Imaginarium: A Tool for Casual Constraint-Based PCG

Ian Horswill
Department of Computer Science, Northwestern University, Evanston IL
ian@northwestern.edu

Abstract

Constraint programming offers a concise and powerful approach to a variety of procedural content generation problems (A. M. Smith & Mateas, 2011). However, it requires considerable technical expertise. No constraint-based PCG languages come close to Compton's goal of *casual creators* (Compton & Mateas, 2015).

In this paper, we describe *Imaginarium*, an interactive environment for developing ontologies for constraint-based PCG, targeted toward players of tabletop role-playing games. The system allows players to interactively describe the ontologies of their games using an *Inform*-like English subset, and explore the spaces of artifacts they generate. While much less expressive than a full logic programming language, it is quite competitive in terms of conciseness and readability.

Introduction

Procedural content generation (PCG) systems use generative rules to explore a space of possible artifacts, be they 3D models (IDV, 2009), dungeon levels (Toy *et al.*, 1980) or whole galaxies (Wright *et al.*, 2008). Rule systems can be embodied through code (Adams & Adams, 2006), machine learning systems (Summerville, *et al.*, 2018), or constraint satisfaction systems (G. Smith *et al.*, 2011), among others.

The technical background involved in constraint satisfaction makes it difficult for non-specialists to build their own systems. While a number of systems have been developed targeting end-users for specific problems, such as Mario level generation (Guzdial *et al.*, 2018; G. Smith *et al.*, 2011), broad coverage, end-user systems are difficult to develop.

Imaginarium is a constraint-based PCG tool designed for use in tabletop role-playing games. It allows users to interactively specify an ontology using a subset of English, then generate random instances of those objects, rendered as English descriptions. These objects can be thought of as constraint-based Mad Libs (Price & Stern, 1974) in that they involve making random choices for the objects' different degrees of freedom, subject to user-specified constraints.

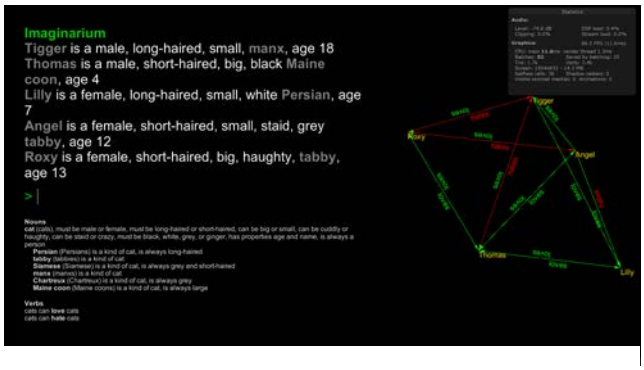

Figure 1: Screenshot of the tool in use

Imaginarium is heavily influenced by Nelson's work on *Inform 7* (Nelson, 2006b, 2006a). While ultimately a declarative programming language, the system seeks to leverage the affordances of natural language to allow non-programmers to use it without having first to learn the subtleties of first-order logic formalization or the deeper subtleties of answer-set programming (A. M. Smith *et al.*, 2012).

The user interacts with *Imaginarium* by typing (or loading from files) declarative statements about the objects to be generated, such as "humans are a kind of animal", "humans can be blond, brunette, red-head, or anime-haired," or "soldiers are crew-cut." The user uses these assertions to sculpt the possibility space of the artifacts to be generated, adding degrees of freedom with assertions like the first two, and removing them with assertions like the last. It compiles these ontologies to an SMT language and uses an off the shelf solver (Horswill, 2018) to randomly generate objects.

Example

Suppose we wanted to generate random cats.[1] We might type something like this:

> *imagine a cat*

[1] There are in fact several cat-based TTRPGs, including the recently Kickstarted second edition of Hanson's *Magical Kitties Save the Day!*

This tells the system to generate random cats. It implicitly, also tells the system that *cat* is a class of thing, membership in which is indicated in English using the noun *cat*. Since, this is all the system knows about cats, it can say only:

The cat is a cat.

We can remedy this by telling it:

A cat is large or small.

This tells the system that cats come in two flavors, large and small, the distinction being indicated using the adjectives *large* and *small*. The system now responds with one of:

The cat is a large cat.
The cat is a small cat.

Hitting return repeatedly generate new cats, but they will always be one of these two. The system interprets the form "a cat *is* large or small" as "a cat is *always* large or small." If we want to allow the cats to be neither large nor small, we can say:

Undo
Cats can be large or small.

We now allow cats whose size is unmarked, but there are still only three possible cats. So let's give the system more degrees of freedom to play with:

Persian, tabby, and Siamese are kinds of cat.

This tells the system *Persian*, *tabby*, and *Siamese* are all nouns and that they denote subclasses of the class cat. We can type more statements to tell it about more kinds of cats. When generating a cat, it will always be of one of the specified breeds. We might also tell the system:

Cats are longhaired or shorthaired.
Cats are grey, white, black, or ginger.
A cat can be haughty, cuddly, crazy, or Nietzschean.

Which define two mandatory properties (color and coat length) and one optional personality property, which can be left unmarked. We can now generate cats such as:

The cat is a large, shorthaired, white Persian.
The cat is a longhaired, ginger, cuddly tabby.

However, the first of these is problematic, since Persians are longhaired by definition. Adding the assertions:

Persians are longhaired.
Siamese are shorthaired.
Siamese are grey.

Prevents the system from generating cats who violate these constraints. Finally, we name our cats and let them have ages:

Cats have a name from cat names.
A cat is identified as "[name]".
Cats have an age between 1 and 20.

The first of these tells the system that all cats have a string-valued property called "name" that is drawn randomly from the list in the file `cat names`. The second tells the system to describe the cat it using her name rather than the generic identifier "the cat." We now get outputs like:

Puck is a Nietzschean, ginger tabby, age 12.
Mr. Muffins is a large, grey, Siamese, age 2.
Rover is a small, white, crazy Persian, age 9.

Finally, we say:

Cats can love other cats

Which introduces a verb *love* that represents an anti-reflexive, binary relation. If we now say:

Imagine five cats

The system will display a set of five random cats, together with an interactive visualization of the *loves* relation as a directed graph.

Comparison with traditional logic programming

The foregoing 13 commands are roughly equivalent to the 16-line AnsProlog (Baral & Baral, 2009) program :

```
entity(1..5).
cat(X) :- entity(X).
cat(X) :- persian(X).
cat(X) :- tabby(X).
cat(X) :- siamese(X).
1 { persian(X) ; tabby(X); siamese(X) }
1 :- cat(X).
1 { age(X, 1..20) } 1 :- cat(X).
1 { name(Cat, Name) : cat_name(Name) } 1
:- cat(Cat).
0 { large(X) ; small(X) } :- cat(X).
1 { long_haired(X) ; short_haired(X) } 1
:- cat(X).
```

```
1 { black(X) ; white(X) ; grey(X) ;
ginger(X) } 1 :- cat(X).
0 { cuddly(X) ; haughty(X) ; crazy(X) ;
nietzschen(X) } 1 :- cat(X).
long_haired(X) :- persian(X).
short_haired(X) :- siamese(X).
grey(X) :- siamese(X).
{ loves(X, Y) } :- cat(X), cat(Y).
```

Using Clingo (Gebser et al., 2010) produces the output:

```
entity(1) entity(2) entity(3) entity(4)
entity(5) cat(1) cat(2) cat(3) cat(4)
cat(5) long_haired(2) persian(2)
long_haired(3) long_haired(5)
persian(5) short_haired(1) siamese(1)
short_haired(4) grey(1) grey(4)
tabby(3) tabby(4) age(1,7) age(2,2)
age(3,12) age(4,1) age(5,7)
name(1,fido) name(2,fido) name(3,ru)
name(4,chi) name(5,puck) large(1)
small(1) small(2) large(3) large(4)
large(5) small(5) ginger(2) black(3)
ginger(5) cuddly(1) crazy(2) cuddly(3)
crazy(5) loves(1,1) loves(3,1)
loves(4,1) loves(5,1) loves(1,2)
loves(3,2) loves(5,2) loves(3,3)
loves(4,3) loves(1,4) loves(3,4)
loves(5,4) loves(1,5) loves(3,5)
loves(4,5) loves(5,5)
```

While far less expressive than AnsProlog, *Imaginarium* compares favorably in both conciseness and readability within its domain of competence.

Moreover, *Imaginarium* can infer most of the rules for translating a model into natural language directly from the syntactic structure of the program text. Other languages would require additional declarations. These properties, together with the system's interactive design, make much more novice-friendly.

Simplification of Natural Language

Although based on English, *Imaginarium* interactions are not true natural language. True NL is both complicated and ambiguous. Consider, for example, the assertion:

(1) *People can be friendly*.

Most English speakers would take this to means roughly that entities of the class people are sometimes, but not always, friendly. It licenses one to use sentences such as "Chris is friendly" or "Chris is not friendly." This is the meaning that our system applies to the sentential form "Xs can be Y."

Now compare this to the assertion:

(2) *People can be friends*.

Fluent English speakers understand this to mean that *pairs* of people can have the property of friendship. It defines an adjective *friends* whose domain is pairs of people, and which is roughly equivalent to the phrasal verb *be friends with*, so that (2) might be read as semantically interchangeable with:

(3) People can be friends with people

Unfortunately, we only understand this because we already know a great deal about the word "friends" and the concept of friendship more generally. An English-language learner who had not yet learned the word friend might hear (2) and interpret it in terms of the more common schema of (1), expecting that "friends" is a property of individual people, as in "Chris is so friends."

This introduces the question of whether a system like *Imaginarium* should allow users to use constructions like (2) that use "X can be Y" to introduce a binary relation instead of a unary predicate. This would require the system to be able to use context to distinguish cases (1) and (2). That context would have to be hard-coded into the system or manually entered by the user prior to the use of the construction (2). The user would therefore have to maintain a relatively detailed mental model of what context the system did and did not understand at a given moment, which would likely lead to errors and frustration.

For this reason, the system maintains a relatively direct relation between syntactic form and underlying semantics, one not present in real human languages. Nouns and adjectives always denote unary predicates, while verbs always denote binary relations. Syntactic forms using the auxiliary "can" ("*A*s can be *B*") always indicate possibility without necessity, while forms with no auxiliaries ("*A*s are *B*"), or which use the auxiliary "must" always indicate necessity.

Syntax

The English subset understood by the system is a regular language (Hopcroft *et al*., 2006). That is, it uses no constructions, such as center embedding, that require the full power of a context-free grammar. This is partly due to the limited expressivity of the SMT language to which the commands are compiled. However, the language is also deliberately constructed to have sufficient uses of closed-class words to make it easier to recognize failed attempts and

thereby generate better error messages. For example, in the sentential form:[2]

```
NP [can | must] BE AdjectiveList
```

The parser can determine in advance that the NP must be delimited by the words can or must. Any input that contains the word can or must followed by a conjugation of the copula is a match for this pattern, with the NP being all the tokens before this substring and the AdjectiveList all the tokens after it. Once the parser has parsed at this coarse granularity, it can parse the NP and AdjectiveList segments. This allows it to generate more useful error messages than it might otherwise generate.

The system allows phrasal nouns, verbs, and adjectives, meaning that individual concepts can correspond to multiple tokens in the input stream. This allows the system to accept concepts such as "being friends with" as verbs, without having to have a deep understanding of English syntax. It also allows users to use freely phrases such as "Lovecraftian horror", as in:

(4) A Lovecraftian horror is a kind of monster.

Without having to explain the meaning of either word in isolation. This has the disadvantage of introducing ambiguity, since some users might actually want to use the words in isolation:

(5) A horror is a kind of monster.
(6) A horror is Lovecraftian or Trumpian.

If, upon seeing (4), the system entered "Lovecraftian horror" into its knowledge base as an atomic, phrasal noun, then upon seeing (5) and (6) it would not know to update its existing knowledge about Lovecraftian horrors. Therefore, users must use nouns and adjectives in isolation before combining them. For similar reasons, phrasal nouns and adjectives may not be prefixes of one another.

Subject to these constraints, the system allows an NP to be any noun, optionally preceded by any number of modifiers (adjectives or other nouns) and/or a determiner or explicit quantity, such as:

Cats
A cat
Five scruffy cats
Lovecraftian horror cats
Five scruffy, Lovecraftian horror cats

[2] Here NP means, "noun phrase" and "BE," means some conjugation of the verb "to be", aka the copula.

The system does its best to inflect nouns and verbs for number (singular or plural) based on the standard rules for English inflection patterns, but these can be overridden, either by adding new rules to a configuration file, or by issuing a command:

The plural of Siamese is Siamese.

The built-in syntactic rules of the system use the standard English verbs *be* and *have*, along with the auxiliaries *can* and *have*. User-defined verbs can be any phrase that does not introduce ambiguity. They currently must be transitive.

Sentential Forms and their Semantics

The system currently understands the follow syntactic forms for declarations.

S `can/must` V `one/many/other` O
Asserts that the verb V defines a relation between instances of the classes defined by the NPs S and O, i.e. $V \in S \times O$. Modals and quantifiers add additional axioms:

- `can/many`
  Add no additional axioms.
- `must`
  There is at least one O for every S: $\forall s \in S. \exists o \in O. sVo$.
- `one`
  V is a function; for any S, there is at most one O. $\forall s \in S, o_1, o_2 \in O. sVo_1 \wedge sVo_2 \Rightarrow o_1 = o_2$
- `other`
  (when S and O are the same class). V is anti-reflexive. $\forall s \in S. \neg(sVs)$.

The construction "an S can V an O" is detected by the parser, but rejected because its semantics are ambiguous (it's unclear whether it means an S can V one or many Os).

S `can't/must` V `themself/themselves`
V defines a relation over $S \times S$. If the modal `cannot`, or synonyms (cannot, never, etc.) is used, the relation is anti-reflexive. If the modal `must` (or synonym always) is used, it is reflexive. Other reflexive pronouns (himself/herself/itself) may also be used.

S `can` V `each other/one another`
V defines a symmetric relation over $S \times S$.

S `is/are a kind of` O
S`, …, and` S `are kinds of` O
The noun(s) S define subclasses of O. An object being an S implies it is also an O. The subclasses of a noun form a

partition so that an O must also be exactly one of its subclasses.

S `can be/is/are` A
A is an adjective (monadic predicate) defined over the class S. The `is`/`are` form states that objects that are Ss must also be A. The `can be` form merely states that they may be.

S `can be/is/are` A`,…,or` A
Objects that are Ss can be at most one of the As. If the `is`/`are` form is used, then they must be exactly one.

S `has a/an` P `between` N_1 `and` N_2
All objects that are S have a numeric property named P in the specified range.

S `has a/an` P `from` F
All objects that are S have a string property named P whose possible values are given in the specified file.

V_1 `implies` V_2
V_1 `is mutually exclusive with` V_2
States, respectively, that verb V_2 is a generalization of V_1, or that they are mutually exclusive. For example, `love and hate are mutually exclusive`, or `loving implies knowing`.

V `is rare/common/very rare/very common`
Adjusts the probability of two individual being related by the verb V in the initial model used by the SAT solver.

Implementation

The system is written in C# and implemented using the Unity3D engine (Unity Technologies, 2004) and the *CatSAT* SAT/SMT solver (Horswill, 2018). It maintains a semantic network of the nouns, verbs, and adjectives that have been defined by the user, and the constraints applied to them such as implications and cardinality constraints. When the user gives an *imagine* command, such as "imagine five left-handed Balrogs," it creates a new CatSAT problem object and walks the semantic net, starting from the concepts mentioned in the imagine command, to generate the relevant propositions and constraints.

By only generating propositions and constraints for the parts of the ontology relevant to the specific generation request, the system can greatly reduce the amount of work that the SAT solver has to do to generate the results. This allows the user to create large ontologies, provided generation commands only touch upon manageable chunks of the ontology.

Having built the SAT problem, the system then calls the solver to find a model (solution). It then re-walks the semantic net to determine which nouns and adjectives potentially apply to each generated individual and tests their truth against the model, listing all relevant adjectives, but only the most specific relevant nouns (the lowest ones in the is-a hierarchy). Thus, if the generator knows the object is a Balrog, it will not bother to add the redundant information that it is also a monster.

Limitations and Future Work

Many things one would like to express in Imaginarium are not currently possible. Some of these are due to its having been in development for a little over two months. For example, the system cannot presently represent mixed-breed cats, such as tabby-Persians, but could easily do so simply by adding more complicated cardinality constraints to the language. Another example would be the ability to specify defeasible rules, such as "poodles are usually large," that could be overridden by other rules, such as "toy dogs are small," thereby allowing it to understand that toy poodles are small. While conceptually straightforward, these are not yet implemented.

Inherent technical limitations

A more fundamental problem is the system's reliance on bivalent logic. If a user tells the system that NPCs can be grumpy, grumpiness is an all-or-nothing trait. There is no notion of being somewhat grumpy, or situationally grumpy, save for the user enumerating those as new, separate, all-or-nothing traits.

Another limitation is its inability to specify probabilities for its random generation. While the system does provide commands for changing the probabilities of various propositions in the initial assignments used by the SAT solver's random walk, there is no simple relationship between a proposition's probability in the initial truth assignment and its probability in the final one, save that the latter is monotone in the former.

Normativity and inclusivity

Role-playing games are simulations: simplifications of some real or imagined world. The technical limitations discussed above add further pressure toward simplification.

These simplifications can become problematic when they involve social identity, such as race, class and gender. The system avoids these issues for the moment by leaving it to the player to formalize everything themselves: players can then adopt whatever models they prefer. However, this

strategy breaks down if we imagine a user community sharing and collaborating in the development of ontologies.[3] An ontology might have a very simple model of gender such as:

People are male or female.
Male people have given names from boy names.
Female people have given names from girl names.

However, a player wishing to use it might prefer a more nuanced model such as:

People are masculine-named or feminine-named.
People can be male presenting, female presenting, or non-binary.
Male presenting people are masculine named.
Female presenting people are feminine named.
Masculine named people have given names from boy names.
Feminine named people have given names from girl names.

However, players who don't wish to force names to be gender-marked, who come from cultures with different gender systems, who are playing games set in cultures with different gender systems, or who just don't want characters to be gender marked at all, must either accept the system's ontology or descend into the code to try to fix it. Players trying to combine ontologies created separately, such as mixing a "modern North American character" ontology with a magic user ontology in the hopes of making an ontology appropriate for urban fantasy, may find the components have irreconcilable models of social identity.

We are unlikely to find a general technical solution to these issues. However, a module system could potentially mitigate the issues by allowing gender (or class, race, etc.) systems to be authored independently of other systems and then combined by the user. The question here would be how to design such a module system, and to expose it to naïve users in an accessible English syntax.

Related work

A number of designer-facing rule-based systems have been developed for games. Perhaps the first such system is Nelson's *Inform 7* interactive fiction system (Nelson, 2006a, 2006b), which was a major influence on this work. Nelson argues that programming languages based on natural language, while inappropriate for general-purpose programming, are a good match for tasks like IF authoring, since the domain for computation is itself natural language text.

The Sims 3 used a simple forward-chaining production system to allow designers to author relations between character personality and behaviors (Evans, 2009). The *Versu* interactive fiction system (Evans & Short, 2013) used a far more sophisticated logic programming system. However, it proved extremely difficult for authors to use, and so Nelson developed *Prompter* (Nelson, 2013), an *Inform 7*-like natural language front end to *Versu* that proved more accessible.

Apart from *Inform*, the most successful designer-facing rule system is likely Compton's *Tracery* (Compton et al., 2014), a tool to allow naïve users to develop text generators based on context-free grammars. *Tracery* has a remarkably large user community, with many thousands of twitter bots alone having been built using it.

Finally, there are a few examples of designer-facing constraint-based PCG tools. The first and most successful of which is *Tanagra* (G. Smith et al., 2011), a constraint-based Mario level editor. More recent examples include *Gemini*, a game generator that uses ASP to reason about the aesthetics of the games it generates (Summerville, Martens, Samuel, Osborn, & Mateas, 2018), and *AutoDread*, a backstory generator for IF characters (Horswill & Robison, 2018).

Conclusion

Imaginarium is a very simple interactive environment for SAT-based PCG intended for use by non-programmers in tabletop role-playing scenarios. It provides an English-based declarative language in the style of *Inform* that allows users to incrementally create and share ontologies for in-game entities, and see randomly generated instances generated based on those ontologies.

While limited in its expressiveness compared to full logic-programming systems, its conciseness and familiar semantics make it a promising tool for use by non-specialists. That said, the system is early in its development and has not yet been tested by non-programmers. The next step is to put it in the hands of players who can evaluate it. We hope to do this in an undergraduate course in the coming year.

Acknowledgements

Many thanks to Ethan Robison, Adam Summerville, and Willie Wilson for advice and comments on the system, and to the reviewers for the comments on the draft.

[3] Such sharing and collaboration are important to the long-term success of any technology such as *Imaginarium*.

References

Adams, T., & Adams, Z. (2006). Slaves to Armok: God of Blood Chapter II: Dwarf Fortress. Bay 12 Games.

Baral, C., & Baral, C. (2009). Declarative problem solving and reasoning in AnsProlog*. In *Knowledge Representation, Reasoning and Declarative Problem Solving*. https://doi.org/10.1017/cbo9780511543357.005

Compton, K., Filstrup, B., & Mateas, M. (2014). Tracery : Approachable Story Grammar Authoring for Casual Users. *Papers from the 2014 AIIDE Workshop, Intelligent Narrative Technologies (7th INT, 2014)*, 64–67.

Compton, K., & Mateas, M. (2015). Casual Creators. *Proceedings of the Sixth International Conference on Computational Creativity June*. https://doi.org/10.1074/jbc.M409039200

Evans, R. (2009). AI Challenges in Sims 3. In *Artificial Intelligence and Interactive Digital Entertainment*. Stanford, CA: AAAI Press.

Evans, R., & Short, E. (2013). Versu. San Francisco, CA: Linden Lab.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., & Thiele, S. (2010). *A User ' s Guide to gringo , clasp , clingo , and iclingo ∗*. Potsdam.

Guzdial, M., Liao, N., & Riedl, M. (2018). Co-creative level design via machine learning. In *CEUR Workshop Proceedings*.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation* (3rd ed.). Pearson.

Horswill, I. (2018). CatSAT: A Practical, Embedded, SAT Language for Runtime PCG. In *AIIDE-18*. AAAI Press.

Horswill, I., & Robison, E. (2018). What's the Worst Thing You've Ever Done at a Conference? Operationalizing Dread's Questionnaire Mechanic. In *AIIDE-18 Workshop on Experimental AI in Games (EXAG-18)*. Edmonton, Canada: AAAI Press.

IDV. (2009). SpeedTree. Interactive Data Visualization, Inc.

Nelson, G. (2006a). Inform 7.

Nelson, G. (2006b). Natural Language, Semantic Analysis, and Interactive Fiction. Cambridge, UK: Unpublished white paper.

Nelson, G. (2013). *Writing for Versu*. San Francisco, CA: Linden Lab.

Price, R., & Stern, L. (1974). *The Original #1 Mad Libs*. Mad Libs.

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.

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. https://doi.org/10.1109/TCIAIG.2011.2158545

Smith, G., Whitehead, J., & Mateas, M. (2011). Tanagra : Reactive Planning and Constraint Solving for Mixed-Initiative Level Design. *IEEE Transactions on Computational Intelligence, AI and Computer Games*, *3*(3), 201–215.

Summerville, A., Martens, C., Samuel, B., Osborn, J., & Mateas, N. W. M. (2018). Gemini : Bidirectional Generation and Analysis of Games via ASP. In *Proceedings of the Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2018)* (pp. 123–129). Edmonton, Canada: AAAI Press.

Summerville, A., Snodgrass, S., Guzdial, M., Holmgard, C., Hoover, A. K., Isaksen, A., … Togelius, J. (2018). Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games*. https://doi.org/10.1109/tg.2018.2846639

Toy, M., Wichman, G., Arnold, K., & Lane, J. (1980). Rogue. Computer Science Research Group, UC Berkeley.

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

Wright, W., Hutchinson, A., Chalmers, J., Gingold, C., & Librande, S. (2008). Spore. Redwood City, CA: MAXIS/Electronic Arts.