Lost Rituals: Generating Text Using Behavioral Data Objects

B.T. Franklin, BACS
Dunesailer Research, Phoenix, AZ, USA
www.dunesailer.science
e-mail: brandon.franklin@gmail.com

Abstract

Many systems have been created for the purpose of generating interesting, novel, entertaining, or insightful text. However, a common shortcoming of these systems is that they do not often include internal structural context about the objects and concepts being described by the generated text, and therefore produce spontaneous gibberish or nonsense, or even simple grammatical errors. This undermines the illusion that the text was generated by a thinking individual and exposes the fact that the output was produced by an algorithm rather than a mind. We describe the system implemented to drive the "Lost Rituals" application, which attempts to address these shortcomings using a complex framework of modeled objects. We describe the implementation of narrator backstory and voice selection as a method for deepening user perception of experiential validity and realism. We explain the technique of using the association of state, behavior, and modification capabilities of represented objects to the generated text as a path to improving descriptive coherence and avoiding or minimizing the loss of suspended disbelief. Examples of generated output are provided, along with discussion for areas of future improvement and exploration.

1 Introduction

Conceptually, the most common forms of artistic outputs from procedural generation systems can be broadly categorized into visual art, musical art, interactive experiences (such as games), architecture, and language-based art such as literature. This final type can be seen through many lenses, with varying levels of conceptual coherence and the presence or absence of traditional narrative structure. When narrative structure is present, maintaining coherence throughout the narrative is an especially difficult challenge. Consider, for example, that at the time of this writing, there has never been a convincingly human-competitive novel-length narrative text story produced by a procedural system.

The NaNoGenMo project, for example, has, since 2013, encouraged participants to write program code that generates a 50,000-word novel[1], and many entries have been created since the project's inception[2], but none of these has ever been a complete human-competitive narrative. The majority of these novels are created through what are often creative and clever structural tricks, none of which depend on narrative plot advancement throughout.

We have endeavored to develop a possible route to achieving the goal of long-form generative text with a coherent and unified narrative. "Behavioral data objects" are programming structures that combine state-related data and evaluation logic with the generation of specific textual segments that can be

used to describe the data, changes in its state, and any associated actions that might be appropriate to the generation goal. It is our position that this strategy has some unique advantages over the most common existing strategies.

2 Common Existing Strategies

There are three primary strategies that appear dominant in the world of procedural text generation: context-free transformation grammars, Markov chains, and neural or deep learning techniques. While each of these approaches can sometimes produce impressive results, each has a tendency to produce output that is revealed to be essentially gibberish upon close reading. The reasons for this vary based upon the system being used.

2.1 Context-Free Transformation Grammars

Context-free transformation grammars rely on a repeating loop of replacements, where symbols are replaced with other symbols, typically drawn from a collection of available options[3]. A very popular system based on this approach is Compton's *Tracery*[4], which has been ported to many programming languages. Context-free grammars suffer from exactly what their name implies: they do not contain context. This makes the production of coherent narrative events especially difficult. Short, standalone snippets of text can be generated easily, and are generally syntactically correct, but the transmission of state and meaning from artifact to artifact is often lost. *Tracery* attempts to overcome this challenge, at least in part, via the use of *ad hoc* variables, allowing already-selected replacement symbols to be consistently re-used in later text. However, what this solution lacks is transmission of state and metadata, since the generated symbol is stored and represented only in its textual form. There is no underlying justification for the selection of any given symbol, and therefore no such data to be conveyed to later pieces of the generation process to maintain context.

2.1.1 Example Output

The following are some examples of generated text using a context-free transformation grammar.[5]

An owl is almost always wistful, unless it is a grey one.

A duck is often indignant, unless it is a purple one.

A unicorn is rarely wistful, unless it is a green one.

An eagle is sometimes vexed, unless it is a grey one.

2.2 Markov Chains

Markov chains are state machines whose state transitions are controlled by stochastic probabilities.[6] In the realm of text generation, a Markov chain is typically configured such that a given state produces a specific word or character as output, and then transitions to another state, whose output is appended onto the text, continuing until some termination state is achieved. The construction of the chain's configuration can be done either manually or (more commonly) through training on a corpus of existing text. The probabilities of transitioning from any state to any subsequent state are modeled upon the probabilities detected in the corpus. When training has been completed, the chain can be used to very quickly produce text output that resembles the corpus in the sense that it will reflect approximately the same state transition probabilities. While this approach is able

to mirror the “feel” of the input corpus, the transition from state to state generally does not convey any contextual information, since each state is arrived at purely through a probability based on the starting state. Even if such contextual information were transmitted across states, it would be nearly impossible to use the contextual information in a meaningful way, since the training of the chain cannot reasonably take context information into consideration when analyzing probabilities. The end result of this is that Markov chains tend to produce output text that meanders through linguistic non-sequiturs, following the most superficial transitional structures of the corpus, but completely failing to replicate the relationship to any underlying information being described. It is, put simply, just a bunch of words strung together.

2.2.1 Example Output

The following are some examples of generated text using a Markov chain.[7]

'I am a master armorer , lords of Westeros , sawing out each bay and peninsula until the'

'Jon Snow is with the Hound in the woods . He won’t do it . " Please don’t'

'Where are the chains , and the Knight of Flowers to treat with you , Imp . ”'

'He thought that would be good or bad for their escape . If they can truly give us'

'I thought that she was like to remember a young crow he’d met briefly years before . “'

2.3 Neural and Deep Learning Techniques

The use of neural-network driven, or more precisely “deep learning,” techniques is one of the more promising and modern approaches being used for generative text production at the time of writing, and there has been a fair amount of success at producing high-quality results using these techniques[8]. The use of deep learning allows for the generative system to be trained against a truly massive corpus of example texts, and to identify complex and multi-faceted structures within the corpus that can be used when generating output. However, while the generated text is certainly able to produce what appears at first glance to be very convincing language, deeper reading will often reveal that the text is actually meaningless, often self-contradictory, and incoherent. As with the previous strategies, the underlying reason for this shortcoming is that while the structure and appearance of the language is being learned, the meaning behind it is not. The generator is producing output without any actual conceptual thread holding it together. It is, at the bottom, simply producing a very opaque simulacrum of a coherent linguistic structure, complex enough to not reveal itself immediately, but breaking down entirely upon careful reading.

2.3.1 Example Output

The following text was generated using a deep-learning engine.[9]

QUEENE:

I had thought thou hadst a Roman; for the oracle,

Thus by All bids the man against the word,

Which are so weak of care, by old care done;

Your children were in your holy love,

And the precipitation through the bleeding throne.

BISHOP OF ELY:

Marry, and will, my lord, to weep in such a one were prettiest;

Yet now I was adopted heir

Of the world's lamentable day,

To watch the next way with his father with his face?

3 *Lost Rituals* Application Description

As a vehicle for the implementation of our generation strategy, we created an application called *Lost Rituals*. The application presents an interactive experience through which the user is introduced to a narrator, and then reads along through a book of fictional rituals in a fantastic world. The rituals are generated as the user turns the pages of the book.

There are two primary forms of text generation used in *Lost Rituals*: generation of the narrator's backstory, and generation of individual rituals.

In order for the generated rituals to be coherent from beginning to end, the generator required the ability to manage contextual state. This prevents the generation of nonsensical, impossible, or implausible actions as part of the generated rituals, and allows for a deeper connection between individual elements.

3.1 Platform Details

Lost Rituals is implemented in the programming language Swift 5.3. It runs on the iOS platform, specifically intended for iPhone devices.

All speech output generation is accomplished through the use of iOS's built-in text-to-speech capabilities. The voices available for selection are dependent upon the voices that have been installed by the user on their device, or the default set if no additional ones have been manually installed.

The application is bundled with a variant that is specialized for use over the Messages instant messaging system included on iOS. This variant allows two human participants to take turns building up a ritual step-by-step as an interactive activity, unlike the primary application mode in which an entire ritual is generated at once.

3.2 Narrator Generation

When the user begins to use the application, he or she is first presented with an introduction by a fictional narrator. Along with the textual introduction, the narrator introduces himself or herself using text-to-speech audio output. The voice of the narrator is selected as part of the generation process, and thus is related to the "character" of the narrator. The user can choose to generate a new narrator as many times as necessary to allow the selection of an acceptable voice for the reading of the rituals. Once a narrator is selected, the user simply presses a button to begin, and the first ritual is generated, presented visually, and read aloud.

The generation of the narrator's introduction is one of the simpler elements of the application, but it immediately reveals a crucial aspect of the value of using behavioral objects: the gender of the narrator is contextual data. The narrator is introduced by name, and the name is gender-associated. The voices provided by the built-in text-to-speech framework in iOS are also gender-associated. By maintaining the gender as a piece of metadata when the narrator's name is selected, it is simple to select a voice that matches the gender. This is a rather uncommon feature in generative text systems, since most of them do not exist in the context of a

complete, standalone runtime application that uses a selectable voice to read the output. In the case of *Lost Rituals*, this feature helps bring the user's perception more "into the world" of fantasy, and deepens immersion.

Beyond voice selection, maintaining the underlying data of the narrator's identity allows more descriptive elements to be added, such as "Sir" versus "Lady" as a name prefix to indicate nobility.

In practice, the generation process for the narrator's introduction is the following:

1. Create an instance of a NarratorIntroduction data structure
2. Creation of the NarratorIntroduction causes creation of an instance of a Narrator class object
3. Creation of the Narrator class object creates a singleton instance, populated with gender, name, institution, and other data
4. The NarratorIntroduction data structure interrogates the Narrator object instance for relevant information, and incorporates it into generated output text

3.2.1 Code Representation

The following is a truncated version of the code representation of the narrator.

```
class Narrator {
  let speechSynthesizer =
AVSpeechSynthesizer()
  let gender: Gender
  let title: String?
  let institution: Institution?
  let givenName:
CommonPersonGivenName
  let surname: CommonPersonSurname
  let voice:
AVSpeechSynthesisVoice?

  private init() {
    gender = Bool.random() ? .male
: .female

    switch Int.random(in: 0...2) {
      case 0:
        institution = School()
      case 1:
        institution = Institute()
      default:
        institution = Church()
  }
```

This is only a small portion of the code used in the initialization of the Narrator, but serves to illustrate the richness that is available through such an approach; the Narrator exists as a collection of defined facts and attributes rather than simply as a name. These facts and attributes can be related to one another using any programming logic desired, and can be passed along during the generation process as part of a rich context definition.

3.2.2 Example Output

The following are example outputs from the narrator introduction process in *Lost Rituals*.

Pleased to meet you! I am Dr. Malaya Mahoney, from Nymoxorr University of Parapsychology.

The University recently acquired this grimoire via an anonymous donation. It was immediately obvious that it contained the details of the most noteworthy rituals from all corners of the world. I have brought it for you to peruse.

Are you ready to start?

—-

I'm Acolyte Miracle Crosby, from the Sacred Orthodoxy of Undying Devotion. Hello!

To better understand the heathen mind, the Orthodoxy has built a library of the details of the most noteworthy

ceremonies from around the world, logging the discoveries in this aged tome, called the Tome of Zemu. I am pleased to present it to you.

Shall we explore?

3.3 Ritual Generation

The process of ritual generation in *Lost Rituals* is built upon the same behavioral object strategy as used in the narrator introduction generation. The rituals contain many more data elements than the introduction, however, and therefore are constructed using a much larger number of branching behaviors and conditional data structures.

In the Swift programming language, any data structure can publish its own textual representation on demand. This "description" facility is leveraged very heavily by *Lost Rituals* to produce text output.

Each concept to be represented in the text is placed through the instantiation of the highest-level concept, such as "Ritual". The process of instantiating this data object, contained within the initialization method of each object, not only establishes state information for the data object itself, but also evaluates various stochastic conditions and uses the results of that evaluation to instantiate smaller concepts that fall within the larger one. For example, a "Ritual" has a "RitualIntroduction" and a "Procedure". The initialization of the Ritual itself is not completed in memory until all of the contained dependency concepts have themselves been fully initialized, and their own contained concepts initialized, and so on. Functionally, this means that an entire tree of established facts is created in memory before any text is generated.

A novel aspect of this strategy is that by utilizing a hierarchical instantiation pattern, a reference-based Context object can be passed along to any contained object to guide and participate in its instantiation, and perhaps even have additional information added to it. This is a powerful approach for allowing context to travel between artifacts, and opens an avenue to address some of the shortcomings of the approaches described earlier.

Additionally, scoped parameters can be provided alongside the common contextual information, based on specific demands and requirements of the contained data object. For example, the "DomesticAnimal" structure allows the specification of whether or not adjectives will be associated with it. This allows code to make use of the concept of either "a pig" or "a well-fed pig". The animal object is responsible for understanding its own attributes (such as, in this example, that the pig is well-fed) but it is not necessary to generate additional facts in cases where they are known to be irrelevant, especially since the ultimate depth of the additional fact tree cannot be known by the containing data object.

The following simplified code snippet illustrates how the instantiation process works, specifically for a "Bottle" object possibly used in a ritual.

```
struct Bottle: OfferableThing, UsableThing
{

  let name: String
  let article: String
  let adjective: String?
  let material: Material
  let methodOfOffering: String
  let methodOfUse: String

  init(singular: Bool = true) {
    adjective =
Bottle.adjectives.keys.map({ $0 }).randomEl
ement()!

    if singular {
      name = "bottle"
      article =
Bottle.adjectives[adjective!]!
    } else {
      name = "bottles"
      article = ""
    }

    material = Crystal()
```

```
    var methodOfUse: String

    if Bool.random(probability: 75) {
      let filled =
Bowl.filledWords.randomElement()!
      let contents: String

      switch Int.random(in: 0...100) {
        case 0...33 :
          contents =
Bowl.nonFoodContents.randomElement()!
        case 34...66:
          contents = Ingredient().name
        default:
          let beverage = Beverage()
          if let adjective =
beverage.adjective {
            contents = "\(adjective) \
(beverage.name)"
          } else {
            contents = beverage.name
        }
      }

      methodOfUse = "\(filled) with \
(contents)"

      switch Int.random(in: 0...100) {
        case 0...25:
          let verb = singular ? "is" :
"are"
          methodOfUse += ", which \(verb)
then \
(Ingredient.offeringActions.randomElement()
!) \
(Ingredient.offeringLocations.randomElement
()!)"
        case 26...50:
          methodOfUse += ", then \
(Bottle.putSomewhereSpecial.randomElement()
!)"
        default:
          break
        }

     } else {
       methodOfUse = "filled with \
(Bottle.specialItems.randomElement()!)"

       if Bool.random() {
         if Bool.random() {
           methodOfUse += ", sealed"
         }
       methodOfUse += ", then \
(Bottle.putSomewhereSpecial.randomElement()
!)"

       } else {
         if Bool.random() {
           methodOfUse += " and sealed"
         }
       }
     }

     self.methodOfUse = methodOfUse

     methodOfOffering =
material.methodOfOffering
  }
}
```

This snippet illustrates several of the key concepts: local state management, contained objects, initialization-time parameterization, custom branching logic and behaviors based on the represented object, and inline text generation through string interpolation.

Once the tree of facts and structures has been established, the process of generating the output text itself is simple and highly performant. The program utilizes the on-demand string interpolation capabilities of Swift to generate a consolidated output of the entire data tree. Because this final step does not need to consider contextual state information (since that is owned by the behavioral objects being rendered as text), this provides an opportunity point to add decorative textual variation that is not part of the state tree's own representation. For example, in rendering the text for a single "ProcedureStep" in a ritual, the text conversion process can stochastically select from a variety of different introductory phrases, as illustrated in the following code snippet.

```
switch Int.random(in: 0...100) {
  case 0...15:
    str = "Next, "
  case 16...30:
    str = "After that, "
  case 31...45:
    str = "Following that, "
  case 46...60:
    str = "When that has been
completed, "
  case 61...75:
    str = "When that is done, "
  default:
    // Do nothing
    break
}
```

The exact phrase selected has no bearing on the state of the represented facts, which is why it can appropriately be selected at render time.

3.3.1 Example Output

The following are example outputs from the ritual generation process in *Lost Rituals*.

Just before the death of a female member of the community in the dangerous province known as eastern Ky, the most distinguished community members use a ritual practice to mourn and to release the soul into the afterlife.

The ritual requires the following four elements, which may be executed in any order.

To begin, a large cut of pig meat is burned.

When that has been completed, the participants chant while clad in blessed shawls and holding very fine bottles.

When that has been completed, one participant speaks the text of a sacred poem while garbed in a blessed robe and carrying a wand.

Finally, the participants, while attired in embroidered magenta silk shawls, chant.

—

The most respected citizens of Vady, just after the passing of an adult, have a sacred rite in order to express their sadness and to ask that the goddess called Casupac take pity on the soul in the netherworld.

The ritual is comprised of the following three steps.

While garbed in a blessed coat, a selected individual chants.

Following that, a spiced dish of sheep meat is burned.

Finally, a bowl is filled with cold white wine, which is then poured on the ground.

—

Prior to the first rain of the year in the lush forests called Lower Medohu, the female shamans have a sacred rite to ensure a thriving economy and to request the favor of the beloved goddess called Fiquad.

First, a single person dances towards a platform, upon which is placed a wand, while grasping a chalice.

Following that, a crystalline bowl is partially filled with wheat, which is then scattered under a tree.

When that has been completed, the participants speak the text of a sacred poem while holding fetishes depicting the mostly unknown demon known as Geposa.

When that is done, the participants in the ritual, while attired in blessed stoles, speak the text of a sacred poem.

Finally, a specially-prepared fig is burned.

4 Conclusions and Future Work

The use of behavior data objects for text generation certainly made the production of the Lost Rituals application easier, in the sense that it enabled the generation of coherent ritual description text that is internally coherent. The approach also shows promise in other projects, especially those that generate short-form coherent passages that address specific, focused subject matter with particular attributes that must vary in description along with the selected detail variants.

Future work includes increasing the amount of contextual information that is used during the creation of the ritual tree. For example, even though the context object already contains information about the purpose of the ritual, this is not currently used for anything during the generation process. It would be interesting to describe how individual steps in the rituals are intended to relate to the overarching goal of the ritual itself. It would also be interesting to have some of the individual procedural steps build

upon the actions of the step immediately preceding them, to create more of a sense of “flow” from step to step.

Finally, this approach shows promise in the goal of creating long-form textual stories. We have not done any substantial work in this area, but the advantages of the strategies described herein offer potential benefits, and these bear investigation.

5 References

[1] NaNoGenMo. Github.io. Retrieved 20:15, November 8, 2020, from https://nanogenmo.github.io
[2] Why I love National Novel Generation Month. Medium.com. Retrieved 21:05, November 8, 2020, from https://medium.com/@liza/why-i-love-national-novel-generation-month-b8f6e58c6422
[3] Context-free grammar. Wikipedia.org. Retrieved 16:04, November 8, 2020, from https://en.wikipedia.org/wiki/Context-free_grammar
[4] K Compton, B Kybartas, M Mateas, “Tracery: an author-focused generative text tool”, Springer, Cham, International Conference on Interactive Digital Storytelling, 2015
[5] Tracery Writer. Beaugunderson.org. Retrieved 16:05, November 8, 2020, from https://beaugunderson.com/tracery-writer/
[6] Markov Chains. Brilliant.org. Retrieved 16:07, November 8, 2020, from https://brilliant.org/wiki/markov-chains/
[7] Markov Chains: How to Train Text Generation to Write Like George R. R. Martin. Kdnuggets.com. Retrieved 16:09, November 8, 2020, from https://www.kdnuggets.com/2019/11/markov-chains-train-text-generation.html
[8] Ziang Xie, “Neural Text Generation: A Practical Guide”, Stanford, web, 2017
[9] Text generation with an RNN. Tensorflow.org. Retrieved 16:12, November 8, 2020, from https://www.tensorflow.org/tutorials/text/text_generation