UC Irvine
Software / Platform Studies

Title
Shaping Stories and Building Worlds on Interactive Fiction Platforms

Permalink
https://escholarship.org/uc/item/6pk7s4n6

Authors
Mitchell, Alex
Montfort, Nick

Publication Date
2009-12-12

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Shaping Stories and Building Worlds on Interactive Fiction Platforms

Alex Mitchell
Communications and New Media Programme
National University of Singapore
alexm@nus.edu.sg

Nick Montfort
Program in Writing & Humanistic Studies
Massachusetts Institute of Technology
nickm@nickm.com

ABSTRACT
Adventure game development systems are platforms from the developer's perspective. This paper investigates several subtle differences between these platforms, focusing on two systems for interactive fiction development. We consider how these platform differences may have influenced authors as they developed systems for simulation and storytelling. Through close readings of Dan Shiovitz's *Bad Machine* (1998), written in TADS 2, and Emily Short's *Savoir-Faire* (2002), written in Inform 6, we discuss how these two interactive fiction authoring systems may have influenced the structure of simulated story worlds that were built in them. We extend this comparative approach to larger sets of games, looking at interactive wordplay and the presentation of information within the story. In concluding, we describe how critics, scholars, and developers may be able to more usefully consider the platform level in discussions of games, electronic literature, and digital art.

Keywords
Platform studies, software studies, interactive fiction, authoring, interactive storytelling, adventure games, object-oriented programming, wordplay, paratexts.

1. INTRODUCTION
There are many adventure game development systems that provide mechanisms for defining a simulated world, the user interface, and ways of displaying graphical or text fragments. These systems are platforms that developers select to facilitate the creation and distribution of games. They include special-purpose programming languages for interactive fiction (text adventures), such as Inform 6 and TADS 2, which have been freely available and widely used since 1996, as well as Inform 7 and TADS 3, both released in 2006.

Because conventions are well established in both text and graphical adventure gaming, competing systems provide very similar high-level capabilities to author/programmers. Our investigation, therefore, is of subtle platform differences in adventure game development, particularly in the development of interactive fiction. We focus on how these differences may have influenced authors as they created games that involve simulation and storytelling.

We offer close readings and analyses of function, considering adventure games developed in two commonly used authoring tools. We begin with close reading of two interactive fictions. Dan Shiovitz's *Bad Machine* (1998), implemented in TADS 2, simulates an intricate, systematic world full of robots, made of components and functioning together in curious ways. The world model acts in ways that are mechanical and nested, making the code's class structure seemingly evident as one plays the game. In contrast, Emily Short's *Savoir-Faire* (2002), written in Inform 6 and of similar complexity, exhibits less obvious inheritance and compartmentalization. This game offers the possibility of magical relationships between objects that have similar appearance, objects which can be linked by the player character. Inform 6 uses attributes and properties more heavily, and sub-classing less often, to determine behaviour, an approach which relates to the type of similarity which governs the game's magic system. We follow this comparison with an extension to other games, comparing these games in terms of the use of interactive wordplay and the ways in which information is organized within the game.

We conclude by describing how critical and scholarly practice is able to better take into account the platform level and the development system when it comes to the analysis of games, electronic literature, and digital art. We also consider these platforms from the perspective of creators who are choosing a development system, noting some less-than-obvious ways in which these systems might influence the shaping of stories and worlds.

2. Background
2.1 Adventure Games and Interactive Fiction
Text-based adventure games, or interactive fiction [6], are text-based simulations that present a spatial representation of a world. The player, or interactor, types in text commands to a "parser", which interprets the interactor's commands. Responses are given to the interactor in text. Typically, the text presents a second-person description of the simulated world and the actions of the interactor (see Figure 1). The simulation of the world is persistent, and actions taken by the interactor alter the state of the world. Interacting with a text adventure often involves solving puzzles to accomplish some goal. A narrative is told in the process.

In the example in Figure 1, taken from Graham Nelson's 1996 reconstruction in Inform 6 of Will Crowther's *Adventure* (1976), the interactor's character is standing at the end of a road in front of a small brick building. The interactor types the command `enter building` to the parser, which responds by updating the position of the character in the simulation of the game world, and displaying a description of the new location. In the new location, the interactor asks for more information about one of the objects in the world (`examine lamp`), and then chooses to pick up the lamp

© Digital Arts and Culture, 2009.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission from the author.
Digital Arts and Culture, December 12–15, 2009, Irvine, California, USA.

(`get lamp`). As with the change of location, this second command changes the state of the simulated world, moving the lamp from the building to the character's "inventory". Finally, the interactor examines the contents of her inventory by issuing the command `inventory`. This interaction is a typical example of a session with a text adventure game.

```
At End Of Road
You are standing at the end of a road before a small brick building. Around
you is a forest. A small stream flows out of the building and down a gully.
> enter building
Inside Building
You are inside a building, a well house for a large spring.
There are some keys on the ground here.
There is tasty food here.
There is a shiny brass lamp nearby.
There is an empty bottle here.
> examine lamp
It is a shiny brass lamp. It is not currently lit.
> get lamp
Taken.
> inventory
You are carrying:
  a brass lantern
>
```

Figure 1. Interacting with a text adventure.

Historically, adventure games can be traced back to Will Crowther's *Adventure*, written in 1976 on a PDP-10. This original text adventure was expanded by Don Woods, which was quickly ported to other platforms. The first commercial adventure game, Scott Adams' *Adventureland*, a loose adaptation of *Adventure*, quickly followed, and the 1980s saw a range of successful commercial releases by companies such as Infocom and Sierra On-Line. After the demise of the commercial text adventure, graphical adventure games such as Lucasarts' *Maniac Mansion* and the *Monkey Island* series continued to offer similar puzzle-based spatial simulations, with text representation replaced by graphics. These graphical adventure games form part of the trajectory leading, through games such as *Mystery House*, *King's Quest*, and *Myst*, to modern games such as *NeverWinter Nights, Bioshock* and *Mass Effect*. At the same time, the wide availability of free development systems such as TADS and Inform has nurtured a healthy independent community of developers of text-based adventure games [1, 4, 6].

2.2 Platform Studies

The study of new media artifacts can be considered on five levels [9]. The first, *reception/operation*, focuses on the experience of the work. This encompasses approaches such as reader-response theory and reception aesthetics. The next layer, *interface*, concerns the interaction between the user and the core of the program, and the ways in which this impacts the use of the program. This includes fields such as human-computer interaction (HCI) and Bolter and Grusin's notion of remediation. The third layer, *form/function*, looks at the core of the program: the rules of a game, the nature of a simulation, the abilities of computer-controlled opponents, and so forth. Approaches such as game studies/ludology, cybertext studies, and narratology are chiefly concerned with this layer.

The fourth layer, *code*, involves the study of the source code of a program, and organizational and individual capabilities for software development. This includes looking at the comments, variable names, program structure, and choices made when writing a program, and is largely the domain of software engineering and new fields such as software studies. Finally, the *platform* layer is the abstraction layer beneath the code, underlying all the above areas. Platform can range from a standard or specification document, to a computer operating system, programming language or an environment on top of an operating system. Basically, a platform can be whatever it is that the programmer takes for granted when developing a program, and the user is required to have to use that particular software [9].

2.3 Related work

There have been a few critical discussions of interactive fiction that consider the implementation of the work. Most notably, in "Somewhere Nearby is Colossal Cave: Examining Will Crowther's Original "Adventure" in Code and in Kentucky" [4], Jerz talks about the cultural and social background of Adventure:

> "Adventure" was written for fun and shared for free; it was the cultural product of an educated, puzzle-loving, and fundamentally altruistic geek culture. Had it been better suited to the expectations of the non-technical public, it would likely have been less interesting to the community of computer specialists and entrepreneurs who responded by creating their own variations."

Jerz goes on to present a detailed discussion of the code, looking at the verbs and the help text and comparing the original Crowther version with Woods' later version. He examines the different features and details added by Woods. He also talks about the actual cave system that *Adventure* is based on, presenting a detailed discussion/ "multimedia intertextual analysis" of the game, map, and transcript. He extensively discusses the date of the game's development. While this analysis did not explore Fortran or the PDP-10 in depth, this type of close reading and bibliographic work shows a useful approach to studying a work as code for a particular platform.

3. STRUCTURING THE IF WORLD

In this section we consider Dan Shiovitz's *Bad Machine* (1998) and Emily Short's *Savoir-Faire* (2002), using these close readings to investigate the ways in which the platform used to implement an interactive fiction influences how the author structures the world within the work. *Bad Machine* (1998), implemented in TADS 2, simulates a factory filled with different types of robots. The robots are made of various standard sorts of components which themselves can be of different subtypes. The game looks like code (and error messages, and the outputs of an erroneous program, thanks to the unusual way that text is presented) but it also acts like an object-oriented program in very overt ways, presenting agents who are evidently of different subclasses and who are made of parts that are of different subclasses. In contrast, the similarly complex *Savoir-Faire* (2002), written in Inform 6, does not exhibit class structure as clearly. It features a notion of sympathetic magic as a way of creating behavioural relationships between objects, along with a system for recalling past episodes. Inform 6, unlike TADS 2, makes heavy use of attributes and properties to determine behaviour, an approach that relates to the type of similarity that governs the game's magic and remembering systems.

3.1 Bad Machine

In *Bad Machine*, the player controls Mover #005, a robot in a vast hive-like factory/warehouse who has suddenly developed the ability to act independently. On the surface level, *Bad Machine* may seem to be little more than a pastiche of computer code and other computer-like texts (see Figure 2). However, there is a deeper sense in which *Bad Machine* resonates with, and is heavily influenced by, the platform in which it is written.

```
reclamation / Area 17 / Warehouse IV                                   $/4
Un^t Sta&us:  B@$ xxxxxxxxxxxxxxxM39_I*~   [press any key]
Re-try:             BAk M3_eIN~   [pr3SSSS @ny k?y]
Re-try:                      BAp MAxHIN~   [pppp^^^^xxx @ kkkk$]
Re-try:                               BAD MACHINE [.]

BAD MACHINE

A TADS Adventure by Worker59
[Developed with TADS, the Text Adventure Development System.
Type 'about' for details, help, and licensing information]

Reclamation Sector (2)
  Cleared area amongst to-be-reforged bodies; gap(s) movement(allow) west, north;
other exits apparent lacking.
To the north you see salvager-class machine.

?examine self
Mover #005 :: Mover #005 I [Mover-class machine * Serial 27-005 * Power: 61 *
GO0d M3CH!N*]
Mover-class id #005; surface invent: mobility 100% (full operation) [unit(s): 317 318
319 320 321 322, CPU active <***WARNING: bad=machine***>, torso normal [560],
head normal [434].
Held (lef#/fr0nt) = [null]
Held (r2ght/fro_t) = [null]

?
```

Figure 2. *Bad Machine*'s heavily code-influenced prose

```
?examine energizer
Energizer-class machine id #<unknown>; surface invent: theoretical mobility 12.7%
(damaged operation) [unit(s): 903 nil nil nil 907 nil, CPU non-responsive, torso
<inaccessible>, head possible use (transfer possible) [704].
  Climber-class machine moves west.

?index 704
KEYWORD: 704
TYPE: Part
SUBTYPE: Headpart
(Standard model) energizer$$HEAD (normal)
Functions_supported: vision(standard),lowlight_vision()
–additional_note: <nil>
  [north]Climber-class machine leaves up.

?detach 704 from energizer
##Disconnect {perf!r!ed} *PART* re+moved

?detach head
Deactivating viSIon module ~~~ switching to (default hardware)
##Disconnect {perf!r!ed} *PART* re+moved

?attach 704 to self
Attachment successful.Activating vision modul0 ... OK OK radius = 2
```

Figure 3. Attaching a part to Mover #005.

The output seems to be code or "codework," with some of the messages appearing to be status messages and erroneous outputs. It perhaps reflects Mover #005's confusion and malfunctioning circuitry and effectively puts the reader in the position of the main character [7]. However, there is more than the surface play of confusion going on here. To the north is a salvager-class machine, and, as the interactor rapidly discovers, there are many other robots moving around the world of the Warehouse. Each of these robots behaves in different ways and executes its routines with robotic efficiency. The structure of the world, and its inhabitants, exhibit an inherently code-like, object-oriented nature, in a rather deep sense.

Specifically, modularity and the concept of an *interface* that form an integral part of object-oriented programming can be seen in the structure of the inhabitants of the Warehouse. As seen in Figure 2, Mover #005 has a number of *properties*: power, mobility, and so on. It also consists of a number of modular *parts*: a torso and a head plus 6 legs. The player will soon discover that it is possible for these parts to be removed and interchanged with other parts. Once a part has been attached to the player's character, the character inherits the properties and behaviours of that part.

For example, there is a puzzle early in the game where the player is confronted with a dark passageway that Mover #005 cannot enter, as it is unable to see in the dark. However, another robot, an "energizer," happens to have night vision. By removing the head from the energizer, removing the player character's head, and then attaching the energizer's head to Mover #005, the player can solve this puzzle (see Figure 3).

Not only the surface appearance, but also the structure and form of *Bad Machine* conveys an object-oriented, code-like aesthetic.

3.2 Savoir-Faire

The world of Emily Short's *Savoir-Faire* is more organic and magical. In this game, the player controls a young man who has returned, heavily in debt, to his childhood home, which he discovers has been abandoned.

Written in Inform 6, *Savoir-Faire* contains an interesting system of "sympathetic magic, called "Lavori d'Arcne", which lets the player *link* objects together based on similarities between the two objects, such that, for example, what happens to one object will also happen to the other object. This linking process succeeds or fails based on how "similar" the objects are. For example, it is possible to link a white, painted, openable teapot with a pair of white, painted, openable doors (see Figure 4).

```
Dining Room                                      Score: 13    Moves: 175

> examine doors
A pair of white-painted doors that lead into the upstairs corridor of the house. Each
door panel is decorated with the family crest, picked out in ostentatious gold, as
though to warn servants not to wander that direction uninvited.

> examine teapot
A hinge-lidded teapot, glossed in white and decorated with the crest of the family,
just as though it belonged to the family china pattern. The lid is attached with a
hinge, and there is a long and delicate spout.

> link doors to teapot
Bending your will, you form the link between the double doors and the hinge-lidded
teapot.

> close teapot
You close the hinge-lidded teapot.

The double doors close.

>
```

Figure 4. Linking a white, painted, openable teapot to white, painted, openable doors is successful..

This linking mechanism is consistently implemented, and does not appear to be hard-coded to specific, special cases that fit within the puzzle or story within *Savoir-Faire*. In fact, the player can attempt to link any object to any object, and the rules of the simulated world will apply. For example, attempting to link the same teapot to a little, grimy, linen (but openable) bag seems as

though it would succeed, if only the two objects were a bit *more* similar (see Figure 5).

> examine bag
A little linen bag on a grimy drawstring, able to be worn on the wrist or neck and protect one's valuables.

> link teapot to bag
You valiantly try to link the two objects, but they are just a little too different. Maybe if they looked more alike.

Figure 5. Linking the teapot to a little, grimy, linen (but openable) bag is not quite successful.

In the case where objects seem to have nothing in common, such as in an attempt to link the teapot to a clove of garlic, the complete failure of this attempt serves to re-enforce the consistency and completeness of the simulation (see Figure 6).

Savoir-Faire comes across as a more organic world, and not only because it is filled with household objects rather than robots. In *Savoir-Faire*, the visual and formal similarity of objects, rather than their compartmentalization and their place in a hierarchy, is the dominant feature.

> examine garlic
A clove of garlic, still good.

> link teapot to garlic
You valiantly try to link the two objects, but they seem as though they will never go together.

Figure 6. Linking the teapot to a clove of garlic is completely not possible.

4. IF DEVELOPMENT SYSTEMS

From the previous analysis of form and function within *Savoir-Faire* and *Bad Machine*, it is evident that there are differences in the way that the two works approach the simulation of the story-world. *Bad Machine* contains a very mechanistic, object-oriented world, whereas *Savoir-Faire*'s world is one of similarity and sympathetic magic. What about the platforms could have influenced the authors to take these very different approaches? We investigate this by looking closely at TADS 2 and Inform 6.

4.1 TADS 2

TADS 2, released for free in 1996, is an object-oriented programming language by Michael J. Roberts. It was designed specifically to support the creation of text-based interactive fictions.

The most important concept in TADS 2 is that of the *object*. A TADS 2 work basically consists of a set of code objects, each of which represents a physical object (or part of an object) in the game world. Every object belongs to at least one *class*. The notion of objects and classes is a basic principle in object-oriented programming. *Classes* define types or categories of things, such as "animal", "vegetable" or "mineral". In most object-oriented programming languages, *objects* are defined as *instances* of a specific class. As explained in the introductory chapter of the TADS 2 *Author's Manual* [12], "An object's class defines how the object behaves and what kind of data it contains." Each object has a number of *properties* and *methods*, which may be *inherited* from its class, or may be unique to this object. Properties contain data that tells TADS 2 about the object. Methods contain code that can be executed, typically providing access to and updating this data. Special properties such as *noun* and *adjective* tell the parser how a player can refer to an object.

In TADS 2, as in many object-oriented programming languages, a class can be a *sub-class* of one or more other classes; these are its *super-classes*. Any properties or methods of these super-classes are inherited by the sub-class. These properties and methods may be used directly, or the sub-class can *override* them, creating its own unique versions. This allows, for example, a programmer to create an "animal" class, which can have sub-classes "cat" and "dog", each of which would share the common attributes of an animal, but would provide specific behaviours unique to cats and dogs. Inheritance and object-orientation are addressed very early in the TADS 2 manual; object classes are mentioned in the first paragraph of the line-by-line discussion of the sample game.

TADS 2 includes a detailed *class library*, which contains a number of predefined classes, such as *item* (a standard item which can be, for example, picked up and dropped), *fixeditem* (which cannot be taken), *surface* (which can support other objects), and *chairitem* (which can be sat on). To create new objects with specific behaviours, the author can sub-class from one or more of these pre-defined classes and override behaviours as desired.

For example, a *bench* could be defined as follows:

```
bench: chairitem
  sdesc = "bench"
  ldesc = "The cold metal bench is, at least,
somewhere to rest."
  noun = 'bench'
  location = startroom
;
```

The first line, `bench: chairitem`, defines the new object, "bench", to be a sub-class of the `chairitem` object. The new object will inherit all the properties and methods of the `chairitem` class: the ability to be sat on, to support other objects, and so forth. The properties listed characterize the specific properties of a bench: its short description (`sdesc`) and long description (`ldesc`), which will be used at various different times by TADS to present the object to the player; the `noun` property which determines how the player can refer to this object (see below); and the `location` property which determines where the object is in the world. In this case, the bench is located in the `startroom`, which is itself an object, most likely an instance of the `room` class.

Another important part of any interactive fiction system is the *parser,* the subsystem that handles player input (text strings) and recognizes intended actions based on this input. In TADS 2, unlike other systems such as Inform 6, the parser is built into the interpreter.

To allow the player to take an action in TADS 2, the author needs to create a *method* on an object, one that will be called by TADS 2 when the player types the corresponding command into the parser. The system uses the `noun` and `adjective` properties on the object to determine which object the player is referring to. An important point to note here is that methods to implement verbs are written *as methods on objects*. As we will see below, this is in contrast to Inform 6, which defines verbs separately from objects and uses before/after properties on objects to create customized rules for specific objects.

The robots that inhabit the world of *Bad Machine*, with their distinct sets of behaviours and autonomous action, seem to almost be a consequence of the particular nature of TADS 2, which lends itself to the creation of sub-classes of objects. Using an object-oriented approach, it would be fairly straightforward to create multiple instances of, for example, a "salvager-class machine", which would actually be an instance of a class that inherits the basic robot behaviour from a "robot" or "machine" super-class. Similarly, the interchangeable parts on the robots, with their common *interfaces* and the *inheritance* of behaviours from super-classes, very much reflects the programming paradigms dominant in the development platform. Shiovitz was not, of course, somehow forced to make a game of this sort by TADS 2. Just as he used the texture of code and obviously computational outputs to constitute the surface appearance of *Bad Machine*, he used the underlying system of TADS to create a simulated world that is evidently object-oriented, providing an environment, puzzles, and figuration.

4.2 Inform 6

Inform 6 is a programming language with libraries developed by Graham Nelson specifically to support the requirements of authors of interactive fiction and released in 1996. Based on entries to the Interactive Fiction competition and the contents of the IF Archive, it has been the most widely-used interactive fiction development system since 1996; TADS 2 comes in second. There are three key concepts in Inform 6: the object tree and properties/attributes — discussed next — and verbs, discussed later.

When writing a game in Inform 6, the developer creates a set of *objects*, which are related hierarchically in terms of containment, in a type of graph called a tree. Every object has a position in the object tree, which indicates a parent-child relationship between objects. An object that is the child of another object is said to be "contained" in the parent object. This object hierarchy provides a concept of physical space, with top-level objects tending to represent rooms, and objects within top-level objects representing physical objects in the world. The player character and any non-player characters are also represented as objects within the object tree. Since any object can contain other objects, it is straightforward to create, for example, a container such as a box that can hold other objects.

Another key concept in Inform 6 is the notion of *attributes* and *properties*. Attributes are true/false values that are used to determine if an object "has" a certain attribute, whereas properties are variables that can have any value. These two concepts are used extensively in Inform 6 to determine how, for example, a verb should be applied to an object.

For example, a *bench* could be defined as follows:

```
Object  bench "bench" startroom
  with  description "The cold metal bench is, at
least, somewhere to rest.",
        name 'bench',
  has   static scenery enterable supporter;
```

The `Object` keyword specifies that we are defining a new object, which will be referred to as `bench` and will be a child of the `startroom` object in the object tree. The `with` directive tells Inform 6 that the object has a property named `description`. The description of the bench is containing in the text that follows. The `name` property tells Inform 6 that the object is named `bench`. Similarly, the `has` directive tells the system that the object has the following attributes: `static`, `scenery`, `enterable`, and `supporter`. Attributes can be defined globally, and then used by objects as required.

Inform 6 is an object-oriented language, providing the ability for programmers to define a *class* from which new objects can inherit. As with TADS 2, Inform 6 provides for class declarations and inheritance. In fact, the concept of objects is introduced in Chapter 3 of the *Inform Designer's Manual* [10], and on page 72 of the *Inform Beginner's Guide* [2]. However, in general, authors who work with Inform 6 tend to create a series of unique objects, all based on the built-in *Object* meta-class. In contrast to TADS 2, the most straightforward way to create interesting behaviours on an object is to define an attribute and then give the object that attribute using the `has` keyword. Classes in Inform 6 are distinct from objects; in TADS 2 any object can be a class. This makes it much easier for a TADS 2 author to decide to create a new object as a subclass of an existing object. An Inform 6 author has to plan a class hierarchy ahead of time, deciding which classes to create and then instantiating objects based on those classes.

As can be seen from the two definitions of `bench`, in TADS 2 and in Inform 6, there is a difference of emphasis in the structure of the code. In TADS 2, the first thing that the programmer needs to do is specify the *super-class* for a new object, in this case *chairitem*. In Inform 6, this can be done — in fact, the `Object` keyword is specifying the class which the new object belongs to. If we had, for example, defined a `Chair` class, we could have started our definition of bench with `Chair bench "bench" startroom`. The fact that the new object is deriving its attributes and properties from the Object class isn't as clear. It can seem to a programmer as if `Object` is simply a keyword that defines the start of an object definition, as opposed to actually specifying the class that this object is an instance of. The foregrounding of the *class* concept in TADS 2 affords consideration of class structure during the design of a work, whereas the emphasis on attributes and properties in Inform 6 focuses the author's attention more on these features of the system.

In Inform 6, a series of *libraries*, sets of program code that extend the basic functionality of the core system, provide the parser, a basic set of verbs, and grammar. The library also implements a *world model*, which provides concepts such as directions, food and drink, clothing, containers, doors, etc., and a simple turn-based model of time.

Finally, in Inform 6 verbs are defined as procedures that are separate from objects. A verb's default behaviour is specified within the procedure itself. It is possible to provide logic that determines different behaviours based on the subject and object of the verb. However, a more commonly used, and more flexible, approach is to make use of the `before/after` keywords in an object to customize the ways in which a verb is applied to specific objects. As a result, Inform 6 works often consist of a large number of objects, a large number of verbs, and before/after rules on objects that modify how the verbs apply to the object based on the attributes/properties of the object and other objects.

Clearly, *attributes* and *properties* provide an obvious way to approach understanding and implementing the sympathetic magic in *Savoir-Faire*. A number of attributes, such as openable, grimy, wooden, linen, and so on, could be defined, and objects compared based on these attributes. Similarly, properties such as colour and

shape could be assigned values and used to determine if a link is successful. Using *before/after* properties to define how to handle actions on objects, checking whether they have been linked or not and acting accordingly, would also be a natural way to handle the results of linking. Although a similar system could be developed in TADS 2, the focus on inheritance and class hierarchies does not seem to lend itself to this type of sympathetic magic. A set of base classes could define objects with certain sets of similar properties, but it is straightforward to implement these as properties in code.

4.3 Source Code Analysis

After developing the previous platform-based readings of *Savoir-Faire* and *Bad Machine,* we asked Shiovitz and Short for the source code to the games and for permission to discuss this source code in our writing at a high level. They provided us with the most recent Inform and TADS files. We believe that the sort of analysis we have done here applies in cases where the source code is lost (as might be the case with some early programs) or unavailable (as would be the case with current commercial games). So, we do not want to overemphasize the importance of source code for this general approach or to suggest that access to these files is essential. However, our ability to examine the source in this case has allowed us to see whether it bears out some of our specific claims.

The *Bad Machine* code is organized into multiple files, reflecting its highly object-oriented structure. Two files, named *parts.t* and *machines.t*, contain definitions of the classes for the robot parts and the specific robots.

The first file, *parts.t*, defines a complex class hierarchy used to implement the robot parts. A base class, `bodypart`, implements the common behaviour for robot parts. This base class has several sub-classes: `legPart`, `headPart` and `torsoPart`. There is also a `body` base class, which has sub-classes `inactiveBody` and `activeBody`. There is a further sub-class for `activeBody`, `Me`, which represents the player-character, Mover #005. This class hierarchy fully defines the base behaviours shared by the robots and their body parts. The second file, *machines.t*, contains a series of classes that are sub-classed from the classes defined in *parts.t*. These classes define the specific robot types. For example, there is an `energizer` class, sub-classed from `activeBody`, which defines the energizer robot that we described earlier. There is also an `energizerHead`, `energizerTorso`, and a series of energizer legs, which represent the various parts of the energizer. These are all sub-classed from the appropriate super-classes in *parts.t*.

The other files, such as *instances.t* and a series of files containing definitions of specific rooms within the game, make use of *machines.t* to instantiate specific objects representing the various robots and robot parts.

In total, *Bad Machine* contains 188 class definitions (excluding the standard library files), and has a maximum class tree depth of 7. For example, `boxClimber` is sub-classed from `climber`, `activeBody`, `body`, `item`, `thing`, and `object`. If we take out the classes from the standard library (`item`, `thing`, and `object`), this still gives a depth of 4 (`boxClimber`, `climber`, `activeBody`, and `body`).

This brief analysis of the code of *Bad Machine* confirms that, as discussed during our close reading and platform analysis, *Bad Machine* has a very elaborate class structure, very much in line with the game world and play experience.

The code for *Savoir-Faire*, in contrast, is largely contained in a single file, *stub.inf*. This file contains the definitions for the majority of the objects in the world, such as the rooms and their contents. It also contains the definition of a class, `Enchant`, which, together with the verbs `LinkSub`, `HLinkSub`, `RLinkSub`, `BadLinkSub` and `LinkableCheck`, contains the implementation of the magic system. Another file, *Mobile.h*, contains the definitions of the various types of objects that are used with the system of sympathetic magic. This file contains a list of attribute definitions, such as `flammable`, `fragile`, `hard`, `heavy`, and so on, and a list of values describing the material and shape of world objects. It also contains a base class, `Mobile`, from which both the `Enchant` class and a series of material-specific classes, such as `Stone`, `Metal`, `Cloth` and `Glass`, are sub-classed. Interestingly, these sub-classes consist largely of a list of attributes and properties. For example, `Metal` has its material property set to `METALMAT`, and has attributes `hard` and `heavy`. Specific objects, defined in *stub.inf*, are defined as instances of these material-specific classes.

Savoir-Faire contains a total of 33 class definitions (excluding the standard Inform 6 libraries and any extensions which may have been used). The maximum class tree depth is 5 (for example, `Chink` is sub-classed from `Mirror`, `Enchant`, `Mobile`, and `Class`). Taking out the standard base class, `Class`, this leaves a depth of 4.

Comparing *Bad Machine* and *Savoir-Faire* at the source code level, we see that, interestingly, the depth of the class tree (leaving aside the standard library classes) is the same. However, the number of class definitions in *Bad Machine* is much greater: 188 as compared to 33. Unlike the extensive use of classes for inheritance of behaviours seen in *Bad Machine*, the implementation of sympathetic magic in *Savoir-Faire* makes use of a smaller number of classes, largely for the inheritance of attributes and properties. The platform differences between TADS 2 and Inform 6 clearly do not prohibit the use of classes, but in these two game, which present themselves to the player as similarly complex systems, there does seem to be a difference in the degree to which classes are used, with TADS 2 possibly encouraging more extensive use of classes as compared to Inform 6.

4.4 Simulationism

Both *Bad Machine* and *Savoir-Faire* can be seen as examples of what has been termed *simulationism* in the interactive fiction community [8]. This term has been used frequently in, for example, discussions on the Usenet group rec.arts.int-fiction. A rough definition of simulationism is as follows:

> Simulationism is the tendency towards deeper and less abstract simulation of physical (and possibly emotional) properties of the game world, not for limited domains that the author has chosen, but as a general framework. Additionally, the "physics" of the world are likely to interact with each other leading to unforseen [*sic*] consequences. [5]

In *Savoir-Faire*, the use of *likeness* allows the player to *link* objects. This linking is not limited to specific, special cases determined ahead of time by the author. Instead, there is a rich, consistent simulation of a set of rules about the world, which the player can explore freely. In *Bad Machine*, there is a similar

consistency and richness to the world, where robot parts can be interchanged to create different behaviours, not just in pre-defined ways which match the solution to puzzles, but in a general way, a *simulation* of a specific world. Object-oriented programming is designed to model the world in terms of categories of things and sub-categories with similar properties, through a system of classes and inheritance. In *Bad Machine*, it is not a naturalistic world that is being modeled; instead, the author has taken an object-oriented model and made a world *out of this model*. Nevertheless, the result is, as with *Savoir-Faire*, a consistent, detailed simulation of a fictional world. Both of these are directly based on a classification model; both are *simulationist* in some way. What is interesting is *how* they approach that position, and the very different end results.

We are not claiming that choosing between Inform 6 and TADS 2 influences authors to be more or less simulationist. As can be seen in our two example works, both systems enable authors to create highly simulationist works. However, it is possible to go along with the mechanisms and features provided by the platform — with specific class structures in the case of TADS 2, or with attributes in Inform 6 — to build games that embody a simulationist perspective in specific ways. The platform provides a certain way of approaching problems that is more natural; the platform affords a particular approach.

5. WORDPLAY AND THE PARSER

Certain interactive fiction games implement interactive wordplay; they require the player to participate in making puns, using alliteration, or undertaking other linguistic tricks in order to solve puzzles and move forward in the game. These games can also create new languages that the player must figure out both to understand the game's text and to enter commands. Wordplay may characterize the entire game, in cases such as *Ad Verbum*, or may constitute one or more puzzles, as in *The Leather Goddess of Phobos*. Wordplay in interactive fiction can be done for many purposes: for humor value, for instance, or to connect to literary questions and practices of contemporary writing.

All text-based interactive fiction, by definition, involves the player typing in some *text*, which is read by a *parser*. The parser is that part of the interactive fiction system which is dedicated to breaking down the player's text into machine-understandable fragments, which can then be used to determine which verb the player wants to activate, and on which object(s) within the game world. The parser has a difficult task, as players may enter ambiguous sentences, make use of unexpected sentence constructions, or use any number of synonyms for verbs or objects.

In TADS 2, the parser, as described in *The TADS Parser Manual* [13], is partly built into the TADS interpreter, the program that actually runs a TADS game. The other portion of the parser is contained in TADS game code, which can further reside in two places: some code will be in the standard library provided by TADS, in the file `adv.t`, whereas additional code may have been written by the game author, and reside in the game-specific code. The code in the interpreter cannot be changed by a game author, although TADS 2 does provide hooks, opportunities to override what the parser does and to change the behaviour to match the needs of the game. In addition, code that is provided in the standard library can be changed or replaced. The parser at the interpreter level does not include any verbs, objects or prepositions — these are provided at the standard library level. This means that, according to *The TADS Parser Manual*, "there's very little of the built-in parser that you can't override".

In Inform 6 there is even more flexibility, however. The parser, as well as the grammar which it uses and the standard library defining the default world of a game, are all implemented as *libraries* which the author can choose to include (or not) in their game. As such, it is possible to, for example, replace the grammar used by the parser with that for another language, such as Spanish. In addition, it is possible to modify, or entirely replace, the parser itself. The Inform 6 parser also provides "hooks", allowing the author to selectively override specific behaviours. A major difference between the approach taken by Inform 6 and TADS 2 is that the Inform 6 interpreter does not include any of the implementation of the parser, or any other behaviour specific to interactive fiction. In fact, there is no such thing as an "Inform 6" interpreter — Inform 6 code is compiled to "z-code", a standard bytecode format which has its roots in the classic Infocom games of the 1980s, which can then be interpreted by a z-code interpreter such as Zoom or Frotz. All of the behaviours required to create the experience of interactive with a text adventure are implemented at the library level.

Based on these descriptions, the main difference between the parser in TADS 2 and Inform 6 is that, in TADS 2, some portions of the parser are contained in the interpreter or virtual machine, whereas in Inform 6 the entire parser is situated in the standard libraries. Both systems provide "hooks" for customization. However, despite these seemingly similar systems, which both allow for customization of the parser, it seems that there is a much greater propensity for authors to use Inform 6 for wordplay games, which often require extensive customization of the parser.

Table 1: Wordplay games by platform.

Platform	Game title and author
IFDB search: "wordplay"	
Inform 6	*Ad Verbum* (Montfort); *Exterminate!* (Martin); *Goose, Egg, Badger* (Rapp); *Letters from Home* (Firth); *The Gostak* (Muckenhoupt)
ZIL	*Nord and Bert Couldn't Make Head or Tail of It* (O'Neill)
IFDB search: "linguistics"	
Inform 6	*For a Change* (Schmidt); *Suveh Nux* (Fisher)
Inform 7	*rendition* (nespresso)
IFDB list: "Word-play games"	
Alan 2	*Puddles on the Path* (Raisanen)
Inform 6	*Beat the Devil* (Camisa); *Large Machine* (Ingold); *The Edifice* (Smith)
ZIL	*Leather Goddess of Phobos* (Meretzky)
Baf's Guide	
Inform 6	*Logic Puzzle Sampler* (Plotkin); *This is the game that I wrote* (Welbourn)
T/SAL	*Quest for the Sangraal* (Partington, originally written in T/SAL, ported to Inform 6)
MSDOS	*T-Zero* (Cunningham)
Spectrum	*Hide and Seek* (Brown)
TADS 2	*ASCII and the Argonauts: Astral Plane* (Berman)

We compiled wordplay games from several resources (see Table 1). We searched *The Interactive Fiction Database,* an online repository of interactive fiction information, for the keyword "wordplay," obtaining six games. We added to this three results from searching for "linguistics." To these, we added the games on the IFDB recommendation list "Word-play games," created by Emily Short. And, finally, we searched *Baf's Guide to the IF Archive*, an index to the IF Archive, for wordplay games. Of the 20 games that resulted, 12 are implemented in Inform 6, only one is in TADS 2, and one of the others is in Inform 7; none of the remaining games are in any version of Inform or TADS.

It is quite possible that other interactive fiction games exist with a substantial wordplay component. The IF Community resource *ifwiki* lists the TADS 2 games *Things* (2004) by Sam Kabo Ashwell and Jacqueline A. Lott, and *Verb!* (1998) by Neil deMause and describes them as wordplay games, but because of the nature of *ifwiki,* which is not categorized in the same way as the other resources, it is not clear if there are more Inform 6 games (or TADS 2 games) of this sort that are also listed there but which perhaps do not include the term "wordplay." There seems to be little reason to believe that the information at *IFDB* (which is hosted at the TADS website) and *Baf's Guide* would not be fairly representative, or would understate the number of TADS wordplay games relative to Inform games.

Based on 998 z-code and 335 TADS 2 games that Baf's Guide lists as available in the IF archive, the ratio of available Inform 6 to TADS 2 games is at most 3 to 1. (It is actually less, since the z-code games include some games that were not created in Inform 6.) But for wordplay games in particular, the ratio of Inform 6 to TADS 2 games seems to be 12 to 1. Although both TADS 2 and Inform 6 provide the ability for the author to customize the parser, Inform 6 seems to be the platform of choice for authors embarking on wordplay games. That platform may invite authors to add wordplay elements to games they are developing.

6. BOXES AND MENUS

There are some features for the presentation of information in Inform 6 that are not available be default in TADS 2. One of these, which may initially seem rather trivial, is the ability to display a *box*, which shows information in a way that is visually distinct from other information. For example, the first thing that a reader of *Curses* by Graham Nelson sees is a quotation presented in a box (see Figure 7).

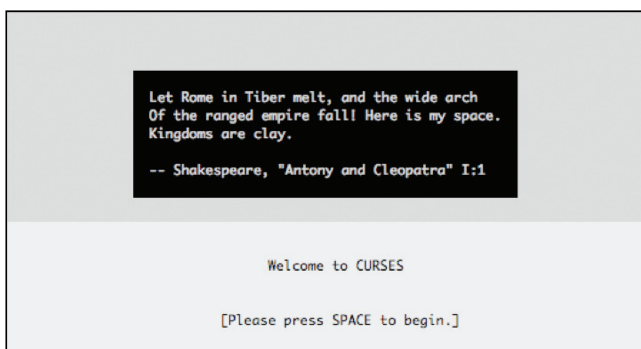

Figure 7: The use of a box to present a quotation in *Curses*.

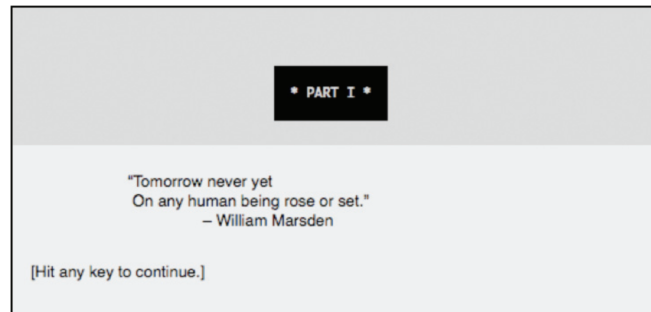

Figure 8: The initial screen of Meretsky's *A Mind Forever Voyaging*.

Information presented in this way is distinct from the in-game information shown as the player moves through the world of the game [6].

As Genette discusses, "text is rarely presented in an unadorned state, unreinforced and unaccompanied by a certain number of verbal and other productions, such as an author's name, a title, a preface, illustrations" [3]. These paratexts or *thresholds* have several significant functions; one is helping to situate a text within a specific form. For example, the title, endorsement and table of contents help to situate a *book* as a book for the reader. A book with no title page seems wrong to anyone who handles books. Similarly, interactive fiction works frequently contain certain paratexts. The use of boxes to mark the start of sections of an interactive fiction work, combined with epigrams or quotations, can be seen early on, for instance, in Steven Meretsky's 1985 *A Mind Forever Voyaging* (see Figure 8). Inform certainly refers back to the legacy of Infocom by using the z-code format. It also does this by allowing these Infocom-style paratexts to be easily generated.

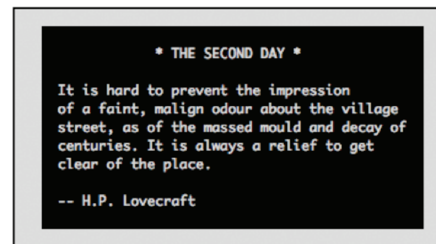

Figure 9: The use of boxes to present chapter headings in *Anchorhead*.

Many Inform 6 games make use of this form of paratext to create a specific texture. Boxes allow the author to present long passages of texts, separate from the in-game text. The use of quotations, for example, can create a very distinctive experience for the player, as can be seen in the T.S. Eliot quotations in *Curses*, and the H.P. Lovecraft quotations in *Anchorhead* (see Figure 9).

In addition to connecting these works to the body of IF works in the tradition of Infocom, the use of quotations and other literary paratext makes a strong connection between works such as *Curses* and *Anchorhead* with *books* themselves. Although these connections could arguably be made by a game written in TADS 2, the author would have to make a special. Boxes afford making these connections for the player.

Another feature that Inform 6 provides is the ability to create a pop-up menu, outside of the main text of a game, from which the player can make choices. These menus can be used to, for example, create a "help" system which exists outside of the world of the game, as can be seen in Admiral Jota's *Lost Pig* (2007) (see Figure 10).

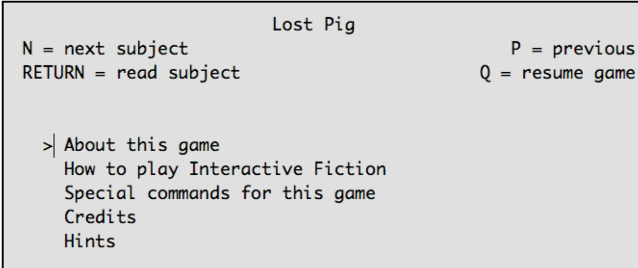

Figure 10: The use of menus to present out-of-game information in *Lost Pig*.

The way in which this information is presented is in stark contrast to the manner in which in-game information is conveyed. Descriptions, actions, and even the player's inventory are very clearly presented in the voice of the main character, Grunk (see Figure 11).

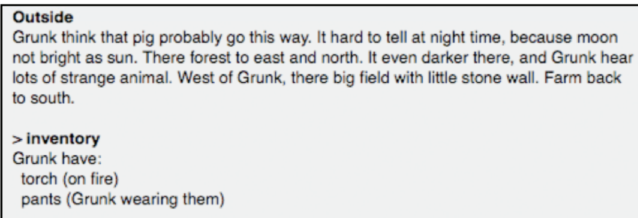

Figure 11: In-game presentation of information in *Lost Pig*.

Compare this with the help given in Suzanne Britton's *Worlds Apart*, which was written in TADS 2. Here, the help text is clearly *not* spoken by an in-game character. Nevertheless, it is typographically indistinguishable from in-game text (see Figure 12).

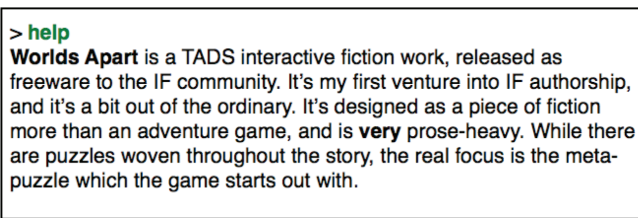

Figure 12: Help information presented typographically the same as description and narration in *Worlds Apart*.

However, as has been mentioned earlier, the fact that a platform does or does not provide a certain feature does not *prevent* the author from taking a given approach to a work, it merely makes it easier. As a counter-example, *Bad Machine* contains a hierarchical menu system (see Figure 13) even though TADS 2 does not provide this built-in facility. Interestingly, the menu is partially in-game — although it provides access to the credits screen, information on how to play the game, and so forth, some of its entries are "corrupted", presumably due to the damage to Mover #05.

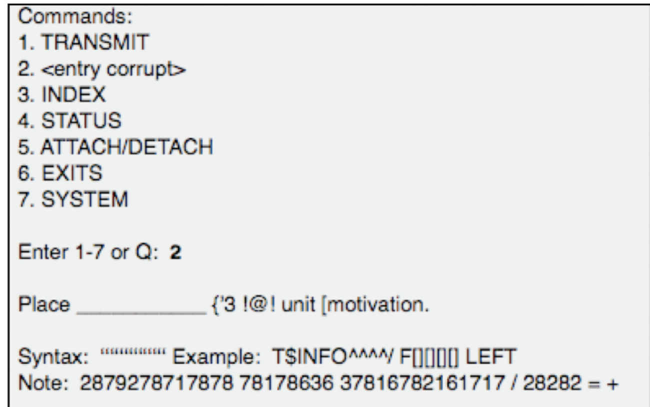

Figure 13: In-game help in *Bad Machine* suffers the same corruption as other parts of Mover #05's system.

Menus can also be used to present information about the world of the narrative. For example, in *Anchorhead*, written in Inform 6, many documents and artifacts, such as the box of newspaper clippings that the player discovers in the cellar (see Figure 14), are revealed to the player through the use of hierarchical menus. This information greatly enhances the richness of the world and the story being conveyed.

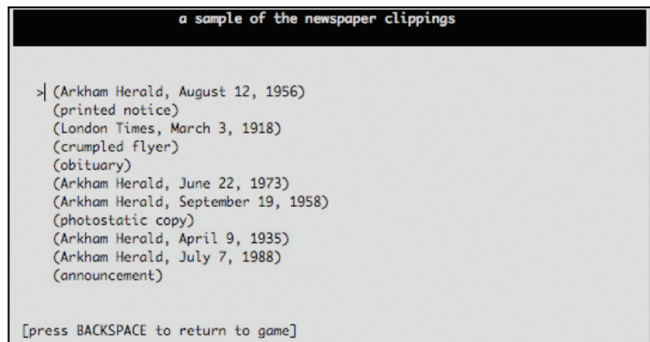

Figure 14: Documents and artifacts, presented as menus, are used to reveal the backstory in *Anchorhead*.

This can be contrasted with the way that information in a work such as Neil deMause's *Lost New York*, written in TADS 2, gradually presents information to the player through the environment and in-game descriptions. For example, as the player climbs the Statue of Liberty, towards the start of the game, a stone tablet can be seen at the top of the pedestal of the statue, just before the stairs leading to the observation deck. By typing `read tablet`, the player can read the poem "The New Colossus", together with a piece of racist graffiti scrawled on the wall beside the tablet, with both presented as in-game text.

Inform 6 provides a facility for creating separate menus as well as boxes, so it is straightforward to use these menus for help systems and to present information in a hierarchical manner. In TADS 2 there is no readily available facility to do this, although it is certainly possible for an author to create an ad hoc menu system, as in *Bad Machine*. The Inform 6 menu facility encourages authors to present more information and to do so in an out-of-game or at least off-the-command-line manner. In TADS 2, the author is encouraged to present information in more of an in-game fashion. Platform differences can be seen as *affordances* [11]. The pertinent question is not what a platform makes is possible, but what it makes *easier*.

The Inform 6 platform encourages the author to approach the creation of a world of interactive fiction from a certain perspective, and to present information in certain ways. Boxes create a kind of paratextual reference to a body of earlier work, namely the early Infocom games such as *A Mind Forever Voyaging* and *Trinity*, while at the same time encouraging the use of intertextual references and connections to literature, as seen in the use of T.S. Eliot quotes in *Curses*. Similarly, the menu system available in Inform 6 encourages the author to present information in an out-of-game fashion, in a manner not seen in TADS 2 works.

7. CONCLUSION

Different platforms accrete different types of games. Even subtle differences, such as those that exist between TADS 2 and Inform 6, can influence the ways in which developers approach creating new media works.

This type of analysis is of course not limited to text-based interactive fiction. Platforms for the development of graphical adventure games include Adventure Game Studio (AGS), used to create about a thousand games over the past decade, and the more recent Wintermute, which supports 3D characters and higher resolutions. Both provide similar capabilities (as with Inform 6 and TADS) but differ in minor ways. StorySpace and HyperCard may have greater differences, but both systems have been put to similar uses by authors of hypertext poetry and fiction. Home computers of different sorts also provided similar capabilities with significant minor differences. In all of these cases, a comparative analysis of platform and creative work could be enlightening.

For academics, it is important to take into account platform differences when it comes to the analysis of games, electronic literature, and digital art. Awareness of how platforms work, how they differ, and why developers and artists choose one over the others will help to inform the analysis of the aesthetic and cultural dimensions of the works.

From the perspective of creators who are choosing a development system, it is useful to consider the less-than-obvious ways in which these systems might influence the shaping of stories and worlds. Being aware of what a certain platform affords, will help a developer to make a more informed choice as to which system to use when considering a new work. This need for awareness extends to the influence of the platform on the outcome of development.

Finally, creators of new development tools and platforms should be aware of how the choices made in the design of these platforms will have an impact, directly or indirectly, on the works created on these platforms.

The platform analysis presented in this paper has shown that it is useful to examine implementation platforms in detail when analyzing new media works, even when the platforms are very similar. We hope this approach will prove to be of value to both academics and developers.

8. ACKNOWLEDGMENTS

Thanks to Dan Shiovitz and Emily Short for kindly sharing the source code for *Bad Machine* and *Savoir-Faire*. This research is supported by the Singapore-MIT GAMBIT Game Lab research grant “Tools for Telling: How Game Development Systems Shape Interactive Storytelling”.

9. REFERENCES

[1] J. Douglass. *Command Lines: Aesthetics and Technique in Interactive Fiction and New Media*. PhD thesis, University of California Santa Barbara, 2007.

[2] R. Firth and S. Kesserich. *The Inform Beginner’s Guide*. Dan Sanderson, third edition, 2004.

[3] G. Genette. *Paratexts: Thresholds of Interpretation*. Cambridge University Press, 1997.

[4] D. G. Jerz. Somewhere nearby is colossal cave: Examining Will Crowther’s original “Adventure” in code and in Kentucky. *Digital Humanities Quarterly*, 1(2), 2007.

[5] James Mitchelhill. Simulationism and IF (long). Usenet posting on rec-arts-int-fiction, 1 October 2005. [online] http://groups.google.com/group/rec.arts.int-fiction/msg/4ee60bfd7626bc42.

[6] N. Montfort. *Twisty Little Passages: An Approach to Interactive Fiction*. MIT Press, 2003.

[7] N. Montfort. A bad machine made of words: Review of *Bad Machine*, interactive fiction by Dan Shiovitz. *trAce*, 17 August 2004.

[8] N. Montfort. Playing to solve *Savoir-Faire*. In T. Krzywinska and B. Atkins, editors, *Videogame/Player/Text*, pages 175–190. Manchester University Press, 2007.

[9] N. Montfort and I. Bogost. *Racing the Bean: The Atari Video Computer System*. Platform Studies. MIT Press, March 2009.

[10] G. Nelson. *Inform Designer’s Manual*. The Interactive Fiction Library, fourth edition, 2001.

[11] D. Norman. *The Design of Everyday Things*. Doubleday, 1990.

[12] M. J. Roberts. *TADS 2 Author’s Manual*. 2002.

[13] M. J. Roberts. *The TADS Parser Manual*. 2000.