StoryAssembler: An Engine for Generating Dynamic Choice-Driven Narratives

Jacob Garbe
UC Santa Cruz
Santa Cruz, CA
jgarbe@ucsc.edu

Max Kreminski
UC Santa Cruz
Santa Cruz, CA
mkremins@ucsc.edu

Ben Samuel
University of New Orleans
New Orleans, LA
bsamuel@cs.uno.edu

Noah Wardrip-Fruin
UC Santa Cruz
Santa Cruz, CA
nwf@soe.ucsc.edu

Michael Mateas
UC Santa Cruz
Santa Cruz, CA
michaelm@soe.ucsc.edu

ABSTRACT

Choice-driven narratives, such as those created through systems like Twine, are a compelling form of interactive storytelling that have been around for many years. But as long as this form has existed, it has grappled with a persistent design problem: consistently presenting choices that feel both effective and relevant. Brute force can achieve the desired effect, but usually at the cost of prohibitively high authorial burden. To tackle this, generative approaches, such as Mawhorter's *Dunyazad*, facilitate authoring procedural choice content for reuse and recombination. However, many such systems, while successful on technical levels, have yet to to be used to author large enough structures to support a full game, and require a high technical threshold for authors to use. To further development in this space, we present StoryAssembler, an open source generative narrative system that creates dynamic choice-driven narratives. It formed a critical part of *Emma's Journey*, an interactive narrative game, the initial version of which was collaboratively authored by a team of six writers. In the course of the game's creation, useful authoring patterns and design lessons were learned, as well as techniques that made the system approachable for first-time users.

CCS CONCEPTS

• **Human-centered computing → Hypertext / hypermedia**; **Systems and tools for interaction design**; **Hypertext / hypermedia**; • **Software and its engineering → Interactive games**; **Interactive games**.

KEYWORDS

Procedural narrative, interactive narrative design, narrative generation, choice-based narrative

ACM Reference Format:
Jacob Garbe, Max Kreminski, Ben Samuel, Noah Wardrip-Fruin, and Michael Mateas. 2019. StoryAssembler: An Engine for Generating Dynamic Choice-Driven Narratives. In *The Fourteenth International Conference on the Foundations of Digital Games (FDG '19), August 26–30, 2019, San Luis Obispo, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337722.3337732

1 INTRODUCTION

Hyperfiction, a form of interactive media which typically relies on links between passages of text to tell a story, is a form which has been with us for many years—yet despite that still offers new modalities for storytelling, each with their own attendant design challenges.

While the canon is amorphous and the form endlessly variegated, typical hyperfiction (such as works written in Twine) can be thought of as a static graph, where the nodes are pieces of story, and the edges the links available for the player to click from each node.

StoryAssembler was created as part of a larger project, where it worked in tandem with a mini-game generator. Both generators were designed to operate off an input list of design constraints, which would then result in two generated artifacts in dialogue with each other—a mini-game, and a hyperfiction. Thus, in contrast to existing hyperfiction systems, we required a dynamic planner system which could assemble choice-based hyperfiction, driven by a list of design constraints, as opposed to explicitly linked nodes. The dynamism of the constraints meant that if we chose a static approach to the problem, the authorial burden would escalate very quickly. Therefore, we created a system that could result in very large, complex graphs from relatively few nodes, due to the reusability of nodes and the dynamism of pathways between them.

While this emergent complexity was promising, it came with attendant authoring and design challenges. Our need to create a fully-realized playable experience drove us to further refine the system, and categorize several design approaches that helped harness the dynamism of our system, which we feel are generalizable to other systems as well. It also resulted in developments to facilitate authoring such that writers without programming skills could author scenes collaboratively.

2 PREVIOUS WORKS

2.1 Hyperfiction

StoryAssembler generates dynamic choice-based narratives, similar in output to hyperfiction or Choose Your Own Adventure (CYOA) books. These types of narratives have been with us in digital form for quite some time, with early systems like HyperCard [11] and Eastgate's Storyspace [2] creating a fertile initial ground.

Hyperfiction experienced a cultural revival of sorts with the rapid adoption and proliferation of Twine works in the early 2010s [9], although there has been a steady stream of systems released throughout the years such as ChoiceScript [6], Inkle Studios' Ink [14], and Undum [18] (with its later enhancement, Raconteur [3]) to name just a few.

Similarly, there is a rich existing field of hyperfiction design practice which is outside the scope of this paper to detail, with a myriad of different works making use of their respective software's affordances to achieve various aesthetic effects through the structuring of their choices and links. In a more codified approach, some experts have analyzed such authoring patterns, such as Mawhorter's work on Choice Poetics [17] and Ashwell's *Standard Patterns in Choice-Based Games* [1]. Such affordance analysis exists on top of the implicit design practice outlined in software-specific authoring guides also tackling the issue, such as with ChoiceScript [7] or InkleWriter [13].

StoryAssembler exists in the same family, in that its output resembles that of these systems. However, the highest-level procedures it uses to generate choice-based narratives are planner-based, rather than explicitly coded by the author, which affords potentially richer and dynamic structures to be created, but requires different design sensibilities.

2.2 Planner-driven story systems

StoryAssembler's core is in line with threads among the planner-based story generation community, such as Lebowitz's *Universe* system from 1983 [16], which is one of the earliest examples of planning systems focused on plot generation. Planner-based interactive narrative generation has remained an active area of research to present day, with systems like Robertson's *General Mediation Engine* [22] making strides towards parser-based generated narrative worlds.

In comparison to hyperfiction systems, which oftentimes rely on explicitly linked passages for much of their authoring, planner-driven story systems rely on heavier computational lifting to assemble their stories dynamically from a library of authored actions/operations. This has been characterized as having two main approaches: simulationist and deliberative [21]. Simulation-based (or emergent) systems rely on operating rules for the world and characters to form the basis of stories, which emerge organically from the processes. The deliberative approach characterizes situations set up to be resolved, or desirable states posited to the system, which has a series of available actions it can take to effect state changes to reach the desired state. StoryAssembler would be characterized as optimized for deliberative approaches to generation, although simulationist approaches could be potentially integrated.

StoryAssembler differs from many narrative planners in that it exposes and diegetically surfaces valid planner operations to the reader through narrative choices. This surfacing can also be shown in scene introductions, where players can read parameterized goal states as diegetic text, and alter them to suit their tastes before the planner executes to generate the story. This is expanded upon in Section 3.1. StoryAssembler is still tightly related to these other planners if one considers the potential generative space of their system output as a sort of "choice-based" narrative, where the planner makes the choices and then shows the resulting traversal to the player, instead of the player choosing from the myriad options at each step of execution.

Additionally, it should be noted that historically, the media artifacts generated by many planner systems are often not as fully realized as ones generated by people authoring with hyperfiction systems, as the primary computational research goal for planners can often be accomplished with simple sentences used as story beats, or even through re-purposing existing texts by breaking them into their component parts and procedurally traversing them, such as Yu and Riedl's work with CYOA books [27]. It was important in the development of StoryAssembler that the focus remained on the final product being recognizably a choice-based narrative that could hold its own alongside more traditionally authored Twine-based hyperfiction, which necessitated a hybrid approach in system development. In some cases this meant starting from simple planning structures with little generativity that were easy to author, then adding in more robust and generative operations and states later on as part of the editorial process, which required more complex design thinking to achieve. More details on that can be found in Section 6.

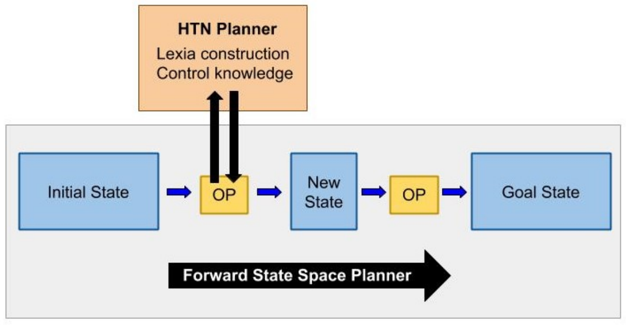


Figure 1: StoryAssember works as an HTN planner co-routining with a forward state-space planner.

3 SYSTEM DESCRIPTION

StoryAssembler takes a library of content and assembles it together into a choice-based narrative. Specifically, it assembles **Fragments** (3.3) from a **Fragment Library** (3.2) to satisfy all the items on the **Story Spec** (3.1), using **Templated Text** (3.4) to further communicate state to the player.

It accomplishes this through two distinct algorithms working in tandem (Fig. 1) to create a graph of connected "fragments", where a fragment is composed of displayed body text, and a list of choices that lead to other fragments.

At the highest level, StoryAssembler is a forward state-space planner [10]. Each scene has a "starting state" with initial blackboard values of booleans, strings, or integers, and a "story spec", composed of narrative scene goals also represented as blackboard values, such as `introduceNemesis eq true`.

When executed, StoryAssembler looks at the starting variables' states, looks at the story spec as a list of desired variable states, and then assembles fragments to push it closer to that goal. It does this by recursively searching the library for components—be that body text or choices—to maximize progress towards the story spec state. Additionally, it scores fragments higher if they link to other fragments that make goal progress, or incorporate into themselves other fragments that make goal progress as compound fragments. In this way, the search is similar to one done by a hierarchical task network (HTN) planner [5], where each fragment in the library specifies a set of sub-tasks to pursue (e.g. finding choices or body-text with specific properties).

The final assembled fragment represents the best next step for the forward state-space planner, and it continues in this manner until the scene ends. In short, StoryAssembler greedily optimizes for fragments that fulfill the maximum number of spec entries, as well as every choice in that fragment going to subsequent fragments that fulfill the maximum number of spec entries.

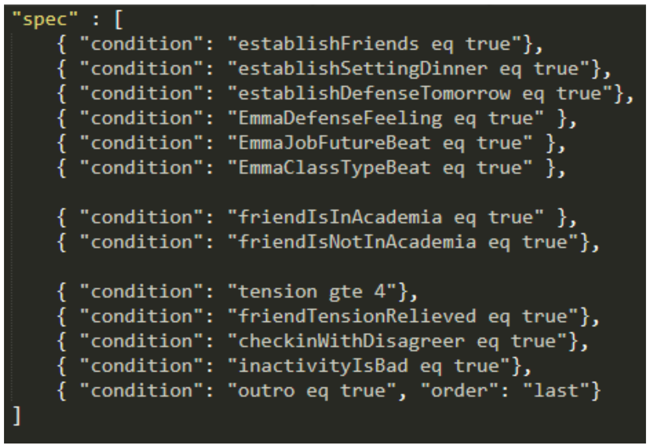
It must be noted that choice-based narratives are a particularly interesting and fruitful domain for planning systems, due to the additional combinatoric factor in the assembly of choices and displayed framing text. Because the framing text, the choice label, and the destination of the choice link are each their own atomic unit, it means those fragments can combine effects as compound operations, opening up a wide space of possibilities.

After each choice the reader clicks, the system re-evaluates the current state, looks at the story spec, and re-plans to assemble the next fragment accordingly. This "lazy evaluation" was chosen because in its initial project (*Emma's Journey*) a mini-game continuously runs in parallel, affecting StoryAssembler's blackboard state. Therefore, it made sense to not evaluate the path forward until the moment of the player's click, in case an originally valid state was invalidated by gameplay while the text was displayed, or the reverse.

Additionally, while StoryAssembler operates in "live" mode, it can change displayed text and choices asynchronously in reaction to other game processes, or elapsed time, as detailed in Section 3.3.

3.1 Story Specs

Each scene in StoryAssembler has a corresponding story spec containing a series of "spec entries" (Fig. 2). A spec entry, like blackboard entries, can contain boolean, string, or integer values, and authors may use them for anything from dramatic beats (`introduceFriend eq true`) to mood conditions (`tension gt 5`) or flags to inform the system where the reader has been, or which choices they've made (`complimentedWaiter eq true`). Spec entries can be roughly ordered through possession of the optional tags "first" or "last," but in general StoryAssembler treats them as an unordered list. The system will end the scene once every spec entry has been satisfied at some point in the story (if a later fragment sets a spec entry bool such that it is invalid, it doesn't "unsatisfy" that entry). This design decision was made so that we could have more flexibility in the design of beats within a scene to change underlying state to multiple values as the scene progressed. For example, if we wanted two characters to increase their `friendliness` stat after having an argument, we could put two spec entries `friendliness lt 2` and `friendliness gt 2` in our story spec, and take care that reconciliation content that increased `friendliness` had pre-conditions of low `friendliness`, so that it didn't prematurely trigger. This was done also to facilitate splitting up authoring tasks in scenes without incurring too much cognitive load in having to keep all the story spec items in mind while authoring.



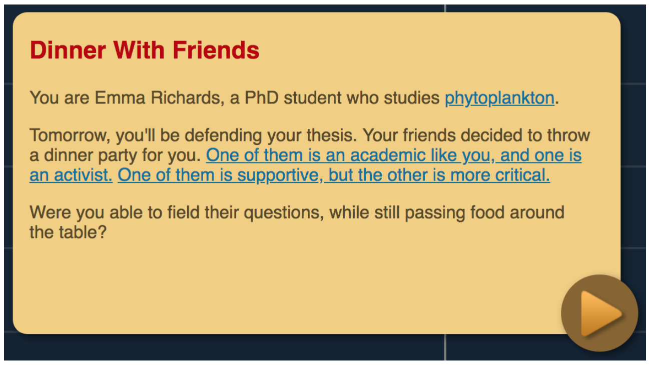
```
"spec" : [
    { "condition": "establishFriends eq true"},
    { "condition": "establishSettingDinner eq true"},
    { "condition": "establishDefenseTomorrow eq true"},
    { "condition": "EmmaDefenseFeeling eq true" },
    { "condition": "EmmaJobFutureBeat eq true" },
    { "condition": "EmmaClassTypeBeat eq true" },

    { "condition": "friendIsInAcademia eq true" },
    { "condition": "friendIsNotInAcademia eq true"},

    { "condition": "tension gte 4"},
    { "condition": "friendTensionRelieved eq true"},
    { "condition": "checkinWithDisagreer eq true"},
    { "condition": "inactivityIsBad eq true"},
    { "condition": "outro eq true", "order": "last"}
]
```

Figure 2: A sample story spec, detailing beats to be hit (boolean values) and mood conditions (tension).

Spec entries can also be flagged as persistent, which means the system will always prioritize content that satisfies them, but will not mark them as satisfied when such content is displayed. This can be used to make the planner prioritize for repetitive actions, such as a professor calling on students during a lecture, or taking a turn in a conversation. Typically pre-conditions are used with such fragments of repetitive actions to control ordering.

A good example of how spec entries work is the first scene of *Emma's Journey*: the night Emma dines with friends before her PhD defense. This scene's story spec contains key items that both communicate setting and initiate the narrative dynamics. The story spec requires two characters to make their cases whether Emma should pursue academia or activism, that the tension in the room passes a certain threshold (via fragments that increment a `roomTension` variable), and that Emma discusses her own career aspirations. A library of fragments have been authored that can achieve these goals in a variety of configurations, which are assembled by the system into a choice-based narrative.

One of the interesting capabilities with advanced authoring in StoryAssembler is the ability to make spec entries dynamic, and exposed to the player as scene settings before the story starts (Fig. 3). For *Emma's Journey*, we implemented this as "cycling links" in the scene description. Players can click on the links to change their text values, which in turn affect the spec entries, and the way the scene



Dinner With Friends

You are Emma Richards, a PhD student who studies phytoplankton.

Tomorrow, you'll be defending your thesis. Your friends decided to throw a dinner party for you. One of them is an academic like you, and one is an activist. One of them is supportive, but the other is more critical.

Were you able to field their questions, while still passing food around the table?

Figure 3: A scene intro, where dynamic spec entries allow players to change friends to be activists or academics, and supportive or critical, as well as Emma's research focus.

plays out once they begin. Therefore, if the player decides to have both friends more critical, the resulting story the planner assembles will have them challenging the player more on their career choices.

3.2 Fragment Libraries

For each scene in StoryAssembler, JSON files of fragments are specified for use. The ability to specify multiple files became an asset during authoring in teams, as it allowed writers to each use their own file, and substitute in placeholder copies of fragments handling other spec entries that other writers were working on. This allowed us to easily develop the game with a sort of version control, without having to negotiate problems such as merge conflicts, which can be confusing to writers unfamiliar with such systems.

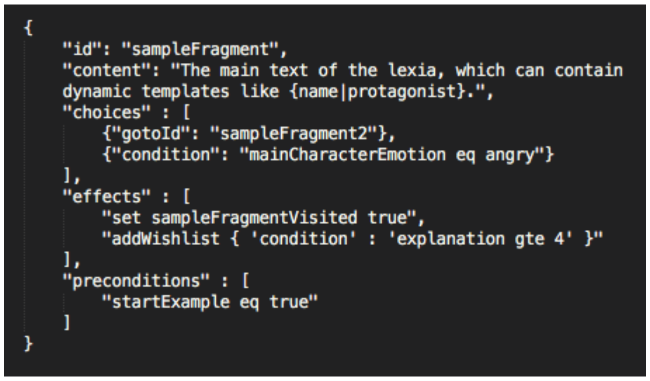
In terms of narrative design pragmatics, this also makes it possible to split up fragments organizationally by other features, such as tone, character, or chronology. Or, as in early prototypes of *Emma's Journey*, specify a global library of fragments with highly dynamic, condition-driven text, that could potentially appear within any scene in the narrative.

3.3 Fragments

StoryAssembler's core unit of narrative is the fragment. In their simplest form, fragments contain a main section of displayed text, and a list of choices. This is similar to the basic "passage" in Twine.

Fragments can contain pre-conditions that control when they're available. This can be used for causal or temporal ordering. Lastly (and most importantly) fragments can contain effects, which change the state blackboard. This is what is evaluated to determine if the assembled fragment makes progress towards the goal state specified in the story spec. Additionally, state modifications can carry through between scenes, allowing choices to affect later points of the narrative. An example of an authored fragment can be seen in Figure 4.

Fragments can also be compound, or composed of other fragments. For example, a fragment might contain explicit choices, but in place of "content" (the main text of the fragment) might instead contain a "request" for any valid fragment that increments tension. A valid fragment that satisfies that might have content but



```
{
    "id": "sampleFragment",
    "content": "The main text of the lexia, which can contain
    dynamic templates like {name|protagonist}.",
    "choices" : [
        {"gotoId": "sampleFragment2"},
        {"condition": "mainCharacterEmotion eq angry"}
    ],
    "effects" : [
        "set sampleFragmentVisited true",
        "addWishlist { 'condition' : 'explanation gte 4' }"
    ],
    "preconditions" : [
        "startExample eq true"
    ]
}
```

Figure 4: A sample fragment showing both static and dynamic choices, as well as modifying spec items through effects.

no choices. The resulting compound fragment would trigger the effects for both of the partial fragments, and would have the text from the second fragment, with the choices from the first.

There are a variety of methods to link between fragments, ranging from hard-coded and static to dynamic and state-driven. One can directly link (as done in traditional hyperfiction) to other fragments by fragment ID. More interestingly, dynamic links can be made by linking through desired state effects. For example, a choice linking via `roomTension incr 1`, will procedurally link to any fragment that fires the intended effect (as long as the preconditions are true). Identical state conditions can be used for multiple choices, and the system will choose different paths to satisfy them.

For example, one could write a fragment with three choices, two of which increase the tension in the room, and one that alleviates it. StoryAssembler will search the fragment library for unique choice paths that lead to those state changes, and link them in. This paves the way for potentially surprising juxtapositions of paths, and emergent readings that are driven by the underlying state changes and tracked stats.

Fragments don't always have to link to other fragments. If no choices are specified in the fragment, the system finds a new fragment (with valid preconditions) that satisfies one of the remaining spec entries, and links to it by creating a "Continue" choice. This frees up authors to author dramatic "beat" structures that have internal consistency, but are independently ordered.

Again, StoryAssembler's process is to construct fragments, then evaluate recursively through the choices afforded by the fragment, choosing paths that maximize state changes to fulfill spec entries. In general terms, this means StoryAssembler is always greedily assembling fragments that both contain as many goal-oriented effects as possible, and linking to the most goal-oriented fragments possible. A fragment's effects, choices, and preconditions therefore all play into that process, and must be kept in mind while authoring.

As mentioned before, StoryAssembler can also operate in a "live" mode, where at a given interval it re-evaluates the displayed text and choices. If the state has somehow changed while the player was reading the fragment (through an accompanying game mechanic, as with *Emma's Journey*) choices can become unavailable and grayed out, or similarly become available. This affordance can be used to inject urgency into player actions, or make procedurally poignant effects. For example, in a tense conversation with the dean, Emma might lose access to choices to steer the conversation where she wants, if the player's performance in the accompanying mini-game is lackluster.

3.4 Templated Text

Templated text can also be used to change the presentation of both fragment content and choice labels. This can be driven by scene state—for example, authors can add hedges like "umm..." to dialogue if the state variable "confidence" falls below a threshold. This is especially useful when used in tandem with live state modification, as with *Emma's Journey*, where it signals time-sensitivity to the player, and communicates their mini-game performance has narrative consequences.

We can also change content more substantially, so that fragments change under different state contexts. For example, we may decide to create a fragment "complimentHost" where Emma compliments the host's decor if dinner hasn't started yet, but compliments the food if that fragment is triggered when `dinnerStarted eq true`. In either case, the fragment's effect to decrease tension is fired, but the text surfacing that is contextual to the current situation.

Templates can also be used to reference character names, pronouns, and other individual mix-ins, so that fragments can be written that are not tied to a specific scene, but rather globally mixed into scenes if their pre-conditions are valid. This also gives authors the ability to cast scenes dynamically, or choose different character qualities before the scene starts that have a perceivable effect on the story. For example, a challenging character can be authored simply as "antagonist" and in a previous scene, set as whoever the player has a disagreement with.

4 AUTHORING PATTERNS

In the course of authoring *Emma's Journey*, some useful design patterns were formalized for procedural choice-based narratives.

4.1 Player-Driven Spec Entries

As mentioned earlier in Section 3.1, templated text can be combined with spec entries and exposed to the player to give them control over what types of goals they want to have in the scene. In earlier versions of the *Emma's Journey* interface, this was controlled via toggle switches and sliders, although the final version was a combination of clicking different temperature lines on a climate change graph, and cycling links on an initial scene description (Fig. 3).

Using this pattern, writers can manage complexity entirely through the dynamism of the story spec, and allow themselves to code more staticly-linked fragments within the actual work. In practice this led to story graphs where small sections were relatively self-contained, and then could be grafted on to each other depending on how the reader modified the spec entries before starting the scene. While this approach decreases the combinatorial possibilities for fragments (in that the self-contained sections are usually what get combined) making use of templated text can also keep this from seeming too similar between playthroughs, or nonsensical if a certain spec list combination is chosen that the writers didn't account for.

4.2 Effect-Driven Spec Entries

In addition to defining a fixed story spec with a number of entries as the starting place for the entire scene, it is also possible to define fragments within a scene that, when chosen, extend the story spec with additional entries through fragment effects (see `addWishlist` under "effects" in Fig. 4).

This feature allows the author to group related fragments causally through such effects and maintain internal consistency, without cluttering up the main story spec with items that are too specific to a particular branch or cluster of fragments. This could be especially useful for segmenting branches or clusters that may not always be included in the story, such as a collection of fragments corresponding to an optional subplot). In other words, this feature allows the author to define a nested hierarchy of spec entries, in which StoryAssembler picks fragments to satisfy "top-level" or initial spec entries that then augment the spec with additional entries to further flesh out the goals for this particular instantiation of the scene.

4.3 Parameterized Spec Entries

A scene's initial spec entries can be parameterized via templates, allowing state variables modified by other fragments to change them. In a StoryAssembler project that shares state between multiple scenes (such as the *Emma's Journey* project, discussed in greater detail in the following section), this feature permits the player's choices in one scene to directly modify the story spec that is used to generate subsequent scenes.

4.4 Fragment Available Both As Choice and Dynamically

When StoryAssembler searches for fragments to automatically continue the story (by bridging to them with a "Continue" link), it ordinarily ignores any fragment with a specified choice label. This prevents it from pulling in fragments that are meant to appear only as choices within other fragments. However, by creating an intermediary fragment with an appropriate choiceLabel but no main content of its own that then immediately "redirects" to another fragment, it is possible for the latter fragment (the target of the redirect) to be selected both as a choice in direct response to another fragment, and as a standalone fragment. This feature affords us a flexible way to hit mandatory story beats, either through chosen options that are more contextual, or forced through a Continue link if the player doesn't choose in that manner.

4.5 Multiple Choice Labels Into One Fragment

A variant of the previous pattern allows you to create multiple intermediary fragments—potentially with different effects, originating "parent" fragments, or choiceLabels—that function as multiple pathways into a single node. This pattern can be used to implement something like the dynamic choice labels used by Yu and Riedl [28] to present players with more personally enticing labels for the same choices given what the system could determine about the player's preferences.

5 CASE STUDY: *EMMA'S JOURNEY*

Emma's Journey is an experimental narrative game that juxtaposes a choice-driven narrative (implemented as a series of StoryAssembler scenes) with a succession of abstract mini-games generated by the Gemini system [25]. It served as a testbed for the development of StoryAssembler and is the first complete game to make use of the system. The side-by-side presentation of the narrative and generated games is reminiscent of Molleindustria's "two-channel narrative game" *Unmanned* [19]. Players experience a series of scenes from the life of the titular character Emma, initially a climate researcher preparing to defend her PhD dissertation, and make choices that influence the progression of Emma's career while the world's climate changes in the background.

In each scene the player is presented with a single generated mini-game, which may interact with the narrative in a variety of different ways. In one scene, for instance, Emma is eating dinner with her friends, and the player must play the mini-game to pass food around the table. Failure to do so with sufficient frequency will result in the game interrupting the narrative, blocking progression until the player passes the food again. In other scenes, the player must play the game to clean up a beach while holding a conversation with fellow volunteers; to maintain Emma's concentration during a lecture in front of her class (resulting in a narrative failure that ends the scene prematurely if concentration is totally lost); or to keep Emma's thoughts organized as she meets with the dean to discuss a possible change in position. [23]

Gemini, the system responsible for generating the mini-games that accompany each narrative scene, is a game generation tool based on what Treanor et al. term *proceduralist readings* [26]: interpretations of a game's "dynamics, aesthetics, and higher-level meanings" deduced directly from its mechanics, in conjunction with a corpus of cultural knowledge. Gemini accepts as input a file of desired "readings" or interpretations for the generated game, specified in the form of a logic program, and uses answer set programming to work backwards from these specifications and construct games that can be read in the desired ways.

These mini-games are in constant dialogue with StoryAssembler, in some cases affecting story variables in real time. We use this to toggle availability of narrative choices gated on character qualities, in a manner reminiscent of Quinn's *Depression Quest* [20]. Indeed, some of the most poignant affordances can result from this: a mini-game which requires constant attention as the reader watches Emma's narration grow increasingly nervous, or confident action choices graying out as the player's performance in the game lags behind.

The trade-off is that StoryAssembler must re-plan at each reader choice, since the mini-game may change the state in the interim before the next click, such that those pathway destinations are no longer optimal or even valid.

6 AUTHORING FOR *EMMA'S JOURNEY*

For *Emma's Journey*, six writers were brought on to tackle the task of writing eight scenes of content. At the time, they were all starting in an undergraduate games program, and thus had—at the time—somewhat limited exposure to programming, although an avid interest in it, and of course a high interest in games. A fair number of them had experience with systems such as Twine, but had never pushed into more procedural narrative territory.

As many practitioners might well know, authoring procedural narratives adds additional challenges to the myriad design considerations particular to creating static digital narratives. In the course of onboarding our writers, we fell into a 3-step rhythm of crafting high-level considerations, then moving into more granular authoring in later sessions.

Typically, the first design meeting for a new scene would be spent discussing and formulating the scene's beats and dynamics. These would be formalized as spec entries for the scene, so that we had a solid—if abstract—idea what would take place. These spec entries would be divvied up between writers as their section of the scene to write. In the next meeting, we would focus on drafting in fragments with placeholder text, but with appropriate effects to satisfy spec entries, and preconditions to trigger in an appropriate manner. Once those were locked in, only in subsequent meetings did we zero in on the fragment text itself, and integrated templates where necessary to surface the underlying state to the reader. This included things such as hedges when confidence dropped, or more confrontational language when anger was higher.

As mentioned above, fragments in StoryAssembler are stored in JSON files, and while there were occasional syntax debugging problems, we found it overall to be an accommodating and lightweight data format for writers to use. We also found the use of Tim Jansen's HanSON (JSON for Humans) plugin [15] incredibly useful, as it allowed authors to comment their JSON files with information necessary for narrative design, and to assist when collaboratively editing files together.

In feedback gathered at the end of the project, we found that a common thread of frustration for the writers was in "no path found" errors, which is when the system still had items on the story spec to fulfill, but no valid way to instantiate content from the library to fulfill them. The reasons for this could be myriad: an added fragment that shows up earlier than anticipated and invalidates a later fragment's preconditions; a typo on an effect that slips by; a removed spec item that decreases the weight of the expected fragment such that it no longer appears.

The knee-jerk reaction when confronted with these problems was to prescriptively restrict fragments with stringent pre-conditions, so that the writer knew the one place it could show up, and plan accordingly. However, that strategy undermines the core goal of the system to create generative narratives. If content is state-driven, but there's only one possible point the content can appear, with one set of choices, then it might as well be statically linked. Ideally, most content in these narratives should be capable of appearing in at least one other narrative position, in order to make the most use of the system affordances.

What was found, however, was that as we pushed more dynamism, the narrative debugging process became very slow and sensitive. Essentially, the disconnect was that changes authors made frequently had unintended or nonsensical effects on the structure of the story. This was further exacerbated by the push for longer scenes, which meant small tweaks to fragments that previously appeared at the beginning of the playthrough could change things near the end, which wouldn't be apparent unless you played through making the correct choices to get to that segment.

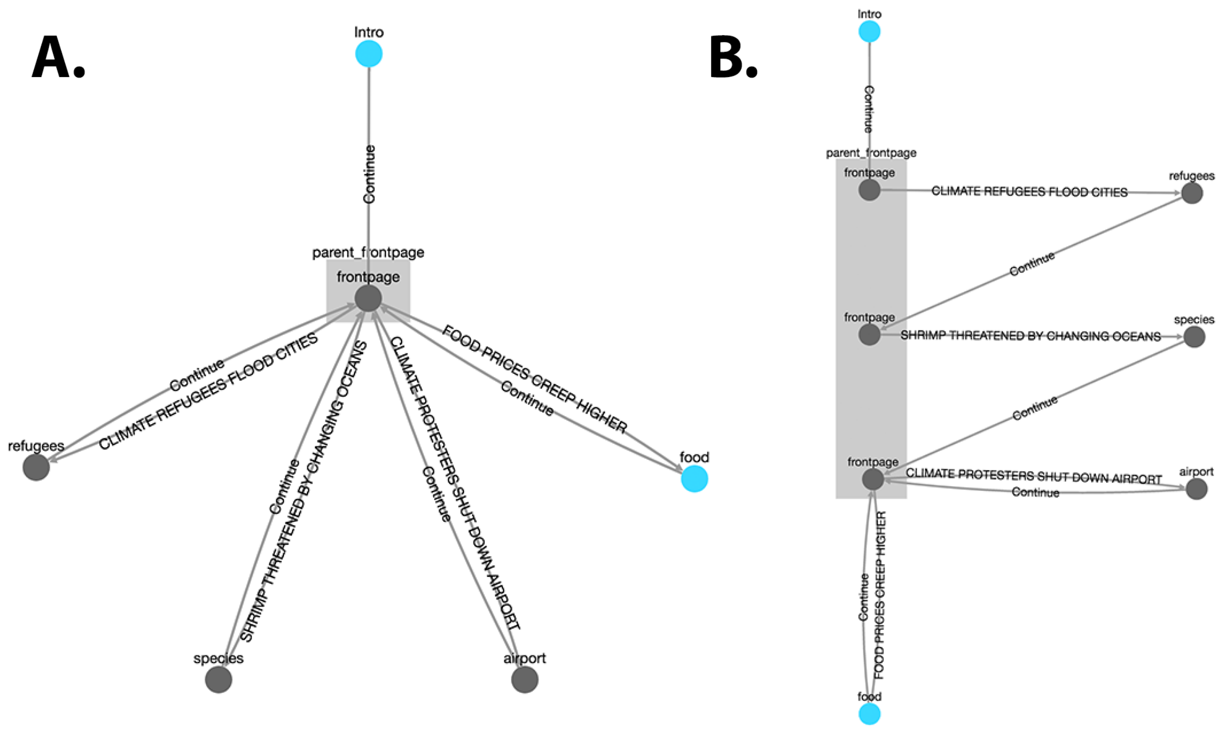


Figure 5: A test sequence with two different structures, based on whether nodes with articlesRead values are considered the same, or different.

In short, once we started creating narratives that were longer than test cases, it became highly desirable to see the total potential structure of the narrative, thus making the effects from small changes more visible.

6.1 Data Visualization

Without a tool, the only way to double-check content authoring is through exhaustive traversal of the choices. Due to their dynamic assembly, this means the entire structure must be re-verified with each added fragment or choice, to ensure a changed or added fragment isn't showing up in an undesired spot due to unforeseen state conditions.

Therefore, the visualization solution created for this problem needed to show three things: the assembled choice structure, when spec entries were being fulfilled, and how content was being re-used. Given that the code underlying StoryAssembler was under active development, we chose to collect data by aggregating automated playthroughs using the same procedures called in-program, so that any updates to how stories were formed would be reflected in the visualization. The resulting data was displayed as an interactive directed graph using the Cytoscape.js library [8]. We chose (and recommend!) this library because it comes with built-in graph traversal and layout algorithms, which streamlined some of the development. In addition to graph layout, other strategies were used to expose some of the underlying structure.

In the visualization, we specify a subset of the narrative's state variables to establish whether a node in the structure could be considered structurally identical under different contexts. A good example is a short test segment where Emma is sitting in an airplane reading the newspaper. After an introduction (which fulfills an introductory spec entry) she has a choice to read any of four articles, incrementing a state variable `articlesRead`. When four articles are read, the last spec entry is fulfilled and the segment ends.

If we set the visualization to not consider `articlesRead` as a differentiating state value, we get a flower-like structure with the central node as the recurring fragment (Figure 5a). If we set `articlesRead` as a differentiating state value, however, we get a different structure (Figure 5b) where the recurring node, although identical in content, appears as a separate entry. However, they are still grouped together in a box, due to their shared content ID. This type of viz flexibility is desirable, as there are cases where authors may want to change what is considered a revisited node, in order to ensure the proper state changes are occurring. This may also be a good strategy to use in works where most of the dynamism results from state, and the viewer can toggle which variables are being considered to differentiate a node as unique.

Blue nodes are used to signal when fragments are satisfying spec entries. Right-clicking nodes brings up a contextual menu, allowing the viewer to highlight narrative paths leading to this node. Additionally, the viewer can also use this menu to see which fragments were considered valid and invalid by StoryAssembler when it selected that fragment as the best fit in the progression. This is useful if the author has in mind a specific spot for a fragment, but it isn't currently being assembled there. Lastly, paths between nodes are also aggregated into thicker lines in the case where multiple unique playthroughs traverse the same connection. This helps convey the scale of difference between main paths and unique side paths at a glance.

Because of these features, authors were able to locate out-of-place nodes in initial scene drafts quickly, and see if tweaks to those nodes (be that in conditions, effects, or some other detail) resulted in correct placement across many branches. It would also show when nodes were inadvertently orphaned by changes to other parts of the story (typically by modifying state the orphaned nodes depended on) thus saving time in some of the initial stages of scene composition.

7 LESSONS LEARNED

7.0.1 Pragmatics. In terms of content authoring, while JSON is a lightweight data format to write in, it is still error-prone and unforgiving when commas or quotation marks are forgotten (errors that even seasoned programmers consistently make). Especially for authors who find dynamic narrative systems intimidating, simple syntax errors can be mistakenly attributed to more technical/systemic problems, discouraging them from engaging the system on more than a superficial level. Additionally, more complex fragments are composed of multiple data fields that are required for StoryAssembler to correctly parse, and errors ensue if any are forgotten.

To combat this, an effort was made to quickly bootstrap a webform-based authoring interface that would output the necessary JSON files [4]. While initial usage was favorable, more work was needed to handle edge cases resulting from advanced authoring structures. Rather than discouraging content using such advanced structures by splitting authoring into "tool-based" and more difficult "manual" groups, we opted to stick with direct JSON editing, leaving the tool in the realm of future work. When taken to completion, however, we believe this tool will significantly decrease simple bugs and greatly increase author productivity, especially when used in combination with the existing graph visualization tool.

There are two main takeaways from creating *Emma's Journey*:

(1) dynamic authoring hinges heavily on authors understanding the impact of new content on existing content dynamics
(2) a single procedural narrative system can enable dramatically different narrative designs, which can each have their own unique structural affordances and challenges

7.0.2 Dynamic Authoring and Gulf of Execution. Shneiderman's concept of "direct manipulation" is a useful framing for talking about authoring challenges for dynamic narrative systems. Essentially, direct manipulation is a goal for interfaces that can be used by novices easily, experts with fluency, and with a tight loop between an action taken in the tool, and its effect [24]. Because dynamic narrative creation already requires authors to keep many things in mind–the characters, the story, the aesthetics, the "possibility space" of the provided choices and how they may change given state–we want to minimize mental overhead for system-driven requirements as much as possible. Additionally, we want to enable authors to see these possibility spaces, and understand how their content changes affect it.

Hutchins et al. elaborated on direct manipulation as being the result of minimizing the *Gulf of Execution* and the *Gulf of Evaluation* [12]. The former regards the distance between action and realizing one's goal, the latter the overhead required to understand if one has reached it.

We found that StoryAssembler's ability to support both static and more dynamic processes meant that, when hard-to-find narrative bugs manifest, authors tend to reduce the procedurality of their story structures and link choices together statically just to "get something reliably working". Essentially, authors have a comfort zone for gulf of execution and evaluation, and will reduce content complexity to stay within that area. Given that we wanted to push the dynamism as much as possible, this capitulation over time somewhat undermined the long-term dynamic goals of the project. Our JSON editor attempted to shorten the gulf of execution, and the visualization the gulf of evaluation, but tool enhancements were needed to reliably represent more complex structures in longer narratives, which is exactly where authors need the most support. For the editor, an example is the aforementioned issues with edge cases in complex fragments (with dynamic choices or content compounded from several nodes). For the viz, a particularly thorny example is how best to represent the traversal graph where the indexicality of each node (fragment) is very high, but differentiating between different cases of state-dependent linkages is very important.

In the future, this might be avoided by treating authoring requirements and system capabilities more as equal partners. For *Emma's Journey*, it might have served better to balance complexity such that the authoring tools kept pace with system capabilities, rather than extending system capabilities beyond the reach of the authoring tools. While we did push the dynamism of the system a great deal, complex structures many times required extensive "design debugging" to ascertain why certain content was being displayed at certain times, which cut into time that could have been used to deepen the narrative content.

7.0.3 Narrative Design. Our initial goal was to reduce "authorial burden" usually incurred by static choice-based narratives, by virtue of a dynamic system that could re-combine and re-use content contextually. While the system certainly afforded us the capability to do so, we incurred a different burden in the form of dynamic narrative design challenges. More than some "gulf" of execution or evaluation in regards to an author's plan, authors (even ones with experience in other dynamic narrative systems) found it challenging just creating dynamic narrative structures that used the full system capabilities. The formalization of authoring patterns helped alleviate this, but committing time up front to working out the implications of complex system design, what narrative design is enabled by it, and what effect is desired through it, is something future projects in this space should account for.

For example, the three StoryAssembler authoring patterns regarding spec entries (player-driven, effect-driven, and parameterized) gave rise to very different narrative designs, emphasizing dynamism at different inflection points of the writing. For player-driven, it was all about state management of variables set before the scene even started. For effect-driven, it was more about establishing semi-autonomous groupings of content and managing state before and after a grouping executed, such that if one grouping ran, it didn't inadvertently invalidate another group's conditions. For parameterized spec entries, a longer multi-scene design approach was required, in order to properly surface how choices taken in earlier scenes were reified as plan changes in later scenes.

Recognizing the design patterns one's system affords and playing to those strengths is critical to content creation. Many times with procedural content systems, there comes a "tipping point" in authoring where the system's expressivity far outstrips what could be achieved with conventional strategies. However, until that point is reached, many times the artifacts the system presents to readers or players aren't representative of the system's full capabilities. Thus, discovering and developing these design patterns accelerates progress towards crossing that boundary, and the system expressing itself to its fullest capabilities.

As perhaps an even more generalizable lesson for systems in this space, *many of these issues only crop up during the process of creating fully-realized narratives with said systems.* Furthermore, the design process and formalization of how content is authored within these systems is an integral part of their contribution to the field. Procedural narratives require different design thinking than traditional narratives, often-times tuned to the unique affordances of the systems themselves. And while system re-implementations may be rare, the kinds of dynamic narrative experiences these systems are pursuing many times overlap. Therefore, system-driven design insights, gained through creating fully-fledged experiences with them, can be just as valuable as the technical insight gained. There's a rich field of inquiry in this direction to explore, and it's our hope that the continuation of work in this area results in a concomitant advancement in its design theory.

8 OPEN SOURCE AND FUTURE WORK

StoryAssembler has been open sourced as a JavaScript library, with the hope that others might try their hand at writing and further developing procedural choice-based narratives.

In keeping with the lessons learned through the creation of *Emma's Journey*, the focus of future efforts is on facilitating authoring. From an engineering standpoint, this means tackling further development of the authoring environment and feedback messages, such that authors without backgrounds in planning systems or even dynamic story generation can create interesting stories with it. Currently it's possible to author a wide variety of hyperfictions—from traditional fare (though dynamically assembled) to JRPG-style visual novels—complete with stat bars and state-driven character portraits (as seen in *Emma's Journey*). However, the development of template projects and tutorials are also much needed to jump-start the writing process for beginners, and provide a "recipe book" for all the different capabilities this system affords.

On the more technical side, StoryAssembler's architecture is set up such that, with a bit more work, it can facilitate swapping out different algorithms for the current forward state-space planner. It's our hope, with a bit more streamlining, that perhaps future contributors could find the library useful even as a game framework to interface with their own story generation systems.

9 PROJECT LINKS

- StoryAssembler is available as an open source library at https://github.com/LudoNarrative/StoryAssembler, along with a guide to getting started.
- *Emma's Journey* can be played online at https://emmasjourney.soe.ucsc.edu/.
- A demo of StoryAssembler's viz tool can be found at https://games.soe.ucsc.edu/storyassembler.

10 CONCLUSION

Hyperfictions, and specifically choice-based narratives, are an expressive and fertile medium for computational research. The persistent design challenge of crafting choices that account for player affordance and context without incurring prohibitive authorial burden is ripe for computational intervention, through systems that can dynamically assemble choice structures to meet those needs. StoryAssembler is a good first step in this process, enabling the creation of dynamic narratives without requiring programming knowledge. However, the revealed complexities of dynamic narrative design present their own unique challenge, and their own corresponding burden. To address this in the course of authoring StoryAssembler hyperfictions, design patterns were formalized that we feel will help in future work to streamline and simplify the design process, ultimately decreasing the burden of authoring. Regardless, StoryAssembler can serve as a happy medium between more technical systems tackling story generation, and hand-authored works from hyperfiction systems.

11 ACKNOWLEDGEMENTS

The authors would like to thank our many playtesters for their time and valuable feedback, as well as our dedicated team of writers. This material is based upon work supported by the National Science Foundation under Grant No. 1409992.

REFERENCES

[1] Sam Kabo Ashwell. Jan 26, 2015. Standard Patterns in Choice-Based Games. Retrieved Jan 3, 2018 from https://heterogenoustasks.wordpress.com/2015/01/26/standard-patterns-in-choice-based-games/

[2] Mark Bernstein. [n. d.]. Storyspace. Retrieved Jan 3, 2018 from http://www.eastgate.com/storyspace/

[3] Bruno Dias. [n. d.]. Raconteur. Retrieved Jan 3, 2018 from https://sequitur.github.io/raconteur/

[4] Jeremy Dorn. [n. d.]. JSON Schema Based Editor. Retrieved Jan 3, 2018 from https://github.com/json-editor/json-editor

[5] Kutluhan Erol, James Hendler, and Dana S Nau. 1994. HTN planning: Complexity and expressivity. In *AAAI*, Vol. 94. 1123–1128.

[6] Dan Fabulich. [n. d.]. ChoiceScript. Retrieved Jan 3, 2018 from https://github.com/dfabulich/choicescript

[7] Dan Fabulich. [n. d.]. Introduction to ChoiceScript. Retrieved Jan 3, 2018 from https://www.choiceofgames.com/make-your-own-games/choicescript-intro/

[8] Max Franz, Christian T Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D Bader. 2015. Cytoscape. js: a graph theory library for visualisation and analysis. *Bioinformatics* 32, 2 (2015), 309–311.

[9] Jane Friedhoff. 2013. Untangling Twine: A Platform Study.. In *DiGRA conference*.

[10] Malik Ghallab, Dana Nau, and Paolo Traverso. 2016. *Automated Planning and Acting*. Cambridge University Press, 37–39. https://doi.org/10.1017/CBO9781139583923

[11] Danny Goodman. 1988. *Danny Goodman's Hypercard developer's guide: the ultimate guide to Hypercard stack development.* Bantam Books, Inc., New York, NY.

[12] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct manipulation interfaces. *Human-computer interaction* 1, 4 (1985), 311–338.

[13] Jon Ingold. [n. d.]. Writing With Ink. Retrieved Jan 3, 2018 from https://github.com/inkle/ink/blob/master/Documentation/WritingWithInk.md

[14] Jon Ingold and Joseph Humfrey. [n. d.]. Ink. Retrieved Jan 3, 2018 from https://www.inklestudios.com/ink/

[15] Tim Jansen. [n. d.]. HanSON - JSON for Humans. Retrieved Jan 3, 2018 from https://github.com/timjansen/hanson

[16] Michael Lebowitz. 1983. *Creating a Story-Telling Universe.* MIT Research Lab Technical Report CUCS-055-83. Columbia University, New York, NY.

[17] Peter Andrew Mawhorter. 2016. *Artificial intelligence as a tool for understanding narrative choices.* Ph.D. Dissertation. UC Santa Cruz, Santa Cruz, CA.

[18] Ian Millington. [n. d.]. Undum. Retrieved Jan 3, 2018 from https://github.com/idmillington/undum

[19] Jim Munroe. [n. d.]. Molleindustria: Unmanned. Retrieved Jan 3, 2018 from http://unmanned.molleindustria.org

[20] Zöe Quinn. [n. d.]. Depression Quest. Retrieved Jan 3, 2018 from http://www.depressionquest.com/

[21] Mark O. Riedl and Robert M. Yound. 2010. Narrative Planning: Balancing Plot and Character. *Journal of Artificial Intelligence Research* 39 (2010).

[22] Justus Robertson and R Michael Young. 2018. Perceptual Experience Management. *IEEE Transactions on Games* (2018).

[23] Ben Samuel, Jacob Garbe, Adam Summerville, Jill Denner, Sarah Harmon, and Gina Lepore. 2017. Leveraging Procedural Narrative and Gameplay to Address Controversial Topics. In *2017 Workshop on Computational Creativity and Social Justice (CCSJW '17)*, G. Smith, A. Sullivan, and Brown D. (Eds.).

[24] Ben Shneiderman. 1981. Direct Manipulation: A Step Beyond Programming Languages (Abstract Only). *SIGSOC Bull.* 13, 2-3 (May 1981), 143–. https://doi.org/10.1145/1015579.810991

[25] Adam Summerville, Chris Martens, Sarah Harmon, Michael Mateas, Joseph Carter Osborn, Noah Wardrip-Fruin, and Arnav Jhala. 2017. From Mechanics to Meaning. *IEEE Transactions on Computational Intelligence and AI in Games* (2017).

[26] Mike Treanor, Bobby Schweizer, Ian Bogost, and Michael Mateas. 2011. Proceduralist Readings: How to find meaning in games with graphical logics. In *Proceedings of the 6th International Conference on Foundations of Digital Games.* ACM, 115–122.

[27] Hong Yu and Mark O Riedl. 2012. A sequential recommendation approach for interactive personalized story generation. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1.* International Foundation for Autonomous Agents and Multiagent Systems, 71–78.

[28] Hong Yu and Mark Owen Riedl. 2013. Data-Driven Personalized Drama Management. In *9th Artificial Intelligence and Interactive Digital Entertainment Conference.*