Fiascomatic: A Framework for Automated Fiasco Playsets

Ian D. Horswill
Northwestern University, Evanston, IL
ian@northwestern.edu

Abstract

We present *Fiascomatic*, a mixed initiative system for generating consistent scenarios for the indie storytelling RPG *Fiasco*. Players can repeatedly generate scenarios, locking down aspects of a scenario they like and regenerating aspects they don't, until they arrive at a scenario they find entertaining. It is not a story generation system; it generates scenarios from which players then generate stories. Nor is it intended to generate optimal scenarios; it generates random scenarios which the players can then curate according to their taste.

Fiascomatic presents an interesting intermediate point between non-automated table-top RPGs and fully automated systems such as story generators or autonomous characters. It is a tool that can be used by *Fiasco* players to speed the generation of game setups while preserving creative input on the part of the players, and by *Fiasco* playset authors to make automated playsets.

Introduction

Fiasco (Morningstar, 2009) is one of the best known of the new generation of so-called "storytelling RPGs": indie table-top role-playing games that emphasize improvisational acting over combat or complicated rule systems. It is a collaborative, GM-less game in which players improvise a two-act story in the style of the films of the Cohen bros. There is no winner or scoring system; the goal is simply to produce an interesting story. For an example of a *Fiasco* playthrough, see *TableTop* (Wheaton, Haislip, Burton, & Rogers, 2012).

The initial phase of a *Fiasco* game involves the players collaborating to design the scenario, called the *setup*, in which the action will take place. At the end of this setup phase, each player's character has an assigned relationship with the characters of the players sitting to either side of her, as well as an additional scenario element: a need the character has, an object in the game, or a location in which action may occur. The elements of the scenario act as resources for improvisation, rather than as rules, requirements, or objectives.

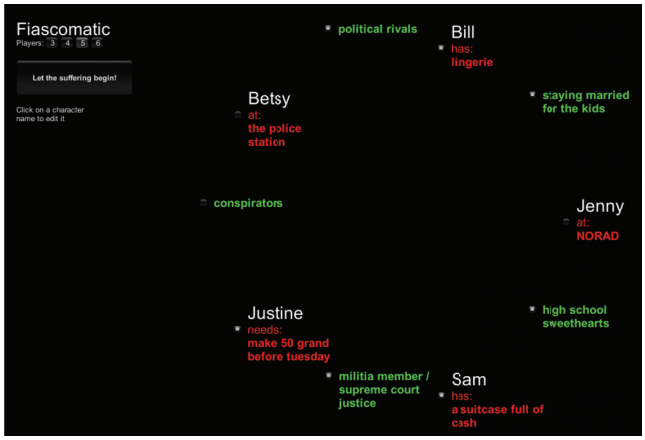

Figure 1: An example scenario. Relationships, shown in green, are read clockwise.

When designing the setup, players use a *playset* that provides lists of candidate relationships, objects, needs, and locations that are relevant to the theme or genre of the playset. Playsets encode conventions and tropes of particular genres, and are often explicitly designed as distillations of particular subgenres of cult films. For example, the *Touring Rock Band* playset (Morningstar, 2010) essentially formalizes the tropes of films such as *This is Spinal Tap* (Reiner, 1984).

One difficulty with playing *Fiasco*, however, is that while it is short compared to a *Dungeons and Dragons* campaign, it is still long enough to take multiple play sessions. The setup phase alone can easily take an entire play session. *Fiascomatic* is a framework for authoring AI-based playsets and generating scenarios from them. Playsets authored for *Fiascomatic* can generate complete setups for differing numbers of players, making it straightforward to play one-act versions of *Fiasco* in a couple of hours.

Figure 1 shows an example setup generated from a playset based on Washington politics. In it Bill and Jenny are politicians who staying married for the kids, Jenny had been high school sweethearts with Sam, who is now a

member of a right wing militia. Justine is a Supreme Court justice, who needs to raise $50K by Tuesday, and Sam has a suitcase full of cash. Justine is involved in a conspiracy with Bill's political rival Betsy. The specifics of all these relationships – what is the conspiracy, what are the aims of the militia, whether Sam is attempting to bribe Justine with the suitcase full of cash, or perhaps whether Justine needs the money so she may give a suitcase full of cash to Sam, who is blackmailing her – are all determined by the players during the course of the improvisation.

Fiascomatic is written in Prolog (Bratko, 2011) and runs under Unity3D (Unity Technologies, 2004). It began as a problem set for a course on knowledge representation for game AI. It solved the pedagogical problems of letting students play *Fiasco* in 50-minute class session and of providing an interesting domain in which students could see examples of simple knowledge representation.

Scenario generation

In normal human play, scenario (setup) generation is simply a matter of selecting relationships, objects, locations, and needs from lists specified in the playset. The selection process involves a dice pool and so is semi-random, but is primarily a deliberative process in which players take turns making selections, discussing with one another their options, and filling in elaborations such as backstory as they proceed.

In *Fiascomatic*, players choose a playset (authored in a KR language embedded in Prolog), select the desired number of players, and specify the names of the characters. By pressing a button, they can generate a complete randomized setup, and either accept it or generate a new one. If they generate a new setup, they can specify aspects of the previous setup to keep in the new scenario by clicking check boxes next to those elements.

A scenario consists of a set of relationships and details, one each per character. Relationships are represented as Prolog terms:

`relationship(`*CharacterA*, *Relation*, *CharacterB*`)`

where *CharacterA* and *CharacterB* are the two characters involved, and *Relation* is the name of the relationship holding between them, e.g. `old_flames` or `parent/child`. Details are also represented as the Prolog terms:

`detail(`*Character*, *Type*, *Value*`)`

where *Character* is the character the detail applies to, *Type* is the sort of detail it is (`object`, `location`, or `need`), and *Value* is the particular object, location, or need. For example, if Sheila needs to get $50,000 by Tuesday, that would be represented as:

```
detail(sheila, need,
       get_50K_by_Tuesday)
```

A setup for a three-player game between characters Sheila, Bob, and Chris would then be of the form:

```
relationship(sheila, R1, bob)
relationship(bob, R2, chris)
relationship(chris, R3, sheila)
detail(sheila, T1, V1)
detail(bob, T2, V2)
detail(chris, T3, V3)
```

Automated scenario generation thus consists of selecting random values for the italicized variables above from the relevant collections defined by the playset. However, many random combinations are nonsensical. For example, if Sheila is the manager of Bob, it's problematic for Sheila to also be the middle-school pupil of Chris.[1] For this reason, *Fiascomatic* playsets specify not only the possible elements of the scenario, but their logical implications so that the system can avoid contradictory combinations. These implicatures collectively define a genre logic for the playset, allowing authors to express rules such as that superheroes shouldn't have evil goals (assuming of course, the playset encodes a straight version of the genre and not a subverted or deconstructed version).

Playset definition

The playset definition file describes the playset's elements (relationships, objects, needs, and locations), along with the logical consequences of selecting particular elements, and what consequences are logically contradictory. The playset provides Prolog clauses (facts and rules) for *Fiascomatic*'s built-in predicates such as $relationship(X)$, $need(X)$, etc. However, the playset can contain arbitrary Prolog code. An example playset is given in the appendix.

[1] It is only problematic, however, not impossible; one can certainly imagine plots, particularly comedic ones, in which a middle schooler is also a manager or even a business owner, for example if they're operating a secret middle-schooler drug ring, or have unexpectedly inherited the family business, or they had been forced to drop out of school at a young age and are now returning as an adult. While these are all entirely imaginable scenarios for *Fiasco*, having too many of these sorts of dissonant combinations in a given scenario would make it difficult to play. It's therefore important to give the author a mechanism for filtering out unwanted dissonance.

Relationships

The most important playset elements are character relationships. In *Fiasco*, these are always binary relations such as "business partners" or "mother and child." Each player's character has a specified relationship with the characters of the players sitting to either side of them.

Relationships are declared to the system using the `relation` predicate. That is, the clause:

```
relation(lovers).
```

Declares that characters may be lovers. In this case, `lovers` is a symmetric relationship so one would actually use the declaration:

```
symmetric(lovers).
```

Telling the system that it can automatically infer that if the `lovers` relation holds between Sheila and Bob, then it also holds between Bob and Sheila. (Symmetry is not assumed by default.) Relations can also be declared to be anti-symmetric (meaning that aRb and bRa are contradictory), transitive, anti-reflexive, right-unique (meaning aRb and aRc implies $b = c$), and/or left-unique (bRa and cRa implies $b = c$).

Relations also have generalizations. If we declare:

```
generalization(lovers, friends).
generalization(friends,
               acquaintances).
```

Then the system will understand that friends are automatically acquaintances, and that lovers are automatically friends and therefore also acquaintances (the former may not be an appropriate assumption in some genres).

Again, playsets can be (nearly[2]) arbitrary Prolog code, so relationship declarations can take the form of more general Prolog rules such as:

```
symmetric(nth_cousins(N)) :-
   member(N, [2,3,4,5]).
```

Which declares that there is an `nth_cousins(`*N*`)` relationship for all *N* from 2 to 5.

Roles

Many relationships in *Fiasco* assign roles to the characters. For example, the parent/child relationship makes one character the parent, and another the child. These can be declared explicitly via the `roles_relation` predicate:

```
roles_relation(boss/flunky).
```

Tells the system that `boss/flunky` is a possible character relationship and that if Sheila is in the `boss/flunky` relationship with Bob, then Sheila is a `boss` and Bob is a `flunky`. Role relations are never symmetric.

There are a number of roles that are incompatible with one another. One can't, for example, be both a politician and a lobbyist at the same time. These can be controlled via the `conflicting_roles` predicate:

```
conflicting_roles(boss, flunky).
```

This states that one can't be the boss of one character while being the flunky of another. `Conflicting_roles` can also specify a list of mutually incompatible roles.

Objects, Locations, and Needs

Objects, locations, and needs are simpler. They are defined simply through the respective predicates:

```
object(ObjectName)
location(LocationName)
need(NeedName)
```

Like relations, names of objects, locations, and needs can be arbitrary Prolog terms (i.e. they can be record structures), and they can be defined through arbitrary Horn clauses. So for example, in the Washington DC Politics playset (see appendix), a class of needs is defined through the clauses:

```
need(hide_my_addiction_to(X)) :-
  drug(X).

drug(meth).
drug(crack).
drug(bath_salts).
drug(human_blood).
```

Stating that characters can have the need to hide their addiction, and the addiction can be any of: meth, crack, bath salts, or human blood.

Implications and Contradictions

Compatibility between different game elements is determined by finding the implications of each element and testing for contradictions.

More formally, the system determines a set of facts determined to be true in the scenario. Any selected scenario

[2] The current version of the solver does require that it be able to compute all possible relationships, needs, locations, and objects in a finite number of steps. Hence a version of the rule that allowed N to be any integer would throw the solver into an infinite loop.

element – a relationship, need, location, or object, e.g. `relationship(sheila, political_rivals, bob)` – is a fact. Other facts can be inferred using the implies predicate:

```
implies(AntecedantFact, ConsequentFact)
implies(AFact1, AFact2, ConsequentFact)
```

Which assert that *AntecedantFact* ⇒ *ConsequentFact*, and *AFact1*∧ *AFact2* ⇒ *ConsequentFact*, respectively. For example:

```
implies(relationship(A, R, B),
        relationship(B, R, A)) :-
   symmetric(R).
```

is used to implement symmetric relations. Similarly:

```
implies(needs(C,
              make_my_parent_suffer),
         role(C, estranged_child)).
```

states that that if a character *C* wants to make their parent suffer, then they must be an estranged child.

Consistency is tested using the `contradiction` predicate, which holds when two facts are inconsistent. For example:

```
contradiction(
  needs(C,
        make_50_grand_before_tuesday),
  role(C, billionaire)).
```

Which states that it doesn't make sense for a billionaire character to have a goal of making $50K, since for them, that is a small amount of money.

Finding a scenario

The system chooses a scenario using a randomized search of the space of possible elements (relationships, needs, objects, and locations), repeatedly trying combinations until one is found that is logically consistent. A combination is consistent if all facts in the deductive closure of the elements are pairwise consistent; that is, if they are consistent with one another, as are their implications, the implications of their implications, etc.

Elements are selected by first computing all possible elements of the desired type (relationship or detail) and then making a uniform, random choice from within the set.

Filtering for consistency is performed using forward-chaining inference. The system maintains a set of facts (Prolog terms) known to be true given the elements chosen so far. This list contains both the elements chosen, and their implications. When a new element is added, both it and its implications are tested for consistency with the existing fact set, and added to the set if they are consistent. If they are inconsistent, the choice of the new element is backtracked. The algorithm for adding to the database is simple:

add(NewFact, OldFacts)
  if NewFact ∈ OldFacts, return OldFacts
  if NewFact contradicts some OldFact, then **fail**
  let Implications = all implications of NewFact
  return addList(Implications, OldFacts)

addList(NewFacts, OldFacts)
  if NewFacts empty, return OldFacts
  return addList(rest(NewFacts),
                 add(first(NewFacts), OldFacts))

The actual code is written in Prolog, but is more concisely expressed in the pseudo-functional form above. Note that the above form is still non-deterministic in that it assumes a fail operation to backtrack the most recent choice of elements.

User interface

The original version of *Fiascomatic* used in class was a command-line program implemented in SWI Prolog (Wielemaker, Schrijvers, Triska, & Lager, 2012). The current version is implemented in Unity3D (Unity Technologies, 2004) using a Prolog interpreter that runs natively inside Unity. This gives non-programmers an easy way to install it and interact with it through a GUI.

The GUI provides controls for selecting the number of players, specifying the names of their characters, (re)generating a scenario, and locking individual generated elements so they are preserved when regenerating the scenario. The scenario is then displayed, with relationships displayed in green between the related characters, and details displayed in red below the character to whom the detail belongs.

Related work

While we are not aware of any work on this exact problem, *Fiascomatic* is certainly related to a number of other research areas.

Fiasco is a form of structured improv acting. Some researchers, such as Magerko and colleagues (Hodhod, Piplica, & Magerko, 2012; Magerko et al., 2009; O'Neill, Piplica, Fuller, & Magerko, 2011) have investigated the problem of making AI systems that can be full-fledged

partners in improv exercises. This is a much more difficult problem than the one solved here.

Fiascomatic could also be compared to story generators, although it would be extreme flattery to call it a story generator. There's been a great deal of work on story generation, dating back at least to TALE-SPIN (Meehan, 1977), and continuing to the present day with systems such as Pérez ý Pérez and Sharples' *MEXICA* (2001) and Ware and Young's conflict-based partial order planner (2011), Gervas et al.'s work on case-based reasoning (2005), and Zhu and Ontanon's work on analogy-based generation (2010).

There has also been a growing body of work on logic programming and constraint programming for procedural content generation in games, particularly using Answer Set Programming (Smith & Mateas, 2011). This has included modeling of formal rule systems (Smith, Nelson, & Mateas, 2010), level design (Smith, Andersen, & Mateas, 2012), and story generation (Chen, Smith, Jhala, Wardrip-Fruin, & Mateas, 2010). Other logics, such as linear logic (Martens et al., 2014) and exclusion logic (Evans & Short, 2014) have also attracted attention for interactive narrative.

Future Work

Although certainly useful in its current form, there is still much that could be done to extend the current system. One obvious extension would to add support for additional kinds of constraints, such as cardinality constraints. The standard rules of *Fiasco*, for example, require that there be at least one need in the game, but the current system does not enforce that requirement. Cardinality constraints are straightforward to implement in Prolog using attributed variables (Neumerkel, 1990), so this could be done straightforwardly in code. But it's less obvious how to present that capability within the GUI.

Another useful addition to the GUI would be the ability for players to specify attributes of their characters, so that players who wanted to play a character of a particular gender, race, age group, etc. could do so. This would require individual playsets to include the sufficient inferential information (implications and contradictions) to ensure the system could detect violations of the attributes.

One serious weakness of the current system is that one needs to understand Prolog to author new playsets. It might be preferable to use a structured natural language front end to allow authors from non-CS backgrounds to more easily author. This has been used very successfully in Inform 7 (Nelson, 2011). This would involve some loss of expressiveness over raw Prolog, but as Nelson argues, English is already well adapted to expressing the particular kinds of ontological assertions important to story worlds. In some cases English is more compact than the equivalent predicate logic assertions, e.g. because of the conciseness of quantifiers in natural language.

Finally, it should be observed that *Fiascomatic* is a nearly perfect application domain for Answer Set Programming (Brewka, Eiter, & Truszczyński, 2011). Implementing it using ASP rather than Prolog would allow considerably more expressiveness than the current system. The choice of Prolog was due primarily to run-time system constraints: a Prolog that could run inside a game was available, whereas running ASP inside a game engine is considerably more painful. However, this issue will presumably be rectified in the course of time. At that point, a reimplementation in ASP would be very appealing.

Conclusion

Storytelling RPGs such as *Fiasco* are an interesting point in design space for intelligent narrative researchers. Because of their improvisational character, they're truly interactive and so arguably a better model for future computational interactive narrative systems than standard Aristotelian narrative.

Making systems that can truly participate in storytelling RPGs like *Fiasco* is extremely difficult. However, simple kinds of computational assistance for these games, such as scenario generation, are more practical. *Fiascomatic* is low-hanging fruit in this space: it's a demonstrably useful system that is actually used by players in our group, but that uses relatively modest inference technology.

An interesting question is how one can incrementally extend a non-interactive scenario generation system such as *Fiascomatic* to actively monitor an ongoing game and inject interesting elements (complications, plot twists) without having to either implement AI-complete characters, or require a non-linear narrative (e.g. a set of branching plot points) to be authored in advance. If this were possible, it would provide a useful intermediate step between simple systems like the one presented here, and full-blown AI-based interactive narrative systems, which are difficult to build at all, much less make aesthetically successful.

Acknowledgements

I'd like to thank the reviewers for their suggestions and comments, and the students of EECS-395 for testing out the codebase and writing playsets.

Appendix: Sample Playset

The following is an early version of a playset in the domain of Washington DC politics. It is loosely based political satires such as *Yes, Minister* (Jay & Lynn, 1980), *The*

Thick of It (Iannucci, 2005), and *Alpha House* (Trudeau, 2013). It is provided to give the reader a sense of both the generativity of the system, and the limits of that generativity. At just under a page of text, it shows that playsets can be expressed relatively concisely.

```
%
% Relationships
%

roles_relation(politician/lobbyist).
symmetric(political_rivals).
implies(relationship(X, political_rivals, _),
        role(X, politician)).
implies(relationship(_, political_rivals, Y),
        role(Y, politician)).
roles_relation(politician/strategist).
roles_relation(politician/estranged_child).
symmetric(old_flames).
roles_relation(journalist/politician).
roles_relation(politician/billionaire).
roles_relation(politician/staffer).

conflicting_roles(
   [ politician, lobbyist, strategist,
     journalist, billionaire, staffer ]).

% A staffer or strategist can only work for
% one politician
right_unique(strategist/politician).
right_unique(staff/politician).

%
% Needs
%

need(hide_my_addiction_to(X)) :-
  drug(X).
drug(meth).
drug(crack).
drug(bath_salts).
drug(human_blood).

need(kleptomaniac).
need(streaking).

need(make_my_parent_suffer).
implies(needs(C, make_my_parent_suffer),
        role(C, estranged_child)).

need(retire_with_a_cushy_wall_street_job).
implies(
   needs(C,
         retire_with_a_cushy_wall_street_job),
   role(C, politician)).

need(become_the_first_tourettes_patient_to_be_ele
cted_president).
implies(needs(C,
become_the_first_tourettes_patient_to_be_elected_
president),
        role(C, politician)).

need(get_the_big_scoop).
implies(needs(C, get_the_big_scoop),
        role(C, journalist)).

need(make_50_grand_before_tuesday).
contradiction(
    needs(C, make_50_grand_before_tuesday),
    role(C, billionaire)).

need(hide_the_body).

%
% Locations
%

location(the_capitol_building).
location(a_corn_farm_in_iowa).
location(a_fact_finding_tour_in_the_bahmamas).
implies(
  at(C, a_fact_finding_tour_in_the_bahamas),
  role(C, politician)).
location(the_oval_office).
location(a_dc_pickup_bar).
location(the_watergate_hotel).
location(the_david_letterman_set).
location(the_police_station).

%
% Objects
%

object(ronald_reagans_ouija_board).
object(an_experimental_truth_serum).
object(half_a_kilogram_of_heroin).
object(the_nuclear_football).
```

References

Bratko, I. (2011). *Prolog Programming for Artificial Intelligence*. *International Computer Science Series*. Addison-Wesley.

Brewka, G., Eiter, T., & Truszczyński, M. (2011). Answer set programming at a glance. *Communications of the ACM*. doi:10.1145/2043174.2043195

Chen, S., Smith, A. M., Jhala, A., Wardrip-Fruin, N., & Mateas, M. (2010). RoleModel : Towards a Formal Model of Dramatic Roles for Story Generation. *Proceedings of the Intelligent Narrative Technologies III Workshop*, 1–8. doi:10.1145/1822309.1822326

Evans, R., & Short, E. (2014). Versu - A Simulationist Storytelling System. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(2), 113–130.

Gervás, P., Díaz-Agudo, B., Peinado, F., & Hervás, R. (2005). Story plot generation based on CBR. In *Knowledge-Based Systems* (Vol. 18, pp. 235–242). doi:10.1016/j.knosys.2004.10.011

Hodhod, R., Piplica, A., & Magerko, B. (2012). A formal architecture of shared mental models for computational improvisational agents. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 7502 LNAI, pp. 440–446). doi:10.1007/978-3-642-33197-8-45

Iannucci, A. (2005). The Thick of It. BBC Television.

Jay, A., & Lynn, J. (1980). Yes, Minister. BBC Television.

Magerko, B., Manzoul, W., Riedl, M., Baumer, A., Fuller, D., Luther, K., & Pearce, C. (2009). An Empirical Study of Cognition and Theatrical Improvisation. In *Seventh ACM Conference on Creativity and Cognition* (pp. 117–126). doi:doi: 10.1145/1640233.1640253

Martens, C., Ferreira, J., Bosser, A.-G., & Cavazza, M. (2014). Generative Story Worlds as Linear Logic Programs. In *Intelligent Narrative Technologies 7*. Milwaukee, WI: AAAI Press.

Meehan, J. R. (1977). TALE-SPIN, an interactive program that writes stories. In *Proceedings of the 5th international joint conference on Artificial intelligence* (pp. 91–98). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Morningstar, J. (2009). *Fiasco*. Chapel Hill, NC: Bully Pulpit Games.

Morningstar, J. (2010). *Touring Rock Band*. Chapel Hill, NC: Bully Pulpit Games. Retrieved from http://www.bullypulpitgames.com/wiki/index.php?title=Touring_Rock_Band

Nelson, G. (2011). Natural Language, Semantic Analysis, and Interactive Fiction. In K. Jackson-Mead & J. R. Wheeler (Eds.), *IF Theory Reader*. > TRANSCRIPT ON PRESS.

Neumerkel, U. (1990). Extensible Unification by Metastructures. Retrieved from http://mips.complang.tuwien.ac.at/ulrich/papers/PDF/meta90.pdf

O'Neill, B., Piplica, A., Fuller, D., & Magerko, B. (2011). A Knowledge-Based Framework for the Collaborative Improvisation of Scene Introductions. In *Proceedings of the 4th International Conference on Interactive Digital Storytelling*. Vancouver, Canada.

Pérez ý Pérez, Mike Sharples, R. (2001). MEXICA: A computer model of a cognitive account of creative writing. *Journal of Experimental & Theoretical Artificial Intelligence*. doi:10.1080/09528130118867

Reiner, R. (1984). *This is Spinal Tap*. Metro Goldwyn Mayer.

Smith, A. M., Andersen, E., & Mateas, M. (2012). A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game. In *International Conference on the Foundations of Digital Games*. Raleigh: ACM Press. Retrieved from http://users.soe.ucsc.edu/~amsmith/papers/fdg2012generation.pdf

Smith, A. M., & Mateas, M. (2011). Answer Set Programming for Procedural Content Generation : A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(3), 187–200. doi:10.1109/TCIAIG.2011.2158545

Smith, A. M., Nelson, M. J., & Mateas, M. (2010). LUDOCORE : A Logical Game Engine for Modeling Videogames. *Elements*, 91–98. doi:10.1109/ITW.2010.5593368

Trudeau, G. (2013). Alpha House. Amazon Studios.

Unity Technologies. (2004). Unity 3D. San Francisco, CA.

Ware, S. G., & Young, R. M. (2011). CPOCL: A Narrative Planner Supporting Conflict. In *The Seventh Annual International Conference on Artificial Intelligence in Interactive Digital Entertainment*. Stanford, CA: AAAI Press.

Wheaton, W., Haislip, A., Burton, B., & Rogers, J. (2012). TableTop, episode 8: Fiasco.

Wielemaker, J., Schrijvers, T., Triska, M., & Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*. doi:10.1017/S1471068411000494

Zhu, J., & Ontanon, S. (2010). Story representation in analogy-based story generation in Riu. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG2010* (pp. 435–442). doi:10.1109/ITW.2010.5593324