# *Procedural Generation of Narrative Puzzles in Adventure Games: The Puzzle-Dice System*

# Procedural Generation of Narrative Puzzles in Adventure Games: The Puzzle-Dice System

Clara Fernández-Vara
Massachusetts Institute of Technology
telmah@mit.edu

Alec Thomson
Massachusetts Institute of Technology
alect@mit.edu

## ABSTRACT
This project tackles procedural generation of narrative puzzles found in adventure games. The challenge is not only generating the puzzles in games which traditionally only have one walkthrough, but also making the development process accessible to designers. Given that the goal is to make these games playable and easy to develop, the focus of this project is facilitating the immediate development of these games. This paper describes the system of procedural generation of one game, *Symon*, which was the reference and inspiration for a standalone toolset, the Puzzle Dice System to create other adventure games with procedurally generated puzzles. The toolset has been put to the test with another game, *Stranded* in Singapore, and it is still being expanded and improved on at the moment of writing.

## Categories and Subject Descriptors
D.2.2. [**Design Tools and Techniques**]: Object-oriented design methods – *games.*

K.8.0 [Personal Computing]: General – games.

## General Terms
Algorithms, Design, Economics, Experimentation, Human Factors.

## Keywords
Procedural generation, adventure games, puzzles, narrative.

## 1. INTRODUCTION
Procedurally generated content in videogames has been relatively common since games like *Rogue*. While it has become easier to generate game spaces and levels, as well as mathematical puzzles, one of the current challenges is incorporating narrative into procedurally generated content. Previous academic research has focused mainly on procedural level generation for video games [2, 4]; other specific work on procedurally generated content is specific to puzzle research: Ashmore [1] focuses on "lock and key" puzzles while Doran and Parberry [3] have constructed a procedural quest model based common structures of quests from MMORPGs.

This project brings procedurally generated content to a different application of narrative games—the generation of narrative-driven puzzles in adventure games. For our purposes, "adventure games" refers to the family of games whose design derives from Crowther and Woods' *Adventure*, a.k.a. *Colossal Cave Adventure*. They are story-driven games, where the player controls a character or characters; the main mode of gameplay is solving puzzles that are integrated in the environment; the solution to the puzzles is found by exploring the space and objects and the possibilities to manipulate them [5]. Traditionally, the player advances in the story of the game by solving puzzles; since the narrative is usually pre-set, the puzzles and the story always remain the same. Even when there may be multiple endings, these endings are usually pre-determined by the designer.

The challenge tackled here is generating narrative puzzles procedurally, to make adventure games replayable. The puzzles must both make sense from a logical standpoint as well as narrative. This paper will overview the design challenge, as it was first tackled in the game *Symon* [7], and the subsequent development of tools to develop games that make it easier to create similar adventure games with procedurally generated puzzles. The project focuses on procedural generation as a design problem, rather than a computational issue.

*Procedural Generation Content Workshop at Foundations of Digital Games 2012*, May 29th, 2012, Raleigh, North Carolina, USA.

## 2. THE PROBLEM
The initial idea was to create an adventure game where the puzzles were different whenever one started the game. Thus the player would find new challenges in subsequent replays. There are many types of narrative puzzles in adventure games, which were the ones we were aiming at generating. Here are a few examples:

- giving an object to someone who will provide a reward in exchange.
- combining two items that result in a new object.
- disassembling an object.
- restoring an object to its original state.
- saying the right thing to convince a character to help the player.
- providing a key that gives the player access to a new area.

These high-level examples are patterns that derive from adventure games [5], and constitute the foundation of the system. The gameplay would be based on a set of puzzles, which would concatenate these patterns to make up the structure of the game, which we will call puzzle map. The map establishes the relationships between puzzles, so that solving a puzzle provides an item or information to solve another puzzle. For example, by taking the battery from a radio (disassembling an object), one can use it to make a flashlight work (restoring an object to its original state). See Figure 1 as a sample map of how puzzles would relate to each other. During the conceptual phase of this project, the

designer would come up with both the puzzle patterns and the puzzle map; the procedural generation would consist of inserting the patterns in the pre-designed map, so that the outcome of one puzzle would be necessary to resolve another puzzle or finish the game.

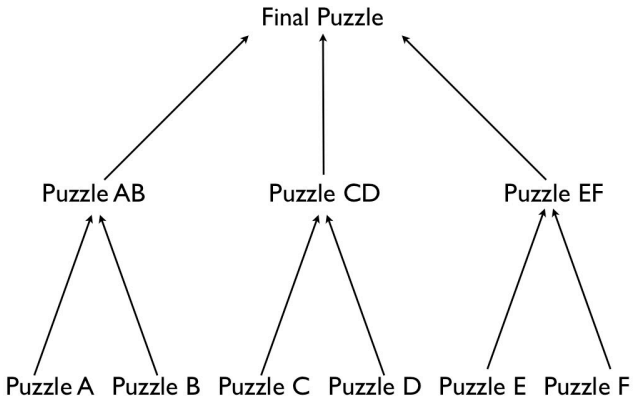


**Figure 1: Sample puzzle map of an adventure game.**

# 3. THE INITIAL SYSTEM – *SYMON*

In order to create the system that would generate the puzzles, the structural idea was inspired by the GRIOT system [6] to generate poetry procedurally. GRIOT allows writers to define first a general structure of stanzas and topics, which is the higher level structure that will be populated with sentences. Then the system allows poets to define grammatical structures, which are later filled up with words according to specific semantic criteria and grammatical categories (such as noun, verb, etc.). In a similar way, the puzzle map and puzzle patterns like the ones described above constitute the higher level, and the puzzle patterns could be populated with characters and objects that fulfilled specific criteria.

Each of the characters and objects to populate the pattern would have a set of properties (for example: character, temperature, color, taste, carryable); and each property would have a set of possible values (for instance: temperature could be hot or cold). The system would fill out the puzzle pattern according to their categories and attributes. For example:

```
Give character1 item1 [temperature=cold] then reward with item2 [edible]
```

or

```
Combine item3 [temperature=hot] with item4 [temperature=lukewarm] then change item4 [color=red].
```

This defines two different areas of design: on the one hand, the designer devises the puzzle map and a series of puzzle patterns, which acted like the grammatical structure for the puzzle. The other part of the system consisted of a database of objects and characters. The generation takes place by selecting items in the database that fulfill the requirements of the grammatical structure of the puzzle.

The advantage of this system is that it sidesteps the need to have an AI solver, which checks whether the puzzles are solvable or not or whether there are duplicate objects, because the items must fit in the corresponding slot to be solvable, and once an item has been placed in a slot, it cannot be reused. If an object does not fulfill the condition that the puzzle pattern requires, it will not be placed in that slot, and if the generator cannot complete a map with objects that fulfill all the conditions, it will discard the generated puzzles and start over.

This system was the basis for the game *Symon* [7], which was developed by a team of students during the summer 2010. The premise of the game is that the player is in the dreams of a paralyzed patient; the dream-like quality of the puzzles would be supported by the procedural generation. For example, in a dream setting, using paint to change the flavor of a box of chocolate is plausible, giving the design of the game more room for experimentation.

## 3.1 Evaluation of the Initial system

The design of *Symon* succeeded in creating a point-and-click adventure game that was replayable. Each playthrough takes 5 to 15 minutes to complete, depending on the familiarity of the player with adventure games, and the game can be replayed hundreds of times. There are three pre-determined endings, so that each time the game starts, the system selects an ending; if the player completes the session, that ending is marked as complete so that in the next playthrough the system will select one of the other two endings. This system of endings creates a story arch that provides insight on the story—each ending refers to one aspect of the character's life.

Because of specific design decisions, which will be described below, the game also becomes relatively repetitive after several playthroughs (enough to see all the endings, and get a sense of all the objects). Some of the issues of the design and the development process derive from problems encountered during development.

One of the first problems was that there is only one basic puzzle pattern, which is a simple fetch quest.

```
NPC or station wants item1[state1] → player gets Item2 [state] or finalItem.
```

Example: Carnivore plant wants a box of chocolates [bitter] → get bouquet flowers [red]

This basic pattern admits variations that result in different patterns, depending on whether the state is relevant or not to complete the puzzle, for example:

```
NPC or station [state1] wants item1 [state2] to change NPC or station to [state3] → player gets [item2]
```

Example: Children [asleep] want music box [raspy] to make Children [awake] → get [family photo]

or

```
NPC wants Change item1 [state2] Item2 [state2] is changed by combining it with station or item2 [state2].

Give item1 [state] to NPC → get item2 [state] or finalItem.
```

Example: Quack doctor wants Diamond ring [blue]. Diamond ring [red] is changed by Desk Lamp [cold] Give Diamond [blue] to Quack doctor → get finalItem [window]

The puzzle map pre-determined in the form described in Figure 1, which the algorithm populates. The system starts by selecting the

end puzzle, which is giving three pre-determined items to Symon's doppleganger in the dream. The generator then proceeds backwards towards the initial state of the game, to have all the puzzles unsolved and all the items and characters involved in their initial slots. The final items are the reward for three separate puzzles. Each puzzle might also involve a fourth puzzle entailing changing a single property of the object that the NPC requires.

The system populates the map by trying different combinations until one fits; although it may not be a very computationally efficient method, it works for the purposes of the game. The fact that there is one basic puzzle pattern with slight variations, and that there are only 40 objects in the database makes this brute-force process not an issue during the puzzle generation. The focus of this experiment was having a working game, it was a design problem rather than a computational one. Improving on the puzzle generation algorithm is one of the aspects that will be addressed in future development.

The workflow created by the procedural generation was the most pressing problem during the process of development. The designer could not modify the puzzle patterns directly, which not only slowed down development, but also prevented iterating on the design. The designer only had direct control over the database in the form of a spreadsheet; incorporating the changes to the game was also a problem, because the spreadsheet could not be directly exported to the code and needed a programmer to integrate it in the game.

These problems with the game pipeline also got in the way of creating more complex puzzle patterns, and also combining them in different ways. Although this was not an obstacle for the algorithm to generate the puzzles, it was certainly the most salient issue to address. Thus the next step of the project became developing tools that would both improve the pipeline by giving the designer more control, and by allowing the designer to create more complex puzzle patterns and overall structures.

# 4. THE PUZZLE-DICE SYSTEM

The next step after the development of *Symon* was to work on a toolset that would facilitate the development of similar games. The goal of the project became developing a more generalized version of the system used in *Symon* to generate narrative puzzles for adventure games procedurally. The result was the initial version of the Puzzle-Dice system, which consists of a puzzle generation library and a basic editing tool for .NET platforms.

The initial basic approach was to separate puzzles into discrete reusable units (puzzle building blocks), which could be recombined and customized by designers making use of designer tools. This modular approach facilitates expanding and adding complexity to the system by adding building blocks one by one. The first version of the toolset was developed during the spring 2011, then put to the test during the development of another adventure game, *Stranded in Singapore* [8], during that same summer. The puzzle generator algorithm and toolset are being expanded and improved at the time of writing, as will be described in sections 4.1 and 4.2 below.

## 4.1 Part 1: Puzzle Generator Library

The following sections are a high-level technical description of how the Puzzle-Dice puzzle generator algorithm works, including the inputs that are given to the algorithm, the outputs that are produced, and advice on how these inputs and outputs can be integrated with a game in a general way. While the puzzle generator algorithm relies on the behavior of many puzzle building blocks, these blocks all follow a very similar process described in section 4.1.3.

### *4.1.1 Inputs to the Puzzle Generator Algorithm*

There are two main inputs to the puzzle generator algorithm: a database of items and characters (referred to as the "item database") and a recursively defined puzzle map representing the narrative structure of the puzzles to be generated. Additionally, the puzzle generator takes as input an item name as input to generate as the reward for solving the puzzle, and (optionally) any desired properties the reward should have. This reward can be replaced with something more concrete for an actual game, such as the game being completed, credits rolling, etc. To simplify the current implementation, the reward is an item the player receives upon completion of all the generated puzzles. This structure was mainly chosen to support the recursive nature of the high-level puzzle map.

#### *4.1.1.1 Database*

The item database is a collection of items that can exist in the game world along with their properties and relationships to other items in the database. An item database is most easily constructed using the database editor tool. The puzzle generator uses the item database to select items for its puzzles; it also uses the relationships between database items to filter out items that may not fit the puzzle being generated.

For example, for a Combine puzzle, where the player must bring two items together and the intended reward of the puzzle is "water", the puzzle generator will look at the "water" item in the database. The database information states that the water item can be the result of combining the items "hydrogen" and "oxygen". The puzzle generator can then use this information to create an appropriate Combine puzzle.

#### *4.1.1.2 Puzzle Map*

The puzzle map represents the structure of the puzzles that will be the basis for the game. Rather than a series of interconnected puzzle patterns as the one described in Figure 1, the map is a tree structure resulting from the combination of building blocks.

The puzzle structure can be viewed as a chronology of all the actions a player takes while moving from the beginning of the puzzle to the end of the puzzle. The puzzle map is composed of puzzles and areas and the connections between them. These components of a puzzle map are collectively referred to as building blocks in the library and tools.

An example of a more complex puzzle sequence than what was present in *Symon* that can be defined by the puzzle map is as follows:

```
NPC1 is in area1 → NPC1 wants item1
[state1] → player gets key to area2
→ NPC2 is located in area2 →  NPC 2
wants item2 [state2] → player gets
item2
```

example: Librarian is in the quad → Librarian wants a banana [green] → player gets key to the Library → Professor is in the library → Professor wants a book [old] → player receives a good grade.

#### *4.1.1.2.1 Puzzles*

Puzzles are objects that represent a single set of actions the player performs to get from a set of input items (hydrogen and oxygen in the example above) to an output item (the water in the example above)

#### *4.1.1.2.2 Areas*

Areas are objects that represent a set of "rooms" within the game world that are all accessible from each other. Figure 2 shows a basic puzzle map for a simple combine puzzle example.

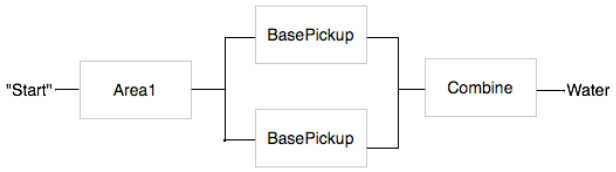


**Figure 2: Puzzle map for a simple combine puzzle with final output "Water".**

The original version of the puzzle map did not have any notion of where the generated items should be placed in relation to each other; this feature was required to implement the Door Unlock puzzle type easily and properly. In this type of puzzle type, having an unlocked door means that there is an area that is not accessible to the player, where there are objects that are out of reach; therefore the generator must make sure that the items behind the door are not part of the puzzle that requires opening that same door, making the game unsolvable.

In the current version, the puzzle map is created by the designer, thanks to the puzzle editor tool. Given a certain database, not all puzzle maps will be able to generate puzzles. Some maps will be able to generate more puzzles than others; the more items in the database and relationships between them, the more likely it will be to generate a complete puzzle map.

### *4.1.2 Outputs of the Puzzle Generator Algorithm*

The output of the puzzle generator is a list of items and relationships to spawn in the world.

#### *4.1.2.1 Items*

The list of items produced by the puzzle generator is a list of items to be spawned. Each item also has a list of properties for the item to possess, as well as relationships to other objects. For example, items of the container type may also provide a list of items they can initially contain.

Items also provide a unique ID representing the area in which they should be spawned. This information is mainly relevant for Door Unlock puzzles, as the items needed to unlock a door must by necessity be accessible without opening the door; the information also comes into play for puzzles that require non-carryable items to be in a specific room to solve the puzzle.

#### *4.1.2.2 Relationships*

The list of relationships produced by the puzzle generator represents a list of relationships necessary to produce the puzzles created by the generator. "Relationship" is a loose term that can refer to a number of different things. Relationships are similar to items in that the underlying game can define the exact way in which relationships are spawned within the game.

- Combine Relationship

  Combine relationship represents a relationship between two items that can be combined to produce a third item. The relationship itself provides the names of the two "ingredient" items along with a copy of the "reward" item (and all properties it possesses). For example, a combine relationship could exist between a "hydrogen" item and an "oxygen" item that produces a "water" item when the two are used together.

- Property Change Relationship

  Property Change relationship represents a relationship between two items, where one item can be used to change the property of the other. The relationship provides the names of the "changer" and the "changee" along with the name and desired value of the property to change. For example, a property change relationship could exist between a "red egg" and "green dye" that changes the color property of the egg to green when the items are used together.

- Insertion Relationship

  An Insertion relationship represents a relationship between two items where one item can be inserted into the other. For example, an insertion relationship could exist between a "water" item and a "bottle" allowing water to be inserted and removed from a bottle in the player's inventory.

- Request Relationship

  Request is a special kind of relationship that represents a set of conditions within the game that are constantly being checked. When the conditions are met, something changes in the game. For example, an Insertion Request might constantly check to see if a certain container has been filled with a certain item. When this happens, the player may receive an item or a door may open. For example, a "Professor" NPC could request a "book" from the player and provide a "key" as a reward.

- Area Connection Relationship

  Area Connection relationship represents a required connection between two areas. While extra connections between areas can be created, the area connection relationship outlines a specific connection that is needed for the puzzles to be solvable. The relationship provides the unique IDs of the two areas requiring the connection.

- Start Area Relationship

  Start Area relationship identifies the area in which the player begins the game. It simply provides the unique ID of the area where the player should be spawned.

### *4.1.3 General Pattern of Puzzle Algorithms: How the Puzzle Generator Works*

The actual details of most of the puzzle generation depend on the building blocks used to construct the puzzle map. Fortunately, almost all of the building blocks follow a very general pattern when generating puzzles.

When a building block is asked to generate a puzzle, it is given the following parameters: the name of the desired output of the puzzle and a list of desired properties for the output to have. With this information, the building block then follows the steps outlined

below. If the building block happens to fail during any of these steps, it passes the responsibility of handling the failure to whatever object asked it to generate a puzzle (be it another building block or the main puzzle generator algorithm). As will be shown below, when inputs to a building block fail, the building block simply tries another combination until it runs out of options, at which point it fails itself.

#### *4.1.3.1 Step 1: Attempt to generate the output*

The building block attempts to generate the output requested of it with the desired properties. If for some reason the output cannot be generated (i.e. the output was already spawned by another puzzle or the output cannot possess the desired properties), the puzzle generation fails. Depending on the building block, the output is not always spawned. For instance, a Property Change puzzle will have its inputs generate the desired output while it merely generates a Property Change relationship.

#### *4.1.3.2 Step 2: Attempt to generate the input*

After the desired output has successfully been generated, the building block then uses a series of filters to construct a list of items in the database that might work as inputs to the puzzle. These filters include:

- Filters internal to the particular building block. For instance: a Combine Puzzle will only consider items that can be combined to produce the output.
- Filters dealing with the desired properties of the output. For instance: a Property Change Puzzle usually has a specific property it hopes to change.
- Filters derived from user-defined constraints.

Once a filtered list of items has been produced, the building block randomly shuffles the list and tries to have its inputs generate the items. Its inputs, themselves building blocks, attempt to generate these items as their outputs. If its inputs fail, the building block simply moves on to the next item in the list. If it runs out of items in the list, the building block itself fails.

Once all of its inputs manage to successfully generate items, the building block has succeeded in generating a puzzle and moves on to step 3.

#### *4.1.3.3 Step 3: Build the Relationships*

Using the successful inputs and outputs it managed to generate, the building block then creates relationships between the inputs and outputs relevant to the type of building block (i.e. a Combine Puzzle will generate a Combine Relationship). The building block has now successfully generated a complete puzzle. It returns a list of items and relationships composed of the items/relationships it and its inputs generated during the entire process.

### 4.2 Part 4: The Database Editor

The database editor is a tool for designers, which allows the user to create and edit a database of items for the generation library to construct puzzles. Since Puzzle-Dice expands a system for procedurally generating narrative puzzles, the database editor is the tool that allows designers to explicitly link potential puzzles with narrative by adding textual descriptions and hints to the items and characters that the player might encounter. Currently, the database editor exists as a .NET Windows application (see Figure 3) with simple spreadsheet controls and a limited amount of user customizability.

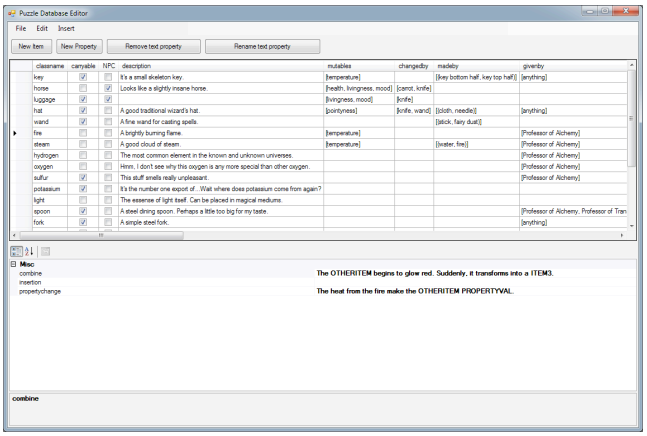


**Figure 3: Screenshot of the prototype database editor.**

### 4.3 Part 5: The Puzzle Editor Tool

The puzzle editor is a visual tool for designers, which allows a user to create and edit different puzzle maps. These maps act as a description of the general shape of the puzzles for the puzzle generator to construct. For example, Figure 2 above is a graphical representation of a very basic puzzle map. The puzzle editor tool can also be used to construct an undirected graph representing a map of areas the player can explore. The initial prototype of the tool was built as a .NET Windows application for the development of the game *Stranded in Singapore* [7]; which has been later rebuilt as a cross-platform application (see Figure 4) with a stronger UI and new features including a textual representation of the puzzle map.

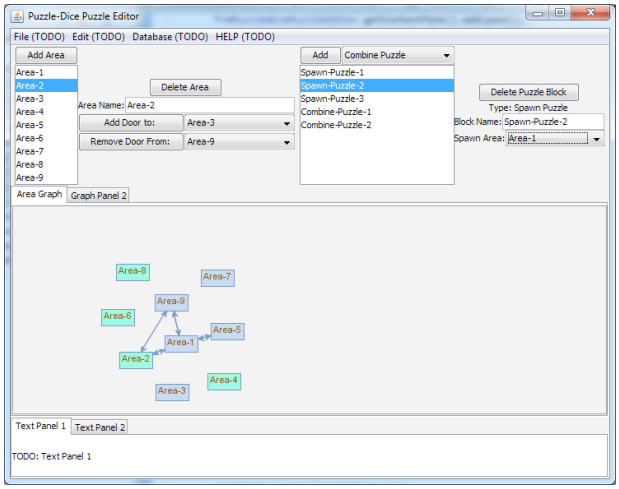


**Figure 4: Screenshot of the prototype puzzle editor**

## 5. Initial Evaluation of the Puzzle-Dice System

The initial set of tools was used and expanded on during the development of *Stranded in Singapore* during the summer of 2011. The tools certainly helped improve the pipeline, providing the designer with direct access to the code, and being able to change puzzles independently of the programming team. This allowed for iteration on the puzzles, a basic process of design that is actually difficult to incorporate to the development of point-and-click adventure games in general, since they depend on assets so heavily.

*Stranded in Singapore*'s scope is comparable to *Symon*: each playthrough takes 5 to 10 minutes to complete, depending on the abilities of the player; the number of database items and assets is also 40. The advantage is that the tools helped create a wider variety of puzzles, including the Door Unlock puzzle type, where players do not have access to a specific area until they solve a puzzle. On the other hand, the way the generator bases its generation on trial and error can also be a problem when generating larger games, so that certain puzzle maps can take longer to generate. This is not a major problem with a game of this scope, but may become unwieldy for larger games.

## 6. Further work

The ultimate goal of this project is to release a polished procedural puzzle generation system under an open source license, which can be used in combination with other game engines. The puzzle generation algorithm was initially developed in Action Script 3.0; the new version was coded in Python and then exported into Action Script for *Stranded in Singapore*. The goal is to make the algorithm work in other engines or development environments, such as Inform, Adventure Game Studio, Unity, or XNA.

Current work focuses on improving the UI of the designer tools to make them more accessible to designers, as well as optimizing the puzzle generator to create new types of puzzles. In preparation for releasing the tools to the public, the plan is to make a new cross-platform database editor with a more polished UI and optimized performance.

## 7. ACKNOWLEDGMENTS

The authors wish to acknowledge the groundwork of the development team of *Symon*, particularly programmers Caroline Sugianto and Kang Hean Jin and designer Justin Tan, for having worked on the first version of the generator system.

Thanks also to Michaela Lavan, co-developer of the Puzzle Dice generator system in its first implementation.

The *Stranded in Singapore* development team also put the Puzzle Dice system to the test and worked on some of the early improvements to it during development. Special thanks must go to programming team Adin Schmahmann, Abhishek Ray, Lee Weizheng, as well as designer Teng Howe Lim.

## 8. REFERENCES

[1] Ashmore, C. 2006. *Key and Lock Puzzles in Procedural Gameplay*. Georgia Institute of Technology.

[2] Compton, K. and Mateas, M. 2006. Procedural Level Design for Platform Games. *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE)* (Jun. 2006).

[3] Doran, J. and Parberry, I. 2011. A prototype quest generator based on a structural analysis of quests from four MMORPGs. *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games* (New York, NY, USA, 2011), 1:1–1:8.

[4] Dormans, J. 2011. Level design as model transformation: a strategy for automated content generation. *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games* (New York, NY, USA, 2011), 2:1–2:8.

[5] Fernandez Vara, C. 2009. Play's the Thing: A Framework to Study Videogames as Performance. (Brunel University, West London, 2009).

[6] Harrell, F. 2005. Shades of Computational Evocation and Meaning: The GRIOT System and Improvisational Poetry Generation. Proceedings of the 6th Digital Arts and Culture Conference, 133-143.

[7] Sugianto, C. and Kang,, H.J. 2010. *Symon*. Singapore-MIT GAMBIT Game Lab.

[8] Ray, A. et al. 2011. *Stranded in Singapore*. Singapore-MIT GAMBIT Game Lab.