this take on PCG thus captures the original grand goals presented by Togelius, et al. [5], mentioned in section 2.2.

As such, the individual modules in a Modular PCG system can be any PCG algorithm, meaning it should be possible to include a variety of different content generators; however, one has to consider who the user(s) will be. If the user is a human designer, the module should be easy to interface. If it is a player, the module most likely needs to facilitate some form for adaptation to player's desires or actions (see section 2.1.2 about Experience Driven PCG). Finally, if the user is another PCG module, the two must be able to talk to each other and adaptation is most likely also required. In any case, the module has to be specific and autonomous enough to handle

demands in a sufficient way, but at the same time, if interacting with a human designer, be flexible and transparent to grant design agency and empowerment [25]. Considering this, the definition of PCG presented in section 2.1: *"the algorithmic creation of game content with limited or indirect user input"* [24], is still valid; however one additional comment has to be made in relation to Modular PCG. I propose that "user" should be understood as both game designers, players and other PCG modules, and it should be possible for modules to have multiple users, e.g. a designer specifying a level layout, a player "requesting" more enemies, and a quest module requesting NPC locations.

4.2.2 Modular PCG as a Research Area

There are many interesting areas of research within PCG, which must not be neglected with the introduction of Modular PCG. This section will therefore review some research topics within PCG suggested by other researchers [5] [4] [8] [1] and explain how Modular PCG is able to cover these topics. Hendrikx, et al. [1] suggests five areas of research. The first is the research in the generation of what they see as higher level content, i.e. Game Scenarios, Game Design and Derived Content (see Figure 5). They also suggest is research in more detailed generators, specifically in relation to Game Space and Game Systems, and suggest that research should focus on the interaction between generators as well.

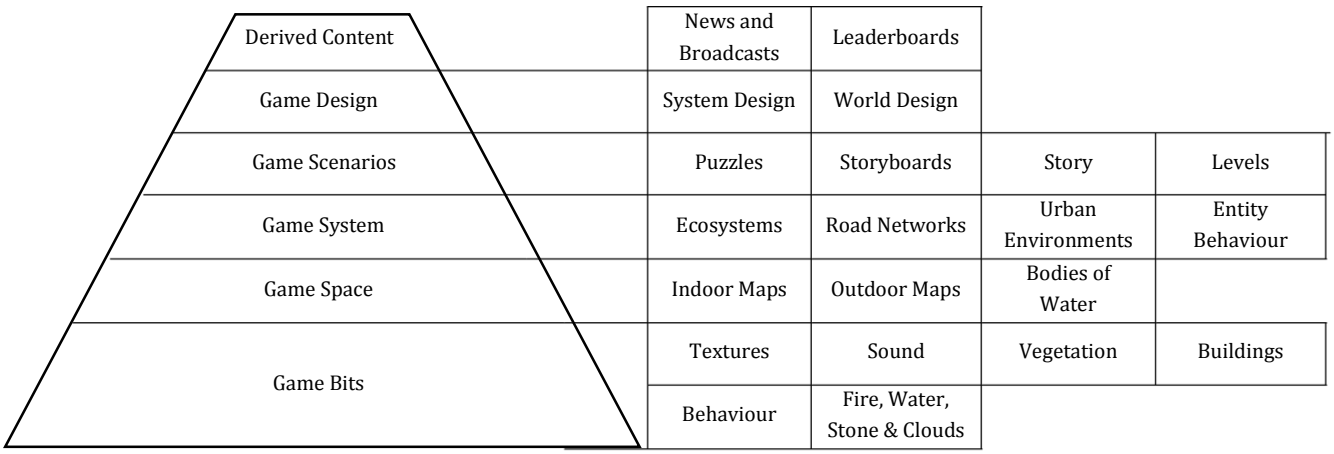


Figure 5: Types of game content that can be procedurally generate [1].

Even though there have not been many examples of generators capable of generating content from the top of the pyramid, it should be possible to create generators for all the different types of content listed by Hendrikx, et al. [1]. Because Modular PCG should be seen as a framework and a way of structuring different procedural algorithms, it is possible to incorporate many different algorithms as modules. The issue is therefore not which content can be generated, but how modules should communicate with each other and the general structure of the system. Within the scope of this project, it might be too comprehensive to create a Modular PCG system that

includes the top most content from Figure 5. What I consider feasible within the scope of this project, and a good starting point for proving the validity of Modular PCG, is to create a system based on high-level modules designed to generate Game Scenarios or Game Systems. The reason for starting with higher-level content is that this type is more designer oriented and it can be theorised that well designed, i.e. accessible, high-level modules will be able to aid a designer more efficiently, i.e. help structure, author and plan. If modules were designed for each of the categories in the content pyramid in Figure 5, the high-level modules should be able to control lower-level modules generating Game Spaces or Game Bits. This method can therefore be seen as a top-down approach, where low-level content is controlled by higher-level content and thus the designer interaction lies with the high-level content. Contrary, it would be possible to create a bottom-up approach, where the designer interaction lies primarily with the low-level content that in turn controls the higher-level content.

Hendrikx, et al. [1] also suggests utilizing multi-core computer systems or multi-node computer networks to enhance the quality of PCG in relation to the individual Game Bits, making generation faster and more time efficient. Such advancements could easily be incorporated within Modular PCG, by designing modules to utilise these techniques.

Togelius, et al. [4], who talks about Search-Based PCG (see section 2.1.1), mentions the investigation of content representation, i.e. genotypes, and fitness function design as a research topic. This is a more general concern and something one always has to consider when designing search-based PCG. Search-Based PCG has both strengths and weaknesses; in broad terms, one can say that it will create content perfectly suited to a given situation but on the other hand it is a very closed circuit and generation time can vary a lot, thus Search-Based PCG is best used in offline generation, i.e. during development. That said some modules might benefit from Search-Based PCG. To fit design requirements from higher or lower-level modules, modules can use Search-Based PCG to generation the best-suited content that links the game together. It can also be used to create modules that will adapt to the player and the player's actions (see section 2.1.2 about Experience Driven PCG). This relates to their subsequent paper where Togelius, et al. [8] suggests investigating player models and how these can be integrated into the evaluation functions. This could lead to investigation of how to incorporate the player model into the generation process, which could help optimise the evaluation and ideally, if content could be generated to a satisfying standard the first time, make the evaluation redundant. A player model can be setup by one individual player before or during play or be created based on a theoretical approximation of player desires and expectations. Because the interactive, i.e. player-driven, and the theoretical player model are capable of different things, a topic of research could be to investigate in which cases either is usable and how the two could be combined.

In [8] they also mentions some more general research topics, which illustrates some general concerns when creating PCG algorithms, and Search-Based PCG. First, one could identify which

type of content is suitable for generation and how it should be represented in the search space. This would of cause depend on the application and whether or not it should be done online or offline, and if the content is optional or necessary (see section 2.1). They also suggest research within optimisation of PCG algorithms, namely how to make them more reliable and precise and how to speed up the generation process. This research is something that could benefit regular PCG algorithms and thus also the modules in Modular PCG.

The last topic, which they [8] suggest, is related to the evaluation of the generators themselves. Since PCG are capable of generating an array of different things it is difficult to compare and evaluate generators against each other. Therefore, they suggest setting up a framework for testing generators, where PCG algorithms must solve the same problem using the same API. This suggesting is highly relevant when talking about Modular PCG since different modules has to communicate and integrate with one another, and therefore a common API would be the ideal. This can of cause be seen as a limitation, since creators has to make their modules work within the same API and work within some general design requirements and specifications. On the other hand, a Modular PCG system would enable developers to use different modules from different designers and apply them in their own development. I believe this should be the grand goal of Modular PCG, however, Modular PCG will have to be defined further if it should be acknowledged as a part of PCG research and become a research area on its own. Further investigation should determine which modules should be produced, and how the inputs and outputs should be designed. Research in Modular PCG should also take designers and players, i.e. users, into consideration, allowing designers to direct and shape the generation and it could be useful to implement player adaptation into certain modules, allowing the generation to change and adapt to players and playing styles accordingly.

4.2.3 Modular PCG in the Industry

As described in section 2.2, Togelius, et al. [5] lists three grand goals of PCG, one of which is complete game generation where a PCG system should be able to generate a complete game including all assets and the engine itself. As mentioned, it could be a fruitful research area, however this is not what the industry wants, and it is only logical that PCG research aims to fulfil the needs from the industry. This is backed up in [3], co-written by people working for Electronic Arts, stating, *"Game artists aren't looking for a one-button procedural solution. Instead, they're interested in procedural methods that help with tedious tasks and provide results that adjust to gaming constraints"* [3]. PCG should fit within the already established workflow and *"free artists to spend time creating and polishing, rather than performing mundane, repetitive, and time-consuming tasks. […] Game*

Electronic Arts

Founded in 1982 this American developer, marketer, publisher and distributor of video games are known for Need for Speed, The Sims, Medal of Honor, and other game titles.

artists are looking for procedural methods for modeling organic objects that meet asset budgets and yet remain convincing" [3]. In relation to this, the tools provided should be easy to use and intuitive to the designers and resemble well-known functions such as soft selection, drag and drop, insertion and deletion [32], and in general accessible to designers [25].

Specifically related to city modelling Lipp, et al. [32] asked artists and programmers about their needs, and found that previous work within PCG were missing an easy way to implement handcrafted assets and that the artists were missing their direct artistic control. This strengthens the assumption that if the game industry should adapt Modular PCG, and PCG in general, the tools and design metaphors should resemble what designers are familiar and comfortable with.

4.2.4 Applied Modular PCG

To my knowledge the concept of Modular PCG has never been discussed before; however, some applications are using methods similar to the ones proposed in relation to Modular PCG. For instance, in section 4.2 it was pointed out that CityEngine [31] and SpeedTree was some of the few tools widely used in the industry, and they in fact proves as good examples of how Modular PCG should be used and understood. CityEngine, SpeedTree and other applications will thus be discussed in this section to illustrate how Modular PCG could be useful to game designers, and ease the development.

In short, CityEngine is an application used for planning and designing urban architecture and cities. It uses a procedural approach based on L-systems to generate streets, building, etc., and was originally presented by Parish & Müller [31] in 2001, but became commercial in 2008. Watson, et al. [3] also describes procedural modelling in relation to city creation and describes how to incorporate CityEngine into the workflow of game development and movie production. They study how procedural urban modelling has been used in the Need for Speed game series. Lipp, et al. [32] proposes a system compatible with CityEngine for structuring city layouts with focus on relatable editing options, such as drag and drop. The techniques presented resembles in many ways the editing options in Sketchaworld presented by Smelik, et al. [30] (see section 4.1 and 6.4). These techniques are relatable and flexibility and returns some design agency to the designers and is therefore a good example of how Modular PCG should be used and how tools should be implemented. In the later years, these editorial options, together with many more, have been incorporated into CityEngine, and it has become a rather complicated piece of software.

As mentioned, SpeedTree is another example of an application that to some degree resembles how Modular PCG should be structured. SpeedTree represents the many applications that are

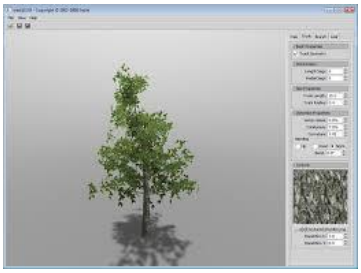
An Ivy Generator

A small tool used to procedurally grow a virtual ivy on 3D objects. The ivy can then be exported as an .obj file and used in other 3D programs.

Xfrog

A procedural organic 3D modeller used to create and animate 3D trees, flowers, nature based special effects or architectural forms.

Tree[d]



An easy to use tree generator with a user interface that allows creation of nearly any type of tree within minutes.

designed to generate plants procedurally, and this is a large research field in itself [12]. The problem at hand, like city generation, is that huge environments requires many graphical assets, i.e. geometry and textures. Instead of reusing assets by changing colour and size, which is often noticeable, PCG can help create a variety of assets and thereby improve the realism [12]. In [12] 12 examples of tree and plant generators is listed and described. It is clear that some of the earlier applications are not capable of generating anything useable for computer games, but are mainly usable to illustrate procedural methods, such as L-Systems. Examples of more useful[5] applications include An Ivy Generator, Xfrog and Tree[d] and, as mentioned before, SpeedTree.

SpeedTree, together with the other examples, can be seen as modules for Modular PCG, even though they are not completely integrable with other programs. It could be interesting to have these tools integrable within a common API, e.g. a game engine, allowing fast and productive development. This will enable designers and developers to choose the generators that suits their needs, which previous was mentioned as the grand goal of Modular PCG (see section 2.2). Having such tools within a common API with other modules would also allow them to use each other and thus a more autonomous system can be designed.

To make this a reality a lot of work is required regarding the architectural design of the system and modules and determining how these modules should communicate with the main engine and each other. Inspired by the most successful examples of plant generators, modules working with geometry and 3D models could be designed to use the .obj file format and other industry standards. This might ease interaction between modules and make the generation more

[5] In this context useful means that the application is able to generate files compatible with other programs, for instance by generating .obj files, that the application is somewhat user friendly and that the generated content is of a relatively high quality, thus useable for computer game production.

relatable to designers and developers, and enable them to import objects from other programs if necessary.

City and plant generators, and generally modelling generators, have the benefit of generating something that is visible and physicalized as a 3D object, and is often the end state of generation. Things such as determining what buildings to place in a certain area of a city or which types of plants should populate a forest can be seen as higher-level generation. This was also discussed in section 4.2.2 with inspiration from Hendrikx, et al. [1] who categorised content from low-level to high-level. As described the architecture of Modular PCG could be designed such that higher-level modules determines higher-level content and lower-level modules generates more simple content based on requests from the higher-level modules (referred to as a top-down procedure).

4.3 Summary of Analysis

The analysis has now analysed and discussed different areas based on the problem statement: "*How can a PCG system designed for complete game generation be made accessible to human designers and how can it be integrated within the development pipeline?*"

In the analysis examples of complete game generation and multi-level multi-purpose generators from games and the research community has been analysed. It was discovered that there have not been many successful examples of complete game generation, and the few examples from research have not been very accessible to designers. Thus, a few examples of multi-level multi-purpose generators was discussed and it was found that these had more focus on the controllability and accessibility and gave some initial ideas towards designer interaction. Based on the examples of multi-level multi-purpose generators a new view on PCG was proposed, named Modular PCG. The term Modular PCG describes a system of multiple individual PCG modules that acts on their own and facilitates easy and relatable game development when combined. The modules should be seen as regular PCG algorithms, with the added ability to interact with each other and react according to changes.

The problem at hand is that most PCG systems are designed to generate only one type on content and often offers very little interaction. This makes it difficult to adapt PCG into game development, as designers might not understand the capabilities of the PCG system and are not able to adjust the outcome and thereby loose some design agency. Modular PCG would facilitate a creation of an array of ready to use plug-and-play algorithms, which designers and developers could easily implement into their game. It should be possible for them to interact with the modules to achieve a desired outcome, making Modular PCG very flexible and relatable. Modular PCG should make it easier for game designers and developers to create games and enable them to choose only the modules needed for their implementation. The grand goal of Modular PCG

should be to enable users, designers and developers, to choose many different modules from different designers, and shape and apply them in their own development.

It has been described how Modular PCG fits within the current PCG research and is able to capture the existing research topics, which illustrates the usefulness of this proposed framework. With further work is would be possible to create Modular PCG systems capable of creating whole games through either a top-down or bottom-up approach, incorporating player experience, i.e. Experience Driven PCG, and the powers of Search-Based PCG. It is the hope that Modular PCG will make the process of game creation more streamlined and accessible to human designers.

In section 4.2.3, it was described that game designers are not looking for a one-button procedural solution to generate all aspects at once. What they seek is procedural methods that ease the completion of tedious tasks and provide tools with well-designed metaphors that give them the design agency they need. This is something that Modular PCG should aim to provide, and in section 4.2.4, it was argued that examples of Modular PCG, or modules, can already be found in some of the more successful PCG applications. Because of this and the requirements from the industry, it can be argued that Modular PCG is the right direction for PCG research and this way of thinking will help PCG integrate within the current game industry.

Research within Modular PCG should determine which modules to develop and how inputs and outputs should be designed such that modules can be combined. This, of cause, only represents a fraction of the research needed in Modular PCG, but this report will serve as initial research to prove the validity of modular PCG. This will help shape and clarify Modular PCG and determining its strengths and weaknesses in relation to game development. The architecture of Modular PCG should be investigated further; Figure 6 however, illustrates an initial architecture to describe Modular PCG.

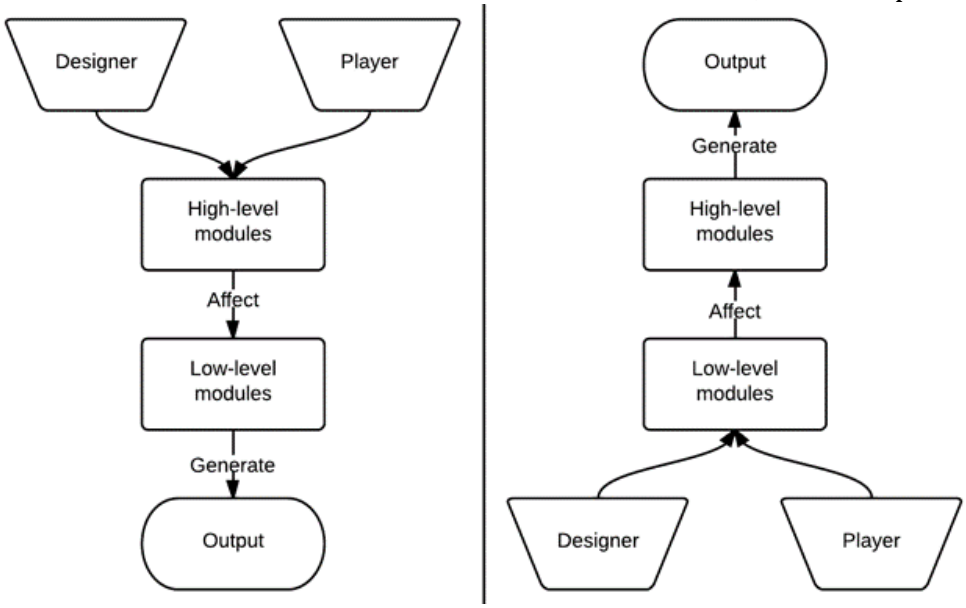


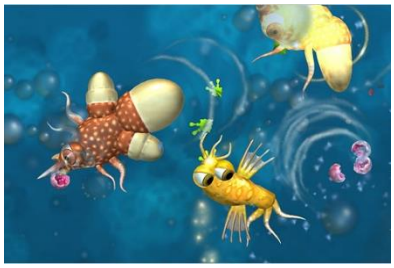
Figure 6: An initial architectural design for Modular PCG. Left: Top-down approach, where designers and players influence high-level modules that affect low-level modules, which generate the final output. Right: Bottom-up approach, where high-level modules adjust to the requirements from lower-level modules, which are controlled by designer and player.

CHAPTER 5
DELIMITATIONS

Because suggesting a new research field is like opening Pandora's Box, this report cannot cover all areas of Modular PCG. The goal is therefore to give a basic holistic vision of Modular PCG, regarding it overall architecture and how new modules should be created and linked to other modules. To focus the research and to give a concrete example of how Modular PCG can be used, this section will review a few research suggestions by Togelius, et al. [5], that can be addressed *"already today"*, to see if any of these would benefit from Modular PCG.

What Togelius, et al. [5] suggests is five actionable steps, which could help advance the state of the art of PCG in general. The first suggestion is to reduce the complexity of PCG focussing on a constrained space of games, similar to old Atari 2600 games. This is because games today are very complex and it can be very difficult to achieve this complexity. They state that within the limitations, one could create a PCG system for complete game generation, which will address one of the grand goals mentioned in section 2.2. Secondly, they suggest research in procedural animations for generated creatures, which will help overcome the animation bottleneck of PCG. The game Spore is the most promising attempt to do this, even though their creature space is rather limited. Thirdly, they suggests creating games with a sense of purpose, and state that procedural generators often create content that looks very generic and does not offer much variation. Generated levels often *"lack meaningful macro-structure and a sense of progression and purpose"* [5], and rarely offers creative design innovations. This is a paradox within

Spore



Single-player god game / life simulation game designed by Will Wright, developed by Maxis and released by Electronic Arts in 2008.

PCG; on one hand, you might want to have a predictable outcome and on the other, you might want innovative, creative and original design. In any case, one should always strive for a purposefully designed game. They also list this issue as a research challenge suitable for a PhD thesis, but as a concrete suggestion for a minor project, they suggest using the Mario AI Benchmark for accomplishing this. This is because it can provide a lot of material for comparison, both level generators and professional created levels. Next they suggest working with player-directed generation to optimise the generation and diverse the content. This relates to [8] who suggests using player models together with Search-Based PCG, which can be seen as Experience Driven PCG [2].

Lastly and most interestingly, Togelius, et al., 2013 [5] suggests investigating the merge of quest and map generation. In the best-designed games, the quests often interact with the game world and vice versa, which often help tell the story and subsequently helps the player explorer the game on both the spatial and narrative level. They state that there are very little work done on generating quests and maps together, whereas there are multiple examples of generators capable of generating only one of the two. [33] [28] can be mentioned as examples of the first, whereas [34] [35] are examples of the latter. In relation to this project, it could be very interesting to investigate this type of complex game generation, because it will test the capabilities of Modular PCG. If Modular PCG is as powerful as suggested throughout this report, it should be possible to create different modules that together in a common architecture will facilitate complex game creation, and allow a human designer the necessary freedom.

Following Hendrikx, et al. [1] both quests and maps can be seen as higher-level content, which strengthens the assumption that quest and map generation will be an excellent example of how two high-level generators should interact and how they should solve the conflicts that might arise. As mentioned higher-level modules might need multiple lower-level modules to generate lower-level content. This was referred to as a top-down approach, and through that, it would be possible to illustrate the interaction and interdependence between modules of different complexity.

Specifically related to quest and map generation, Togelius, et al. [5] present four methods that could direct research. One method could be to use an algorithm that has already been proved to work well for either quest or map generation, and then integrate generation of the other into this. Another way could be to have a quest generator and a map generator take turns generating content and in the process responding to each other's generation. A third option could be to invent a new algorithm that could generate both quest and map synchronously. The final method they propose involves human intervention at any phase of the generation process.

All these methods links very closely to Modular PCG, although they require slightly different structures. The first implies a waterfall approach where either the quest or the map generation module generates its content and then controls, or puts constraints on, the other. This means

that the first has higher priority than the other does. It can be suspected that if the quest represents the main storyline, this will have to be generated first, but if the quest element represents side quests and the map the main element, a map might have to be generated first. The second method requires an interaction between the two modules, meaning that the two will have equal priority and conflict resolution will have to be implemented. This could be used to illustrate how high-level modules interact and solve conflicts as mentioned before. The third method might be outside of Modular PCG since it implies creating a new generator capable of generating both quest and map, which will be the outcome of a modular system, and single multi-content generators is against the concept of Modular PCG. The last method can be seen as a more general thing and can be linked to the first and second mentioned. The idea behind Modular PCG is to let the designer interact whenever possible and logical, to help integrate PCG into the game development pipeline.

Based on the discussion above, Modular PCG will be illustrated through complex game generation, and more specifically by a top-down approach of quest and map generation where both represent high-level content. The two will be linked through either a waterfall or an interactive approach depending of the concrete scenario, however this cannot be decided until further analysis has been carried out. The high-level modules should allow some human interaction and the generated result should look like it has been purposeful designed with a logical progression throughout, thus helping the player explorer both the spatial design and the quest structure.

CHAPTER 6
MODULAR PCG

This project set out to answer how PCG, when used for complex game development, can be made more accessible to game designers, and as a solution the term Modular PCG was proposed, which describes is an architecture that facilitates human design better than previous attempts. Modular PCG describes a system of multiple individual PCG modules that acts on their own and, when combined, facilitates easy and relatable game development. It should allow game developers to choose different modules from different designers and apply them in their own development. In Chapter 5, it was decided to illustrate Modular PCG through complex game generation, and more specifically to use quest and map generation to illustrate the use of Modular PCG. This will facilitate both high-level and lower-level content generation and a top-down approach will form the basis for the structure.

This chapter will describe an initial architecture for Modular PCG and analyse its different elements in order to synthesise one final architecture that describes Modular PCG and how it

should be applied in game development. The purpose of the architecture is thus to describe how it will be possible for multiple content generation modules to form a common architecture enabling coherent generation of complex games and leaving sufficient control to a human designer.

6.1 Initial Architecture

This section will describe some basic thoughts and ideas about the initial architecture of Modular PCG. Previously it was stated that modules should be able to interact and act on their own, enabling designer and developers to use different modules of their choice. Following Figure 6 in section 4.3 there must exist at least two types of interaction between modules, namely high-level modules instructing low-level modules and vice versa. Beyond that, there must be an interaction between the modules and the designers and/or players. First, designers must be able to instruct either high- or low-level modules depending on the approach. Secondly, the modules responsible for the generation should be able to make the generated content available to the player and the designers during development. In connection to the player, some modules might need player input to some extent, either directly or indirectly, and should thus facilitate this and be able to collect the input. As a last type of interaction, modules might need to interact internally, this again might depend on the approach, but it seems logical that modules responsible for the structuring and planning, whether it be high- or low-level, must be able to interact with other modules. This leaves seven types of interaction:

1. High-level modules instructing low-level modules
2. Low-level modules instructing high-level modules
3. Designers instructing high-level modules
4. Designers instructing low-level modules
5. Modules providing content to players and designers
6. Modules getting input from players
7. Internal interaction between modules

Figure 7 shows an example of a Modular PCG system using a top-down approach, which illustrates these different types of connections. The example is purely fictional and the numbers in the model correspond to the number from the list above.

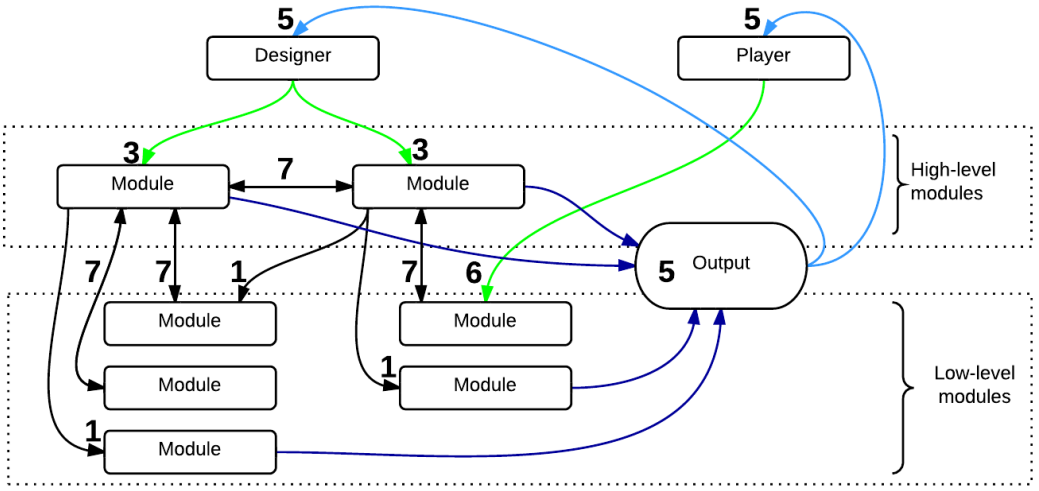


Figure 7: Example of a top-down Modular PCG system, numbers describe different connection types. 1: High-level module instructing low-level module. 3: Designer instructing high-level modules. 5: Modules providing content to player and designer. 6: Modules getting input from player. 7: Modules communicating internally.

These interactions will have to be investigate further in relation to complex game creation and specifically to quest and map generation. The following sections will thus analyse each of the proposed connections using examples from literature, and thereby establishing how these connections should be structured in the common architecture. This will also either verify of disprove the existence of each connection and clarify which is needed and if some can be merged into one.

6.2 High- and Low-level Modules

Firstly, this section will discuss whether modules can be divided into high- and low-level, and how and if this structuring can benefit the architecture.

In section 4.2.2 it was established that both high- and low-level modules exists based on the categorisation of game content by [1], namely: Derived Content, Game Design, Game Scenarios, Game System, Game Space, Game Bits. Generally, generators within Game Scenarios and Game System should be seen as high-level modules, because they can be seen as authors of more complex content. Low-level modules thus generates content within Game Space and Game Bits. The reason why Derived Content and Game Design was not seen as high-level modules, even though they lie higher in the hierarchy, is that Derived Content can be seen as something outside of the game itself, and Game Design are at the same level as human designers. That said one could create an artificial game designer that could replace the role of the human designer and author a complete game experience using a Modular PCG system just as a human designer would.

The division into high- and low-level modules should be seen as categorisation of modules, where high-level modules generate more complex and abstract content as opposed to low-level modules that generally generate simpler and more concrete content. This categorisation might be more relevant when discussing modules of different types and the approach one uses, either top-down or bottom-up, and in fact unnecessary in relation to the general architecture. As an example, let us consider a map generation module capable of generating rough-like maps and a story generation module. In a top-down system the map generation module will be required to adapt to instruction from higher-level modules, such as the story generation module, but it might still be desirable for a designer to draw a basic map layout. In a bottom-up system, the map generation module will put constraints on higher-level modules, and thus the story module will have to adapt to the generated map. In this case, a designer also needs the ability to influence the map layout. In both cases, this interaction between modules seems to be identical on the architectural level. In both cases one module puts constrains on the other, however, since modules should have the ability to adapt to various unknown modules they cannot interact with each other directly. If one module were directly controlled by another, the second module could be seen as a sub-module and the two would as such be seen as one module. The module and sub-module would form their own architecture and the interaction between the two would be internal, which, together with the fact that modules should not interact directly, eliminates the last of the suggested interaction types, "internal interaction between modules".

Instead of direct interaction between modules, one module should change the virtual world and the other should react to this change. This means that both top-down and bottom-up can be achieved by ranking modules in a hierarchy, where modules near the top have greater influence on the virtual world and modules further have to adjust to the changes. That said modules could still be designed to be near the top or further down this hierarchy. This type of structuring can be related to the implementation of Sketchaworld [30]. In Sketchaworld, the content is organised in a top-down fashion in five layers[6] based on semantics and relationships between features, which means that the generation will start with the most abstract structures working down towards structures that are more concrete. Since features in the Sketchaworld implementation are able to interact with each other, it differs slightly from Modular PCG; however, organising modules in layers could be a useful method for designers and developers to prioritise modules and it seems to be a very practical organisational tool.

In relation to the general architecture of Modular PCG, Sketchaworld is a rather limited example because it is designed for the creation of virtual worlds only, and it is therefore difficult to see any logical connection to for instance the non-physical content found in complex games, e.g.

[6] The layers are as follows: "1. Urban layer: e.g. cities, districts, parcels, buildings. 2. Road layer: e.g. highways, local roads and streets, bridges. 3. Vegetation layer: e.g. natural forests, planted vegetation. 4. Water layer: e.g. rivers, canals, lakes, oceans. 5. Landscape layer: elevation profile and soil material" [30]

quests and the content quests might entail. Ideally, it should be possible to create modules that enables generation of both physical and non-physical content. The main issue with non-physical content is that interaction is not straightforward, especially since the modules should not interact directly. It has been established that modules designed for physical content, e.g. forests, cities, grass and buildings, should interacts through the virtual environment, and thus modules for non-physical content generation should interact through a similar space. This interaction space is something, which will be determined in section 6.3, about interaction.

Following the discussion above the same modules can be used in both a top-down and a bottom-up implementation, and modules can therefore not be divided into high- and low-level modules when it comes to the architecture. The terms can however still be used to describe which types of content the modules are designed to generate and the general purpose of the modules. Consequently, since the architecture cannot and should not distinguish between high- and low-level modules, the first two interaction types, "high-level modules instructing low-level modules" and "low-level modules instructing high-level modules", can be merged into one. As said modules should interact through the virtual environment, or a similar space for non-physical content generators, and not directly with each other, which means that the new interaction cannot be named "modules instructing modules". Instead, it will be named "virtual world interaction", which will cover the interaction that arises when modules instruct the virtual world and when they adapt to it. In addition, "designers instructing high-level modules" and "designers instructing low-level modules" can be merged into "designer instructions". Note that this is a one-way interaction, and that the interaction from module to designer and player therefore remains as their own category for now.

6.3 Virtual World Interaction

As described in section 6.2 this interaction type is the result of considering high- and low-level modules as equal on an architectural level, and the decision to avoid any direct interaction between modules. This type thus covers the interaction that arises when modules instruct the virtual world, i.e. when modules makes decisions that affects or changes the virtual world, and the reverse interaction that arises when modules adjust to fit the virtual world. To distinguish between the two, the first type will be called *Instructive* Interaction (I) and the second *Adaptive* Interaction (A).

To illustrate virtual world interaction, the application Sketchaworld can be used to exemplify how generation can be both *instructive* and *adaptive*. In Sketchaworld, terrain features interact with each other, which means that some features *instructs* the generation, some *adapt* to the generation and in many cases features have to solve conflicts between one another meaning both

have to be *instructive* and *adaptive*. In Sketchaworld, features not only have a geometric description, but also a semantic description that defines connections with other features in relation to geometric and functional constraints. This allows many different features to interact and content are specifically designed for these interactions, for instance in a situation where a road and a river intersects, a bridge can be automatically generated. On one hand, this creates a very adaptive and flexible system generation wise, however, it also makes the system very entangled and it is far from straightforward to create new types of content. With Modular PCG new generators, or modules, can be made without considering which other modules exists in the system, which is why modules must be able to interact with the virtual environment and not necessarily with other modules. The reason for this structure is to make is possible for other researchers and developers to contribute to the development of PCG in a more practical and applicable framework.

The content generated by various modules needs to interact with the virtual environment at level that facilitate fast and efficient instruction and adaptation, i.e. contains the right amount of detail. Because of computational complexity and computation time, it might not be optimal to let modules interact with a fully detailed virtual environment. In Sketchaworld, features are divided into three levels of abstraction, which represents different levels of detail. The first is the specification level, which enables designer instructions; the second is the structural level at which features are represented as simple structures allowing interactions; the third and final level is the object level, which is the final detailed generation:

1. "Specification level: user-sketched coarse outline and input parameters (e.g. a forest specification)"
2. "Structural level: the layout of the feature and the area it encompasses (e.g. the contour of the forest)"
3. "Object level: all individual semantic objects making up the feature that will result in concrete, geometric objects (e.g. the set of individual trees)" [30]

Related to Modular PCG, abstraction level 1 facilitates designer input, i.e. designer instructions (see section 6.4), and level 2 and 3 can be seen as the content representation available to designers and target audience respectively (see section 6.5). Since these levels are able to describe input and output of modules, they will be used prospectively to describe the general internal architecture from which each module should be build. Using these descriptions and in relation to virtual world interaction, it would be most beneficial to let modules interact with a representation of the virtual world at abstraction level 2.

Since features in Sketchaworld interact directly with each other, their interactions cannot be applied directly to the architecture of Modular PCG; however, inspiration can be drawn from how conflicts are handled. In Sketchaworld whenever there is conflict between two features, each feature can give one of two requests. Either a *claim*, where the feature requests control over

for instance a terrain area, or a *modification*, where the feature requests a local terrain modification for instance an elevation or material change [30].

In relation to Modular PCG both *claim* and *modification* can be seen as *instructive* interactions, because both represents an instruction from the module to the environment, however, *claim* implies a higher priority than *modification*. As described in section 6.2 it would be useful to organise modules in a hierarchy, where priorities are decided from the order in which the modules are arranged. Within this hierarchical structure, a *claim* would refer to an *instructive* interaction from a higher priority module, and a *modification* would refer to a module of equal priority requesting a change in an already generated part of the environment. Note that this structure does not allow modules to influence modules higher up the hierarchy.

To manage priorities, modules could each be given a priority ID, which would make it possible to group modules by given them the same ID. Modules with the same priority ID would have to adapt to each other, i.e. *adaptive* interaction, and resolve conflicts by requesting *modifications*. In the context of Modular PCG, a *modification* might be better described with the word *proposal*. A *proposal* should be an alternative layout of the content of the proposing module, which the other module should then *counter*, i.e. return a new *proposal*, or *accept*. A maximum amount of proposals could be included in the implementation to stop infinite counter request loops. When a proposal has been accepted or the maximum number of proposals has been reached, both modules will generate their content based on the last proposal. That way both modules will have adjusted their content through *adaptive* interactions. To direct the generation of alternative layouts and in general *adaptive* interactions, a scoring system could be built into each module, allowing them to rate how well they are able to generate their content. This could then be used to generate the most optimal layouts and to avoid sacrificing too much content in situations where a module needs to propose an alternative layout. Rules about how and which content could be rearranged or excluded could be built into the modules allowing better reconstructions of layouts. Figure 8 illustrates how modules with different or equal priorities will affect each other's generation and which type of virtual world interaction will arise.

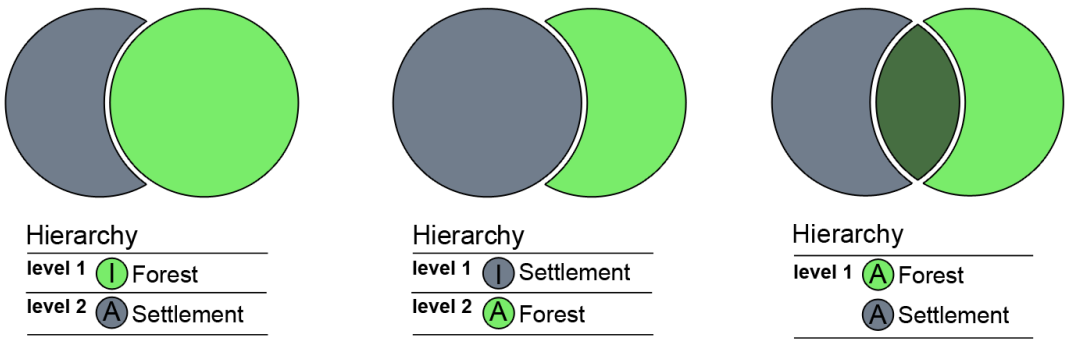


Figure 8: How hierarchical structure will affect generation and change which elements becomes Instructive (I) and Adaptive (A).

The *instructive* or *adaptive* interaction that arises between modules can be related to what Smelik, et al. [30] calls *feature interaction*, which in Sketchaworld occurs when two terrain features *claim* the same area. In Sketchaworld different priorities determine if *feature interaction* should be resolved through either cooperation, e.g. when a bridge is created over a river, or conflict, e.g. when a city overlaps a forest and the forest no longer have rights to occupy that area. In Modular PCG priorities is determined through a hierarchical structure and thus cooperation can be said to be what happens when two modules of equal priority adaptively generates content, and conflict can be said to occur when a higher priority module *claims* a part of the virtual world, thus restricting generation of lower priority modules.

In relation to *adaptive* interaction in general, modules has to interpret the virtual world on the structural level, as discussed earlier in this section, and determine how new content can be adapted to the existing content in the world. With physical content, such as houses, rivers, roads, etc., this is relatively straightforward, since these features can be represented with basic geometry, which modules can access through the virtual environment and avoid with methods such as pathfinding. Differently from physical content, it can be very complex to represent non-physical content on a structural level, making adaptation difficult. As stated in section 6.2, non-physical content should interact through a similar space as the physical, however, since non-physical content can be very diverse and represented in many ways, it is difficult to imagine a non-physical structural level capable of including all possible types of content. Therefore, a solution would be to have parts of the non-physical content linked and represented as physical content in the physical space. This should be possible, since non-physical content very rarely interacts with other non-physical content, and when it interacts with physical content, it is on a structural physical level. This is of cause a statement, which will have to be examined in a proof of concept illustrating how Modular PCG can be used in complex game generation (see Chapter 7).

To represent non-physical content as physical, new types of content might be needed and new data types might arise. On a physical structural level, to enable unknown modules to interpret the content, this might be represented as 3D content that block out occupied areas, but other modules might be designed to interpret this information and use it in their generation. This will consequently mean that the second module will be a sub-module of the first and the two will be linked. As an example, a story generator might generate a dummy NPC in the environment with some basic variables. This dummy NPC could then be regenerated by an NPC generator module, using the variables and possibly some designer input to create a detailed NPC. The other way around a NPC generation module would be able to generate a detailed NPC and a NPC dummy describing its features. A story module will thereafter use the dummy variables to generate a story including that NPC. This way the story and NPC module will form their own sub-system responsible of generating non-physical and physical content in the physical space, but in context of the architecture of Modular PCG they should be seen as one module. The advantage of this

approach is that parts of this sub-system could be interchanged, allowing designers to, for instance, use another NPC module to generate different types of NPCs using the same story. This structure will make some modules depend on one or more sub-modules utilizing what can be called *direct* interaction.

As discussed in this section, modules must represent their content in a way that sufficiently describes it such that other modules can access this information fast and efficiently through the virtual environment. Following the terminology from Sketchaworld, this level should be called the structural level. As described modules will be structured hierarchically and each should have a priority ID describing it place. When two modules try to generate content at the same spot in the virtual environment, the module with the highest priority is said to be *instructive*, and the lower priority module is *adaptive*; the higher priority module *claims* the area. Modules can be given the same priority ID, i.e. given equal priority, which will have the effect that both modules will be *adaptive* when generating content in the same area. When this happens, one module will *propose* an alternative layout to the other, which will either *counter* or *accept* the layout. When a *proposal* has been *accepted*, or a determined max has been reached, both modules will generate their content based on this layout. To optimise *adaptive* interaction, an individualised scoring system could be built into each module. As discussed modules should interact with the virtual environment on a structural level through the physical content. This should also apply to modules designed for generation of non-physical content, and therefore non-physical content must therefore be linked and represented as physical content. This might cause modules designed for non-physical content to be very complex and they could therefore be divided into smaller sub-modules each responsible for some of the generation. In the eyes of Modular PCG, a system of sub-modules would be viewed as one large module; however, within such a system *direct* interaction would be allowed and possible.

In short, there are two main types of interaction, which exists on the structural level between modules, *instructive* and *adaptive*, and one secondary type, which as such is outside the main architecture of Modular PCG, called *direct* interaction.

6.4 Designer Instructions

As mentioned previously, PCG has a long history and it can be argued that due to graphical limitation the differences between generated content and manually designed content was not significant in earlier examples, which meant that even simple PCG could be applied without sacrificing quality. Because of dedicated professionals in the industry, this has changed and most modern games have high graphical standards, complex and detailed level design and well-written stories, which makes it hard for PCG alone to meet the expectations of the audience. PCG

has many benefits, but it must not compromise the quality and should not oppress the imagination of human designers, which is why designer interaction with modules of Modular PCG is such an important issue.

In section 4.1, Sketchaworld [30] was used as an example of how modules could interact with the virtual world. In relation to designer instructions, the interface of Sketchaworld allows designers to direct the procedural generation by interacting with features on the specification level, the first of three levels of abstraction. This enables designers to sketch high-level terrain features and specify desired features, e.g. designers can draw a few points to represent a road. In Sketchaworld, this was made possible by having all tools integrated into the application, and designer could use these to sketch a desired layout in rough details in near real-time (see Figure 9). This illustrates how designer instructions can be designed, however, because modules in Modular PCG should be able to stand-alone, the tools would have to be integrated within the modules themselves.

Following the convincing results from the Sketchaworld application [30], input of equivalent complexity would be suitable for modules of Modular PCG. If more control were granted in the generation process, the generation might no longer be called procedural. However, it is believed that designers should have the possibility to adjust the generated outcome at abstraction level 3 (see section 6.4) to fine-tune and lock certain details of a generated level, which also was mentioned as a desirable feature by Smelik, et al. [30].

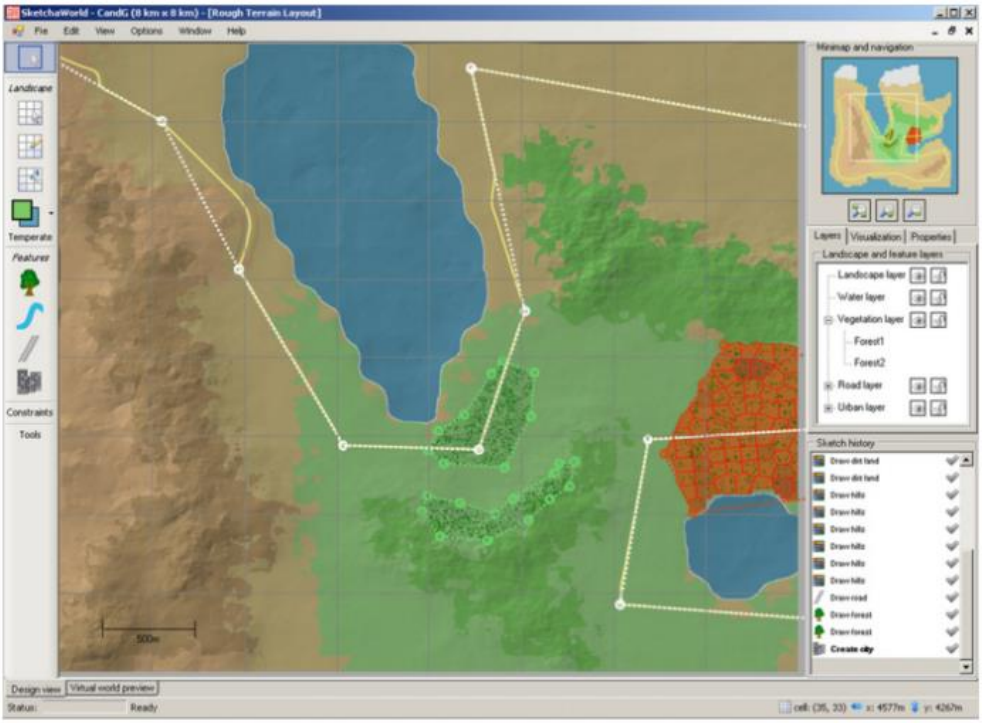


Figure 9: Interface of Sketchaworld showing editing tools for procedural sketching [30].

In the Sketchaworld implementation, designer actions are added to a queue and executed through their virtual world consistency managements system that ensures that features does not conflict with each other (see Figure 10). In Modular PCG, consistency management should be built into each module, ensuring that modules are able to react to unforeseen conflicts as described in section 6.3. To facilitate undo and redo functions the implementation of Sketchaworld keeps a history and manages the state of the random number generator, thus ensuring result are the same after undoing an action and regenerating the world. This implementation can therefore be seen as an important step towards the adaptation of PCG into the workflow of game designers, because it incorporates familiar actions such as undo and redo, and allows designer to sketch a desired layout visually in real-time. This makes the process more accessible than abstract declarations and coding-based examples found in other PCG implementations.

In short, creators of modules should design them to receive input at the specification level, and create embedded designer tools, which makes interaction possible and assessable for the users using familiar editorial options and design metaphors.

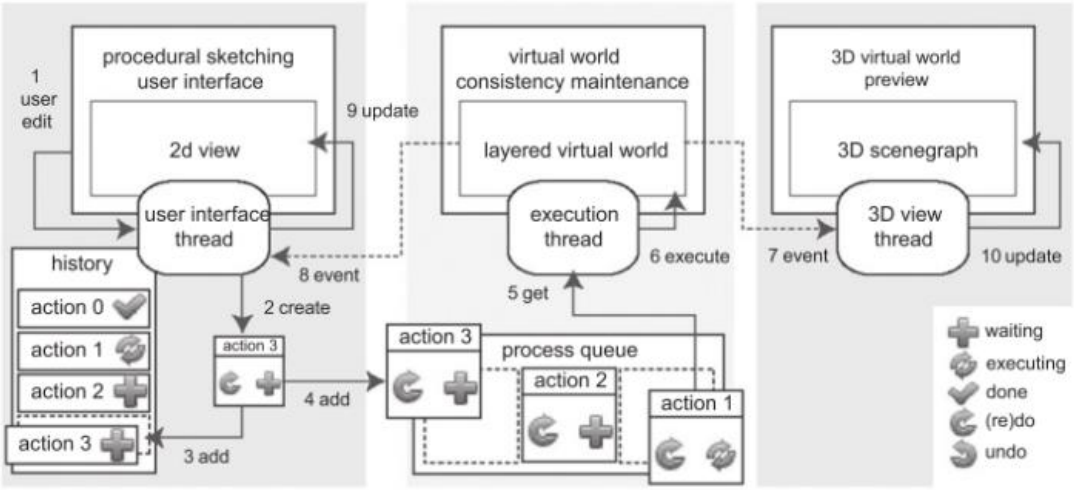


Figure 10: Diagram of the virtual world consistency managements system implemented in Sketchaworld [30].

6.5 Modules Providing Content to Players and Designers

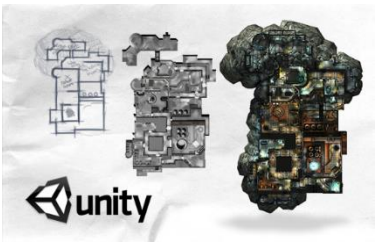
The purpose of Modular PCG is to provide accessible modular game development utilizing the powers of PCG. To make it accessible, designers needs to have to correct tools, and in connection to this, they need to be able to see what is being generated. As discussed in section 6.3 designers should be provided a preview of the generated content at abstraction level 2, the structural level. Designers could also be allowed to view the content at abstraction level 3, the object level (as

discussed in section 6.4), which represents the final state of generation and should be the level at which the players play and interact with the generated content. The structural level can be said to provide designer content, whereas the object level provide player content. One can imagine that the structural level needs fewer graphical details than the object level, but will still need the information necessary for designers to author additionally content with other modules. Since the structural level is also the level, at which other modules perceive the virtual world, it needs to be constructed of simple graphical elements; however, these elements could potentially contain some additional textual information available only to designers and maybe to some specific sub-modules if needed (see section 6.2). A good example of designer content, i.e. content available at the structural level, can be found in the application Sketchaworld (see section 6.4, Figure 9).

6.6 Modules Getting Input From Players

One might think that player input is something that belongs in the domain of Experience Driven PCG (see section 2.1.2) to heighten the experience for the player. However, in Modular PCG player input should be seen as an important basic functionality that allows modules to change the state of the virtual environment upon player request. What this means, is that even the simplest interactions should be seen as a form of player interaction with a module, meaning that for instance, the character controller, which normally would be built into the game engine, can be seen as a single module on its own. This is already the case in the game engine Unity, where developers can drag and drop an already created third person player controller into the scene without having to modifying it (see Figure 11). Some people might argue that a character controller does not include any PCG, and it therefore cannot be considered a module for Modular PCG. However, the movements of the character have to adapt to the terrain and obstacles, and it can therefore be argued that it is procedurally adapting to the environment. As such, a drag and drop character controller meets the requirements for being a self-contained standalone unit, and one could enhance the procedural capabilities to include for instance procedural animation.

Unity



Game development software featuring rendering engine, intuitive tools and easy multiplatform publishing, with a large online asset library.

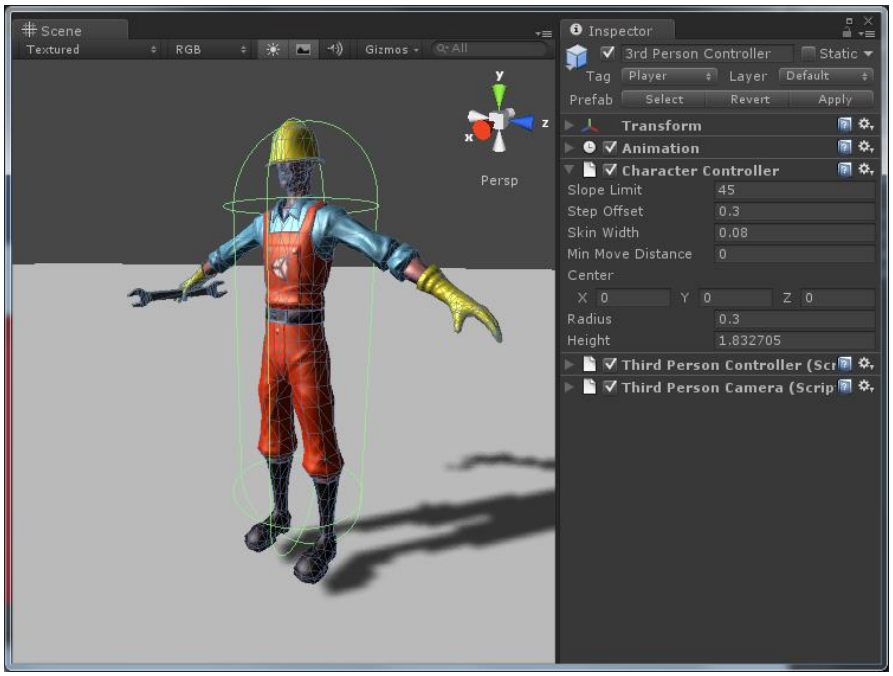


Figure 11: Third person character controller in Unity.

Even though player interaction should be seen as a basic functionality, it is only applicable for some modules and should not be included in every module. In many cases, player interaction would be unnecessary and illogical. If one were to design a complex game, most if not all of the environment for instance would be static and there would be no need for player input. That said it would be possible to imagine a game where player input has an effect on the environment, and such a module could be created.

As described in section 2.1.2 about Experience Driven PCG, player input can be gathered through different methods. According to Yannakakis & Togelius [2] methods can be *subjective*, *objective*, or *gameplay-based*, and under each category lies several other methods. A subjective method can for instance be based on *free-response* or *forced* data, and objective methods often include methods such as electrocardiography (ECG), galvanic skin response (GSR), and electroencephalography (EEG).

6.7 Final Architecture

After the general architecture of Modular PCG proposed in section 6.1 has been dissected and analysed throughout the previous sections, this section will summarize and rebuilt the architecture based on what has been discussed.

Initially seven types of interaction was found, which were able to describe the relations between modules, designers and players. These interactions were meant to describe the structure of the architecture and illustrate how a Modular PCG system could be built. Initially two types of architecture were discussed, namely top-down and bottom-up, and modules were divided into high- and low-level. Through analysis, however, it was discovered that this division were more suitable to describe the intentions of the modules, i.e. whether they were designed for high- or low-level content, e.g. city structure or individual buildings, and both top-down and bottom-up structure could be achieved by the arrangement of the modules in a hierarchical structure. This meant that the number of interactions were limited from seven to five. Furthermore, it has been decided that modules should not be dependent on each other, and that each modules must be able to stand alone, facilitating a modular approach where designers and developers are able to apply modules without considering existing modules. In other words, modules must be self-contained and able to adjust to the virtual environment, meaning modules should not interact directly with other modules, which was why internal interaction was removed from the list as well. However, by allowing the creation of sub-modules, i.e. smaller pieces of modules that together form a single module, the term internal interaction, changed to *direct* interaction, can be used to describe the interaction between these. It is important to remember that in the view of Modular PCG a system of sub-modules would be categorised as one single module.

Following this, the list of interactions was reduced from seven to four main types. For easier referencing, the two unchanged types will be renamed, thus "*modules providing content to players and designers*" will be called "*module output*" and "*modules getting input from players*" will be referred to as "*player input*". To sum up the architecture of Modular PCG will be built around the following four types of interaction:

1. Virtual world interaction
2. Designer instructions
3. Module output
4. Player input

To illustrate these interactions and the architecture in general, the model shown in Figure 12 has been created.

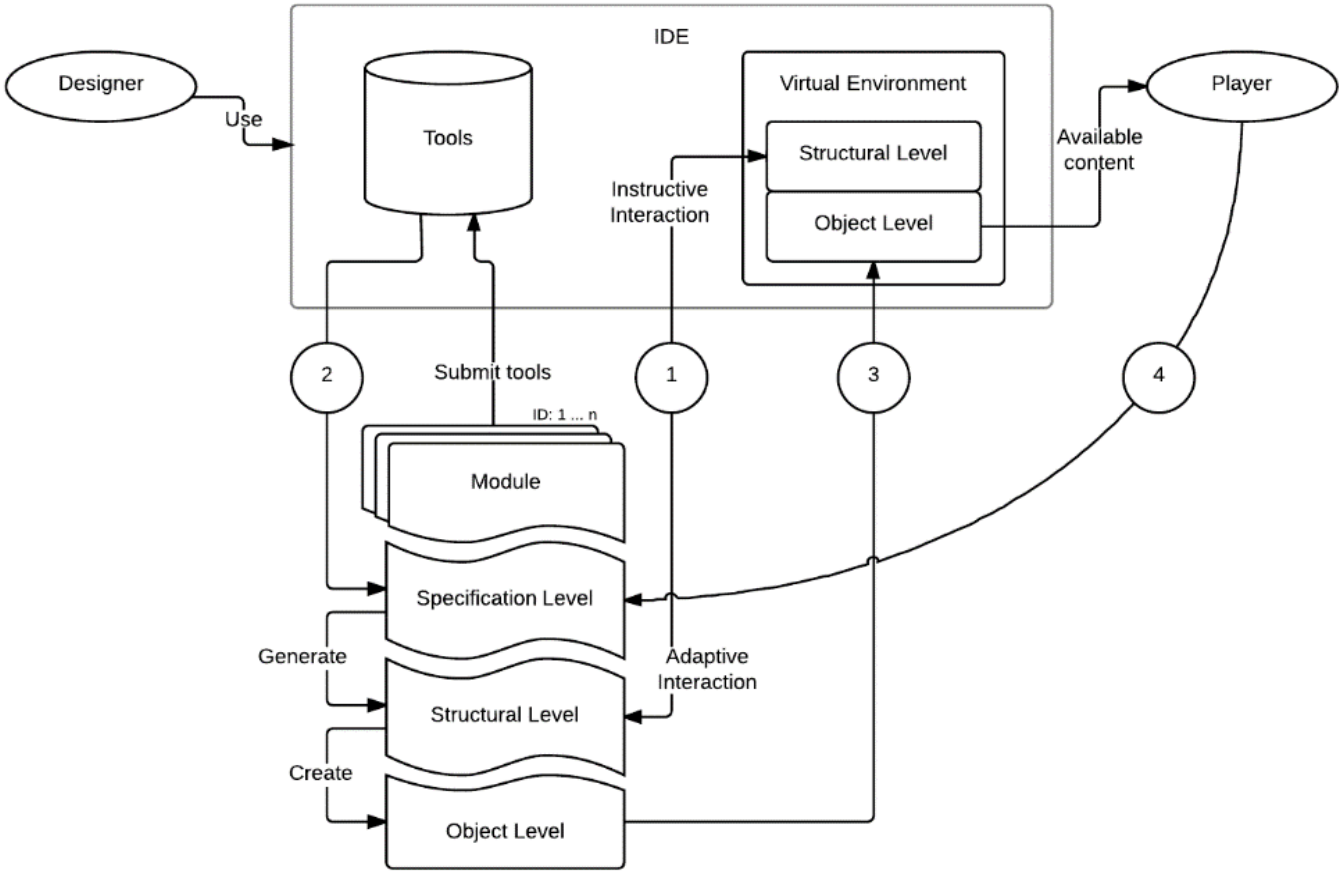


Figure 12: The architecture for Modular PCG. The numbers 1-4, refer to the four types of interaction: 1. Virtual world interaction, 2. Designer instructions, 3. Module output, 4. Player input.

In section 6.3, virtual world interaction was divided into two main interactions, one going from the module to the virtual world, called *instructive* interaction, and the other vice versa, called *adaptive* interaction. Whether a module will be *instructive* or *adaptive*, will be decided by their place in the hierarchical structure, determined by their priority ID, which is an ID each module should be given. Modules can be given the same ID to force a merge of two types of content, but otherwise the module with the highest priority will always be *instructive* and modules with a lower priority will be *adaptive*. In the architecture in Figure 12 connections are only drawn to one module, however, since modules are independent they all have the same connection to the virtual environment, and therefore this one module represents all modules in the hierarchy with priority from 1 to n.

Designer instructions is how designers influence the generation and author the content, and since modules should be self-contained, the tools needed for authoring and designing must be embedded in the modules themselves. Using familiar design metaphors and conventional layouts and tools is highly recommended since this will help designers relate and make the process accessible. As one can see in Figure 12 modules submit tools to the IDE, which designers interact with and thereby instruct modules. Inspired by the literature, modules can be described

through three levels of abstraction, the specification, structural and object level, each representing a different level of detail. On the specification level, modules are able to accept designer instructions, which they will try to accommodate. On the structural level, modules will interact with the virtual world through *instructive* and/or *adaptive* interaction. At this level, content should be represented in rough details with enough information such that other modules are able to interact with it, and since this level provides feedback the designers, content should be represented in an intuitive way that enables designers to make sense of the generation. The object level represent the final state of generation and is the level at which the end-user perceive the content, however, designers could also be allowed to view content at this level to make final adjustments before the product is shipped. These levels of abstractions thus provides a skeleton for the internal architecture of each module, and some can be seen as design guidelines for the creators of modules to follow.

Finally, some modules might need player input, which is the last type of interaction in the architecture of Modular PCG. Player input can be direct or indirect, but is not always needed and there are no specific rules as to how it should be included in the module structure; however, one can follow the methods described by Yannakakis & Togelius [2] in relation to Experience Driven PCG. As illustrated in Figure 12 player input goes directly to the specification level of the modules, however, this communication should to some degree be linked to the virtual environment, since it is through this the players are presented with the content of the modules. The reason why this connection, in the architecture in Figure 12, is drawn as a direct link is that player input should ultimately affect the specification level, thus forcing modules to regenerate some content on the structural level and through *instructive* or *adaptive* interaction change the object level and the content, which is presented to the player.

Following the discussion, some general requirements for modules can be established.

- Modules need to have a priority ID identifying their place in the hierarchy
- The necessary tools needs to be implemented in each module, making them accessible in the integrated development environment (IDE) in which the module is applied
- Tools needs to integrate with the specification level of the module
- Designers and other modules must be able to make sense of a simplified generation at the structural level
- Modules must be able to generate detailed content at the object level

CHAPTER 7
EVALUATION OF MODULAR PCG

Modular PCG has until now been discussed on a conceptual level, making it difficult to see any real world applications of the concept. This chapter will provide an example of how Modular PCG can be applied in complex game development, and illustrate how it should be used to facilitate human designers and developers.

In Chapter 5, it was decided to apply Modular PCG in the context of complex game generation and more specifically generation of games where the quest and map structure are interconnected in a unified experience. This section will therefore illustrate how Modular PCG can be used to create a complex game, and how the architecture described in section 6.7 will make content generation accessible to designers and developers. Since it has not yet been determined what exactly is meant by complex game, other than an experience driven by a closely connected quest and map structure, this first needs to be established. When thinking about games that rely greatly on both quest and map structure, primarily two genres comes to mind, namely the action adventure and the roleplaying genre. Since it should be possible to create any type of game using Modular PCG, the selection comes down to preferences and what can be

illustrated within the timeframe of this project. On one hand, roleplaying games (RPGs) generally have focus on multiple quests, but often has a monotonous map design. One the other hand, action adventure games usually focus on just one main story quest, however, the level design often seems closer connected to the story. According to Dormans [33] the action adventure genre has the added benefit of supporting a more varied gameplay and giving a greater sense of purpose than RPGs, because they rely on more on well-designed levels to create enjoyable exploration, flow and narrative structure. For these reasons, the purpose of this chapter is to illustrate how Modular PCG can be used to create an action adventure game.

It is important to note that the goal is not to implement a fully functioning game with complex gameplay, but to illustrate which modules would be required and how these should be connected and used by designers. This chapter should therefore be seen as a proof of concept illustrating the different aspect of Modular PCG. The proof of concept will provide some initial design specifications for the specific modules needed for creating a complex game with an elaborate quest and map structure.

7.1 METHOD

As described Chapter 7 will provide a concrete example of how Modular PCG could be applied in game development. The purpose of this is to evaluate the usability of the concept and this section will describe the specific method used for evaluating the Modular PCG architecture.

First, since the Modular PCG architecture can be seen as a software architecture, it is possible to apply methods for evaluating computer software. Within software engineering, there exists several techniques for evaluating a software architecture in relation to quality, e.g. usability, maintainability and performance. When evaluating a software architecture, the purpose is to identify risks and ensure that the requirements has been addressed [36]. Of cause, there are some fundamental differences between a software architecture and the architecture of Modular PCG. This means there are some classical quality attributes that cannot be addressed. However, it should be possible to illustrate the Modifiability, i.e. how easy it is to create new modules, Availability, i.e. what it would cost in person-hours to create new modules, Performance, i.e. the speed of generation, and Usability, i.e. how easy it is for users to use module for content creation. These attributes are normally considered in software engineering, and it would be beneficial to keep these in mind when evaluating the architecture of Modular PCG.

Experience-based evaluation is another software evaluation method, which could be applicable for evaluating the architecture of Modular PCG. In this method the developers of the architecture, or consultants, validate the architecture based on previous experience and domain knowledge [37]. Validating a system solely using this method might result in an architecture that only the

developers will see as logical. It is therefore very important to think objectively when evaluating an architecture based in own experience. That said, because of past experience with game design, it should be possible to validate the architecture of Modular PCG using personal experience.

Methods outside of the software engineering domain can also be applied to evaluate the architecture of Modular PCG, and as mentioned in the introduction the evaluation chapter should be seen as a proof of concept. In short, the purpose of a proof of concept is to demonstrate the application of a given theory, model or architecture. This can be done through smaller tests or smaller implementations.

Conclusively, this project will validate Modular PCG by creating a proof of concept illustrating its applicability by describing the architecture in relation to a game development scenario using personal game design experience to validate the necessary interactions and tools. The proof of concept will illustrate how modules should be created, structured and how they could be integrated into a game development software. Because modules will not be implemented, it is unfortunately not possible to demonstrate exactly how modules will generate content and how they will perform. However, the necessary tools and the outcome of the modules will be illustrated through rough mock-ups.

7.2 Game Concept

Because the aim of the evaluation is to give a practical example of how Modular PCG can be applied in game development, this section will describe an action adventure game concept, which will give design requirements for a future implementation. These requirements will determine which modules is needed and how they should be designed. However, before discussing the game concept, this section will describe the typical game development process, to establish where in this process Modular PCG should be used.

In short, a typical game development process can be divided into four stages: Specification and planning, pre-production, production and finally validating and testing. When starting on development of a new game, the process starts with a short description of the proposed game including target group, plat-form, genre, references and a draft of the planning, which is used to validate if the proposed game concept is viable [38]. The pre-production phase is used to create prototypes and general game design. In this phase, most of the major design decisions are taken and usually game developers start making a game design document (GDD) to document the design and the GDD is used throughout the development process to catalogue and organise all elements of the game. There are no right or wrong way of writing a GDD and normally developers will use a style that matches their process and preferences: "*Each game designer usually finds*

what works best for them" [39]. In the production phase, the individual elements are created and pieced together based on the design documented in the GDD. The last phase of development is the validation and testing, where alpha and beta tests conducted and changes are made to the design and implementation based on the test data.

As described throughout the report, the goal of Modular PCG is to make PCG more assessable to designers and developers. In general, PCG is used to ease the implementation by applying procedural methods, but it can also be used as a creative tool during the design phase. Regarding generation methods and applications, Modular PCG is not very different and thus it can be used as a creative prototyping tool during the pre-production stage; however, the main purpose of Modular PCG is to ease the implementation process and combine the strengths of PCG with the controllability of manual content creation.

The proof of concept will therefore illustrate the use of Modular PCG in a production and implementation context. Because the production phase are dependent on the design created in the pre-production, a rough game design has to be established before it can be discussed how Modular PCG can be used in a possible implementation. However, because the pre-production phase can be very time-consuming, the initial design for the proof of concept will be taken directly from an existing GDD describing an imaginary game (see Appendix II). The game described in the GDD is an action adventure game set in the ancient Egypt, where the player, incarnated as the biblical character Moses, fight and quest his way through the Egyptian lands using godly powers to liberate his people from the oppression of the Egyptians. The GDD is not completely exhaustive; however, it does provide the overall gameplay and lists some environments, objects and NPCs, from which the initial modules can be created.

With the basic game-design established, it is now possible to introduce the individual modules that is required. However, before doing this, it is important to discuss how Modular PCG should be integrated within the development environment that is used. Thus, the next section will describe one approach for integrating Modular PCG into one of the popular game engines.

7.3 Module Integration

For this project, it has been decided to describe how Modular PCG could be integrated within CRYENGINE free SDK (CryEngine3). CryEngine3 has been chosen because of the authors previous experience with the engine, and because it features many high quality assets that can be used to illustrate the generation process. Even though CryEngine3 will be used as the example, it should be possible to integrate modules in other game engines in ways similar to what will be described in this section.

To make the integration of Modular PCG logical, it should follow the existing methods for making particles, prefabs, and many other entities available in CryEngine3. Before for instance a particle effect can be applied in the environment, it must be imported into the database, which essentially is a collection of different premade entities, in which some global variables can be adjusted. In a similar way, modules could be imported into the database and designers could adjust some basic properties for each as illustrated in Figure 13. Importing the modules will not affect the virtual environment, but it will make them usable and enable designers to apply them at a later stage.

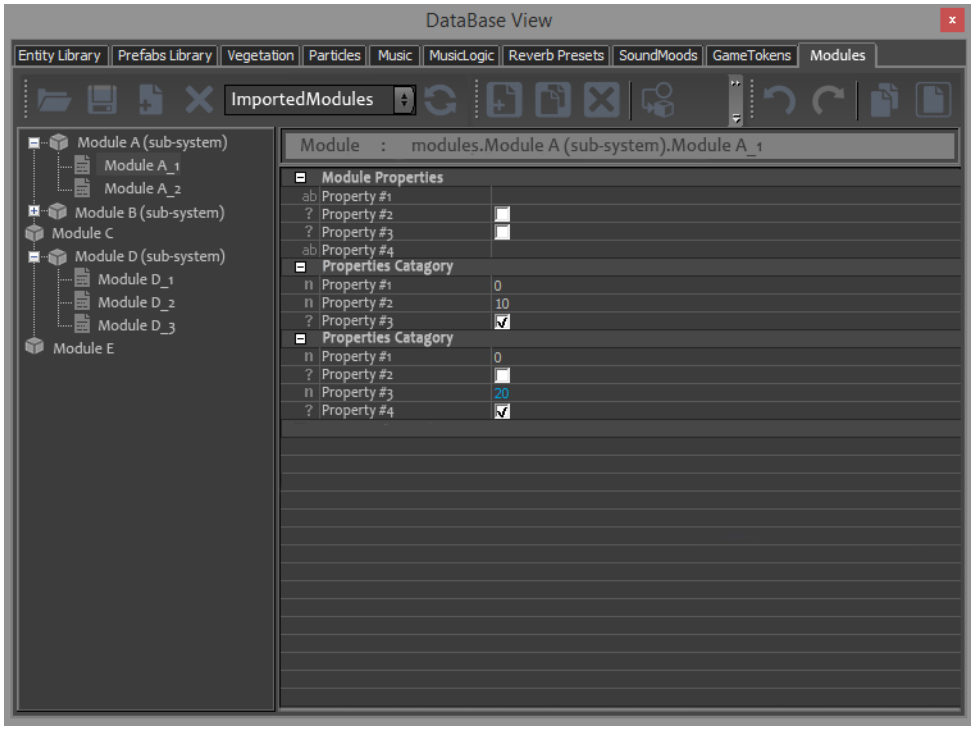


Figure 13: The original DataBase View from CryEngine3 with the added tab 'Modules' open to illustrate how basic properties could be adjusted.

After modules has been made available, designers should be able to use them in the virtual environment. To maintain an easy overview of the implemented modules and to facilitate the creation of a hierarchical structure as described in section 6.2, a separate editor similar to the Layer Editor could be created, in which modules could be added, removed and organised. In CryEngine3, designers use the Layer Editor to create and manage different layers, which are used to organise all objects that are created in the virtual environment. The layers can therefore be seen as the folder structure on your computer.

As said one could imagine an editor where modules could be organised and selected for editing, which could be called the Hierarchy Editor. The Hierarchy Editor should enable designers to assign priorities to the different modules, which could be done by organising the added modules in a folder-like structure as illustrated in Figure 14.

Figure 14: The proposed Hierarchy Editor for adding, removing and organising modules. The three eye symbols in each line represent the visibility of the specification, structural and object level for each module and/or hierarchical level.

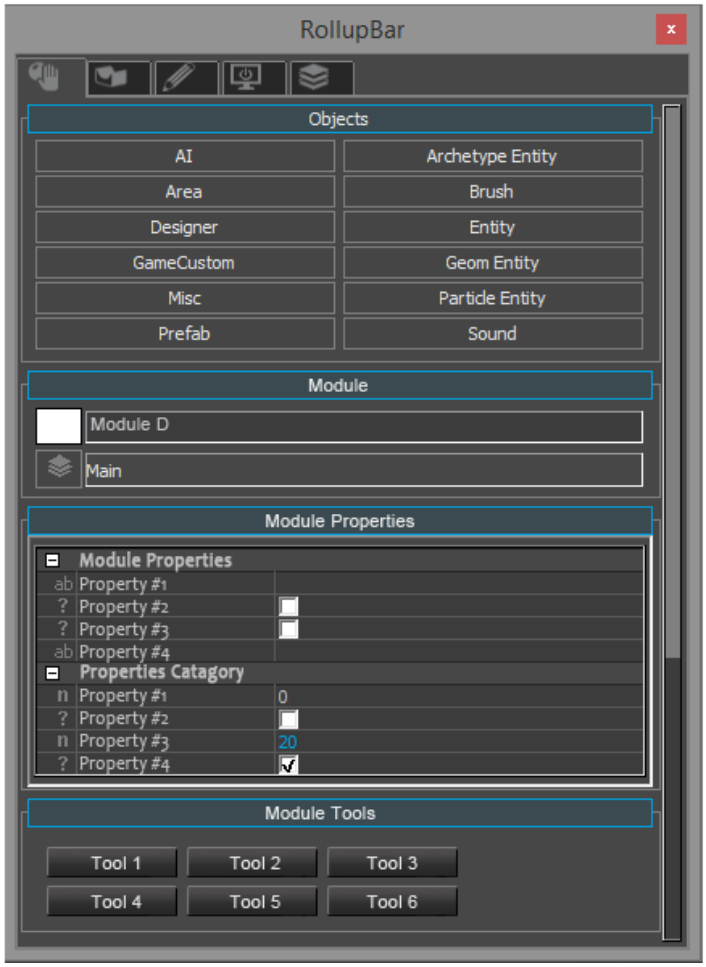


Figure 15: When selecting a module from the Hierarchy Editor its tools and properties will become available in CryEngine3's RollupBar. The RollupBar is where all details and properties of objects that has been selected in the virtual environment can be found.

As described only modules imported into the database can be added to the hierarchy and when selected in the editor, their tools and properties should become available as seen in Figure 15.

In CryEngine3, when designers select an object in the virtual environment, its properties will become available, and it might be best to maintain this interaction, such that it will be possible for designer to select objects that has been created by modules. However, instead of showing the normal properties of the selected object, designers should be informed that it has been procedurally generated and that changing them might affect the procedural generation. To avoid confusion and to maintain the usual interaction with objects, procedurally generated objects could be locked, i.e. designers may not change any parameters. As discussed in section 6.4 about designer instructions, designers should have the ability to fine-tune and lock the procedurally generated objects, i.e. avoid regeneration of the objects. This could be achieved by giving the designers the ability to free individual objects from their parent module. This functionality should remove the object from parent module and add it as a normal static object with the usual properties familiar to designers (see Figure 16).

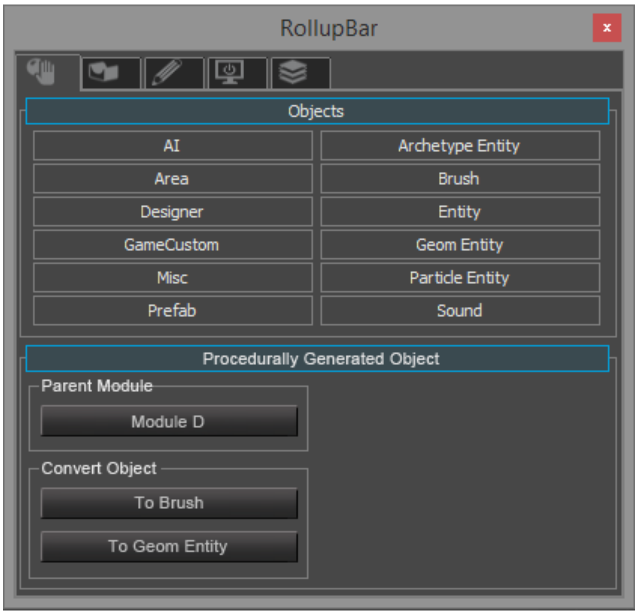


Figure 16: When selecting a procedural generated object in the virtual environment, the designer should be informed about it and allowed to free the object from the module, by replacing it with a copy with the same options as a manually created object.

If one wished to implement Modular PCG into a game engine in the future, this section has provided the initial ideas and specifications. It has been described how Modular PCG could be integrated into CryEngine3, and it has been described how designers should interact with the modules and the generated content in general. The proof of concept has yet to describe which modules are needed for generating the action adventure game concept described earlier, which tools should be implemented and how these modules should be organised and used. The next section will therefore describe some initial modules that can be used for level generation based on what is documented in the GDD in Appendix II.

7.4 Level Design Modules

As said, the purpose of the proof of concept is not to implement any modules, but to give an overview of which modules would be needed for complex game generation and how these modules should be structured and applied in the same game scenario. It is the purpose to illustrate how these modules will form a common architecture and how designer interactions could be facilitated.

Thus, this section will establish which initial modules is needed for creating content for the action adventure game described in the GDD in Appendix II. Because the GDD only describes the initial part of the game and not individual quests and bigger areas, the purpose of this section is to describe a few smaller modules that could be applied throughout the game. In the GDD, it is described that the environment will consist of smaller open linearly connected deserts and caverns, populated by ancient Egyptian scenery such as ruins, markets and streets. It is described that main aspect of the game should be puzzle solving, but that this should be complimented by a fighting (action) aspect, in which the player can fight off vermin, e.g. bats, rats, scarab beetles and jackals, and bigger enemies such as Egyptian guards. Finally, in the GDD the layout and objects of the first level is described (see Figure 17).

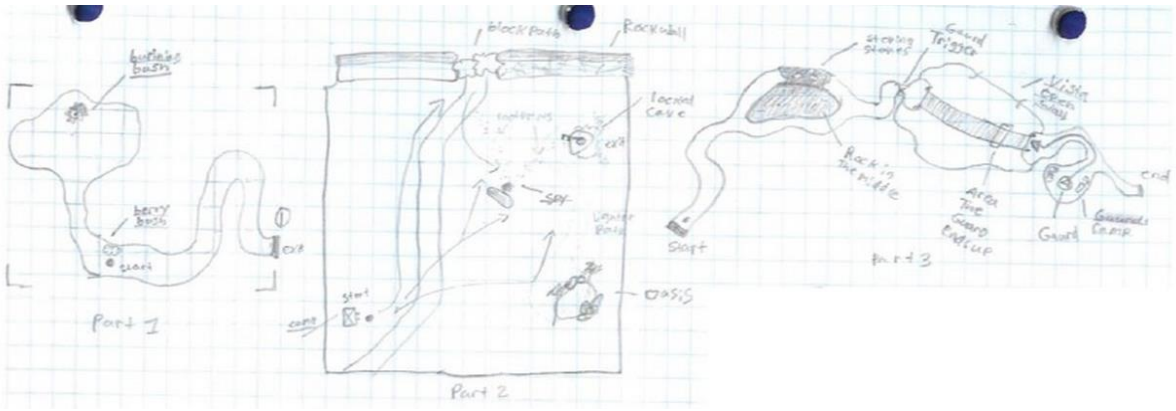


Figure 17: Overview of one level described in the GDD in Appendix II.

With the basic information about scenery and level layout, it is possible to create a list of possible modules that could be useable to create the described game and layout in Figure 17 (See Table 2). Since it would require some work to create each individual module, it is pointless to create modules that will only be used a few time throughout the production phase. In other words, it would be a waste of time to create a module with a specific purpose, if it would take half the time to create the content manually. Therefore, only modules that can be considered reusable has been included in the list of possible modules.

Name	Purpose
Enclosed Desert Area	Module for creating desert pathways and desert areas surrounded by a cliff face. Used to restrict free roaming.
Oasis	Module for generating a desert oasis.
Small Desert Objects	Populate an area with scattered rocks, bushes or grass patches.
Desert Ruin	Generate desert ruins of all sizes.
Desert Path	Module for generating a desert path.
Cavern Entrance	Generate a cavern entrance in a vertical wall or on flat ground.
Cavern	Module for creating an enclosed cavern.

Table 2: List of possible modules, usable for creating the action adventure game described in the GDD.

To illustrate the practicality of the suggested modules in Table 2, let us consider the layout from Figure 17. In the first part, one could use the Enclosed Desert Area module to specify the walkable area and restrict the player from walking off the level. This part of the level could also be populated with a couple of ruins (Desert Ruin module), which will help convey the right atmosphere and lastly the Small Desert Objects and Desert Path modules could be used to add details to the area.

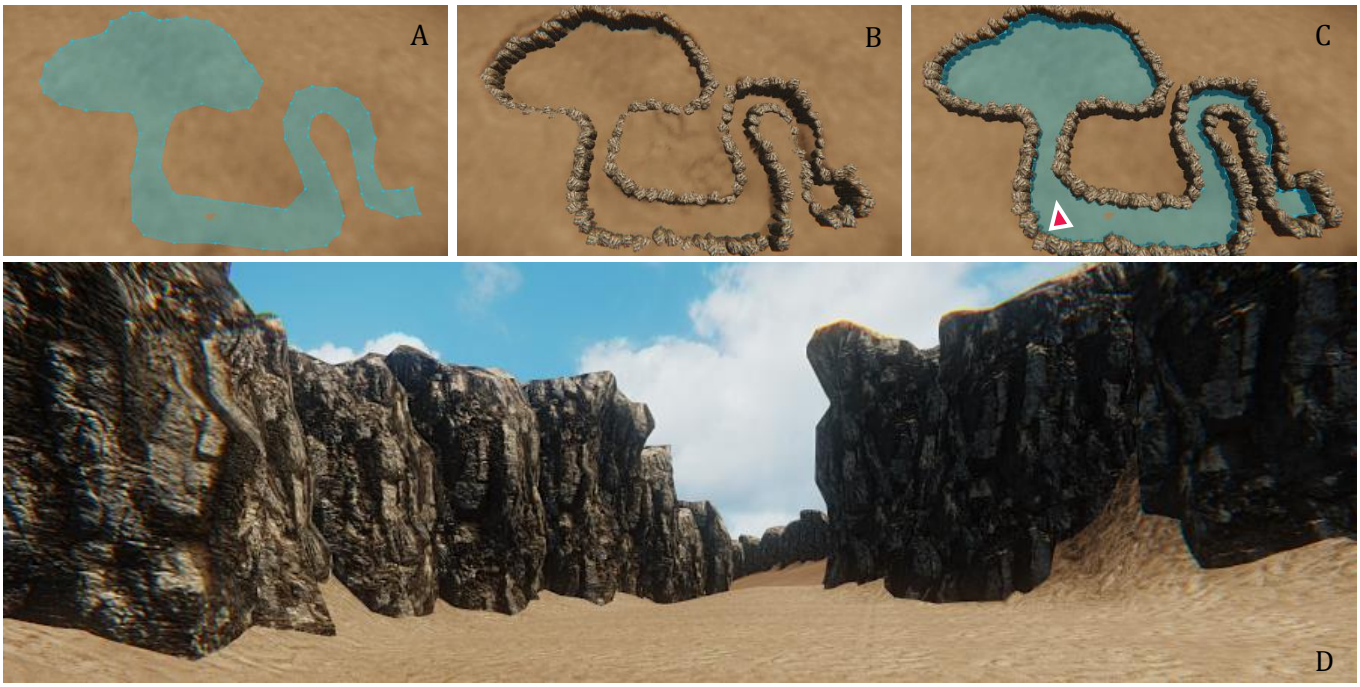


Figure 18: The workflow of the Enclosed Desert Area module. A: Specification of an area. B: The area is encapsulated with cliffs. C: After use. D: Screenshot from the player's perspective from the point illustrated with a triangle in C.

The Enclosed Desert Area module could have a tool for drawing a shape that specifies the area in which the player can move (Figure 18 A). This area will be surrounded with a large cliff face (Figure 18 B), and the terrain will be raised to fit the cliffs (Figure 18 C). The example has been created with little attention to detail and it is very rough, but it illustrates the workflow and generation of this module. Other specifications might include the height of the cliffs and the appearance. To avoid compromising the performance, the module should not be allowed to generate many new objects, as this will increase the number of drawcalls in the environment and thereby affect the performance negatively. Designers could be given an option to adjust the amount of new objects that can be generated, and alternatively, to increase controllability, designers could be allowed to specify which objects should be used for the generation of the cliff face. These options will steadily increase the controllability, and give the designer exactly the amount of control they wish and need. As specified in section 6.3, each module should also be able to adapt to the virtual environment on a structural level. In the case of the Enclosed Desert Area module, its goal would be to create an area that the player are not able to escape. This means that if there already were an object on the edge of the specified area that blocks the player, it would not be necessary for the module to generate content in the area occupied by the object (see Figure 19).

As said Figure 19 illustrates how modules adapt to existing content in the virtual environment, and as described in section 6.3, this adaptation is possible because the modules interact with virtual environment on the structural level, which is a low-resolution representation of all content in the environment. Because of the physics system in CryEngine3, all objects have a low-poly model of themselves attached, which is used for physics collisions and hidden when the game is running. In the view of Modular PCG, this low-poly model can be seen as the structural level of the object (see Figure 20).

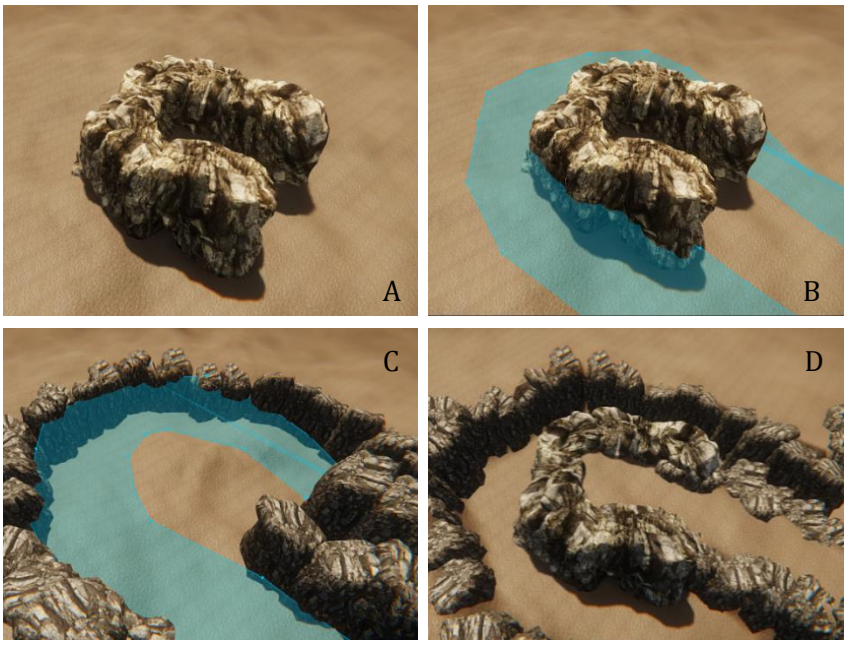


Figure 19: How a module will adapt its content to existing content in the environment. A: Existing content. B: Specification for module generation. C: Content generated by module. D: Generated and manually created content form the final layout.

After the basic layout has been created using the Enclosed Desert Area module, one could use the Desert Ruin module to create some scenery for the player to explore. The purpose of this module would be to create a detailed decorative ruin, which could be used as container for other game elements. As with the Enclosed Desert Area module, designers should be allowed the necessary freedom to create exactly what they want and specify the details they want. First, designers could be allowed to draw a 2D area on the ground, indicating the footprint of the ruin, and for extra control, they should be able to specify the basic 3D shape of the ruin. Alternatively, they might also want to import a shape from another application, instead of using the drawing tools built into the module. After specifying or importing either a 2D area or a 3D shape, the generation should generate a ruin that fit within this area (see Figure 22).

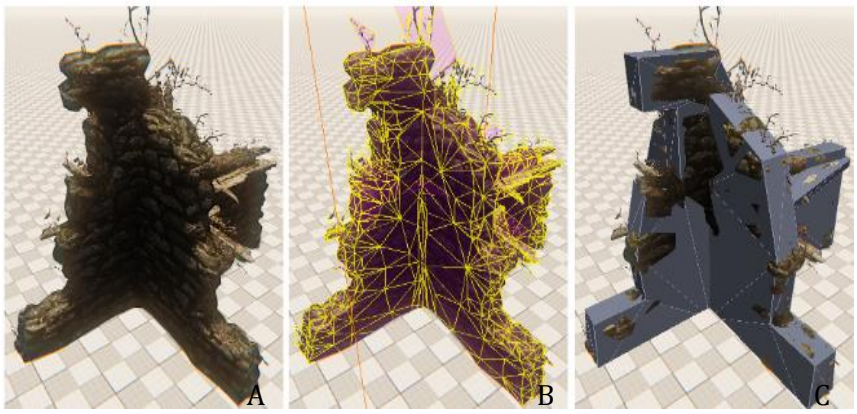


Figure 20: Illustration of the structural level in relation to the object level. A: The object as seen by the player. B: Object level with wireframe. C: Structural level on top of object level.

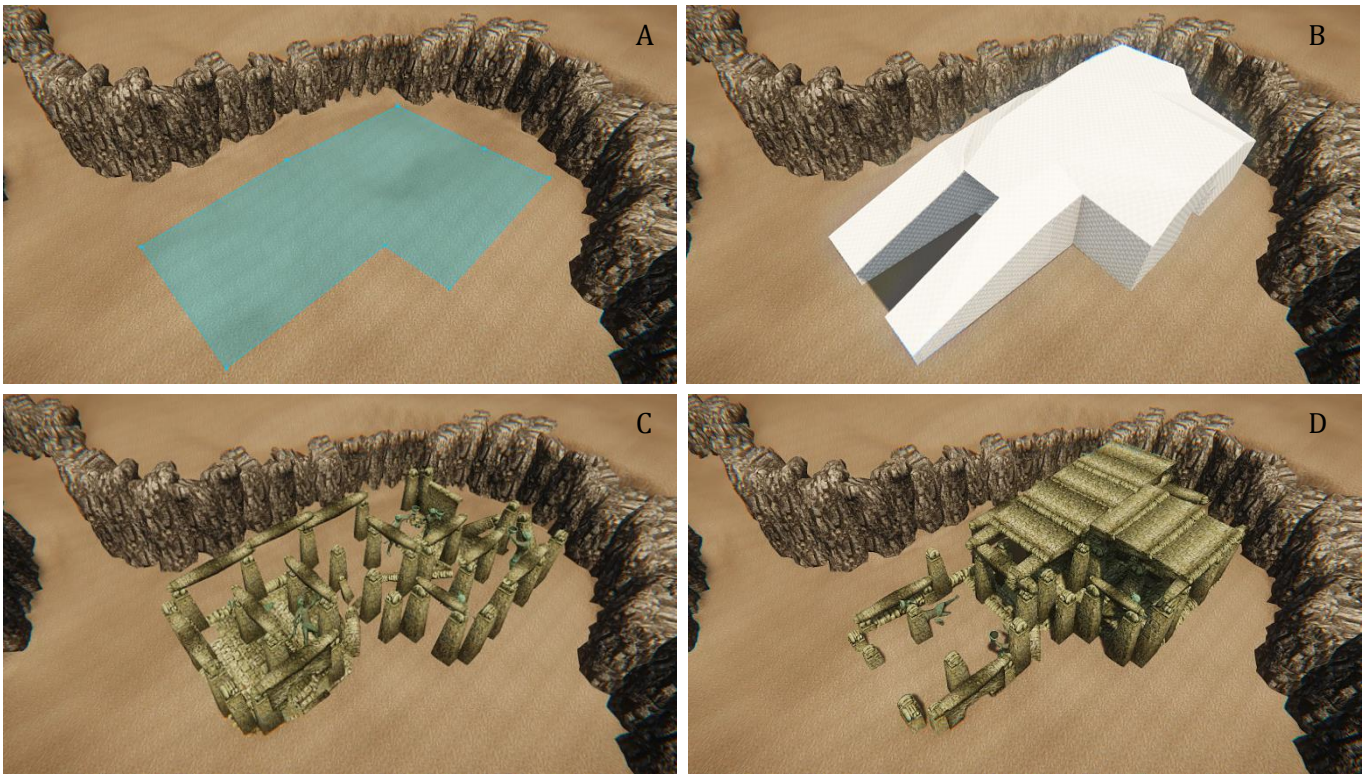


Figure 22: Ruin generation with different specifications. A: 2D shape specification. B: 3D model specification. C: Generated result based on 2D shape. D: Generated result based on 3D model.

As illustrated, the generated result will be different depending on the specification method. Additional specifications could include a density parameter, which could be used to specify the percentage of the area or model that will be filled with content (see Figure 21). Another specification could be the age, or deterioration, of the ruin, which could be used to give the building a worn or destroyed look. As with the Enclosed Desert Area module, the generation should be able to adapt to the existing scenery in the environment and designers should be able to specify either how many new objects should be created or which existing objects should be used for the generation.

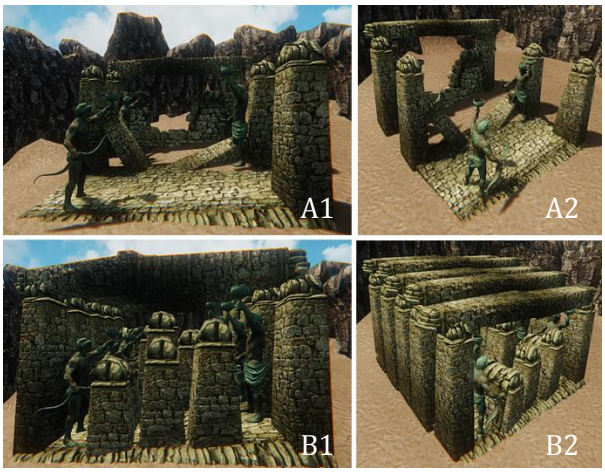


Figure 21: Example of density setting for ruin generation. A: Low density. B: High density.

The last modules that will be discussed in relation to the first part of the level layout from Figure 17 is the Small Desert Objects and Desert Path modules which as described should be used to add extra details to the area. With the Desert Path module, designers should be able to create paths of different sizes fast and efficiently. One way

to facilitate this could be implement a pathfinding algorithm that enables the module to find the best place for a path between two points in the virtual environment. A designer might for instance specify that a path should run from the larger open area to the end of the level at the right. The module would then create a line for the path to follow, and the designer could be allowed to fine-tune this path before the module would generate the path on the object level (see Figure 23).

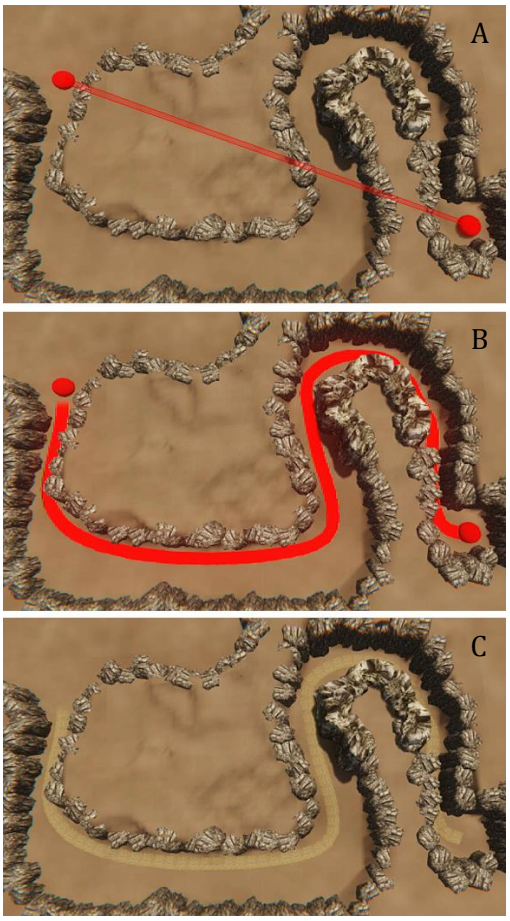


Figure 23: Desert Path module. A: Specification of two points. B: Pathfinding between the two points. C: Generated path on object level.

The last module, the Small Desert Objects module, should be use last to fill the environment with smaller objects such as bushes, rocks and patches of grass. In CryEngine3 designer can already use the Vegetation Editor for this, and it is possible to specify which objects should be procedurally placed on different terrain layers. Designers can, for instance, have two terrain layers with a grass texture and specify that rock and grass objects should be procedurally placed on one of them. With this setup, designer are able to paint areas with objects and areas without, and thereby create variations in the environment. However, this technique is not very flexible and it can be tedious to make changes. In addition, it is not possible for designers to specify different densities, meaning that you have are limited to one fixed density for each object. To overcome this limitation the Small Desert Objects module should enable designers to draw different density-maps for each object, thereby allowing greater control over the generation. A density-map should be single colour overlay that with different transparencies represents different densities. The module should enable designer to import different objects and link them to different density-maps. When an object has been imported and linked, the module should distribute copies of the object based on the density-map (see Figure 24).

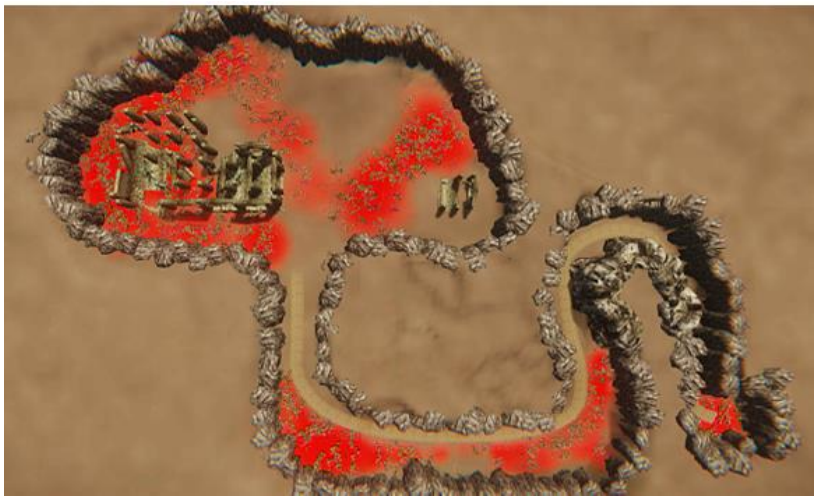


Figure 24: The Small Desert Objects module can be used to distribute rocks, grass and bushes across the entire area. The darker the colour, the closer the objects will be placed together.

This section has now illustrated the use of Modular PCG in a game development context and given concrete examples of a few modules. Different designer tools has been discussed in relation to each module, and their use and application has been explained through examples. Because the game for which the modules has been created has not been implemented before, it is uncertain whether the resulting level (see Figure 25) was what its original designer had in mind when writing the GDD. Nevertheless, the purpose of this section has been to illustrate the usability and accessibility of Modular PCG. To ensure that the modules suggested was logical, they have been created from a design point of view, and it has been the intention to structure the modules in a way that would make sense to a game designer.

Figure 25: Final game generated exclusively using Modular PCG.

As described previously it has been the intention to investigate how Modular PCG could be used to generate a complex game with a well-connected quest and map structure. Because the GDD does not describe the quest structure of the game, it has not been the focus of this section. Furthermore, as said in the beginning of this chapter, modules should be reusable and the time spent creating the individual modules should be made up for by applying them in the production phase. To limit the extent of this project, and because no concrete quest specifications are provided in the GDD, this report will not describe a specific quest generation module which could be applied in the game. The next section will, however, describe the issues with procedural quest generation in general and provide some initial considerations for designers and creators of modules to remember in the future.

7.5 Quest Module

This section will investigate the issues with procedural quest generation and provide some initial considerations to remember if one were to create a quest module. Whenever possible, the game from the GDD in Appendix II will be used as a practical example to describe how a possible quest module should be applied in game production, however, the purpose is not to provide requirements for creating a quest module specifically for the described game.

Before discussing the actual quest module, a definition of a quest has to be established. Dormans [33] defines a quest (what he calls mission), as a series of tasks that keeps the player occupied and provides concrete goals. Similarly, Doran & Parberry [35] defines a quest as a task that includes a challenge and a reward. Ashmore & Nitsche [34] has a more stringent view on quests, and state that it is a way to structure play in a virtual environment. They state that a quest has a space, a challenge, a goal and a setting in which it takes place, and that quests can facilitate personal growth (such as levelling) and spatial expansion (such as exploration and spatial progression). In relation to the GDD, the space would be the levels (or map structure), the challenge would be the individual puzzles and enemies, the goal would be to liberate people from the oppression of the Egyptians, and finally the setting would be ancient Egypt. In relation to space and setting, Dormans [33] states that the quest structure can be independent from the map structure (what he calls space, defined as the geographical layout of the game), but that isomorphism between quest and map structure are seen in many games.

Quests provide challenging elements and concrete goals to the player, but can also be the narrative element that informs the player about the world; they offer the player knowledge and power and can include some dramatic events [35]. Quests are in many games static and linear and offer very little replay value, and even if the quests are non-linear or branching, they still offer very limited replayability. Procedurally generated quests has the potential to overcome this limitation and offer variability and replayability, however, for a quest module to be useful it would have to be fitted to the game, and the module have to know when to generate a quest and must ensure it makes sense in the context of the game. Doran & Parberry [35] believe that procedural quest generation could lead to an increase in player interest because the player will always be provided with a new quest and an alternative gameplay option.

As described the main purpose of quests is to provide goals and activities to the player, but can be used to facilitate narrative and action as well. It is possible for quests to be linked to the map structure, and in Chapter 5 four methods for combining quest and map generation was mentioned, originally suggested by Togelius, et al. [5]. If the quest and map structure should be closely connected, it was theorised that the best way would be to design two separate modules and link the two through either a waterfall or an interactive approach. In relation to the reviewed architecture of Modular PCG described in section 6.7, quest and map generation should be

designed as one module, if the two should be closely connected and directly dependent on each other. However, it would be possible to create a system of two (or more) interconnected sub-modules, one responsible for quest generation and one for map generation.

To determine which procedural approached would be suitable for creating a quest and map module or a system of sub-modules, this section will analyse the existing examples of quest and map generation found within the PCG community. This analysis will illuminate the advantages of procedurally generated quest in relation to game development and illustrate how designer interaction can be facilitated.

The Legend of Zelda: Twilight Princess

An action-adventure game with focus on combat, exploration, and item collection developed by Nintendo EAD and released in 2006.

Dormans [33] investigates the generation of levels for action adventure games, through procedurally generating the overall mission using generative grammar (see Appendix I, for a more detailed description of grammar and other procedural methods). He uses the action adventure game The Legend of Zelda: Twilight Princess to determine the grammar needed for generating an overall mission, and based on the generated mission structure a map is generated. In relation to the proof of concept, an overall mission can be seen as a quest, and the approach by Dormans thus illustrates one way to generate a map based on a quest.

Dormans states that well-designed games generally have two structures, namely the mission and space (map) structure, and suggests that mission and space should be generated using two different grammars designed to suit each task, which is why he uses graph grammar for the mission structure and shape grammar for the map structure. He uses graph grammar because missions can be described as non-linear graphs. A graph grammar produces, instead of strings, graphs consisting of linked nodes, and instead of letters, the alphabet can be other symbols that describe general game concepts, such as obstacle, key and lock (see Figure 26). The start rule can incorporate the overall structure wanted, such as martial art training or Hollywood drama, ensuring that the mission exceeds a minimal length or follows a dramatic arc [33]. Although the initial effort of creating the grammar rules is time-consuming, it is outweighed by the ease with which new content can be generated based on the grammar [33].

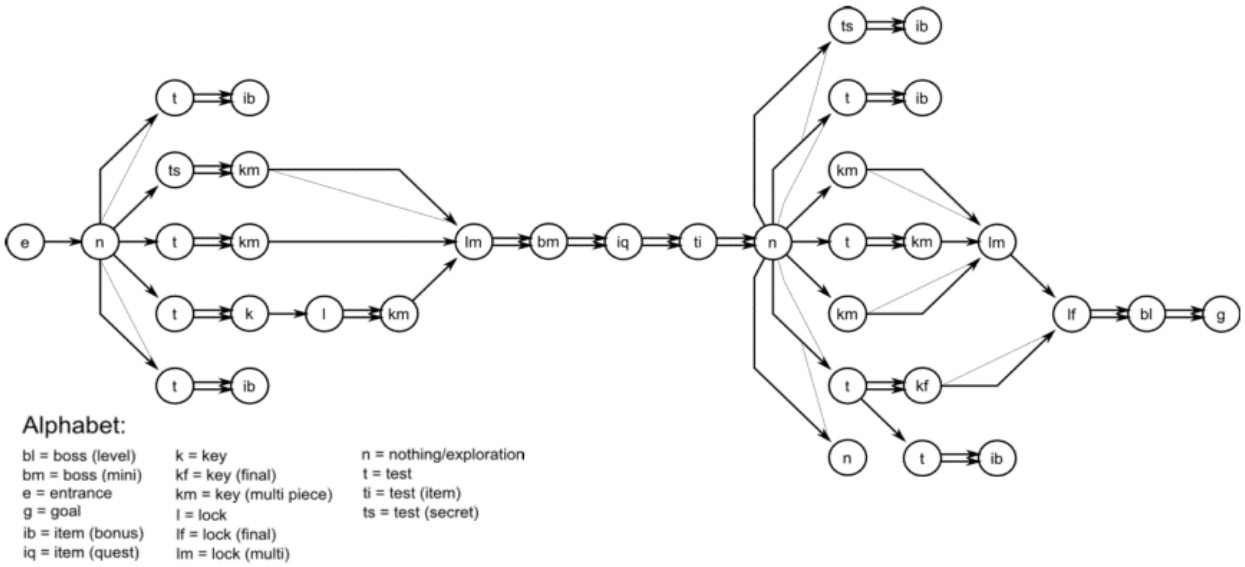


Figure 26: Example of a generated mission structure [33].

For the game space, i.e. map, generation Dormans uses shape grammar, and to ensure the map structure follows the mission structure, the terminal symbols of the graph grammar is translated into symbols in the shape grammar. Instead of symbols or words, shape grammar consists of shapes and rules that define how to reshape the existing shapes. The generation looks for the next terminal symbol in the mission structure and then applies the shape grammar rule that applies to that symbol and find the best suitable location for the rule to be applied (see Figure 27). The shape grammar is extended with some parameters that influence the rule selection in order to create progressive difficulty [33]. If the shape generated after including all mission nodes have any non-terminals, these are replaced with terminal symbols based on a set of finalizing rules.

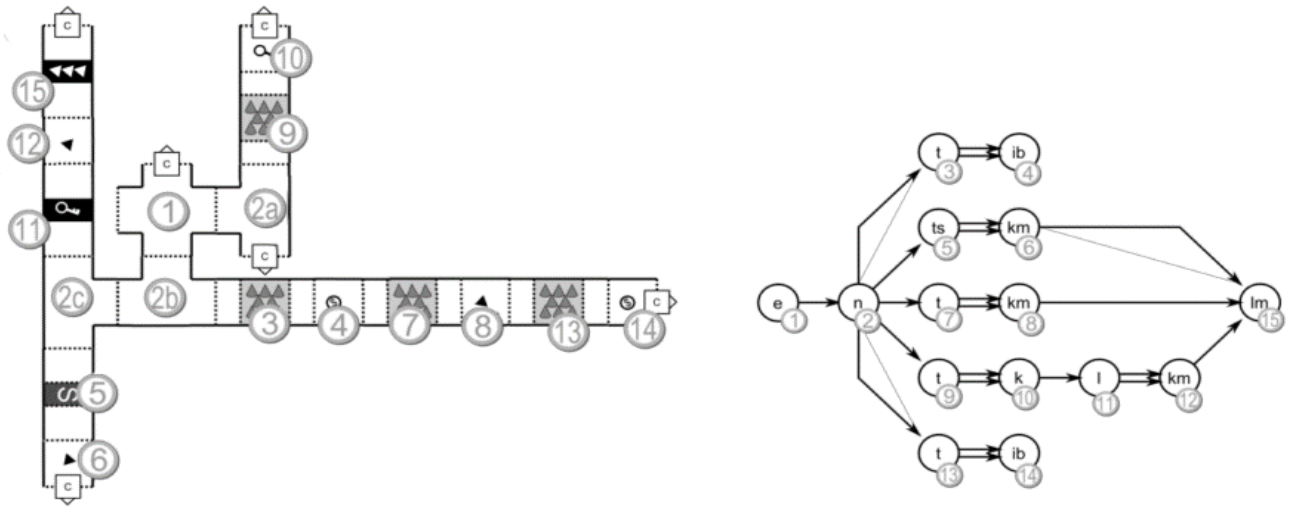


Figure 27: Example of a map (left) generated based on a mission structure (right) [33].

In addition to the static mission and space generation, Dormans discusses the possibility of changing the generation based on player input. For instance, player performance could affect the generation of different features, and allow some parts of the structures to be generated online. One way to facilitate this is to generate the mission before play and the space while the player explorers the game world. This will ensure an overall good mission structure and a game space that fits the player's movement and playing style while minimising the number of dead ends the player encounter. It will ensure varied gameplay and a feedback loop between player performance and generation offers many opportunities [33]. A similar strategy is to leave non-terminals in the game space and/or the mission space and let these be generated during gameplay. These non-terminals should then contain enough information to ensure the overall structure is valid, but the nature of them could be unknown until the player triggers the generation. This could lead to what Marie-Laure Ryan calls fractal stories, where information is added to the story as the player turn his attention towards it [33].

The method used by Dormans has the advantage of ensuring a coherent map and quest structure. By using generative grammar, he ensures that connections are logical and that the structure has a sense of purpose, and because generative grammar functions at the same scale as level design principals, it can be translated into concrete level design elements with relative ease [33]. A disadvantage of using grammar is that it can be difficult for designers to know exactly how the structure will look after generation, especially if many grammar rules have been implemented. Therefore, it is important for designers to have a clear idea of the layout and structure of the game they are creating. However, if designers are interested in exploring different game structures and create new and interesting missions and spaces, it is possible to experiment with the grammar rules. All in all "*mission and space grammars are an efficient way of generating a high variety of quality levels for action adventure games*" [33].

Based on the approach presented by Dormans, it should be possible to create a quest module that uses graph grammar to generate a quest structure, and a map sub-module that can generate a map structure based on the generated quest and a set of grammar rules. To facilitate designer instructions, tools for authoring grammar rules could be created enabling designers to affect and direct the generation. Additional control could be given by allowing designers to create and edit the graph nodes and organise these in the virtual environment. This organisation could be used to instruct the sub-module responsible for the map generation. In relation to the level design discussed in section 7.4, the Enclosed Desert Area module could be part of a map generation module instructed directly by an overall quest module. As such, all the modules discussed in section 7.4 could be applied as sub-modules under a general map generation module.

While Dormans [33] discusses the generation of one overall mission structure, Doran & Parberry [35] discusses a general grammar-based method for generating multiple quests for RPGs. Through analysis of over 750 quests from the four MMORPGs (Massive Multiplayer Online RPGs)

Eve Online, World of Warcraft, Everquest and Vanguard: Saga of Heroes, they discovered that quests share a common structure and propose a general classification based on NPC motivation (see Table 3). These motivations shape the quest and together with different grammar rules, they determine which actions the player should preform to complete the quest. To vary the different quests the NPC motivations changes over time, especially if the player completes a given quest. Each motivation has a number of strategies, which shapes the quest and in Appendix III, an overview of the different strategies can see found. Based on the NPC motivations they have created a prototype quest generator capable of generating quest of similar structure and complexity as the original quests. The generated quest are represented as a tree, where the leaves are atomic actions that can be performed by the player [35] (see Appendix III).

Motivation	Description
Knowledge	Information known to a character
Comfort	Physical comfort
Reputation	How others perceive a character
Serenity	Peace of mind
Protection	Security against threats
Conquest	Desire to prevail over enemies
Wealth	Economic power
Ability	Character skills
Equipment	Usable assets

Table 3: Different types of motivation that can generate quests [35].

The procedure presented by Doran & Parberry can be flexible and very adaptive, and in their implementation, quests could be adjusted to the assumed knowledge of the player. This enabled the quests to vary in length and complexity based on what was assumed known, for instance, if it is assumed that the player does not know the whereabouts of NPC2, a quest could include a sub-quest that tells the player to visit NPC1 to get the location of NPC2. In contrary, this ensures that

Vanguard: Saga of Heroes

A fantasy-themed MMORPG created by Sony Online Entertainment released in 2007.

Everquest

A fantasy-themed MMORPG developed by Sony Online Entertainment. The series was released in 1999 with now 20 expansions.

World of Warcraft

A MMORPG created by Blizzard Entertainment from 2004 with subsequent expansions. Players explore, complete quests, fights monsters and interact with NPCs or other players.

Eve Online

A player-driven MMORPG set in a science fiction space setting, developed by CCP Games, where players pilot spaceships through a galaxy of over 7,500 star systems.

a player is never sent on a quest to find something they already have. A similar functionality could be built into a quest module, enabling designers to adjust quests based on what the player knows, and additionally player input could be used to shape the quests. The quests generated by Doran & Parberry's system follows a generic form and details, such as locations, NPCs and objects, are added to the quests at the end of generation. This replacement technique is also popular in commercial games and suggests that much can be achieved just by changing the details. However, one has to ask what the reason for the use of this technique is – is it because it is cost-effective or because it is the best a most reliable solution. This question raises an interesting discussion, which is beyond the scope of this project and therefore remains unanswered for now. However, this replacement technique could be used in Modular PCG, enabling designers to choose various objects, locations and characters for the different quests, and thereby being able to generate multiple quests from a few simple structures. One could imagine a similar technique where the designers first choose a few elements, such as objects, NPCs and locations, and thereafter the quest module generates a quest that includes the elements in a logical and coherent way.

Similarly to Doran & Parberry, Hartsook, et al. [28] discusses the use of PCG in relation to PRGs, however, instead of generating quests they presents an approach for procedurally generating playable game world based on *a priori* unknown story [28]. This resembles the approach used by Dormans [33], and similarly, a story can be considered a quest in this context. As mentioned in 4.1 the story used for world generation is written as plot points, which can be authored by either a human designer or an artificial designer. This approach has the disadvantage of only being able to use linear stories, however Hartsook, et al. justify this, and states that "*computer games typically have a single main storyline that constitutes the set of plot points that are necessary for completion of the game*" [28].

In their implementation, a map generator uses the plot points and some initial information about story-specific details to create a game world. The generated map consists of *islands*, i.e. the locations connected to specific plot point, and *bridges*, i.e. the areas between the islands. On the bridges non-plot-specific gameplay occurs, e.g. fighting enemies, finding treasures, etc. One clear advantage of the approach by Hartsook, et al. is that they include player preferences when generating the world, and creates a subjective player experience model (player model) based on a pre-game questionnaire about the players preferences. They state that the player model can "*be used to personalize the story and world of the game so as to maximize pleasure and minimize frustration and boredom*" [28]. This player model is used in the generation to determine the branching and length of the bridges. The islands and bridges are generated through a search-based PCG approach, using genetic algorithms. The generation create a space tree representing the game world genotype and rewards the generate content based on the variation between it and the parameters from the player model. The player model thereby determines the fitness. After the generation process has found a suitable layout, the phenotype is generated as a top

down 2D world [28]. In relation to Modular PCG and the creation of a quest module, it should be possible to create a quest module where the designers are able to write a story, i.e. quest, in either an .xml-like language or natural language. The modules should then interpret the input, create plot points, and send these to a map generation sub-module that applies the approach by Hartsook, et al. [28]. Of cause, designers should be given additional options to adjust and fine-tune the generated result, but it could be a nice tool for easy and fast generation of the overall structure.

Where Hartsook, et al. focuses on generating a world for a predetermined story, Ashmore & Nitsche [34] investigates the generation of quests into an already procedurally generated world. This resembles the implementation done by Doran & Parberry [35], because they investigated quest generation as a separate entity as well. The difference, however, is that Ashmore & Nitsche focus on explorative quests, i.e. quests that requires the player to move from location A to location B, and introduces the lock and key metaphor to describe the structure of such quests. The key and lock metaphor means that during exploration obstacles (locks) restricts the movements of the player and he must use items (keys) to overcome these obstacle [34]. This metaphor does not only apply to spatial constraints (such as locked doors and keys), but a lock can be any obstacle that hinders the player, and the key can be any item, skill, etc. that helps the player pass the obstacle.

Doran & Parberry [35] criticises Ashmore & Nitsche and state that the key and lock structure lacks a sense of purpose, they believe that their own system, based on NPC motivation, can express additional types of quests that is not possible with the key and lock structure. It can be argued that the key and lock structure has the potential to express the same types of quests as the system by Doran & Parberry, if the lock was to get an item from an NPC and the key was perform a task given by the NPC. However, the quests that can be generated using the key and lock structure are not very elaborate and complex, and in their implementation locks are materialised mostly as physical barriers. It is a shame that they did not utilize their own system to the fullest potential, and the NPC based quest generation proposed by Doran & Parberry [35] seems to be easier to utilise and follows a more concrete structure, which most likely makes it easier to understand, use and implement. In addition, the implementation by Ashmore & Nitsche illustrates an important issue to remember when using PCG. They did not manage their procedural techniques, which meant that no two playtests were comparable because of the random nature of PCG [34]. This can be avoided by including a random number seed in the generation process as Doran & Parberry did, which enabled them to regenerate the quests for later analysis [35].

CHAPTER 8
CONCLUSION

The conclusion will now give a brief resume of the entire report and restate the most important aspects of the architecture of Modular PCG.

Based on a general interest in PCG and game development, this project started investigating the advantages of PCG in relation to game development. The initial goal was to determine how PCG could be used to facilitate game creation, and through an initial analysis, a more concrete approach to a subject was found. It was decided to investigate the possibility of complete game generation using PCG, and together with the focus on game development, it became the goal to investigate how complete game generation could be made accessible to human designers and how it would integrate within game development. To investigate this the analysis described a few games and research projects that utilises complex procedural techniques, and it was discovered that together with complex PCG comes either very complex or very limited interaction, bordering on inaccessible designer interaction. As a solution to the problem of inaccessible PCG algorithms, the concept Modular PCG was introduced. In short, Modular PCG describes a new way of considering PCG in relation to game development, and it facilitates the creation of individual PCG modules that applies procedural techniques to generate game content. The modules integrates directly into the virtual environment, which means that designers can apply different modules without considering existing content and other modules. For easy and rapid development, the necessary tools for authoring content are included in the modules themselves and work out of the box.

Modular PCG was introduced and discussed in three steps, first it was introduced as a new concept, then the architecture behind it was discussed, and lastly the concept was evaluated as a theoretical proof of concept.

In the introduction of the concept, Modular PCG was defined as a system of individual PCG modules that acts on their own and facilitate easy and relatable game development when combined. Using Modular PCG, designers and developers should be able to choose different modules from different designers and apply them in their own projects. As initial validation of the concept, it was described how traces of Modular PCG can be found in existing PCG applications, and it was argued that because of this Modular PCG is the right direction for PCG research and that it will facilitate an integration of PCG within the game industry. As part of the concept, an initial architecture was described. This initial architecture led to seven types of interaction between modules, designers and players, which was later discussed individually to form a condensed architecture. Initially two types of architecture was said to exists, a top-down (designers and players influence high-level modules that affect low-level modules) and bottom-up (designers and players influence low-level modules that in turn influence high-level modules) architecture.

In the discussion about the architecture, however, it was determined that both a top-down and a bottom-up implementation could be achieved with the same modules by structuring them hierarchically with internal priorities. As an additional change, it was described that the generated content should be represented on three levels, the specification level, structural level and the object level. These levels, represents different levels of detail and are used for different purposes. The specification level allows designer interaction, the structural level allows modules to interact with the virtual world, and the object level represents the final state of generation available to the player. Regarding virtual world interaction, it was decided that modules should interact with the virtual environment in two ways, either affecting or changing the virtual world, called *Instructive* Interaction, or adjusting to the virtual world, called *Adaptive* Interaction. This would enable modules to act independently from each other allowing designers to use different modules without considering existing content in the environment. It was also determined that designers should be allowed to view the generated content on both the structural level, allowing basic adjustments, and the object level, allowing detailed adjustments.

Regarding designer interaction, it was decided that modules should provide designers with the necessary tools for authoring content and controlling generation; in other words, the tools should be included in the modules and integrate automatically with the development environment. Likewise, modules requiring player input should be designed to gather this input automatically. In short, modules should be self-contained, including the necessary authoring tools, and must be able to adjust to the virtual environment without interacting directly with other modules.

Form the discussion of the architecture, the initial seven interactions was reduced to four basic interactions: Virtual world interaction, Designer instructions, Module output, and Player input.

After the architecture was discussed, Modular PCG was evaluated as a theoretical proof of concept. The proof of concept illustrated how Modular PCG could be used to generate a complete complex game with an elaborate quest and map structure, and how Modular PCG could be used in a production and implementation context. The purpose of the proof of concept was not to implement any modules, but to describe theoretically how modules could be created and what creators should keep in mind. It was chosen to describe how Modular PCG could be integrated within CryEngine3, and to make the integration logical it was described in relation to some existing tools and functionalities within the development environment. After describing the integration, the evaluation chapter described a few level design modules that could be usable, and described how these tools should be created and integrated. The creation was based on a game design document (GDD) describing an action adventure game set in the ancient Egypt. Based on the GDD the following modules were described: Enclosed Desert Area module, Small Desert Objects module, Desert Ruin module, and Desert Path module. For each module, different designer tools were described and it was discussed which options could be useful to have as a designer and how much control designers should be given. Because the main purpose of Modular PCG is to give designers better procedural tools allowing easier development, the modules was discussed from a design perspective and have been structured such that it would make sense to a game designer. After describing the level design modules, a section dedicated to quest generation discussed how quests could be generated using procedural techniques and how a quest generation module could be structured. It was stated that if quest and map structure should be closely connected, the two should be designed as one module, possible as a system of two sub-modules. Among the different generation techniques, grammar was mentioned as a viable way of generating both quest a map structure, and even though grammar requires a lot of initial work this should be outweighed by the ease with which new content can be generated afterwards. Inspired by the use of grammar it was suggested that one could create a quest module using graph grammar, and a map sub-module capable of generating a map structure based on the generated quest structure, and thereby ensuring that the two are closely connected. Another example of quest generation included presumed player knowledge, ensuring that players are only given quests that makes sense for them. The described example used a replacement technique, which allowed several quests to be created from the same simple structure. This technique could be used in a quest module, allowing the module to generate several quests based on simple designer instructions, such as a specification of objects, locations and characters. Another example illustrated the generation of a map structure based on a prewritten story structure written in simple plot points. In relation to this, one could imagine a module that was able to generate a map based on a simple story, specified by either a human designer or a quest generation module. In a complete system, a module could be used to generate

the overall quest structure while several sub-modules could be used to generate the different elements of the levels and gameplay.

As a final remark, the purpose of this project has been to help advance the state of the art of PCG, and it is believed that the introduction of Modular PCG has been a step in the right direction. Currently PCG is not widely used in game development, but it is the hope that Modular PCG will increase the use of procedural techniques in the game development industry. Modular PCG has yet to be tested and proven practical in a real game development scenario, however from the theoretical evaluation of the concept, it can be said to be applicable in game development and that it successfully makes procedural techniques accessible to designers and developers. I hope that this project has illustrated the need for Modular PCG as a research field, and I hope that other researchers will use this project as a stepping-stone and continue research in this direction. Modular PCG is the future of PCG.

REFERENCES

[1] M. Hendrikx, S. Meijer, J. V. D. Velden and A. Iosup, "Procedural content generation for games: A survey," *ACM Trans. Multimedia Comput. Commun. Appl.,* vol. 9, no. 1, pp. 1-22, February 2013.

[2] G. N. Yannakakis and J. Togelius, "Experience-Driven Procedural Content Generation," *Affective Computing, IEEE Transactions on,* vol. 2, no. 3, pp. 147 - 161, 2011.

[3] B. Watson, P. Müller, O. Veryovka, A. Fuller, P. Wonka and C. Sexton, "Procedural Urban Modeling in Practice," *IEEE Computer Graphics and Applications,* vol. 28, no. 3, pp. 18-26, 2008.

[4] J. Togelius, G. N. Yannakakis, K. O. Stanley and C. Browne, "Search-based procedural content generation," in *Proc. European Conf. Applications of Evolutionary Computation*, 2010.

[5] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss and K. O. Stanley, "Procedural Content Generation: Goals, Challenges and Actionable Steps," in *Artificial and Computational Intelligence in Games*, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck and J. Togelius, Eds., Dagstuhl, Germany, Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik, 2013, pp. 61-75.

[6] C. Browne and F. Maire, "Evolutionary game design," *Computational Intelligence and AI in Games, IEEE Transactions on,* vol. 2, no. 1, pp. 1-16, 2010.

[7] C. Browne, "Cameron's Yavalath Page," 2013. [Online]. Available: http://www.cameronius.com/games/yavalath/. [Accessed 9 February 2014].

[8] J. Togelius, G. N. Yannakakis, K. O. Stanley and C. Browne, "Search-based Procedural Content Generation: A Taxonomy and Survey," *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG),* vol. 3, no. 3, pp. 172-186, 2011.

[9] J. Togelius, R. D. Nardi and S. M. Lucas, "Making Racing Fun Through Player Modeling and Track Evolution," in *Proceedings of the SAB Workshop on on Adaptive Approaches to Optimizing Player Satisfaction*, 2006.

[10] G. Kelly and H. McCabe, "A survey of procedural techniques for city generation," *ITB Journal,* pp. 87-130, 2006.

[11] R. M. Smelik, K. J. d. Kraker, S. A. Groenewegen, T. Tutenel and R. Bidarra, "A Survey of Procedural Methods for Terrain Modelling," in *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, Amsterdam, The Netherlands, 2009.

[12] A. d. l. Re, F. Abad, E. Camahort and M. C. Juan, "Tools for Procedural Generation of Plants in Virtual Scenes," LA, USA, 2009.

[13] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin and S. Worley, Texturing & Modeling: A Procedural Approach, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[14] G. Smith, J. Whitehead and M. Mateas, "Tanagra: An Intelligent Level Design Assistant for 2D Platformers," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, North America, 2010.

[15] K. Compton and M. Mateas, "Procedural Level Design for Platform Games," 2006.

[16] N. Shaker, G. N. Yannakakis, J. Togelius, M. Nicolau and M. O'Neill, "Evolving Personalized Content for Super Mario Bros Using Grammatical Evolution," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, North America, 2012.

[17] S. Dahlskog and J. Togelius, "Patterns and Procedural Content Generation: Revisiting Mario in World 1 Level 1," in *Proceedings of the First Workshop on Design Patterns in Games*, Raleigh, North Carolina, 2012.

[18] N. Shaker, M. Nicolau, G. N. Yannakakis and J. Togelius, "Evolving levels for super mario bros using grammatical evolution," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, Granada, 2012.

[19] L. Johnson, G. N. Yannakakis and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Game*, Monterey, California, 2010.

[20] T. Mahlmann, J. Togelius and G. N. Yannakakis, "Towards procedural strategy game generation: Evolving complementary unit types," in *Applications of Evolutionary Computation*, 2011.

[21] A. Liapis, G. N. Yannakakis and J. Togelius, "Generating map sketches for strategy games," in *Applications of Evolutionary Computation*, 2013.

[22] J. Togelius, M. Preuss and G. N. Yannakakis, "Towards Multiobjective Procedural Map Generation," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, Monterey, California, 2010.

[23] M. Nitsche, C. Ashmore, W. Hankinson, R. Fitzpatrick, J. Kelly and K. Margenau, "Designing Procedural Game Spaces: A Case Study," in *FuturePlay 2006*, 2006.

[24] J. Togelius, E. Kastbjerg, D. Schedl and G. N. Yannakakis, "What is Procedural Content Generation?: Mario on the Borderline," in *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, Bordeaux, France, 2011.

[25] R. Khaled, M. J. Nelson and P. Barr, "Design Metaphors for Procedural Content Generation in Games," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Paris, France, 2013.

[26] J. Togelius and J. Schmidhuber, "An Experiment in Automatic Game Design," in *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On.*, Perth, WA, 2008.

[27] M. Cook and S. Colton, "Multi-Faceted Evolution Of Simple Arcade Games," in *The Computational Intelligence and Games (CIG)*, 2011.

[28] K. Hartsook, A. Zook, S. Das and M. O. Riedl, "Toward Supporting Stories with Procedurally Generated Game Worlds," in *Computational Intelligence and Games (CIG)*, Seoul, 2011.

[29] R. Smelik, T. Tutenel, K. J. de Kraker and R. Bidarra, "Integrating Procedural Generation and Manual Editing of Virtual Worlds," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, Monterey, California, 2010.

[30] R. M. Smelik, T. Tutenel, K. J. De Kraker and R. Bidarra, "A Declarative Approach to Procedural Modeling of Virtual Worlds," *Computers and Graphics,* vol. 35, no. 2, pp. 352-363, 2011.

[31] Y. I. H. Parish and P. Müller, "Procedural Modeling of Cities," in *SIGGRAPH '01 Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 2001.

[32] M. Lipp, D. Scherzer, P. Wonka and M. Wimmer, "Interactive Modeling of City Layouts using Layers of Procedural Content," *Computer Graphics Forum,* vol. 30, no. 2, p. 345–354, 2011.

[33] J. Dormans, "Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, Monterey, California, 2010.

[34] C. Ashmore and M. Nitsche, "The quest in a generated world," in *Proc. 2007 Digital Games Research Assoc. (DiGRA) Conference: Situated Play*, 2007.

[35] J. Doran and I. Parberry, "A Prototype Quest Generator Based on a Structural Analysis of Quests from Four MMORPGs," in *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*, Bordeaux, France, 2011.

[36] Z. Qin, J. Xing and X. Zheng, "Evaluating Software Architecture," in *Software Architecture*, Berlin, Springer Berlin Heidelberg, 2008, pp. 221-273.

[37] M. Mattsson, H. Grahn and F. Mårtensson, "Software architecture evaluation methods for performance, maintainability, testability, and portability," in *Second International Conference on the Quality of Software Architectures*, 2006.

[38] V. Gal, C. L. Prado, S. Natkin and L. Vega, "Writing for video games," in *Proceedings Laval Virtual (IVRC)*, 2002.

[39] S. Rogers, "You Can Design a Game, But Can You Do the Paperwork?," in *Level Up! - The Guide to Great Video Game Design*, Chichester, United Kingdom, John Wiley & Sons, Ltd, 2010, pp. 57-82.

APPENDIX

I METHODS OF PCG

There are many methods within PCG that each has its strengths and weaknesses, and are suited to produce certain types of content. This section will list some of the methods used in PCG and how these methods can be applied.

The simplest and earliest methods of PCG are based on pseudo-random number generation (PRNG) [1]. Because it is pseudo-random, it can be used to mimic the illusion of randomness found in nature, e.g. mountains, clouds and flowers. Perlin noise is a PRNG-based noise generator, which generates maps of data points through interpolation of points generated by a seeded PRNG. Detail can be added by combining more maps with different scaling.

Another technique is generative grammar (GG), which is sets of rules that operates on words to generate grammatically correct sentences. Generative grammar in general, consist of an alphabet (words) and a set of rules that define rewrite operations of the alphabet. Rules are written as "S →ab", where capital letters describe symbols that can be changed and lowercase letters are terminal symbols that cannot be rewritten. Generative grammar always starts with one symbol, often denoted as "S" [33]. This technique can be adapted to describe and generate correct objects, e.g. in a game level, from elements encoded as words [1]. Sub-systems of GG includes L-systems, split grammars, wall grammars and shape grammars. L-systems was designed to describe the growth of plants. It is today used to generate trees as well as other natural structures and are even used in city generation [33].

Image processing techniques can also be used in PCG, namely image filtering (IF), which is used to emphasize elements of in image or to improve subjective measurements of an image, i.e. give

a certain style. Within IF, one can for instance use binary morphology, i.e. operations on a binary image, or convolution filters, i.e. modify an image with another or a kernel, to modify and change images.

As IF can manipulate images, spatial algorithms (SA) can be used to manipulate space, for instance by storing map data in a grid with the technique called tiling. After the data is decomposed, i.e. cut into sections in a grid for instance each tile can be manipulated. Layering is a technique, where several grids (layers) are combined into one map. Each tile are then constructed by several overlapping layers [1]. To save memory one can use grid subdivision to only divide grid cells close to the player, in order to provide detail, while cells beyond a threshold remains undetailed. Another SA is fractals, which can be described as recursive copies of itself, e.g. snowflakes. One advantage is that fractals can produce objects with seamlessly endless detail. Voronoi diagrams is another way of dividing space into smaller regions. In metric space, a number of seed points (points of interest) are selected and a number of points equally distant from the closest two seed points establishes the borders [1].

Natural phenomena can in some cases not be described with mathematical formulas, and in those cases modelling and simulation of complex systems (CS) can be applied, for instance cellular automata, tensor fields and agent-based simulation. In cellular automaton, the simulation is based on a grid of cells that each has a state and can influence its neighbour cells. The cells are bound by a common set of rules. Tensor fields are a set of two-dimensional vectors (tensors) that describe the shape of the game space. Because it can be visualised, tensor fields are suited to visual interactive design. In agent-based simulation, complex situations are modelled using agents. As the agents interact emergent behaviour arises that can be observed through traditional modelling techniques [1].

One of the great fields of computer science, artificial intelligence (AI), provides some methods usable in PCG. One of which is genetic algorithms that mimics biological evolution, where content is generated, a fitness function then evaluates the result and a mutation and crossover function creates new content [parallel to search-based PCG]. Artificial neural networks are systems of neurons that each take input and give output based on internal criteria. By adjust when each neuron is fired (gives output) the system can learn patterns. The last method within AI, which will be described here, is constraint satisfaction and planning, which can plan what actions needed in order to get from an initial state to an end state. A planner consists of an initial state, actions it can take and a goal test. Planners can be either forwards state-space search algorithms or backward state-space search algorithms depending on in which state they start the search [1].

II GAME DESIGN DOCUMENT

Game design document URL: www.scribd.com/doc/5402045/The-Design-Document-Justin-Kelly

The Design Document

PROJECT SCARAB

Justin Kelly
October-3-07
Torin Lucas

Table of Contents

Section 1 – Game Overview

Working Title
Project Scarab

Genre
Action-Adventure / Religious / Comedy

Audience
Ages 11 – 32, with some gaming background.

The Plot
Commanded by a burning bush, You-Moses, are sent on a quest to liberate your people from the oppression of Egyptians. Infused with the godly power and with your trusty staff at side, you will fight and quest though Egyptian lands, temples and palaces; dodging traps, pits and guards. Until Reaching your final goal the exodus from Egypt.

Section 1.1

Level Abstract
First Level – Introdus

After Selecting **New Game** from the main menu, the first level starts with a cinematic of the oppression of the Hebrews. The Video though careful foreshadowing and panning shows your main objective - finding, meeting and killing/persuading the pharaoh. The Opening cinematic also gives a bit of back-story on Moses the river way and how he obtained his lot in life as well as shows the player what Egyptian guards look like and what Hebrew slaves and non combatants look like. There is no Indication of traps at this time. The Player then is placed into the level where a burning bush calls out Moses' name (Your Character) and you meet the bush and are further informed about yourself and what's going on. You then set out to march towards the pharaoh. The next scene starts and you arise from camp to find your path blocked and a sleeping spy whom has been recording information about your movements. (Gives you information about your march thus far – as 3 days have passed since you left the bush). You find a new path in a cavern used by the military and spies to say stealthy and move people in and out without being seen, with that you proceed to scene three. Scene three is a cavern where you are faced with a small pit trap if you find it and the first wow area and plot trigger. The player walks though a cavern over a wooden bridge and sees a waterfall and large vista. The player is then met with a guard and made to fight to defend himself, as the game teaches combat to the player. After this the player loots the guard's goods and receives more information about why the spy was spying on him and who is after him and who sent the spy and guard. The player then descends deeper into the cavern – scene 4.

After this is outside the scope of this document.

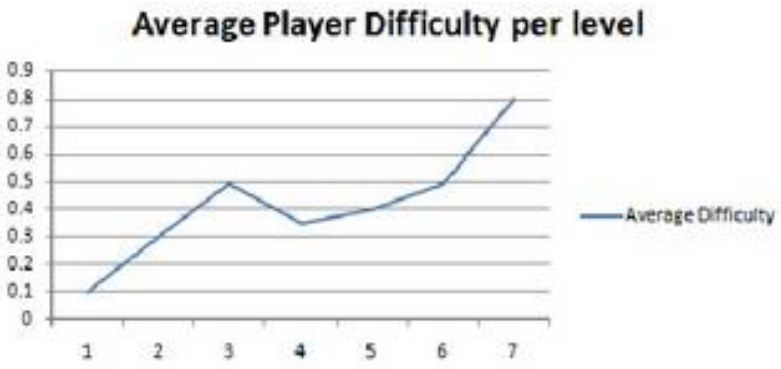
Central Characters

Moses

Objectives: To survive level to level, reach the end boss and liberate your people.

Difficulty: Player Unit / Protagonist

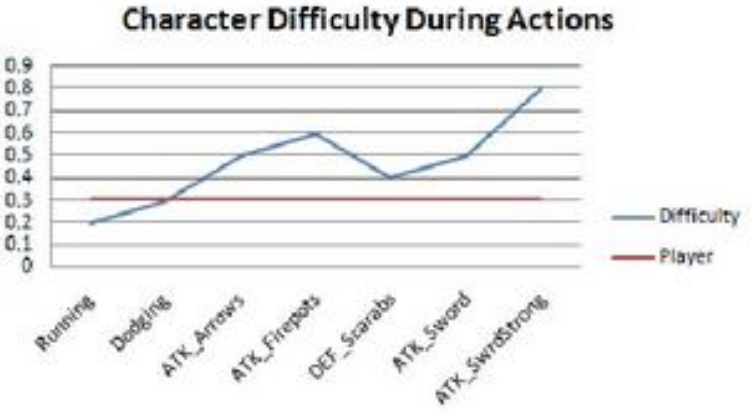
- Difficulty based on player skill level and game Level Modifier.



Pharaoh

Objectives: Lead Antagonist – Stop Moses from liberating the slaves.

Difficulty: Hard - being the end character if the user chooses to fight the pharaoh the must survive a barrage of melee and ranged attacks. The Pharaoh uses ranged and melee attacks, the play must learn to attack the Pharaoh during and after attacks, and dodge ranged projectiles.



Location

Ancient Egypt approximately 1525 B.C, Goshen and Midian Provinces. Upon Exodus the eastern provinces of Egypt and the Red Sea will be visible.

Environment

The Environment will consist mainly of medium to small open areas connected by linear indoor or outdoor paths. Areas consist but are not limited to:

- Open desert levels
- Old and new caverns
- A Egyptian temple
- A Egyptian palace
    - Egyptian palace subfloors
- Egyptian Ruins
- Egyptian Market and Streets

Weather/Environmental Effects

Will consist of:

- Sunny desert
- Lit and dimly lit cavern fog and hazing
- A Dark Brooding storm
- Hellfire (fireballs/ volcano eruption related effects)
- Hail
- Locus Storm
- Frog Rain
- Gas Storm (Green Haze)

Enemies

Enemies consist mainly of basic vermin (bats, rats, snakes, vultures) and some specialized vermin like scarab beetles, Jackals, Palace Leopards. Non Standard Enemies consist of place guards, guards, members of the Egyptian armed forces. All Human units are scaled to their period and posses no abilities other then what is granted to them for being human. There are also a few enemies that NPC and hostile towards the player that can be considered Enemies although they are only spawned once and are unique.

Gameplay elements (summary)

Puzzle Aspect

Moses (The Players Character) will need to navigate dungeons, temples, palaces and other natural landscapes. These areas will be filled with traps and Navigational and dexterity (mouse, keyboard skill) demanding puzzles. Some will include navigating a sewer filled with rats why dodging the water run off as to not be swept into the main flow and pulled under. Another example would be jumping on rock pillars to cross an expanse in one of the caverns or stepping on the correct pattern of stones in a temple. Puzzle gameplay is the primary aspect of the game.

Action Aspect
As with any good puzzle game there is some conflict or action present, be it zombies or Ganondorf. In Project Scarab the Player is faced with a few opportunities where he must defend himself in combat. Action is not a primary aspect of the game. Action is used to break up the puzzle aspect of the game as well as to round of gameplay so that the game doesn't become to stale and narrow.

Humor Aspect
Humour is a debated aspect that may not be included in final release. If the game chooses to be humours it would come in a over the top manner. Humour would be used to create a enjoyable game where the player plays to see the next bit of dialog and to see how the comical Moses will progress.

Storyline
Storyline will be a key aspect to the game. The Storyline is the driving force behind the game, the other gameplay elements will drive the game and the storyline give its shape and context. The storyline will aim to drive fear and a sense of urgency and need into the player. The Storyline if humour is added will become less important but its job of shaping the scope and feel of the game will remain the same.

Position

Before
Main Menu Selection Screen
There is nothing before this level as this is the first level.

After
Ta' Mara Heights
After this level is the outskirts of the Egyptian city where the pharaoh is believed to be presiding.

Section 2 – Characters and Physical Layout

The Following Information is specific to the section of the game being covered by this document.

Central Characters
The Players Character – Moses

Ai Characters
- Sleeping Spy
- Weak Egyptian Soldier

Patrols
There are no Patrols in this level.

Enemy placement
Enemy placement is noted on page 14 in the document

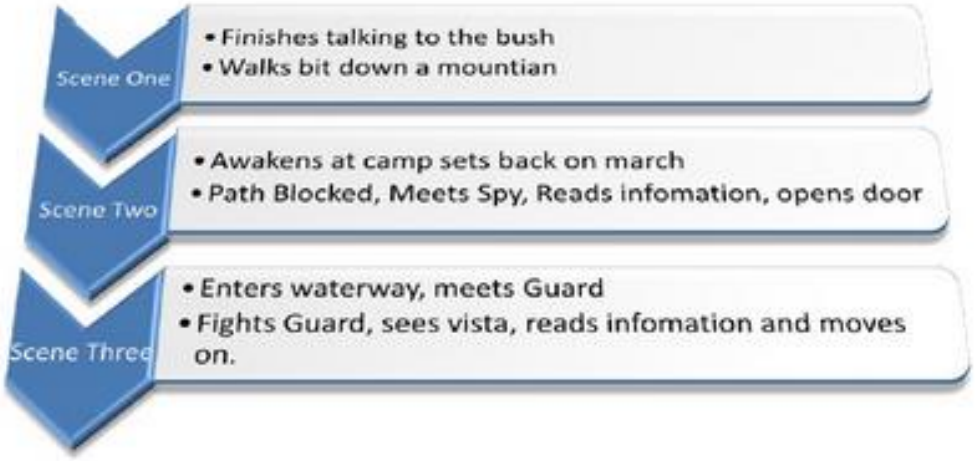
AI Triggers

There is one AI trigger called by the script presiding over the bridge See Level layout on page 14

The Players Perspective

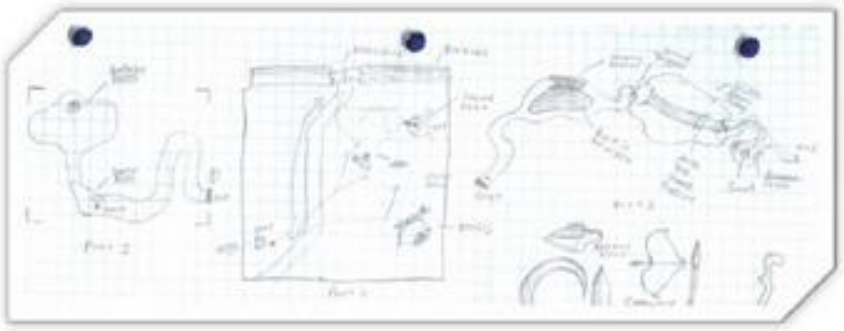
- First Person.
- 90° FOV.
- 122 UU off the ground camera.
- Players Hand and staff displayed, no feet.

Cell Diagrams of action sequence / narrative



Level Layout with description and marker key (blocking)

See Page 14



Section 3 - Supporting documents

Game Metrics

All Human Units

Base Character size

96 UU Standing, 64 Crouched and 25 UU radius wide.

Speed

- **Jump distance**
  64 UU Single Jump
- **Running**
  100 UU a second
- **Walking**
  70 UU a second

Player Values and modifiers

- *Disposition* – A Measurement of how disliked a object or player is to a this unit. [0.0 to 1.0 Value]
- *Fraction Status* – A note/team to which this unit belongs, these values assigns the units hostility's and alliances.
- *Hostility Level* – Combined with the Disposition this value determines the chance this unit take arms and attack the player character or object this is applied to.
- *Current Health* – Current level of Life, 0.0 will result in the death of a object [0.0 to 1.0]
- *Current Status* – Frozen, Attacking, Panicking, Defending, Idle (triggers animations and effects)
- *Objects Name* – For tracking purposes and in game HUD
- *Is Immortal* – Can this person be killed (used to protect main quest characters)
- *Inventory Set* – This Defines the body, Armour, Cloth ect. of the unit.

Moses

Character size

110 UU Standing, 70 Crouched and 25 radius wide.

Speed

- **Jump distance**
  70 UU Single Jump
- **Running**
  120 UU a second
- **Walking**
  55 UU a second

Inventory Set

- Knarled Staff
  - Swing Speed
    - Slow [0.3]
  - Defence Ability
    - Poor [0.3]
  - Damage points Induced
    - 38% of the players strength + 0.3
  - Other
    - Blow Back 15% of the Players weight vs. player's strength.
    - Ability to cast divine providence.
- Moses Robes
  - Upper Body
  - Lower Body
- Moses Beard
- Moses Hair

Powerups

Moses will gain the ability to cast the 10 plagues to which he can use in smaller form against foes.

Purpose

Main Character for the player to utilise to complete objectives.

Visual description

Moses is a long bearded old man with period costume. Moses looks a little worn out and aged which is to add to his solid "no force shall move me" character trait. He stands 5.8 ft tall and holds a wooden snarled staff that as aged as he is. He face looks that of a serious old man he never smiles, nor ever seems hostile.

Weak Egyptian Guard

Character size
Standard

Speed
Standard

Inventory Set

- Short Sword
  - Swing Speed
    - Fast [0.7]
  - Defence Ability
    - Fair [0.5]
  - Damage points Induced
    - 28% of the Units strength + 0.2 (when used by AI)
      - -0.05 – Easy
      - -0.02 – Normal
      - -/+ 0.0 for Hard
    - 38% of the players strength + 0.4 (when used by player)
  - Other
    - Blow Back 1/8th of the time at 5% of the Players weight vs. player's strength.
- Weak Egyptian Guard Gear
  - Upper Body
    - Random chance of equipping - Helmut (non pick up able)
    - Robes with studded leather armour
  - Lower Body
    - Leather Skirt with studded flaps.
    - Sandals (non pick up able)
- Egyptian Facial Hair
  - Randomised from a set of hairs.
- Egyptian Hair
  - Randomised from a set of hairs.

Purpose
To protect the Empire and Pharaoh from all it's/his enemies. As a game element a guard's objective is to delay or kill Moses. Serves as a Standard monster in human parts of the game (palaces) as well serves as a stronger, smarter monster then wildlife (rats, bats).

Visual description
He is a standard troop from the Egyptian Army. Being a weaker version of the armed forces (less trained, and as a result less equipped with expensive armour) the character is shown as presentable and semi professional. Armour is studded leather amour and looks slightly aged as if it was passed down a generation. He is armed with a sword, fairly athletic with a strong face and demeanour.

Section 4 - Game Specific Detail

Light sources

Lighting will be provided by overhead skylight and manual lighting to highlight objectives and provide the scenes with dramatic shadows

Obstacles

Small Bottomless Pit

- Basic Pit for the player to get a feel in a non stressful environment how to traverse platforms.

Blocked Path (Storyline Obstacle)

- The Blocked Path is a physical blocked path that forces a change in the players mission. No longer can he just walk the road into Egypt. Forces the player to look around for another path.

Mission specific structure

The Cavern Entrance

- The Entrance to scene three. Rock faced bulge from the game, with a gate that is visibly locked with a gate.

The Burning Bush

- A large bush consumed in flames that Moses meets in the first level. It's a standard Tumbleweed looking with consumed in bright heavenly flames with a halo glows around it.

The Rope Bridge

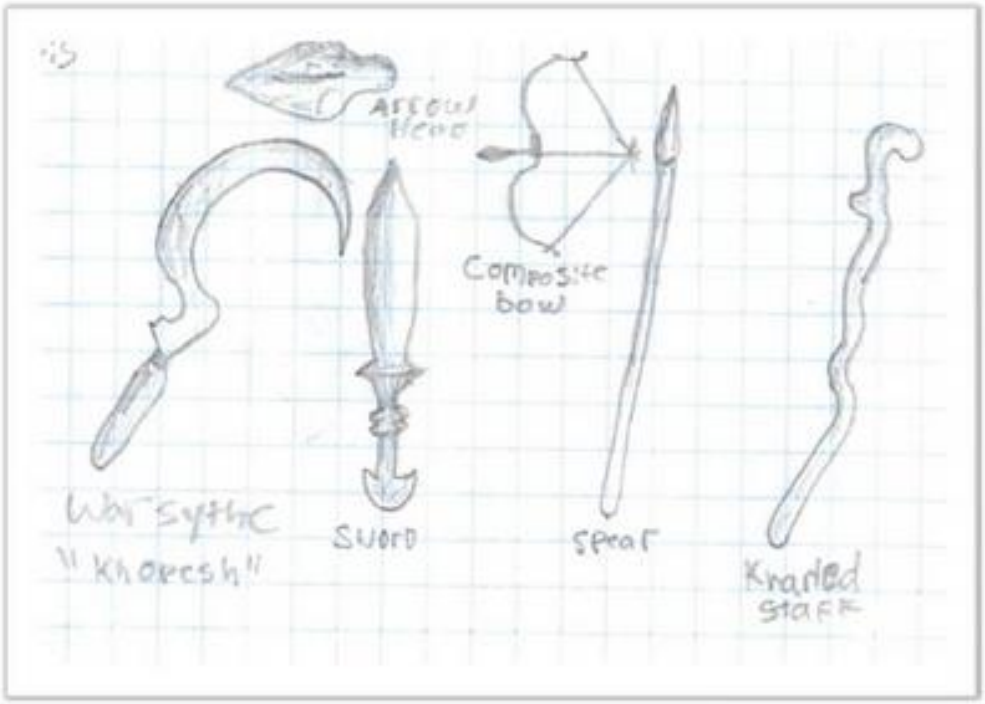
- This Bridge introduces the first Wow area of the game, when the player traverses a large wooden rope bridge that suspends between two parts of a cavern. The rope bridge is suspended over a dark pit that looks very far down, so far it's black. A waterfall flows beside the bridge on the right and pours into the abyss. To the left the player can see a beautiful vista that showcases the height of the cavern as well as the long distance left to travel to the city, seen in the vista.

Doors, Elevators and Traps (Dimension and behaviour)

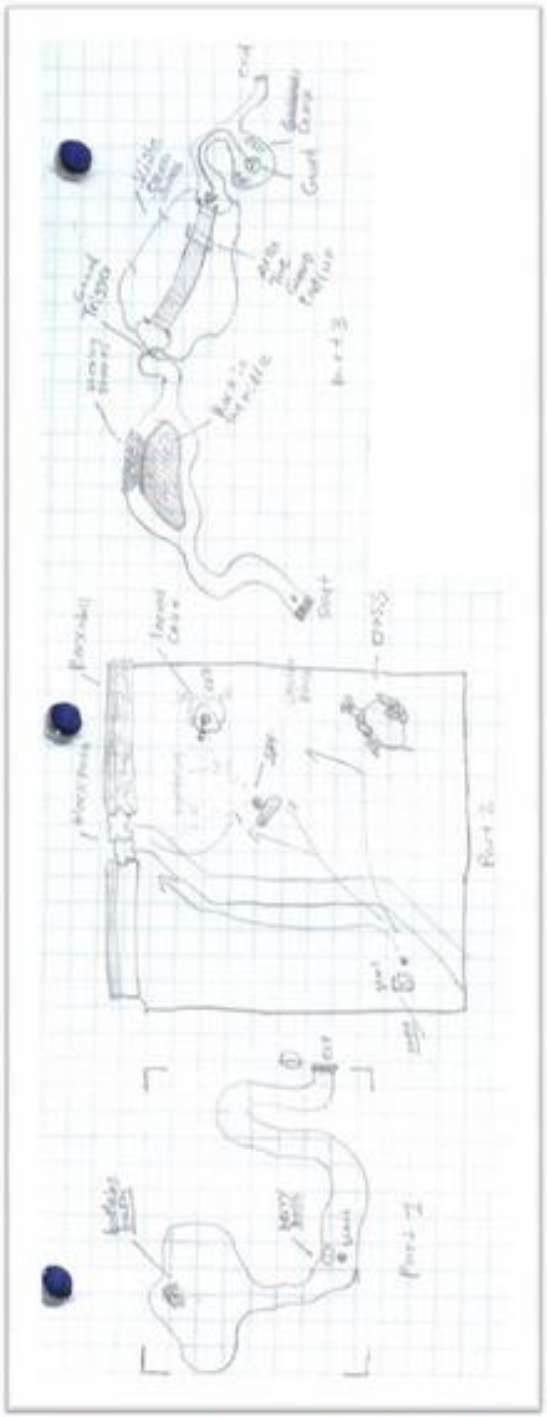
The Rope Bridge

- The Rope Bridge will be powered by the Karma Physics Engine and wobble a bit under movement to give it the illusion of being real and increase the realism level of the game.
- Dimension of the bridge will be proportional to the amount of time it takes the user to realise the features and the speed at which the average player progresses though that part of the level. On final build it should be presumably 1200 UU long and 222 UU wide

Section 5 - Sketches and Brainstorm Documents



Above: Some of the Period Weapons used throughout the game. On the far right is the wooden staff used by Moses.



1 Level is scaled to the size of Moses, The black dot. 55 UU Round.

Left: The three scenes being covered by this level design document. This level is the first of many scenes that will compose levels which will create the overall fabric of the game.

The First Scene consists of a burning bush that calls out Moses by name. At first the call is slow then with time the voice increases in frequency, attracting the player to the bush. If the player attempts to head down the mountain he is blocked by a message saying "Something compels me to stay"

The Second Scene is after the player has left the mountain and he awakens at his camp set the night before. The player awakens to set out down the road to head towards the town which the pharaoh is believed to be located. Only to find the road is blocked. Or if the player fails to follow the road he will run across the spy that outlines 3 days worth of spying on Moses and the intentions of people in Egypt to harm and kill him. The player then gets a Key from the spy (text message no physical key) this triggers the unlocking of a rock mound that leads into a secret cavern used by spies and the military to move in secret.

The Third Scene is where Moses enters the cavern, a bit of navigation and environment effects are placed in the level here. The player if going the top route is faced with a basic pit jump puzzle if too difficult they may take the secondary route. This is not intended to always be used, and is unimportant to the overall level. Moses then reaches a Wooden rope bridge over a massive expanse to which a waterfall flows into, to his left there is a break in the cavern wall showing him the city to which he is travelling to. Moses then fights a guard and further reads (via a note), more information about himself. The guard also has a small camp with health and supplies. Moses then enters deeper

Section 6 – Sequences

In game in its whole entirety the game will rely heavily on scripted sequences and cinematic. They will show up frequently.

Scripted Sequences

In this document's limited scope there is one scripted sequence which consists of Moses meeting the first guard. This introduces the player to what is going on, reaffirms the information gathered by the spy (that Moses is being watched and hunted) and reinforces to the player whom is hostile against him and what he looks like. There is dialog which consists of the guard rushing out thinking that Moses is the spy whom has returned with news, only to realises at some quick sentences that this is not the spy, but in fact Moses – whom he attempts to attack afterwards. This introduces the player to combat against a human foe.

There may also be - depending on alpha testing of this game, a quick trigger and scripted sequence when Moses sees the vista on the roped bridge. It all depends if noticeable to players.

Cinematics

There are 5 Cinematics used. One is used at the end of each scene to transition to the next scene. One is also used from the main menu to introduce the game, Moses and the backstory.

1. IntroductionToScene1
    - Moses and his past is announced the movie outlines Moses lot in life, and foreshadows what he needs to do, Shows him friend and foe and Hebrew oppression also foreshadows that he is a important person, destined for greatness.
2. MosesAndBush
    - Moses Meets the bush and is given images and dialog that solidifies his quest and the bush entrusts him as the only hope for a enslaved race. Outlines his first quest and gives him direction and purpose. Cross Fade to ingame
3. Scene1To2
    - Transition that shows him walking off a mountain then endlessly in a desert before making camp a few days later. Fade to black.
4. Scene2To3
    - Moses Enters a cave dark and filled with a light fog – Pan away and fade to black.
5. Scene3To4
    - Moses Ventures deeper in a cave, rocks rumble and Moses slips a bit and regains his stance –a foreshadow of possible problems and traps ahead.

After this is outside of the scope of the document.

Section 7 – Art Assets & Directory Structure

Textures

Environmental Textures

- Desert Textures
  - Sand
    - 3 Versions
    - +1 version to accent the sand textures
  - Oasis Grass base
  - Oasis Grass Physical
  - Rock Face for Static Mesh/BSB
- Cavern Textures
  - Cavern Rock Face
  - Water Pool Bottom
  - Pure Black – For dark dark areas.
  - Sun Beam Texture for Static Mesh
  - Moss
- Atmosphere
  - Dust Twinkle Sprite
  - Waterfall spray
  - Fire Sprite
  - SkyBox
    - Something of eye-catching nature
    - Broad Daylight
    - Shows the scorching heat by blurring the sun

Static Mesh / Object UV Textures

- Bridge Static Mesh/Karma
  - Wood
  - Rope
- Unit Spy
  - Spy UV Map
- Unit Guard
  - Guard UV Map
  - Helmut
  - Sword

Shaders

No Shaders will be used other then what comes default to the unreal engine.

Bumpmaps

No Bumpmaps will be used other then what comes default to the unreal engine.

Models

- Guard
- Sleeping Spy
- Cavern Entrance Face
- Rock Debris
- Rock Blockade
- Flaming bush

Sound FX

- Ambiance
  - Cavern Dark Sound
  - Desert
  - Bush Burning
  - Creaking of wooden bridge
  - Dripping water
  - Waterfall
- Effects
  - Sword to Moses (flesh)
  - Moses (wood Staff) to Sword/Guard
  - Foot on sand
  - Metal Clinking
  - Snoring

Voice Overs

- Guard
- Moses
- God

Music

There is no music added to this level.

Interface assets

- Custom Hud
  - Clean up the Health and Adrenaline bar and replace it with large numbers in the top left for both.
  - Remove Unreal's Colour Framing
  - Remove Inventory

III NPC MOTIVATIONS FOR QUEST GENERATION

All tables has been taken the article by Doran & Parberry [35].

Motivation	Strategy	Sequence of Actions
Knowledge	Deliver item for study	<get> <goto> give
	Spy	<spy>
	Interview NPC	<goto> listen <goto> report
	Use an item in the field	<get> <goto> use <goto> <give>
Comfort	Obtain luxuries	<get> <goto> <give>
	Kill pests	<goto> damage <goto> report
Reputation	Obtain rare items	<get> <goto> <give>
	Kill enemies	<goto> <kill> <goto> report
	Visit a dangerous place	<goto> <goto> report
Serenity	Revenge, Justice	<goto> damage
	Capture Criminal(1)	<get> <goto> use <goto> <give>
	Capture Criminal(2)	<get> <goto> use capture <goto> <give>
	Check on NPC(1)	<goto> listen <goto> report
	Check on NPC(2)	<goto> take <goto> give
	Recover lost/stolen item	<get> <goto> <give>
	Rescue captured NPC	<goto> damage escort <goto> report
Protection	Attack threatening entities	<goto> damage <goto> report
	Treat or repair (1)	<get> <goto> use
	Treat or repair (2)	<goto> repair
	Create Diversion	<get> <goto> use
	Create Diversion	<goto> damage
	Assemble fortification	<goto> repair
	Guard Entity	<goto> defend
Conquest	Attack enemy	<goto> damage
	Steal stuff	<goto> <steal> <goto> give
Wealth	Gather raw materials	<goto> <get>
	Steal valuables for resale	<goto> <steal>
	Make valuables for resale	repair
Ability	Assemble tool for new skill	repair use
	Obtain training materials	<get> use
	Use existing tools	use
	Practice combat	damage
	Practice skill	use
	Research a skill(1)	<get> use
	Research a skill(2)	<get> experiment
Equipment	Assemble	repair
	Deliver supplies	<get> <goto> <give>
	Steal supplies	<steal>
	Trade for supplies	<goto> exchange

Table 4: Strategies for each of the NPC motivations from Table 2 using actions from Table 5.

	Action	Pre-condition	Post-condition
1.	ε	None.	None.
2.	capture	Somebody is there.	They are your prisoner.
3.	damage	Somebody or something is there.	It is more damaged.
4.	defend	Somebody or something is there	Attempts to damage it have failed.
5.	escort	Somebody is there	They will now accompany you.
6.	exchange	Somebody is there, they and you have something.	You have theirs and they have yours.
7.	experiment	Something is there.	Perhaps you have learned what it is for.
8.	explore	None.	Wander around at random.
9.	gather	Something is there.	You have it.
10.	give	Somebody is there, you have something.	They have it, and you don't.
11.	goto	You know where to go and how to get there.	You are there.
12.	kill	Somebody is there.	They're dead.
13.	listen	Somebody is there.	You have some of their information.
14.	read	Something is there.	You have information from it.
15.	repair	Something is there.	It is less damaged.
16.	report	Somebody is there.	They have information that you have.
17.	spy	Somebody or something is there.	You have information about it.
18.	stealth	Somebody is there.	Sneak up on them.
19.	take	Somebody is there, they have something.	You have it and they don't.
20.	use	There is something there.	It has affected characters or environment.

Table 5: Atomic actions.

	Rule	Explanation
1.	<subquest> ::= <goto>	Subquest could be just to go someplace.
2.	<subquest> ::= <goto> <QUEST> goto	Go perform a quest and return.
3.	<goto> ::= ε	You are already there.
4.	<goto> ::= explore	Just wander around and look.
5.	<goto> ::= <learn> goto	Find out where to go and go there.
6.	<learn> ::= ε	You already know it.
7.	<learn> ::= <goto> <subquest> listen	Go someplace, perform subquest, get info from NPC.
8.	<learn> ::= <goto> <get> read	Go someplace, get something, and read what is written on it.
9.	<learn> ::= <get> <subquest> give listen	Get something, perform subquest, give to NPC in return for info.
10.	<get> ::= ε	You already have it.
11.	<get> ::= <steal>	Steal it from somebody.
12.	<get> ::= <goto> gather	Go someplace and pick something up that's lying around there.
13.	<get> ::= <goto> <get> <goto> <subquest> exchange	Go someplace, get something, do a subquest for somebody, return and exchange.
14.	<steal> ::= <goto> stealth take	Go someplace, sneak up on somebody, and take something.
15.	<steal> ::= <goto> <kill> take	Go someplace, kill somebody and take something.
16.	<spy> ::= <goto> spy <goto> report	Go someplace, spy on somebody, return and report.
17.	<capture> ::= <get> <goto> capture	Get something, go someplace and use it to capture somebody.
18.	<kill> ::= <goto> kill	Go someplace and kill somebody.

Table 6: Action rules in BNF.