38

Procedural Level and Story Generation Using Tag-Based Content Selection

Jurie Horneman

38.1 Introduction

Content selection is something that happens frequently inside games. Abstract data structures need to reference "content"—a mesh, a bitmap, a sound effect, or a line of text—to become something concrete, like an enemy. We must build mechanisms that allow the program to refer to external content, so that nonprogrammers can work without having to modify and recompile code. At the most primitive level, this can just be a filename referring to an external file.

We can do more than simply link entity A to file B, however; we can implement logic to make the selected content depend on dynamic conditions without having to manually spell out every single case. In this chapter, we will look at a simple yet powerful content selection mechanism that uses *tags*. This mechanism can be used for selecting level parts, scenes, events, items, conversations, NPC remarks, audio-visual effects, and more. It is easy to understand and to implement. We will see how and why it works, what the edge cases and pitfalls are, and discuss some alternative or more advanced techniques.

38.2 The Advantage of Simplicity

A simple way to look at content selection is to imagine you have a black box, and you ask it "please give me a piece of content that fulfills the following conditions," and then the black box gives you some content. In *tag-based* content selection these conditions are *tags*, which are represented as simple strings. Behind the scenes you have a pool of content, and each piece of content has been marked up with one or more tags. The algorithm simply looks through the content for pieces that have all of the requested tags, and returns one of them, picked at random (although see Section 38.5 for a more sophisticated approach).

The algorithm is extremely easy to understand. A tag is either there or not; content with the right tags is either there or not. This simplicity is a strength, because of the following:

- It requires little training and documentation.
- It requires little tool support to make the method useable.
- It requires little programmer time to help fix problems, and programmers are typically outnumbered by game designers, artists, sound designers, and so on.
- It does not require people who cross the divides between disciplines (designer-programmers, technical designers, or technical artists).
- It allows people to focus on creating a great game instead of trying to understand complicated systems.

This does not mean that more complicated methods do not have their place, but rather that simplicity has value. But even the simplest method has subtleties, and tag-based content selection is no different.

38.3 Case Study: *Last Mysteries*

In 2010 and 2011, I was part of the team at Mi'pu'mi Games that created *Last Mysteries*, a browser-based multiplayer 2D action role-playing game. Tobias Sicheritz, our technical director, designed a tag-based content selection mechanism to generate dungeons. Our dungeons were 2D arrays where each cell contained a bunch of tags, such as "corridor medieval spooky exit-north exit-west," and we had a number of tagged, hand-created dungeon rooms. The game server would load the 2D array and then find a fitting room for each cell. This is not "classical" dungeon generation: We did not generate dungeon layouts from scratch. Instead, our goal was to be able to design dungeons very quickly.

38.3.1 Dealing with Missing Rooms

What if there is no room with the requested tags? For some content you can simply return null. Perhaps you are asking "does anyone in the party want to say something about the player harassing this chicken?" and it is fine if nobody reacts to it because it does not break the experience or the game.

However, in *Last Mysteries* not finding a room was unacceptable: The player would walk into the void. If the game server could not find a room it would exit. So we had a combination of "hard to test manually" and "catastrophic consequence," and our testing department consisted of just one person.

We solved this problem by writing a medium-sized Python program that read all of the game's data, identified all of the content requests that could be made, and then checked whether those requests could be fulfilled. If not, it would spit out an error. We hooked this into our continuous integration server, and we never had a broken dungeon again. Identifying all the requested tag sets can be tricky, but in our case it was straightforward. As a bonus the tool allowed us to test the content in many other ways. The Expressionist system (Ryan et al. 2015) achieves a similar thing with an automatic, integrated to-do list.

38.3.2 Sparse Content

In a game with dynamic content, it can be helpful for content creators to know where content is sparse. But what does that mean, exactly? For instance, having only one piece of content for a request can be fine if that request is only made once an hour, but terrible if it is made more often. The type of content also plays a role: Spoken text is more memorable and distinctive than a wall texture in the background, and so will suffer more from sparseness.

There is no universal answer to this question, but it is important that you consider and answer it for your particular use case. You can get far with common sense ("The player will be in cells tagged with 'medieval' for a long time, so we need more content there."), but depending on the game you might want to use measurements and statistics, if only to compare reality to the heuristics you used.

Once you know the desired and the actual sparseness, you can display it either in some kind of structure that makes sense for the given content (a table of game events per character showing the number of reactions, say), or with a list showing the "sparsest" content ("number 1: medieval corridors"). Emily Short uses colors to indicate content sparseness in some of her procedural text generation tools.

38.3.3 Optional Tags

In *Last Mysteries* we used a tilde prefix to mark tags that were optional. So we would use "medieval corridor ~spooky" to ask the system: "I would like a medieval dungeon corridor, spooky *if possible*." This made the content selection algorithm slightly more involved. For each piece of content, we had to check whether it had the obligatory tags, and, separately, any optional tags, and select based on that.

We may be tempted to search for the *best* match. But what does that mean? The point of optional tags is to have fallbacks. You can say "If you cannot give me a spooky medieval corridor, just give me any medieval corridor." But if you start packing multiple fallbacks into your tag structure, it becomes unclear what the system should fall back to. For example, when you ask for "spooky ~medieval ~corridor," and there is a "spooky futuristic corridor" room and a "spooky medieval hall" room, which one should the system pick? Is "medieval" more important than "corridor?" Do you add priorities? Do you take content order into account—does earlier content get picked first? Do you look at the number of optional tags a piece of content fulfills? How do you make that truly simple to use?

I recommend picking content from the set that fulfills *any* optional tag, as well as the obligatory tags. This forces users not to be clever. Either something is optional, or it is not. Again, simplicity is a strength. However, depending on the use case, more sophisticated scoring systems can work, as can one of the alternative techniques described later in this chapter.

38.3.4 Excluding Content

Some rooms in *Last Mysteries* had entrances or exits to and from other dungeon floors. The server would start the player in the first room with an entrance, and when the player went to an exit, they would be taken to the next floor, or out of the dungeon altogether.

It was easy to go into the floor description and ask for rooms tagged "medieval" and "entrance." The system would pick the right room. But in the rest of the floor we would ask for rooms tagged "medieval," and this would occasionally lead to entrances being where they should not be, ruining the level flow and causing bugs. A similar problem happened with rooms marked "boss." There was no easy way for us to say "do not pick these rooms."

There are several possible solutions to this problem:

1. Introduce a tag that says "not an entrance, exit, or boss room." Give that tag to all applicable rooms. We did this automatically inside Last Mysteries' export tool—any room request that did not have tags such as "start," "boss," and so on. tags would get a "middle" tag.
2. Make it possible to indicate that a tag must be requested explicitly for the content to be picked. Then tag rooms with, say, "!boss" to mean "boss" must be requested explicitly. The selection algorithm then has to filter out content with explicit tags that are not found in the requested set.
3. If you have a small list of these explicit tags, just treat them as a special case in the selection algorithm. In the case of *Last Mysteries* we only had five.

Solution 1 gives more control to the content creator but involves more tagging, or a magical tag that is never explicitly requested. Solution 2 is less work but complicates the system a tiny bit. Solution 3 is a bit harder to change and slightly more magic. Which solution is best depends on how often you expect these explicit tags to change.

38.4 Case Study: *Mainframe* and Choba

In October 2015, Liz England and I developed *Mainframe* (England and Horneman 2015), a web-based interactive fiction (IF) game for Procjam, a game jam about procedural content generation, using a JavaScript IF system called Choba* (Horneman 2016).

Mainframe makes heavy use of tag-based content selection. With Choba, as in any similar choice-based IF system, it is possible to create a link between two scenes, so that an option in one scene takes the player to the second scene. Using *Mainframe*'s XML-based syntax, we would write:

```
<option nextScene="computer_room_introduction1">
Approach the central mainframe.</option>
```

However, Choba can also select another scene using tags and then inject a link to it:

```
<injectOption tags="option, mechanical" />
```

* Both *Mainframe* and Choba are free and have been released as open source.

This says "find me a scene tagged with 'option' and 'mechanical,' and create an option leading there." Using this, we could express higher level concepts like "give me three options for searching a room with electrical and container elements":

```
<injectOption tags="search, electrical" />
<injectOption tags="search, containers" />
<injectOption tags="search, electrical" />
```

Another use of tag-based content selection simply injects a block of content:

```
<injectBlock tags="mission_desc, act1" />
```

This searches for a block with the tags "mission_desc" and "act1" and inserts it into the scene.

Things got really interesting when we made tag requests dynamic:

```
<injectBlock tags="mdesc, $act" />
```

The "$" prefix indicates the requested tag is a variable, so this command requests a block with the tags "mdesc" and, say, "act1." This runtime evaluation turned out to be immensely useful to allow the game to evolve over time. Depending on certain player actions, the "act" variable is increased, and the game seems to slowly change.

We also used it for a rudimentary injury system. We would occasionally execute:

```
<injectBlock tags="injury, $injury"/>
```

The "injury" variable would start off as "none," and there is an empty block tagged with "injury" and "none." But in certain parts of the game, the "injury" variable could be set to "bleeding_hand," say, and then we had several blocks describing the injury.

Picking some content based on a single dynamic variable can of course easily be done by just using a one-dimensional lookup table, but the advantage of our approach is that we could create both basic and complex mappings quickly and easily in a single line of text.

In *Mainframe*, we used tag-based selection to implement missions, dialog, random remarks, mood text, items, item descriptions, and the injury system. It contains 200 injected scene transitions (using tag-based selection), 249 injected blocks of text, and just 144 standard scene transitions, where the next scene is selected by ID.

38.5 Decks

A subtle aspect of tag-based content selection turned out to be of crucial importance in *Mainframe:* once you have found all pieces of content with the desired tags, which one do you actually pick?

In *Mainframe* we wanted each play-through for each player to be different. We also wanted each piece of content to be seen once before content starts to repeat. Our approach was to shuffle the items with the right tags and then pick the elements in order.

Let's introduce a new metaphor, "decks," that I have found useful in this case, both for the content creator (game designer, writer, etc.) and the implementing programmer. Imagine all pieces of content with a given set of tags as a deck of cards. Immediately,

the algorithm to use for picking content becomes very clear: shuffle the deck, then draw. When you are done, shuffle again, so you do not get recognizable patterns.

The metaphor of a deck of cards makes a subtle edge case more prominent. Let us say you want to draw three cards from the same deck, and you do this:

- Draw a card and use it.
- Put the card on the discard pile.
- Shuffle the discard pile if the deck is empty.
- Repeat this three times.

In this case, it is possible to end up with the same card twice, because your deck ran out, you reshuffled, and a card you had already picked ended up in the front of the deck. This is not an esoteric bug: this can happen in *Mainframe*. As a result, it is possible to be in a room with two exits leading to the same place. The correct approach, as you know when you have played board or card games, is as follows:

- Draw cards until you have the desired amount. If the deck becomes empty during drawing, shuffle the discard pile.
- Use all cards.
- Put all cards on the discard pile.

In *Mainframe*, I picked the abstraction "draw a card" and then repeated it three times, instead of picking the abstraction "draw three cards." "Draw a card" is easier to implement, because I only have to consider each draw in isolation. "Draw three cards" is trickier. We can write this:

```
<injectOption tags="option, containers" />
<injectOption tags="option, containers" />
<injectOption tags="option, containers" />
```

in which case it is obvious we need to draw three "cards" from the deck of scenes tagged with "option" and "container." We could instead have used:

```
<injectThreeOptions tags="option, containers" />
```

But we can also write:

```
<injectOption tags="option, electrical" />
<injectOption tags="option, mechanical" />
<injectOption tags="option, electrical" />
```

or add arbitrary if/then logic to these lines. That makes the logic more complex—you need to track what you have drawn from where, per scene. But at least the clear metaphor of a deck makes clear what *should* happen.

It is important to remember when implementing decks that "dealing" a card modifies the deck, and thus the program's state. This also caused problems in *Mainframe*.

One advantage of tag-based content selection is that the content is decoupled from where and how it is used. However, this can also lead to unexpected behavior. For example,

in *Mainframe* the selected content can itself select more content, or set variables. Because of this, sometimes when we added a block somewhere, a completely different part of the game broke, because that new block got selected. From a programmer's point of view, it is as if you wrote a function and it crashed your program without you explicitly calling it.

38.6 Other Case Studies

Tag-based content selection has been used in several other games.

38.6.1 Irrational Games

SWAT 4, Tribes: Vengeance, BioShock, BioShock 2, and *BioShock Infinite* by Irrational Games used something very similar to tag-based content selection (Cohen 2005). The system matched audio and visual effects to abstract game events at runtime, by examining the nature of the content involved in the event and then choosing the "best match" effect that artists had mapped for that combination. So, for example, when a gun was fired, the game would figure out what was hit and then trigger a parameterized event, like so:

```
BulletHit(holderOfTheGun, bulletType, materialThatWasHit, meshThatWasHit, …
etc…)
```

It considered things like mesh names, material names, designer-controlled "contexts," and even text tags.

38.6.2 Six Ages

The storytelling game *Six Ages* also makes heavy use of tags (Dunham 2016).

Tags in tag-based content selection are typically inside one big global namespace, and big global namespaces do not scale well. Over time, as multiple people work on a game's content, different subsets of tags may start interfering with each other. *Six Ages* uses tags like this:

```
rumors = [self scriptsWithTag: @"rumor" ofType: type_News];
```

"rumor" is a tag, whereas "News" is a type, another way of classifying content. It may have been possible to implement types using tags, but sometimes it makes sense to introduce stronger typing, to reduce complexity, improve performance, and avoid problems with tags overlapping. The script types in *Six Ages* effectively create namespaces.

Additionally, in *Six Ages* tags can be added dynamically to scenes, and tags can be disabled, meaning content with these tags is no longer selected.

38.7 Extensions and Other Approaches

Tag-based content selection is a simple yet powerful approach, but it does have limitations. Let us look at some ways to extend it, or different techniques that do similar things.

One seemingly promising way to extend tag-based content selection is to allow logical predicates as tags. Instead of tagging something with "medieval," we tag it with "area = medieval," and if that condition is true, that piece of content is selected. After all, some tags are already implicit predicates, such as "mood = spooky." And then why not extend that with logical or comparison operators, such as "health <5"?

This seems a small step but in fact inverts the entire system, because it becomes unclear which "tags" are "requested." The metaphor is no longer "give me some content with the following qualities," it is "here is the entire state of the world, which content fits?" This approach requires a much more sophisticated algorithm to find matches. It will need to go through and evaluate every predicate for every piece of content on every request. It is now effectively evaluating a rule set. This can be done with some combination of intelligent ordering of predicates, shortcut evaluation, caching, and so on, but it is significantly more work, and the question of how to identify the "best" match will become an issue.

As a case in point, I once worked on a project that implemented a simple version of a predicate-based system for text selection. Although the selection algorithm was one line of C#, using Linq, that line took three good programmers a whole day to write, and afterward I still spent more time reasoning about why a given piece of content had been picked than I expected or wanted.

Six Ages uses predicates to *exclude* rather than include scenes. This is considerably simpler in that only a chosen candidate needs to have its condition evaluated.

38.7.1 Valve's Dynamic Dialog System

Rule-based systems have their place. For instance, systems that select audio reactions ("barks") for AI-controlled entities are inherently complex, because they need to react to a lot of different stimuli in complex ways. Tag-based content selection may be too simple. One good approach to look at is Valve's dynamic dialog system (Ruskin 2012), as used in *Team Fortress 2, Left 4 Dead, Portal 2*, and other games. It is based on rules that can query any aspect of the world's state. The advantages of this approach are that it is easy to understand and fairly elegant in design, and can be powerful in practice. From discussions I have had with some studios using this technique, one of the downsides is that it may require a dedicated person, with programmer support, to maintain the rules as they grow in complexity. As an alternative, Paul Tozour (Tozour 04) describes a more complex use of tags, rather than rules, to select animations and audio.

38.7.2 Bioshock Infinite's Gameplay Pattern Matcher

Going even further, *Bioshock Infinite* used a gameplay pattern matcher (Kline 2011) for achievements, NPC AI, quest logic, conversations, scripting logic, and more. Tag-based content selection implicitly recognizes game situations, in the sense that requested tags are a limited proxy for what is happening. *Bioshock Infinite*'s gameplay pattern matcher makes this recognition explicit, and goes beyond simple world state queries into temporal logic. Though it is a fascinating approach, one of the developers noted that nonprogrammers sometimes found the asynchronous nature of its temporal matching hard to conceptualize, and its usability could be improved through better tools. It is significantly more complex than tag-based content selection.

38.8 Conclusion

Content selection is a common activity in game development. Tag-based content selection selects content by comparing a set of tags, that is, strings. This technique is simple to implement and to use. Among other things, it can be used for audio and visual effect selection, for level generation, and for procedural generation of interactive fiction scenes.

Despite its simplicity, there are edge cases that must be considered. There may not be content for a given set of tags, or there may be too little content, which can lead to repetitiveness. Tag structures must be designed with care to make sure unwanted content is not selected by mistake. By making tag requests depend on game state, the game can become more dynamic and responsive. When multiple pieces of content with the same set of tags exist, it is useful to think of the content as a deck of cards from which a piece is "dealt." It may be tempting to use logical predicates or even heuristics instead of tags, but this can make the system much more complex. For some use cases, more complex systems can be appropriate, but tag-based content selection is surprisingly versatile and powerful.

Acknowledgments

Thanks to David Dunham, Sebastian Holzfeind, Christopher Kline, and Borut Pfeifer, for their valuable feedback which greatly improved this chapter.

References

Cohen, T. 2005. Moment of impact: Designing an in-game effects system. *Game Developer Magazine*, pp. 21–28.

Dunham, D. 2016. Scene Tags. http://sixages.com/blog/index.php/2016/05/scene-tags/ (accessed July 11, 2016).

England, L. and J. Horneman. 2015. Mainframe. https://github.com/jhorneman/procjam15/tree/choba (accessed July 11, 2016).

Horneman, J. 2016. The Choba engine. https://github.com/jhorneman/choba-engine (accessed July 11, 2016).

Kline, C. 2011. The future of gameplay authoring. *Presented at Cornell University in 2011*. https://twitter.com/korkyplunger/status/525326773563973632 (accessed July 11, 2016).

Ruskin, E. 2012. AI-driven dynamic dialog. *Presented at Game Developers Conference 2012*. http://assemblyrequired.crashworks.org/ai-driven-dynamic-dialog-at-gdc-2012/ (accessed July 11, 2016).

Ryan, J. O., A. M. Fisher, T. Owen-Milner, M. Mateas, and N. Wardrip-Fruin. 2015. Toward natural language generation by humans. *Proceedings of the Intelligent Narrative Technologies*. http://www.academia.edu/14884597/Toward_Natural_Language_Generation_by_Humans (accessed July 11, 2016).

Tozour, P. 2005. A flexible tagging system for AI resource selection. In *AI Game Programming Wisdom 2*, ed. S. Rabin. Hingham, MA: Charles River Media, pp. 351–359.