Procedural Play with Tracery

Tracery is a “story-grammar generation library” written in JavaScript. It was created by Kate Compton and has risen in popularity for its easy-to-follow functions for taking a set of options and creating a grammar that can produce different texts.

Here’s a couple examples of play with Tracery:

- Re:traced Threads Bot
- Generated Retro Blogs
- Interruption Junction
- Daelog (Tracery + Twine 2)
- Tracery Bots (list)
- NaNoGenMo (paper)
- Programming Historian: Intro to Twitterbots

In this workshop, we’ll be working through Tracery in three stages (we might not get through them all in our time together, but this link will stay live):

Writing a Basic Grammar:

1. Before diving in, play with the interactive tutorial and try changing the examples at each stage. Focus on changing the vocabulary and see what impact it has first, then worry about the rest of the elements and particularly the rules.
2. For this exercise, we’ll be creating a simple sentence generator. You will need to write your own grammar for the actual theory bot.
3. To start, we will work with the Tracery visual editor. This tool allows for testing generative text directly. Select "tinygrammar" and "JSON" to show the code.
4. Start by replacing each of the example names. Replace each of the professions as well. Add at least 3 more names and 3 more professions to each of the arrays using the "+" button.
5. Change the origin to: #name#, a #occupation##, -- don’t forget the ",," on the end, as we’ll use that to transition to the next part of our sample sentence. Hit "reroll" to see different combinations.
6. Add a new symbol by pressing the "new symbol" button, then change the name to always
7. Set the rules to array options that provide good substitutions for always (such as never, rarely, etc)
8. Add another new symbol called **expecting** and follow the same pattern with alternatives such as anticipating, imagining, etc.
9. Change your rule to **#name#, a #occupation#, was #always# #expecting# this...**

**Putting our Grammar in a Bot:**

1. Create a Twitter account. If you already have Twitter accounts connected to all your current email accounts, you can use name+bot@email.com to spoof the system. Choose a name that fits your intended bot’s theme.
2. Authorize the "**Cheap Bots Done Quick**" app to connect to this new Twitter account by logging into your account, then following the directions on the website. You will need to have confirmed your account through the email validation before the app will authorize.
3. You will now see a live Tracery editor of the type we've worked with before. Start by replacing the origin with a basic statement:

```yaml
{
  "origin": "Here is a bot."
}
```

4. Take the grammar you were playing with before and copy it into the Tracery editor. Make sure you have retained an "origin", as this is what contains the sentence patterns that will be tweeted. Check the preview to see samples of what your bot will generate.

**Adding Images:**

5. For this exercise I'm going to start by modifying a very basic example, then generate a complex version as we go. Here's the example of an ellipse from w3schools:

   ```html
   <svg height="140" width="500">
   <ellipse cx="200" cy="80" rx="100" ry="50" style="fill:yellow;stroke:purple;stroke-width:2" />
   
   Sorry, your browser does not support inline SVG.
   
   </svg>
   ```
6. Copying this example into Tracery doesn’t work: We need to convert it to SVG syntax that Tracery recognizes. First, wrap it in a object and indicate that the content is SVG and remove the alt text:

```xml
{svg <svg height="140" width="500">
  <ellipse cx="200" cy="80" rx="100" ry="50"
    style="fill:yellow;stroke:purple;stroke-width:2" />
</svg>}
```

7. We’re not quite there -- remember, quotation marks have a special meaning in Tracery as they indicate strings. We need to escape all the quotation marks by adding \ before every quotation mark, and remove all the line breaks within the code. We also need to add some boilerplate to help Twitter recognize the graphic codex:

```xml
{ "origin": "Here is some SVG: {svg <svg xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" width="500"
height="140">
  <ellipse cx="200" cy="80" rx="100" ry="50"
    style="fill:yellow;stroke:purple;stroke-width:2"
</svg>}"
}
```

8. Scroll down and you’ll see the text and svg now co-exist. A good side for a svg on twitter is 1024 x 512, so revise the size to reflect those dimensions:

```xml
{ "origin": "Here is some SVG: {svg <svg xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" width="1024"
height="512">
  <ellipse cx="200" cy="80" rx="100" ry="50"
    style="fill:yellow;stroke:purple;stroke-width:2"
</svg>}"
}
```

9. This is still pretty static: let’s start by making our shape dynamic and random. For every element, create a symbol in Tracery and fill it with options. For example, here’s an option for the radius:

```
"rx": ["50","100","150","200","250","300","350"]
```

10. To add this to our shape, we’ll need to integrate it into the grammar and then use it to change the HTML. Use the same symbol syntax in the middle of the code to replace the number with \#rx#.

```xml
{ "origin": "Here is some SVG: {svg <svg xmlns="http://www.w3.org/2000/svg"
```

```xml
```
11. We can also do this with values like colors: try replacing the fill and stroke with a set of color options using the same symbol for both:

```xml
{ "origin": "Here is some SVG: {svg <svg xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" width="1024"
height="512"><ellipse cx="200" cy="80" rx="#rx#" ry="50"
style="fill:yellow;stroke:purple;stroke-width:2" /></svg>}

"rx":["50","100","150","200","250","300","350"] }
```

```
"color":["purple","blue","green","teal","navy","darkgreen","LightCyan",
"MediumTurquoise","LightSeaGreen","CadetBlue","SeaGreen","PaleGreen",
"ForestGreen",
"Indigo","SlateBlue","DarkMagenta","BlueViolet","DarkOrchid"]
```

12. We can also add other SVG elements and styles, such as opacity, with corresponding symbols. Think about the number of options you include for each and be strategic about the value range to achieve the impact you want:

```xml
{ "origin": "Here is some SVG: {svg <svg xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" width="1024"
height="512"><ellipse cx="200" cy="80" rx="#rx#" ry="50"
style="fill:#color#;stroke:#color#;stroke-width:2;opacity:#opacity#" /></svg>}

"rx":["50","100","150","200","250","300","350"] ,
"color":["purple","blue","green","teal","navy","darkgreen","LightCyan",
"MediumTurquoise","LightSeaGreen","CadetBlue","SeaGreen","PaleGreen",
"ForestGreen",
"Indigo","SlateBlue","DarkMagenta","BlueViolet","DarkOrchid"],
"opacity":["0.2", "0.3", "0.4", "0.5", "0.6", "0.7"]
```
13. Now let's turn our SVG into a moving element: change the cx and cy to options to center the image at different parts of the box.

```
{ "origin": "Here is some SVG: {svg <svg
xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" width="1024"
height="512"> <ellipse cx="#cx#" cy="#cy#" rx="#rx#" ry="#50" style="fill:#color#;stroke:#color#;stroke-width:2;opacity:#opacity#" />
</svg>"},
"rx":["50","100","150","200","250","300","350"],
"color":["purple","blue","green","teal","navy","darkgreen","LightCyan","MediumTurquoise","LightSeaGreen","CadetBlue","SeaGreen","PaleGreen","ForestGreen",
"Indigo","SlateBlue","DarkMagenta","BlueViolet","DarkOrchid"],
"opacity":["0.2","0.3","0.4","0.5","0.6","0.7"],
"cx":["100","200","300","400","500","600","700","800","900"],
"Cy":["100","200","300","400","500"]}
```

Refresh to see the range of places your circle is generated.

14. Now let's make a pattern of multiple shapes: start by moving the ellipse code into its own symbol and calling it from the origin, as in the below snippet:

```
"ellipse":"<ellipse cx="#cx#" cy="#cy#" rx="#rx#" ry="#50" style="fill:#color#;stroke:#color#;stroke-width:2;opacity:#opacity#" />
",
```

15. We can add as many of this type of shape as we want by calling the ellipse repeatedly, as shown here:

```
"origin":"Here is a SVG: {svg <svg
xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink"
" height="512"
width="1024"> 
#ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse##ellipse.
```
16. From these elements, you can create a very complex pattern-making bot. Let’s change the background color of each generated image:

```
"background":"<rect width="1024" height="512"
    style="fill:#backgroundcolor"/>",

"backgroundcolor": ["black", "gray", "darkgray"],
```

Add background to the SVG in origin prior to the ellipse call. Here’s the final code:

```
{
   "origin": "Here is some SVG: {svg <svg
     xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="1024"
     height="512">"#background#"</svg>"},
   "ellipse": "<ellipse cx="#cx#" cy="#cy#" rx="#rx#" ry="50"
      style="fill:#color#;stroke:#color#;stroke-width:2;opacity:#opacity#" />",
   "rx": ["50", "100", "150", "200", "250", "300", "350"],
              "Indigo", "SlateBlue", "DarkMagenta", "BlueViolet", "DarkOrchid"],
   "opacity": ["0.2", "0.3", "0.4", "0.5", "0.6", "0.7"],
   "cx": ["100", "200", "300", "400", "500", "600", "700", "800", "900"],
   "cy": ["100", "200", "300", "400", "500"],
   "background": "<rect width="1024" height="512"
      style="fill:#backgroundcolor"/>",
   "backgroundcolor": ["black", "gray", "darkgray"]
}
```