Creating Dreamlike Game Worlds
Through Procedural Content Generation

Clara Fernández-Vara
New York University
clara.fernandez@nyu.edu

Abstract
This article describes the process of designing a point-and-click adventure game that aimed at using dream logic as the basis to create its narrative puzzles. The technical solution to tackle this challenge was using Procedural Content Generation (PCG) as the design approach, which was used expressively to recreate the instability and changeability of dreams. Although PCG brought about replayability to the game, called Symon, it also created a series of other development problems, which had to be remedied through other design devices. One of the lessons learned during the development of the game is that PCG is not a blanket solution to problems, but rather an expressive tool to be used in combination to other design strategies; human factors are also key both during the development and reception of narrative video games.

Introduction
Innovation in narrative games often starts with a system rather than a story concept. The project here described aimed at generating new game mechanics based on a narrative premise, namely modeling how dream logic would work in the context of a point-and-click adventure game. The idea was designing narrative puzzles that were based on the logic of dreams, where events and solutions to problems make sense but not quite, given the unstable nature of the dream world.

Procedural Content Generation (PCG) was the design approach chosen to generate the puzzles. The puzzle generation system would model the mind of the person in whose dream the player is in, both incorporating the randomness that characterizes dreams and providing the consistency that would make the puzzles feel logical. Thus procedural generation would be used as an expressive tool, apart from generating larger content with a limited number of assets. PCG was also an attractive option because it could give way to a new type of adventure game, which would present different puzzles whenever a game starts. Replayability is not one of the strengths of story-driven games, as Sullivan et al. argue (2011), and especially not of point-and-click adventure games, so exploring the design possibilities of PCG seemed like a further advantage.

The result of this experiment was the game Symon, a web game that was released on a research website but played and played alongside commercial games. Our IRB protocol allowed us to gather the comments of players in public online forums; the original web version of the game also noted that the game was part of a research project. The game thus was also played outside a research lab context, which facilitated obtaining initial feedback from players, who posted comments and walkthroughs on forums without paying attention to the fact that the game was a research experiment.

This paper is an overview of the design and development process of Symon, as well as its extended version, Symon 2.0. This case study describes the difficulties of incorporating PCG into the design process of point-and-click adventure games. The core of the discussion is to examine how procedural generation poses a design problem, rather than a computational issue, and must be complemented by other design solutions to achieve the intended goal, in this case, recreating the logic of dreams in a point-and-click adventure game.

Previous Works
Using computation to create dream-like narratives already has a tradition in the field of computer science. Mueller (1985) designed the Daydreamer system, which explored the workings of daydreaming as a series of recalls or imaginings based on previous experiences. This system
generated a series of daydreams based on a given set of past events, in which the protagonist would imagine alternative reactions and paths to those events. Following Mueller’s trail, Pérez y Pérez and Sosa (2007) designed the Visual-Daydreamer system to generate daydream narratives, this time producing a visual output rather than textual. Using iconic representation, the output of the system elicits an interpretation from the viewer, who fills the gaps and constructs a story (or at least tries to) based on the visual cues. In a similar vein, Zhu and Harrell (2008) used the GRIOT system to create the story of the robot Ales, a micronarrative where the actions of the character intertwine with its daydream memories. (The GRIOT system will be described in more detail below.) The novelty of the Ales micronarrative is that this system admits user’s input in order to choose the actions of the robot. The system also incorporates a scale of intentionality, which allows the player to provide input to the system, but also determines how relevant the input is to the generation of the daydream. The lowest settings on the scale base the reactions to events to a minimum and are based on user’s input, while at its highest, the character takes over the control and provides its own input based on its emotional state.

The systems described above focus on creating narratives, but are not digital games. They also base their generation on a database of past events, incorporating planners that determine the events that take place in the daydream. What these systems evidence is the suitability of PCG to generate dream-like narratives, and how they encourage user’s interpretation—as Pérez y Pérez and Sosa (2007) demonstrate, it is the users who evaluate the dream-like qualities of the output of their system. The discussion below demonstrates that an adventure game does not quite need as complex a system to generate dream-like puzzles, since players have to solve them and thus bring their own interpretation of what the problem may be.

In the specific field of computer games using PCG, there is also a long tradition started with games like the dungeon crawler Rogue (1980). Procedural generation of game spaces and levels, as well as mathematical puzzles, has become commonplace, while incorporating narrative into procedurally generated content is still a challenge. Previous academic research has focused mainly on procedural level generation for video games (Compton 2006; Dorman 2011); other specific work on procedurally generated content is specific to puzzle research: Ashmore (2006) focuses on “lock and key” puzzles while Doran and Parberry (2011) have constructed a procedural quest model based common structures of quests from MMORPGs.

Two holistic approaches to using PCG for narrative games are Harsook et al (2011), a system to generate game worlds and quests for CRPGs, and Sullivan’s Grail Framework (2012), a set of tools to facilitate the creation of narrative games, where designers create a set of high-level rules and the tools help generate the narrative quest that players have to tackle. While these projects also aim at creating story-driven games, the generation of quests as playable stories is different from what Symon’s goal, that is, generating narrative puzzles, which are a problem with a specific solution.

Another game project using PCG to create game narratives in a dream world is Endless Web, a platforming game that generates different levels depending on how the player plays (Smith et al 2012); the game utilizes PCG as similar expressive device, but within a different genre.

The project here described brings procedurally generated content together with narrative-driven puzzles in adventure games, rather than generating narratives or game levels. 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 (Fernández-Vara 2009). 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, they are usually predetermined by the designer.

The challenge of developing a point-and-click adventure game based on dream logic derives from using PCG to create the challenges, since the puzzles have to be solvable and yet should also evoke a sense of not quite making sense. In order to demonstrate the efficiency of PCG, the game also had to be replayable and avoid repetition as much as possible. As these pre-existing works demonstrate, generating gameplay elements or even the narrative is not a problem; the experience of developing Symon shows that creating narrative puzzles that are solvable, enjoyable and intelligible is the real challenge.

**Dream Logic Mechanics**

The project started with several paper prototypes, which explored different ways of understanding how a dream narrative could become playable. The initial approach was creating games that took place in a dream, and would use dream imagery, such as stubborn shopping carts that had to be herded into their rack. This approach, however, relied on the representation to be expressive, but would still be based on pre-existing mechanics that were not dream-like. In order to reach a breakthrough, the key was to understand how dreams worked.

Psychology and neuroscience provided this key: Hobson (1988) argues that the dream experience is delusional, because there is a lack of insight, and the events are accepted uncritically. This absence of insight is a serious hurdle in order to create a puzzle, since one of the
characteristics of puzzle-solving is that it requires insight thinking to solve the problem posed (Danesi 2002). The issue therefore was to evoke dream-like states but maintaining insight thinking—the moment of insight is pleasurable, and one of the reasons humans enjoy solving puzzles. Without insight, there was no game.

According to Hobson, the delusional state during dreaming is part of a larger set of cognitive disturbances, such as non-sequiturs – changes of subject, actions, settings, narrative – defiance of physical laws, and vagueness in the explanation of events. This list of elements provided a series of narrative devices that can evoke the way dreams work, even if insight needs to take place in order to advance in the game.

Thus there had to be a middle ground where we could provide the player with pleasurable narrative puzzles, and yet still evoke the sensation of being in a dream. The idea was to create a system that would be based on the dreamer’s brain: the story of the dreamer, the memories, emotions and mental states would be the information that would be used by a procedural generation system. Solving a puzzle would not only provide insight about the specific problem posed, but also on the dreamer’s personality and their life story—the solution to the puzzles would help getting to know the character.

Procedural generation would provide the instability that characterizes dreams—the game would be different every play-through; the relationships between objects and situations would not have to be explained completely, the actions would not have to follow the laws of physics. For example using a banana to paint a wall yellow as a way to gain access to a room fits with the kind of cognitive disturbances that Hobson describes. How to make this type of action a meaningful part of a narrative game was a further design challenge, which was addressed with other design strategies described below.

Although PCG could provide some of the intended expressive features to produce the puzzles, it also became clear that the narrative of who our dreamer was had to be predetermined and not procedurally generated. The memories and life events of the dreamer had to be written beforehand, in order to create a repertoire of events and objects for the system to generate the puzzles. Thus there would be two narrative levels: the story of the character, which would be embodied in the puzzle generation system, and the story of the player, which would be how players revealed and interpreted the story of the character after successive play-throughs.

### Puzzle Patterns

In order to generate the puzzles procedurally, first we had to come up with an abstract model of how adventure game puzzles would be generated. There are many types of narrative puzzles in adventure games, such as:

- 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 (Fernández-Vara 2009) and constitute the foundation of the puzzle-generation system. The gameplay would be based on a set of puzzles, which would concatenate these patterns to make up the structure of the game—the puzzle map. The map establishes the relationships between puzzles, so that solving a puzzle provides an item or information to solve another one. 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 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.](image)

### The Generation System

The structural idea was inspired by the GRIOT system (Harrell 2005). Initially created to generate poetry procedurally, GRIOT allows poets to define first a general structure of stanzas and topics, which is the higher-level structure that will be populated with sentences. In the Ales
example above, which used the GRIOT system, the past-
story of the robot had been determined beforehand, while
the events and memories were recalled through the system.
This approach seemed to fit very well our purposes of pre-
defining the story of our dreamer beforehand, which the
player would then discover through puzzle-solving.

In its initial conception, GRIOT 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. The puzzle-
generating system is a simplified version of the high-level
structure of GRIOT.

Each of the characters and objects to populate the
pattern would have a set of properties, for example:
character, temperature, color, taste, portability. 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 system defines two different areas of design—on the
one hand, the designer devises the puzzle map and a series
of puzzle patterns, which act like the grammatical structure
for the puzzle. The other part of the system consists 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 a system of this type 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.

Symon 1.0 - The Puzzle Generation System

These high-level concepts resulted in the first version of
the game Symon. The premise of the game was that the
player is in the dreams of Symon, a paralyzed patient in a
hospital bed. In his dreams, he can move around and revisit
dream-like events of his life, which represent his hopes and
frustrations. The items and characters in the database are
drawn from the backstory of the character, which was
written beforehand. As we will see below, the relationship
between objects in the puzzle patterns also creates some
patterns that express who Symon is.

Symon succeeded as a proof of concept for a replayable
point-and-click adventure game. 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 play-
through the system will select one of the other two
endings. This system of endings creates a story arc that
provides insight on the story—each ending refers to one
aspect of the character’s life.

Because of specific design decisions, described below,
the game also becomes relatively repetitive after several
play-throughs, although it is long enough to see all the
endings, and get a sense of most of the objects. Some of
the issues of the design and the development process
derive from problems encountered during the
implementation of the system.

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 is 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 narrative game, it was a design problem rather than a computational one. Improving on the puzzle generation algorithm was one of the issues that was addressed in later work (see Thompson 2013).

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 puzzles, it was certainly the most salient issue to address. Thus the next step for future development 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. These tools have been described elsewhere (Fernández-Vara and Thomson 2012).

The problems during development had a further and more serious consequence: there was hardly any iteration during the process. The two main play-testing sessions gave information about how to improve the user interface, but the puzzle generator was not in a state where the puzzles were intelligible or varied enough. The game only came together in the last week of production, and a last-minute informal playtest allowed us to see that the game worked and players were engaged solving the puzzles. The playtest was too short to evaluate whether players would replay the game or if they even realized that they could, but at least we learned that the proof of concept worked as a playable, solvable game.

**Evoking a dream-like narrative**

In spite of the limitations of the system, the designer still had room to work puzzle design into an expressive narrative element. Using a basic fetch quest, the designer could establish certain relationships between the characters and the objects they were requesting. For example, the carnivorous plant represents how Symon hates to do the gardening. The plant requests items harm it, such as poisonous pills or chocolates. In contrast, when the children appear in the dream, the objects they request have positive effects on them. Therefore, in defining the properties of the objects an NPC may request, the game also reflected the emotional relationship between Symon and the object or NPC.

Apart from procedural generation, the game uses other design devices in order to evoke the qualities of dreams. Since the process of achieving insight had to be preserved for the game to be playable, we based other elements of the game on the kinds of cognitive disturbances that Hobson listed. The game space is structured as different interconnected rooms (see Figure 2 for sample map of the space). All the locations seem to be outdoors, with doors serving as the connection between them. The rooms do not change places during the same play-through and there is no thematic connection between them; their meaningfulness derives from the relationship to different times and memories in Symon’s life, which are also associated with specific NPCs. For example, in the graveyard, a gravedigger asks Symon for some object that he may be emotionally attached to in order to bury it.

The player always starts in the same room, with Symon’s doppleganger lying in bed and providing the list of items that he needs to achieve peace of mind. There are always four other locations, adjacent to each other, that are generated and also change from game to game, depending on which NPCs are involved in the puzzles. For example, if the generated puzzle involves the gravedigger, the game space will include the graveyard; if the frog is one of the NPCs, it will appear in the pond. Each room also had spots for the items that the player would have to pick up. In either case (allocating the rooms or the characters) there was no need to have an algorithm limiting where rooms where or objects may appear, since randomness already provided the type of non-sequitur that characterizes dreams.

**Figure 2: Sample map of the space of Symon.**
The way the puzzles are solved can also defy physical laws—the player turns soups from bland to spicy using a bouquet of flowers, or change plot of a book from happy to melancholy using a fan. If the game were set in the real world, it would need a lot more constraints and a more complex generating system; as it is, these actions make sense in context.

The final design decision that helps evoking cognitive disturbances is the fragmentation of the narrative. Although us developers knew what the story of Symon is, the generator only provides fragments of it, associated with the NPCs or the objects. Descriptions of two characters as “the children I never had” hint at what their significance in his life is, but it does not explain the whole story. By replaying the game, more fragments of the character’s life surface, but always leaving it up to the player to complete the story.

**Initial evaluation**

After its initial release, the game quickly became popular in game review sites. The overnight success of *Symon* brought about an invaluable source of data: the comments sections of review sites, as well as browser game forums, all across the world. Our players were tackling the game like any other browser game, without thinking about it as a research game, even though it was featured in the website of an academic project on a page that explained that the game was a research experiment. Playing happened at home rather than in a research lab, providing us with tremendously useful data.

Players used the comments sections of websites and forums to discuss the game, and try to help each other find the solution. These forums worked as transcripts of how players were tackling the game, and how they were trying to solve the puzzles; it was all relevant information on where and how players were having problems understanding the game.

Since players did not pay attention to the research context, most of them were unaware that the game was replayable. This is a problem stemming from innovating within the conventions of a well-established genre, which creates expectations about what can and cannot be done; since point-and-click adventure games are not known from their replayability, players did not expect variability in the puzzles. Smith (2012) ran into similar issues while playtesting *Endless Web*—players missed that the levels of the game would change if players retraced their steps, but since most platform games expect players to keep playing, they missed the core of the innovation of the project.

The pre-designed endings provided an incentive for replay—once the player has finished a walkthrough, a new picture will appear in the initial screen, up to three pictures to mark one narrative arc. The player can continue playing after completing three walkthroughs, the generated puzzles will still be different, but the set endings will repeat.

The comments and forums evidenced that the game defied their expectations about how to play a point-and-click adventure game. Players usually start describing what objects they got and what they did, for example: “not sure what to do with the happy wind up box. Also still have pills, although used once” or “found flowers placed under water and used on grave to get baby outfit have ring and chocolate have used both under the lamp to turn them red.” They believe that they are writing the walkthrough, but very soon they realize other people have different objects and are meeting different characters. Since they were not in the same physical space, they could not see each others’ screens. A common response from players is utter confusion: a very telling interjection is “Which fan? help!” The walkthrough gets eventually written, although not as a series of solutions for fixed puzzles, but rather as an explanation of how the puzzle generation system works, or at least part of it. For example, one forum poster writes this as a solution, knowing that this only accounts for part of the system:

“To make things HAPPY
Look for something YELLOW or ORANGE
To make things HEARTBROKEN or HEALTHY
Look for something BLUE
To make things RASPY or AWAKE
Look for something BROWN
To make things SPICY or HOT
Look for something RED
To make a man eating plant cough up the goods
Look for something to make it sick”

The pattern of utter confusion before finding the solution, expressed as a breakdown of how the system works, repeats through different forums and in different languages. The fact that players partly figure out the puzzle system is not a problem, since the achievement and the innovation is that players engaged with a point-and-click adventure game in a novel way, and managed to follow along the change in the conventions.

Surprisingly, there were very few comments on the repetitive nature of the puzzle pattern, perhaps because it is also one of the dominant design patterns in point-and-click adventure games. The randomness of the puzzles does not seem to bother players, perhaps because some adventure games are notorious for presenting random, and often illogical, puzzles.

Forums also revealed some of the issues that we had encountered during internal testing. The worst one was players getting to an unsolvable state, where they changed the property of an object that should not have changed. The game menu gave players an option to reset the game, but the option was mostly missed.
Symon 2.0

We had the opportunity to work on the game for another semester, even after the game had officially shipped, with a small student team working only 10 hours a week. The changes and improvements in the game were mostly based on the feedback from game forums. For the purposes of this article, the features that were tackled during this extra production time was the inclusion of another puzzle pattern, and the option to undo a changed object, and extended endings.

The first and most ambitious change was including a new puzzle pattern. Still a fetch quest, it involved a container object, which could “transport” a property from one station object to another station / an NPC. The pattern went like this:

NPC or station wants to be in [state2]
ContainerItem1 [state1] is changed by combining it with station → ContainerItem[state2].
Give ContainerItem1 [state2] to NPC → get item2 [state]
or finalItem.

Example:
Soup wants to be [spicy].
Cotton [sweet] is changed by Fountain [spicy] →
Cotton[spicy]
Give Cotton [spicy] to Soup → get finalItem [window]

The game had all the assets for the necessary objects as well as description texts, but had to be cut because it would make the game crash. The extra time allowed us to find the cause of the crash, and reinstate the container puzzles, which suddenly brought about a refreshing variety to the gameplay. Having an extra type of puzzle changed the pacing of the game, and invited players to be more careful in changing the properties of items. However, given that we only had two objects that could be containers, and two stations – fixed objects that the player could not pick up – that could change them, the new puzzle pattern only generated eight new puzzles. Since the puzzle map was hard-coded, and we could not leave the pattern out, very similar patterns would repeat in every play-through, revealing the limitations of that particular pattern very fast. Since we did not have the time to produce new visual and audio assets for this pattern, it had to be left out again.

We needed a feature that would allow players to undo changing the state of an object, which had become evident after reading the comments of players, and it was an easier feature to implement. A new type of station was included in the form of a grandfather clock, so that it would appear in every play-through. By making it an item in the game instead of a menu button, we made sure that players would see the new item and interact with it. If a player wants to restore an object to its initial state, they can use the object with the clock, which will “rewind” it to its original state.

The last significant feature implemented was a final set of hard-coded endings, using the objects that the player has to give to the doppleganger in order to finish the walkthrough. Based on the feedback from players, once the three objects were given away, the game ended too abruptly. Thus we included a screen for each ending, where the player had to use the three final objects to solve a pre-designed (i.e. not procedurally generated) puzzle, since there was no time to make these puzzles procedural as well. By having to use the objects in a puzzle, players obtained a bit more information about their importance, and enacted the steps that would help Symon achieve his peace of mind.

Symon 2.0 was published in the commercial website Kongregate. This version is still getting comments from players, although these comments have not been as useful to help us understand the reception of the game. Players focus on the tone of the game and the themes, rather than the mechanics; only a few of players commented on the procedural nature of the game, usually in a positive manner. The lack of comments from players stating they got stuck makes us guess that the undo feature is useful.

Continuing work

The development of Symon demonstrated the necessity to develop better tools to incorporate PCG into point-and-click adventure games. An early version of those tools was put to the test in the game Stranded in Singapore, which is also briefly discussed in Fernández-Vara and Thomson (2012). The final version of these tools has taken the form of a toolset to produce narrative puzzles across game genres (Thomson 2013).

The only aspect of Stranded that is relevant to the current discussion is that it uses PCG as a device to bring about replayability to a game based on the real world instead of a dream. The expressive effects of PCG did not quite mesh with the game itself: procedural generation undermined the believability of certain aspects of the game that were based on real-life objects, but which may be exotic for people who are not familiar with Singaporean culture. For instance, one of the puzzles involves preparing an ice-cream sandwich, which involves using sliced bread with a square of ice-cream. This item is actually sold in the streets of Singapore, but players believed that it was a random puzzle.

Stranded in Singapore did not utilize PCG effectively as an expressive device as Symon. Although initially the development team believed the random associations would have humorous effects, that did not quite happen, because the puzzles modeled situations based on the real world, in this case in the city-state of Singapore.
Conclusion

This paper has described the advantages and issues PCG into the design of point-and-click adventure games. Generating the puzzles is not the biggest challenge, since there is already a tradition of generating both narratives and game spaces, but making the generated content both playable and meaningful.

Although initially PCG was approached as a solution to the problem of how to generate mechanics in adventure games based on dream logic, it was not “the” solution. PCG is a tool, with expressive possibilities that related to how dreams work, and had to be supported with other game design choices, as well as trusting players to make narrative sense of the generated puzzles. The resulting game was achieved the goals of the project, in that the novel design was engaging, defied players’ expectations, and yet they did manage to traverse the game.

References


Crowther, Will and Don Woods. 1975-6. Adventure / Colossal Cave Adventure.


