Tracery: An Author-Focused Generative Text Tool

Kate Compton[1(✉)], Ben Kybartas[2], and Michael Mateas[1]

[1] Department of Computational Media, UC Santa Cruz, Santa Cruz, USA
{kcompton,michaelm}@soe.ucsc.edu
[2] Department of Intelligent Systems, TU Delft, Delft, The Netherlands
b.a.kybartas@tudelft.nl

Abstract. New communities of generative text practitioners are flourishing in novel expressive mediums like Twitterbots and Twine as well as the existing practices of Interactive Fiction. However, there are not yet reusable and extensible generative text tools that work for the needs of these communities. Tracery is an author-focused generative text tool, intended to be used by novice and expert authors, and designed to support generative text creation in these growing communities, and future ones. We identify the design considerations necessary to serve these new generative text authors, like data portability, modular design, and additive authoring, and illustrate how these considerations informed the design of the Tracery language. We also present illustrative case studies of existing projects that use Tracery as part of the art creation process.

1 Introduction

What does it mean to make an 'author-focused' generative text tool? We consider the potential authors for such a generative tool to be creative persons who are interested in the expressivity of language and the aesthetics of prose. They may not self-identify as 'programmers' or want to write code, but they want to create generative text that is algorithmically combinatorial and surprising, while still seeing their authorial 'voice' in the finished text. With the success of NaNoGenMo, the text-generating twin of National Novel Writing Month [2], and the increasing popularity of Twitterbots [6], there is a coherent community of practice around generating text that has little relation to many previous academic approaches to narrative generation. Text generators are finding audience and community in new platforms like Twitter, Itch.io, and Twine. These platforms have lower barriers to entry and less structured expectations than traditionally game-oriented hosting spaces like Steam or publishers like Eastgate, so many text generators that could not stand alone as either games or literary experiences now have a chance to build a niche audience. These new works often embrace an aesthetic of nonsense and absurdity, and make use of unexpected, unplanned, yet insightful juxtapositions. Many of them use templates and grammars to create structured yet variable text, a technique used in the ELIZA system [11] but still powerful and popular.

H. Schoenau-Fog et al. (Eds.): ICIDS 2015, LNCS 9445, pp. 154–161, 2015.
DOI: 10.1007/978-3-319-27036-4_14

1.1 What Is Tracery

We designed Tracery as an open source tool to write text-generating grammars, for users who may be academics, IF authors, botmakers, or game-makers[1]. Grammars are written as JSON objects in a simple and readable syntax, and then recursively expanded by Tracery into finished text. The system has been kept as lightweight and syntactically simple as possible to encourage its use in other systems while maximizing accessibility for novice users, even non-programmers. Grammars have been dismissed as insufficient for analyzing and generating stories [1], and fallen out of fashion in interactive narrative. We show that, with a good interface and some small additional features, grammars can be rehabilitated in interactive narrative, and can be used by authors, even casual ones, to create a wide range of stories, poems, dialogue, and even images and code.

2 Related Work

Authors of Interactive Fiction have identified generative text tools as a missing or poorly developed feature of their practice. In Emily Short's survey of the IF community, she notes the desire for: 'the ability to have the computer describe complex world model states or story events without having to hand-author every possible variation,' and catalogs the useful generative text features that *have* appeared in existing IF authoring tools, such as pluralization, option selection, and slurred-speech filters, among many others [8]. Previous story generation systems commonly focus on story structure and the maintenance of narrative causality [7] with planning algorithms or agent-based simulations. The price of this causality, however, is a reliance on a carefully modeled database of world knowledge, provided either as part of the system [12] or authored by the user of the system [4]. The pleasures of writing, like creating dialogue and character, playing with language, writing expressively, take a backseat to performing laborious knowledge modeling. Several storytelling systems do, however, model poetic language use, like Zhu and Ontañon's Riu system [13], which creates analogy-based stories using force dynamics, and Harrell's GRIOT system [3], which uses conceptually-linked axioms to pick content to fill in phrase templates. GRIOT uses templating to create recognizable poetic structure, but requires the author to create an ontology of axioms in order to create a new polypoem with its own logic. Tracery takes a different approach by *not* explicitly modeling world knowledge or axiomatic relationships. The only knowledge of the world consists of the grammar of symbols and their rewrite rules provided by the author. Even with only this very shallow data structure, the case studies included in this paper show that authors can produce structured and interesting generative text with a unique literary voice, even though that text may lack the logic of GRIOT and Riu, or the narrative validity of MEXICA. As demonstrated in the MEXICA case study included in Sect. 5, causality can be handled by an external system which pipes a more rigorous story model into Tracery. To the large, diverse, and

[1] https://github.com/galaxykate/tracery.

mostly non-academic communities engaged in generative text practices outside of knowledge-modeling traditions, these limitations do not seem to matter as much as we might have expected from previous academic theory.

3 Design Considerations

Tracery was intentionally designed to be easy to author with. The content created by an author is a simple object (a formal grammar) written mostly in plaintext, and advanced syntax is kept as readable and minimal as possible (see Sect. 3.1 for details). In addition, we followed other design considerations that we knew would be important to our potential authors: **modularity**, **balancing generativity and control**, and **modifiability**. Tracery is a modular ecosystem of interworking parts: a parseable language, an expansion engine, and many visualization and integration tools for building larger works, each of which can be used independently (see Sect. 4). Modules can communicate through the data format of a *grammar*, a list of symbols and rewrite rules, which provides a lingua franca that can be understood by any tool in our set, even those not authored by us. Maintaining open data formats and good encapsulation is a standard practice in software industry, and it has many useful side effects, such as allowing an independent user to turn it into a Node.js library (Sect. 4), serving Tracery server-side to run Twitterbots. Seeing interesting and unintended juxtapositions is one of the great pleasures of generative text systems. Many users make satisfying generators using only the most basic syntax: hashtags to signal expandable symbols. At the same time, some authors also need control over some facets of generation, like maintaining a persistence of a character's name and pronouns. Advanced users can opt in to more control when they need it, by using higher-level features like push-pop actions and modifiers, or changing the random distribution, while still taking advantage of randomness when it suits them. JavaScript, conveniently the dominant language of the web, is well suited to this project by being highly mutable. If an author includes Tracery in a larger JS project, any exposed library variable or object can be modified and added to, at runtime. They can add new modifier functions, read or write rules, and add new symbols, as in 'Interruption Junction' [9]. No library can provide all features for all projects, so this modifiability allows authors to bend Tracery to their needs.

3.1 Syntax

Symbols and Rules. Tracery's main data structure is a formal grammar, a mapping of *symbols* to sets of *rewrite rules*, as in this example:

```
color: ["red", "green", "indigo", "ecru", "violet"],
animal : ["panda", "ocelot", "meerkat", "platypus"],
mood: ["joyful", "morose", "alert", "sleepy", "pensive"],
pet: ["puppy", "#mood# kitty", "#mood# #color# #animal#"],
tale: ["This is the story of a #pet#...", "Once there was a #pet#"]
```

Each rule can be written as plaintext or with Tracery's special syntax to specify recursively expandable symbols, as in a formal grammar. We use a hashtag syntax to signal recursion: `#animal#` tag in `pet`'s rules can be replaced with the available rules for the `animal` symbol, and as shown in the rules for `tale`, rules can expand recursively to an arbitrary depth. This syntax defaults to allow the user to write in normal natural language, so an author can write: `The hero walked into a bar`, and then incrementally replace parts with symbols and alternative rules to add variability over time, until eventually `#theHero# #walked# into #someBar#` can expand to many different heroes, many bar names, and many ways of walking into them.

Modifiers. Capitalization, pluralization, a/an choice, and conjugation are common hassles of generative text, so Tracery provides functions that can be applied after a symbol is expanded. These built-in modifiers (`.a`, `.ing`, `.ed`, `.pluralize` and `.capitalize`) are provided as convenient though imperfect utilities. While we plan to improve the built-in modifiers, we also encourage advanced users to build their own modifiers for the languages and genres they work in: better pluralization, Spanish verb conjugations, or adding gender declensions for Icelandic nouns. Any function that can take text, modify it, and return it can be a modifier. Functions that work on full texts, such as a pirate-speak filter or a drunkenness filter, can be added, so that `'Aye', said the old sailor, '#longRamblingStory.piratize.drunkenize#'` will produce a story with the necessary flavor, regardless of what `longRamblingStory` originally generates.

Push-Pop Stacks. Often, authors want to save and reuse some generated text, such as the main character's name, species, possessions, or appearance. To this end, Tracery allows rules to write to the grammar when they run, 'overwriting' symbols (or creating them if they don't exist). Each symbol, instead of just having one set of rules, actually has a *stack* of rule sets, only the topmost of which is used to select a rewrite rule. Because these actions affect only their current sub-tree, an author can create a recursive nested story-within-a-story. This technique uses the grammar itself as a sort of blackboard to store and read information over time, balancing random generation with control.

4 A Modular Architecture: One Format, Many Tools

Tracery's main component is a lightweight text-expansion library. A grammar is a key-value array matching symbols to arrays of expansion rules, written in the syntax in Sect. 3.1. This **grammar** is loaded into Tracery, which can then respond to requests to expand Tracery-syntax strings by using the grammar's rules to recursively generate text. Each generated **trace** represents one tree-shaped path through the possibility space of the grammar: each symbol encountered is a choice of available possible expansion rules. Thus the trace can be returned as either flattened plaintext, or as the original tree-shaped path. The **grammar** and the **traces** provide a data structure that can be used by other

independent modules, which may operate independently or alongside the expansion library, depending on the use case. In this section, we present the modules and apps that take these structures as input to create text, visualizations, Twitterbots, or hosted shareable generators. The intentional modularity of Tracery makes such an ecosystem possible.

Visualizations. Tracery grammars can become very deeply nested and interconnected for non-trivial works, but good visualizations can clarify even a complex grammar. We created several visualizations, for understanding connectivity and reuse of symbols in a grammar, and seeing the commonality and range between multiple traces (Fig. 1).

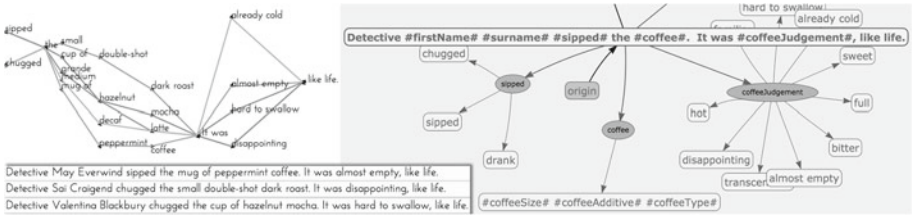

Fig. 1. Different ways to visualize the expansion of "Detective #firstName# #surname# #sipped# the #coffee#. It was #coffeeJudgement#, like life". Right: visualizing the grammar's connectivity shows how each symbol (in dark blue) can expand to many different options (pale blue). Left: visualizing five sample expansions demonstrates where each story varies or remains the same. Bottom-left: fully expanded text, as it would be read by the end reader (Color figure online).

CheapBotsDoneQuick and Twinecery. Tracery is being adopted by members of other established interactive fiction communities, some of whom have built tools to support the genres in which they write. Tracery's portability and common data format enabled these tools to be built with little or no additional support from us. Matthew Balousek, a student who had worked with Tracery, repackaged the expansion library to generate text inside of Twine[2]. The most successful Tracery tool so far was not created by us at all. George Buckenham, an independent game designer, created *Cheap Bots Done Quick!* (http://cheapbotsdonequick.com) independently for a bot-making workshop as a way for novice botmakers to quickly create and host expressive bots. Users sign in with Twitter, add a Tracery grammar, and can use it to preview generated tweets or set it to post automatically. This tool has encouraged both novice and very advanced IF authors (including Squinky, Emily Short, and Porpentine) to try Tracery and bot-making. At least 135 bots have been made so far, including some that generate orcish insults, fussy hipster cocktails and poetic space explorations (@orcish_insults, @HipsterCocktail, @DeepSpaceProbe), as well as non-prose like

[2] https://dl.dropboxusercontent.com/u/8790624/twinecery/twinecery.html.

valid query links for IFDB (@emily_testbot) and ASCII art (@infinitedeserts). Both Cheap Bots and Twinecery would not have been possible if we had not packaged and released Tracery for modification by others.

Hosting: Tracery.io. Though we are delighted by the success of Cheap Bots and new users it brought to Tracery, the bot authors lack a way to show the 'source code' of their grammars, or to learn from the grammars of others. We are building a hosting site (http://tracery.io/) where authors will be able to write and test grammars, browse the work of others, and examine and reuse elements of their grammars. This hosting provides templates hooking generated output to rogue-like RPGS, music generators, and image generators, as well as styling choices for dressing up plaintext Tracery output.

5 Evaluation: Case Studies

For a tool like Tracery, our evaluation metric is whether it has been successfully used to make works, by ourselves and others. Tracery has been developed while working on many of our own fully-implemented works (rather than just toy examples) and in conversation with authors writing real works. This kept our development grounded in the needs of the authors, leading us to the design considerations outlined in Sect. 3. Below are some of the works: several of our own, and one from independent designer Dietrich 'Squinky' Squinkifer, with the lessons we learned from them, and several pilot experiments showing new directions that Tracery may go in the future.

Eternal Night Vale. Eternal Night Vale was the first released stand-alone project using Tracery, and was created by us for ProcJam 2014, the procedural generation jam. It created possible episodes in the style of the podcast Welcome To Night Vale. A deeply nested grammar of about fifty symbols, each with many possible rewrite rules, was used to expand an '#episode#' rule recursively into the highly-structured segments of a Night Vale episode, filling in details with absurd generated vignettes that would be appropriate to the very idiosyncratic world of Night Vale. A review in Rock Paper Shotgun praised the work as 'a great little tool which condenses the tone and main elements of the show into just a few paragraph' [10]. By mimicking the distinctive structure and language of the show, we were able to capture the spirit of the show in a relatively short grammar, in the short time-frame of a one-week jam.

Interruption Junction. 'Interruption Junction' [9] by game artist Squinky is the first major game released using Tracery. In their words, it is 'a short one-button conversation game about being lonely in a group of people!' The player is in a conversation in which three other friends are discussing the activities of mutual acquaintances. The player can mash the space bar to interrupt and begin rambling about video games, but unilaterally dominating or retreating from the

conversation will cause people to fade from the conversation. The dialogue is generated by Tracery, using a grammar by Squinky, and is meaningless, absurd, and endless, a good fit for the theme of the game. Unlike Eternal Night Vale, Tracery dialogue is just one element of this game, which also has interactivity, animation, and sound. Interruption Junction showed that the encapsulation of Tracery works well in actual practice of professional game making.

Neverbar. Neverbar is a scifi dating sim that we are developing in parallel with Tracery to inform Tracery's development with the needs of a real game. Neverbar needs to maintain consistent world state, such as tracking the gender of the protagonist, their name, their current location in the bar, their drink, and the gender, name, and species of their love interest, and other story information. We found this to be a common need for users using Tracery in larger more interactive works like Interruption Junction, so we are developing an optional 'plumbing' layer on top of Tracery to automate the most common tasks we encounter. The 'plumbing' is able to generate options with display text, keep track of their potential values, and specify handlers to be called on activation or cancellation, like pushing the values into the game's world state. It can also move values from the world state back into the grammar to maintain synchronization.

MEXICA: Testing Integration with Other Narrative Systems. Many narrative generation systems focus on knowledge modeling, narrative, or simulation, features that Tracery lacks. Can we pair Tracery with these existing systems to create coherent stories augmented with expressive generative language? A recent paper on MEXICA included an XML output of the generator [5], which we translated into a JavaScript object, and then tasked Tracery with providing interesting and variable text interpretations for this skeleton story. Each concept in the MEXICA data is converted to a Tracery symbol and rules representing ways to describe it. The skeleton of the story remains the same, maintaining structure, but the language used creates very different stories. Although this test was done with exported XML data, any live narrative system that can communicate with JS will be able to use Tracery in the same way.

6 Future Work and Conclusions

We designed Tracery as an 'author-focused' generative text tool targeting the communities of practice around NaNoGenMo, Twine, and Twitterbots. We built a lightweight, modular, and easy-to-use library (and associated tools) using known best practices and design considerations drawn from those communities of practice. Our goal was seeing it adopted and used by a wide range of authors and toolmakers; it has been very successful by that metric. Our next steps will be launching Tracery.io and finishing and releasing the plumbing module and visualization tools. Intriguing recent experiments suggest that Tracery can generate syntax for HTML,[3] SVG, or even valid JavaScript, opening the door for

[3] https://twitter.com/ranjit/status/605149881200713728.

automatic code generation in the future. We look forward to seeing how Tracery continues to evolve as we add new authors and tools.

References

1. Black, J.B., Wilensky, R.: An evaluation of story grammars*. Cogn. Sci. **3**(3), 213–229 (1979)
2. Dzieza, J.: The strange world of computer-generated novels, November 2014. http://www.theverge.com/2014/11/25/7276157/nanogenmo-robot-author-novel
3. Harrell, D.F.: Walking blues changes undersea: imaginative narrative in interactive poetry generation with the GRIOT system. In: AAAI 2006 Workshop in Computational Aesthetics: Artificial Intelligence Approaches to Happiness and Beauty, pp. 61–69 (2006)
4. McCoy, J., Treanor, M., Samuel, B., Tearse, B., Mateas, M., Wardrip-Fruin, N.: Authoring game-based interactive narrative using social games and Comme Il Faut. In: Proceedings of the 4th International Conference & Festival of the Electronic Literature Organization: Archive & Innovate (2010)
5. Montfort, N., ý Pérez, R.P.: Integrating a plot generator and an automatic narrator to create and tell stories. In: On Computational Creativity (2008)
6. Neyfakh, L.: The botmaker who sees through the internet, January 2014. http://www.bostonglobe.com/ideas/2014/01/24/the-botmaker-who-sees-through-internet/V7Qn7HU8TPPl7MSM2TvbsJ/story.html
7. Riedl, M.O., Bulitko, V.: Interactive narrative: an intelligent systems approach. AI Mag. **34**(1), 67 (2012)
8. Short, E.: Procedural text generation in IF, November 2014. https://emshort.wordpress.com/2014/11/18/procedural-text-generation-in-if/
9. Squinkifer, D.S.: New game: interruption junction, January 2015. http://squinky.me/2015/01/19/new-game-interruption-junction/
10. Warr, P.: Welcome to eternal night vale, November 2014. http://www.rockpapershotgun.com/2014/11/19/eternal-night-vale/
11. Weizenbaum, J.: ELIZA: a computer program for the study of natural language communication between man and machine. Commun. ACM **9**(1), 36–45 (1966)
12. ý Pérez, R.P., Sharples, M.: MEXICA: a computer model of a cognitive account of creative writing. J. Exp. Theor. Artif. Intell. **13**(2), 119–139 (2001)
13. Zhu, J., Ontanón, S.: Story representation in analogy-based story generation in Riu. In: 2010 IEEE Symposium on Computational Intelligence and Games (CIG), pp. 435–442. IEEE (2010)