Loading… The Journal of the Canadian Game Studies Association
Vol 10(17): 93-111
http://loading.gamestudies.ca

# Game Studies at Scale: Towards Facilitating Exploration of Game Corpora

**John Aycock**
Department of Computer Science, University of Calgary
**aycock@ucalgary.ca**

## Abstract

Critically playing a game and performing a close reading of a specific aspect of a game are valid game analysis techniques. But these types of analyses don't scale to the plethora of games available, and also neglect implementation aspects of the games which themselves are texts that can be analyzed. We argue that appropriate software tools can support research in game studies, allowing individual games to be read at the level of gameplay as well as the implementation level, the level of computer code. Moreover, these tools permit analysis to scale in a similar fashion as the "distant reading" of digital humanities allows for traditional texts, and can be applied to an entire corpus of games.

We illustrate these ideas within a corpus of games created using the *Graphic Adventure Creator*, a program first released in 1985 for a number of computing platforms. As a proof of concept, we have built a system called GrACIAS – the Graphic Adventure Creator Internal Analysis System – that we have used for both static and dynamic analysis of this corpus of games, effectively allowing them to be internally explored and "read." Furthermore, our system is able to look for game solutions automatically and has solved over 60 game images to date, making the games accessible to researchers, but also to people who may not be expert players or even able to understand the language the game uses.

## Introduction

As the commercial value of older games diminishes over time, we are left with an embarrassment of riches. There are more computer games[1] available than ever before, and more options to play them. Certainly, the prevalence and quality of emulators has improved, and emulation has acquired a strong foothold even in commercial products like Nintendo's recent NES Classic Edition (Gartenberg, 2016) and the Virtual Console before it (Jones & Thiruvathukal, 2012). The most notable digital computer game collection, however, is perhaps the Internet Archive, whose Console Living Room, Internet Arcade, and library of MS-DOS games go beyond collecting software to making it playable in-browser (Internet Archive, n.d.-a, n.d.-b, n.d.-c). What can be done with these collections of games?

While making games available and playable for untold numbers of people is a big step, and one whose importance should not be minimized, it is something that only provides a single view on a game collection, and arguably a superficial one at that. Playing a game allows a person with sufficient skills

[1] We use "computer game" here rather than the familiar "video game" to be more inclusive, because not all computer games – especially old ones – necessarily have a video element.

and knowledge to experience a possibly proper subset of the game that is reachable from the game's start.

Let us examine that last statement carefully. "Reachable from the game's start" implies that not all of a game's content may be seen through normal gameplay, and in fact that is the case. For instance, content may be accidentally or deliberately captured in game media that cannot be seen from within the game (Aycock, 2014; The Cutting Room Floor, n.d.). Beyond the unseen, some games cannot even be completed in their shipped state: the 1984 game *Jet Set Willy* was a famous example of this (Byte-Back, 1984).

"Sufficient skills and knowledge" points to the fact that a game may also demand a certain level of mastery to fully experience, a level that some people simply may never be able to acquire no matter how much time they devote to the task. And, while various forms of cheating may exist (Consalvo, 2007), especially for popular games, they do not always allow the mastery hurdle to be circumvented. Although not something we consider in this work, it is worth remembering that there are some people who are physically unable to play games in their extant form, for whom exploring a computer game collection through gameplay would need a substantially different interface.

Finally, "possibly proper subset" speaks to what people, regardless of skill level, have experienced in games. Time invested scouring a section of a game does not always mean that 100% of that section is completed. From the explorability standpoint, it can be difficult to see all of a game through gameplay even when no other barriers exist.

The seemingly straightforward statement about gameplay clearly has a number of nuances to it when it comes to exploration of a computer game collection. From the computer science point of view, more troubling still is the fact that the playable game is an artifice. What the game programmer or programmers have directly written is the code underlying the game, the execution of which is experienced as the game. The code implementing the game is not explored at all in a gameplay-focused approach. Gameplay is a vantage point of a game from the surface, not an exploration of the internals of the code, and this is a loss because there can be many clever and interesting facets within games' implementations (Aycock, 2016b). A full collection exploration should permit the code underlying a game to be "read" in the same way as reading tools like Voyant (Sinclair & Rockwell, 2016) and Prism (Walsh, Maiers, Nally, & Boggs, 2014) facilitate for textual content.

Who is the audience for this? On the one hand, a toolset that supports broader forms of exploration enhances accessibility for non-experts, as we will illustrate through our system's capability of automatically finding game solutions. The label "non-expert" does not preclude researchers, either. Fernández-Vara's (2015) book on game analysis says that the "goal should be becoming an expert on the game" but concedes that "becoming an expert player requires dedication, and not everybody has the time, the inclination, or the talent to become a top-notch player" (p. 24). A system like ours can thus assist game studies researchers. For example, one can imagine wanting a walkthrough for an obscure or newly discovered game under study, or the ability to compare the vocabulary and language structure across hundreds of games.

On the other hand, our system also allows static and dynamic analysis of a game's internals that can benefit an even wider variety of research audiences. Obviously, computer scientists deal with computer code, but so do researchers in platform studies (e.g., Montfort & Bogost, 2009; Altice, 2015), software studies (Fuller, 2008), and critical code studies (e.g., Montfort et al., 2013). The growing field of archaeogaming, "the archaeology both in and of digital games" (Reinhard, forthcoming, p. 1) is another area where code can play a role.

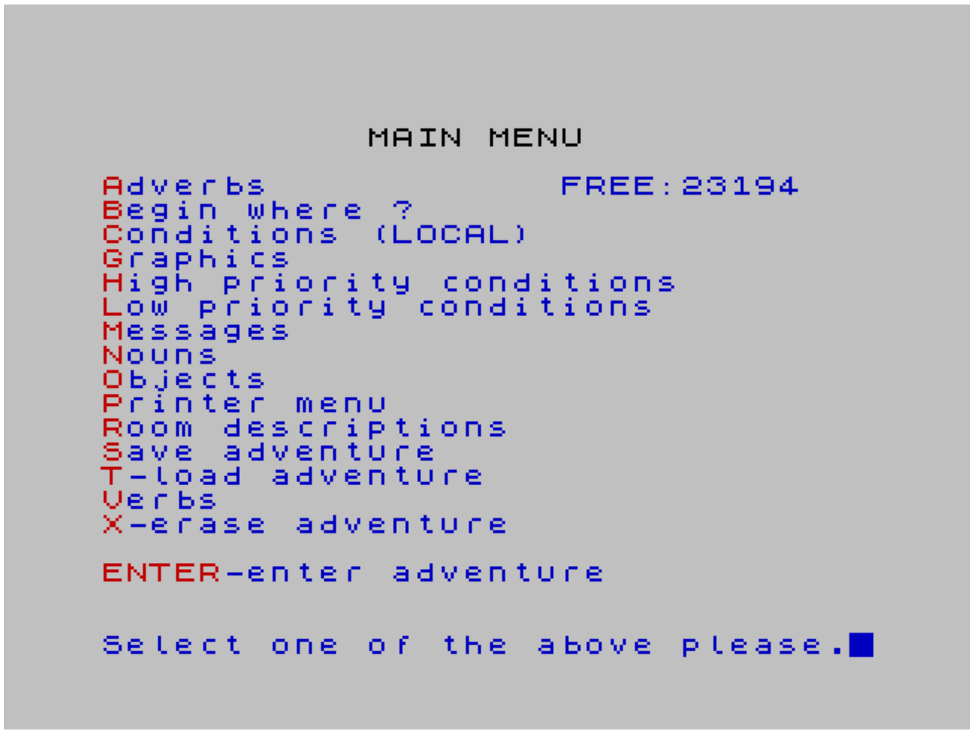
In the remainder of this paper we detail our work towards facilitating exploration by all these groups. We first present the game-creation tool, *Graphic Adventure Creator*, that we used as the basis of our collection and exploration. The two subsequent sections delve into our GrACIAS system, both how it works as well as how it was used. While we have tried to keep the high-level concepts approachable, there is necessarily a fair amount of technical content to explain exactly what we are doing, how it might be repeated, and how future researchers may build on this work. Related work and our conclusions comprise the final two sections.

## *Introducing Graphic Adventure Creator*

We use old, "retro" computer games for our proof of concept for two key reasons. First, they provide an opportunity to work with a set of games that provided manageable functionality; we do not have to worry about complicating factors like downloadable content or game servers' existence in most cases. Second, their implementation could not be overly complex due to technology constraints of the time, constraints on processor power and speed, memory capacity, and secondary storage speed and capacity. This makes working with retrogames much more tractable than modern games – if a proof of concept cannot be constructed for retrogames, then that does not bode well for modern releases.

Specifically, we turn to games in one of the earliest computer game genres: text adventures, also known as interactive fiction, which date back to the mid-1970s with the hugely influential *Colossal Cave Adventure* (Jerz, 2007). For those unfamiliar with this form of game, textual descriptions of the player's surroundings are presented, and the player makes moves in the game by issuing textual commands; part of the game experience, for better or worse, is determining the vocabulary the game understands. Once this genre of games appeared on personal computers with graphics capability, they quickly became enhanced with images to accompany the text, much like illustrations in a book.

Another factor working in favor of text adventures is related to their implementation. Without the same low-latency requirements as, say, an arcade-style game, a number of text adventure games were implemented using interpreters (Aycock, 2016b), an implementation technique that was slower but had advantages like size and portability. In essence, these games' code was expressed in a domain-specific (interpreted) language, and that specificity makes the code exploration we do easier to interpret by target audiences, as opposed to more general purpose code.



```
                MAIN MENU

Adverbs                   FREE:23194
Begin where ?
Conditions (LOCAL)
Graphics
High priority conditions
Low priority conditions
Messages
Nouns
Objects
Printer menu
Room descriptions
Save adventure
T-load adventure
Verbs
X-erase adventure

ENTER-enter adventure

Select one of the above please.
```

*Figure 1*. *Graphic Adventure Creator* main menu.

Tools that programmers used to create text adventure games in these domain-specific languages would typically be in-house and proprietary. In other cases, these tools could themselves become the product, and it is to one of those that we turn. *Graphic Adventure Creator* (GAC) was a 1985 program[2] that, according to Montfort (2003), "proved to be a capable system for creating text-and-graphics interactive fiction" (p. 196). GAC was ported to multiple computing platforms; we use the ZX Spectrum version of it in our work. The ZX Spectrum was "the UK's home computer of choice [...] For a brief moment it was thought to be the world's best-selling computer" (Donovan, 2010, p. 116). Aspiring text adventure creators could use GAC on their home computers to develop their games, by defining vocabulary (nouns, verbs, and adverbs), rooms, objects, and (optionally) drawing images. The finer points of the game would be expressed using GAC's text-based domain-specific language. GAC's menu-driven main interface (*Figure 1*), and its line-based code editing facilities, were not *un*usual for the time period, but were still somewhat spartan: for example, code editing literally only showed a line of code at a time, without context.

Wikipedia states that "Over 117 titles were written using GAC" (Wikipedia, n.d.). The true number is somewhat higher. We have manually aggregated a digital collection of all the GAC-created games for the ZX Spectrum that we can locate, and our corpus consists of 130 games, with 152 game images in total. Some explanation about that distinction is in order: cassettes were the usual storage medium for the ZX Spectrum, and their slow speed and lack of random access meant that these GAC games could not load additional content on demand – they were all "single-load" games that were loaded into the computer's memory in their entirety. Longer games that would not fit into the computer's memory would need to be broken up into parts. One way that this was managed, to force the parts to be played in order, was to have the end of part *n* reveal a password needed to play part *n+1* (Incentive Software,

[2] Dating games accurately is difficult for a number of reasons. Here, the copyright date on the software is 1985, and the port to the ZX Spectrum we use is copyright 1986, the year that Montfort cites.

1986a).[3] Hence we can have more game images than actual games, although from the exploration point of view, each part can effectively be thought of as a separate game. The game *images* refer to memory snapshot images. We manually loaded each of the games' (one or more) parts into the Fuse emulator (Fuse, n.d.), and captured the computer memory's complete contents when it was loaded; each memory image is 64 KiB in size. The games are mostly in English, although our corpus also includes some in other languages: Polish, Portuguese, and Spanish.

## *GrACIAS and Game Analysis*

Having described the collection we have to analyze and explore, we can now discuss the software we have written to perform those tasks. GrACIAS, the *Gr*aphic *A*dventure *C*reator *I*nternal *A*nalysis *S*ystem, has been under development since mid-2015, and currently is comprised of approximately 4000 lines of code, mostly Python with some C used for efficiency in the searching described in the next section.

The first part of GrACIAS is a GAC game interpreter engine written in Python. Where did the information come from? There are two GAC manuals (Incentive Software, 1986a, 1986b) that describe the GAC language to a reasonable degree, although they omit a number of fine points. We initially needed to augment the manuals' information with a number of experiments run in the real GAC to determine its behavior, essentially treating it as a black box. This part has been continually refined over the course of this work as we discover new cases where our interpreter and the real GAC differ. While we are able to do partial reconstructions of the graphic images in GAC games with our interpreter, that has not been our focus. In keeping with the idea of being able to explore the code, we have primarily worked on that aspect in GrACIAS instead, and our GAC language implementation is complete.

The format for a game that our GAC interpreter uses is itself a Python module containing data structures with the game information. While other formats are possible that are more programming-language agnostic, like XML, the game-as-module was straightforward to integrate with the Python-based GAC interpreter; more game formats could be supported if required. A game module contains the following information:

- Nouns, verbs, and adverbs, each a mapping from strings to integers. Multiple vocabulary words may correspond to the same numeric value, allowing aliases like `S` and `SOUTH`. In GAC code, vocabulary words are referenced by number only; `VERB 5` is a Boolean predicate that is true if the user's command used verb number 5, for instance.
- Messages, text that is output from GAC code, which are a mapping of integers to strings. Again, messages are identified by number, and the GAC code `MESS 42` would output message number 42.
- Room information. A room is simply a game location, and need not be a literal room. Associated with each room is a room number, descriptive text, connectivity information to other rooms, a picture number, and any room-specific GAC code.
- "High priority" GAC code executed before a user command is entered.
- "Low priority" GAC code executed after a user command is entered.
- Objects in the game, consisting of an object number, a description, an initial location, and a

[3] The GAC game *Karyssia: Queen of Diamonds* also passed state using a variant of this scheme. A player got one password for part 2 if they were carrying an amulet, and a different password if they weren't.

weight.

- Pictures, a sequence of graphical operations to render each image.
- The game's starting room number.

Obviously, the content in the game modules must come from somewhere. We manually reverse-engineered the format of the GAC game data from a GAC memory image, using a combination of static analysis – studying the GAC assembly code and data when GAC wasn't running – and dynamic analysis, which was performed when GAC was running in-emulator. Once we knew the format, we created the second part of GrACIAS, automating the conversion from a GAC memory image in the collection to a GAC module for GrACIAS. In addition to conversion, this program does some preliminary analyses and may output up to nine types of diagnostic warning. Some are code-related issues, like unreachable code; others are related to GAC's overtolerant nature, such as multiply-defined vocabulary words; still others may indicate errors in the original tape media, and we have remedied some problems by recapturing the memory snapshot image using a different tape image.

Since we are considering GAC code exploration, it is worth noting that there are two views on GAC code within GAC itself. There is the code that the GAC user would have programmed in and been presented with in GAC's programming interface, as described in the manuals (Incentive Software, 1986a; 1986b), and this is the representation GrACIAS uses. There is also an internal representation that GAC uses, a stack-based language similar to the Forth programming language (Brodie, 1987), that helped inspire the internal representation's design (Aycock, 2016a). For example, the GAC code from *King's Ransom*[4]

```
IF ( NO1 = 0 AND VERB 7 ) MESS 19 WAIT END
```

is represented internally as

```
NO1 0 = 7 VERB AND IF 19 MESS WAIT END
```

The auto-conversion process saves this raw internal code representation in the game module for verification and possible future use, but does not make use of it otherwise.

[4] This game was bundled with GAC as an example.

```
...
STARTROOM = 7
NOUNS = {
        'BAR':  4,
        'DOOR': 6,
        'GOLD': 4,
        'IT':   255,
        'KEY':  3,
        'LAKE': 8,
        'LAMP': 1,
        'RAT':  2,
        ...
}
ADVERBS = {
        'A':    2,
        'THE':  1,
}
VERBS = {
        'BASH': 32,
        'D':    6,
        'DOWN': 6,
        'DROP': 8,
        'E':    3,
        'EAST': 3,
        'EAT':  21,
        ...
}
MESSAGES = {
...
1: 'It is old, but still looks servicable.',
2: 'It must be freshly dead, as it is still warm.',
3: 'It is a large, rusty old key.',
4: 'it must be worth a fortune, judging by the size of it.',
5: 'It is an old lamp, shining brightly.',
7: "You can't walk through doors, you know!",
8: 'Its slumber disturbed,the snake rears its foul head and
    bites you!The poison kills you quickly and painlessly.',
9: "Well done! The gold you have brought is enough to pay a
    king's ransom!",
...
}
ROOMS = {
1: [
'You find yourself on the bank of a turbulent stream, babbling
 along the base of the mountain itself. It is crossed by a
 stone bridge leading East to a dark cave entrance.' ,
 ...
],
...
}
HIGHPRI = '''
IF ( AT 7 ) LF MESS 21 HOLD 25000 LF GOTO 1 WAIT END
IF ( RES? 6 ) SET 6 STRE 111 3 CSET 1 END
IF ( RES? 1 AND RES? 2 ) DECR 1 END
...
'''
LOWPRI = '''
IF ( VERB 9 ) LOOK WAIT END
IF ( VERB 10 ) MESS 239 LIST WITH WAIT END
IF ( VERB 11 ) QUIT OKAY END
...
'''
OBJECTS = {
        1: ( 'an old oil lamp', 1, 10  ),
        2: ( 'a dead rat',      5, 20  ),
        3: ( 'a key',           0, 1   ),
        4: ( 'a gold bar',      6, 100 ),
        5: ( 'a lit lamp',      0, 10  ),
}
...
```

*Figure 2*. Auto-converted game excerpt.

*Figure 2* shows an excerpt from the auto-converted *King's Ransom* game. The auto-conversion output effectively provides an X-ray of the game's content: its vocabulary, text presented to the player, and code. This is available directly, and can be seen and analyzed without having to play through the game at all, already addressing some of our goals for this system. However, we can do better.

```
import analyze

class Analyzer:
        def start(self, name):
                print name, len(self.OBJECTS)
analyze.run(Analyzer())
```

*Figure 3*. Simple static analysis example.

For more complicated analyses, and to allow analysis to scale easily to the size of the whole collection, we wrote an analysis framework for GrACIAS. Analysis-specific code uses this framework to iterate through one or more game modules, processing them one at a time. The framework manages all the game module loading, making the game data easily accessible to the analysis-specific code. For example, *Figure 3* is all the code needed to print out the game's name and the number of objects in each game. The `analyze.run()` function iterates through all game modules specified on the command line, calling the analyzer's `start()` method for each. This code can be run with a single command over all game images in the collection.

```
import analyze

class Analyzer:
    def c_if(self, conditions, statements):
        if 'if' in statements:
            print 'nested if in', self.name

    def start(self, name):
        self.name = name
        analyze.conditions(self.HIGHPRI)
        analyze.conditions(self.LOWPRI)
        for r in self.ROOMS:
            analyze.conditions(self.ROOMS[r][3])

analyze.run(Analyzer())
```

*Figure 4*. Static analysis example with code traversal

The analysis framework also includes a code-traversal engine that, in compiler terms, walks the GAC code's abstract syntax tree. *Figure 4* shows a simplified version of one of our analyses, looking for nested uses of GAC's `IF` statement – the ability to nest it was not mentioned in the GAC manuals, but was clearly known to some GAC game developers. The code-traversal engine (analyze.conditions()) must be invoked multiple times, due to GAC's code being split into high- and low-priority portions, as well as each room potentially having associated code. The Analyzer class need only define c_ methods for GAC code elements required for a particular analysis; the traversal engine finds and invokes those methods via reflection when it encounters the corresponding elements traversing the GAC code.

```
import analyze

class Analyzer:
    def c_mess(self, arg):
        if len(arg) != 1 or not arg[0].isdigit():
            print self.name,'has MESS',' '.join(arg)

    def start(self, name):
        self.name = name
        analyze.conditions(self.HIGHPRI)
        analyze.conditions(self.LOWPRI)
        for r in self.ROOMS:
            analyze.conditions(self.ROOMS[r][3])

analyze.run(Analyzer())
```

*Figure 5*. Static analysis example with code traversal; boilerplate code is highlighted in gray.

As another example, GAC code that identifies messages to print using expressions, as opposed to numeric constants, can potentially cause problems for the automatic solution-finding we describe in the next section. *Figure 5* is the analysis-specific code to identify those MESS statements in the GAC code, and output what game they are in along with the offending MESS argument, like so:

```
spectrum/Aureon/image.py has MESS no1
spectrum/Beneath Folly/image.py has MESS no1
spectrum/Black Knight, The/image1.py has MESS ctr 10
spectrum/Black Knight, The/image1.py has MESS no1
```

Here we are interested only in the MESS statement, and therefore only define c_mess in our analysis code. The second parameter arg is a list of GAC code separated into tokens, as were the conditions and statements parameters to c_if in the previous example; these lists can then be examined for features of interest. It is apparent, comparing *Figures 4* and *5*, that much of the code structure is boilerplate for these analysis tasks.

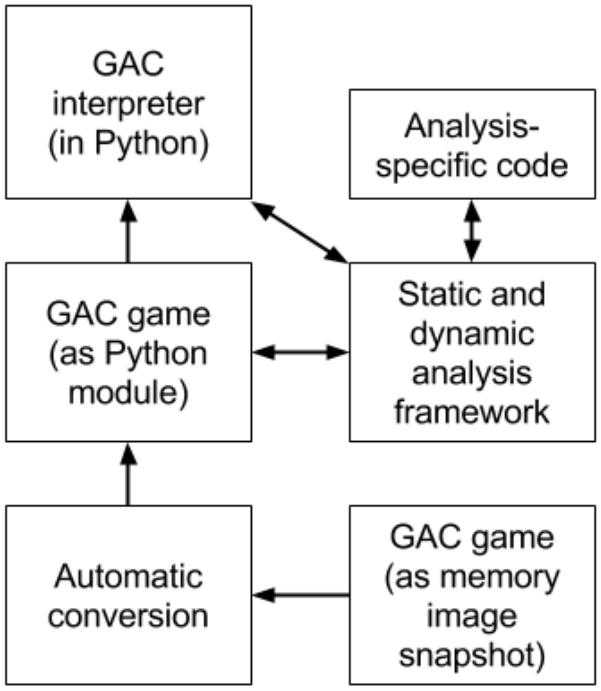

GAC interpreter (in Python)
Analysis-specific code
GAC game (as Python module)
Static and dynamic analysis framework
Automatic conversion
GAC game (as memory image snapshot)

*Figure 6*. GrACIAS analysis system overview.

All these analyses so far have been static, where the GAC code and other game module data have been examined without the game running. Some analyses can be made more precise by running code, though. As an example, a GAC manual mentions that a particular Boolean variable can be used to ascertain if it is the first move in the game, which can then be used for game state initialization (Incentive Software, 1986a). Did GAC game authors actually do this? We started with a static analysis to answer this question, but it quickly became apparent that static analysis could not capture all the ways that initialization happened, because one `IF` statement in GAC code could change the game state so that later `IF` statements were also executed. Thus we extended GrACIAS to perform dynamic analysis: our GAC game interpreter can be used in a "sandbox" mode that allows analyses to execute GAC code. The sandbox mode disables the subset of GAC instructions that have externally visible side effects, allowing it to be safely run non-interactively over arbitrary GAC code. An overview of GrACIAS' full game analysis framework is shown in *Figure 6*.

Given the amount of time GrACIAS has been under development, we have had experience adding new analysis-specific code long after the framework was no longer fresh in our mind. We can anecdotally report that GrACIAS' analysis framework can be effectively used in a cookbook fashion, basing code for new analyses on existing ones. While writing Python code may not be the optimal interface for some types of user, GrACIAS serves as a positive proof of concept that game code within a collection can be explored and analyzed both statically and dynamically at scale, without overly onerous demands on the audience. It would certainly be possible to extend GrACIAS with a graphical interface that would be more user-friendly.

## ***GrACIAS and Automatic Solution-Finding***

Once GrACIAS was able to place GAC games into an accessible and explorable format, we began to wonder what else could be done with them. One interesting application is to see whether or not the games can be automatically solved by the computer. While game walkthroughs do exist, they may not exist for all games in a collection; regardless, automated solution-finding speaks to a more general issue, namely new ways that we can use the computer to facilitate research on, and accessibility of, collections.

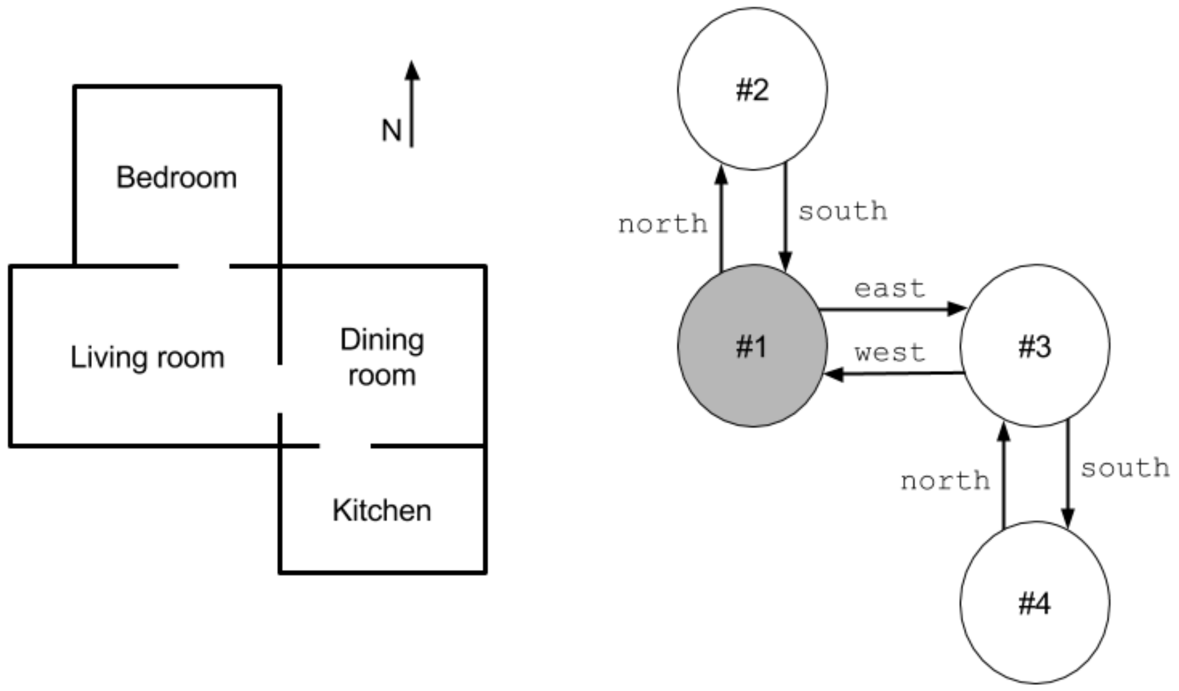


*Figure 7*. Floor plan (left) and the corresponding finite state machine (right).

A text adventure game can be viewed as a Finite State Machine (FSM). While an FSM has a formal mathematical definition, we can explain it informally with respect to text adventure games using the floor plan of a house in *Figure 7*. The house has four rooms, and say that the player starts in the living room. We can represent this house using the FSM shown on the right of *Figure 7*. Each room is abstractly represented as a "state" in the state machine, and possible movements from room to room are represented as arrows between states; an arrow is labeled with the input command the player would give to make that movement. One state in the FSM is distinguished as the start state (here, shown with a gray background). In an FSM, knowing what state the player is currently in encapsulates all necessary information: location and allowable commands. For example, a player starting the game in the living room (state #1) could only issue the `north` command to move to the bedroom (state #2) or the `east` command to enter the dining room (state #3).

For our purposes involving GAC, a state in the FSM incorporates not only the location of the player, but the entire game state, including the locations of objects and the values of all GAC code variables (GAC provides 128 8-bit 'counters' and 256 Boolean 'markers'). As in the initial FSM example, transitions between states in the FSM occur based on input commands from the player. While the resulting state space is large, these games are thirty years old, and one might reasonably expect that the state spaces could be fully mapped and explored with modern computing resources.[5]

To understand when a solution search has succeeded, we need to know what constitutes the goal state for each game. Invariably the text adventure games in the collection print a congratulatory message

[5] Spoiler alert: one would be disappointed. This turned out to be more the exception than the norm.

when the player has completed the game, and if the corresponding `MESS` statement in the GAC code can be found, reaching that statement should be equivalent to game completion (this is not always true, as we discovered. Some games in the collection print their end-of-game message as a side effect of moving to a specific room in the game, but again this is a very specific condition to identify). Happily, GrACIAS extracts all the game text for us to see, and we could find the message we want that way, such as message 9 in *Figure 2*. However, it was more expedient in most cases to search in the code for the GAC `EXIT` command at the end of the game, then see which `MESS` number was displayed immediately prior – it was likely either describing a gruesome death or epic success. This was a particularly helpful technique for games in foreign languages, because it limited the amount of game text that we had to translate. Finding the target message/room number was a manual process we did for all the game images in the collection, although with the `EXIT` search technique it was usually very fast.

Our initial approach to game solving was to leverage an existing solver, KLEE (Cadar, Dunbar, & Engler, 2008). KLEE performs symbolic execution of C code to automatically find program failures, and we modeled success as a failed code assertion: KLEE would detect the failure and tell us the input (player commands) that led to that "failure." The obvious problem was that we had games expressed using GAC code, not C code. The solution was that we wrote a compiler in Python to translate a GAC game image into equivalent C code. Uncertain of success, we used what Morgan (1998) calls a "spike approach" (p. 423), where we implemented enough of the compiler to translate one game, *King's Ransom*, which is a relatively straightforward game; if KLEE could not solve that one, then we would need to take a different tack. The translation that the compiler needed to implement was already known: it is essentially what our GrACIAS GAC code interpreter did from the start.

KLEE succeeded in finding a solution once, sort of. Due to a bug in translation to C code, we forgot to check if an object was present in the current location before picking it up. KLEE's mechanism found this and solved the game very quickly with a single command, `TAKE TREASURE`. Once that bug was fixed, KLEE made no notable headway, even when we moved the goal state earlier in the game to make it easier. Unfortunately it appeared that KLEE was too general, and that we could do a better job if we were able to manage the search process ourselves and incorporate domain-specific information to limit or reorder the search space.

```
 1  initial game setup
 2  while True:
 3      start turn
 4      run high-priority GAC code
 5      if first turn:
 6          save game state
 7      else:
 8          if game state changed:
 9              if game state unreached before:
10                  save new state
11              record input and start/end states
12          else:
13              state unchanged, ignore this path
14      readjust search queues
15      load next game state and input
16      handle room-to-room movement, if any
17      run room-specific GAC code
18      run low-priority GAC code
```

*Figure 8*. Pseudocode for solution-searching algorithm.

Since we already had a partial translation of GAC code to C code, and C code can be optimized more and run faster than Python code, we continued on that route to create a full compiler from GAC to C. Instead of KLEE's symbolic execution, this C code is effectively playing the game, with additional code to save and restore game states, and choose commands that will hopefully result in FSM state transitions into unexplored states – and eventually a game-winning one. The algorithm is shown in *Figure 8*. The structure roughly follows the execution flowchart from the GAC manual (Incentive Software, 1986b), although we had realized when implementing the GAC interpreter previously that the flowchart was not complete, and had needed to perform a number of experiments with the real GAC to understand how it really operated.

Let us delve into some finer points of the algorithm. At Line 3, the start of a turn increments an internal counter whose value is exposed to the GAC code via two of the 'counter' variables. By itself, unremarkable, but we have deliberately *ex*cluded this when saving and comparing game states. Otherwise, two states with no meaningful differences would appear distinct because of the change in those counter variables, and cause the search space to balloon as a result.

The execution of GAC code at Lines 4, 17, and 18, as well as the room-to-room movement in Line 16, is all implemented as game-specific inline C code generated by our compiler. This allows us to naturally leverage any optimizations that the C compiler performs.

The management of game states on Lines 6 and 8–10, along with their representation, is of critical importance to the search process, and has a direct impact on how fast the search proceeds and how much of the search space it can cover. Each game state is 174 bytes, and each is stored along with 12 additional bytes of data: two links to thread it in with data structures, and one index used for input bookkeeping, all of which are described shortly. We allocate a multi-gigabyte memory arena for the states, whose size we can adjust when the search program is compiled; the main compute server we use for extensive searches is a Linux machine provisioned with 95 GiB of memory. Space for a new state is allocated by appending it onto the end of the allocated space in the memory arena, to preserve as much locality as possible. To locate a state (or its absence) efficiently, we use a separately allocated hash table with collision resolution by chaining. All "pointers" for this and other data structures involving states are actually kept as 32-bit unsigned integers rather than 64-bit pointers; this makes the code trickier to write and debug but saves considerable space at scale.

Line 14's readjustment of search queues implies that another data structure exists. We found early on that movement in the search space towards potential solutions seemed to be hampered by having to search a deluge of unproductive states generated near the start of the search. There is unfortunately no reliable way to generally decide if a new state is nearer to solving the game, so heuristic methods need to be used. We began by introducing a bucket queue per room, with one bucket per priority level in each room. States to search are selected in round robin from the rooms, resulting in newly-reached rooms getting as much search attention as earlier ones. The state chosen from a room is taken from that room's highest priority queue.

How are priorities assigned? We have four heuristics, one of which is selected when the search code is compiled:

- On the intuition that progression through many games is governed by in-game objects that have been found by the player, the first heuristic assigns a score based on the number of objects both carried and present in the current room. In addition, one object is designated as a "special" object with an even higher weighting, which is changed round-robin. Since changing which object is special affects the priorities throughout, this is only done at widely-spaced adjustment intervals: for the first two heuristics, this is done after every 5,000,000 game states are tried.
- The second heuristic only includes the object bias, and does not consider what objects are carried or present.
- The final two heuristics mirror the first two, but only readjust the priority queues after a certain number of *new* states are found. This is to try and avoid the situation where more time is being spent readjusting queues than making actual search progress. The queue readjustment threshold for these heuristics is set correspondingly lower, to 50,000.

We are unable to predict as yet which heuristic is best suited for an arbitrary game. Given the small number of heuristics, it is tractable to respond to a failed solution search by simply recompiling the search program to select a different heuristic and running a new search.

In Line 15 of the algorithm, the loading of the game state *and* input reflects the fact that they are conjoined. A naive approach to choosing inputs to try would be to list all possible vocabulary combinations and, because two nouns may be supplied along with one verb and one adverb, that would give many, many combinations indeed – most of which would not advance the search. Even with this naive approach, we would need to keep track of which inputs had already been tried from a particular game state, which explains the need for the input bookkeeping index mentioned earlier. We can do far better in winnowing down the set of input combinations to try, though.

For each room, our GAC to C code compiler creates a single blob of GAC code by concatenating four pieces of code together. First, the compiler generates GAC code that implements room-to-room movement, a task normally done implicitly by a GAC interpreter, not with explicit GAC code. For example, room 2 in *King's Ransom* has an eastward connection to room 3, a fact that is represented in GAC using the abbreviated form `E 3`. In this first step, the compiler would convert this in room 2 to the explicit GAC code:

```
      IF ( VERB 3 ) GOTO 3 WAIT END
```

Second, the compiler appends room-specific GAC code, if any. Third, it appends low-priority GAC code. Fourth, high-priority GAC code is appended. Compared to the normal GAC interpreter loop, this constructed code blob reflects the sequence of operations from input to the bottom of the interpreter loop and around the loop to the next input.[6] We feed this code blob to GrACIAS' static code traversal described in the last section to extract out the `IF` conditions in the GAC code. GAC code almost exclusively consists of guarded statements (i.e., `IF` statements), and our compiler analyzes those conditions to construct an approximation of the minimum vocabulary combinations that will cause the conditions to be true. For example, consider the GAC code from *King's Ransom* given earlier:

```
      IF ( NO1 = 0 AND VERB 7 ) MESS 19 WAIT END
```

This `IF` condition can only be true when the input consists of a noun (`NO1`) with value 0 and a verb with value 7. There is no point trying any other input for this code. The reduction in input combinations to try is substantial. The naive approach in *King's Ransom* would need 8100 combinations for every state, but with our compiler's analysis this drops to an average of just under 32 combinations per state.

We return now to Line 11 of the algorithm. Searching the state space and reconstructing a search solution are separate processes. They are coupled only by a log file that records pairs of state numbers and the input that caused the transition between the two, in a binary format. In fact, the state space search can be run with logging disabled, and for good reason. The log files can be massive, and writing them also slows down the search, so there is no point capturing a log unless necessary. *King's Ransom*, a comparatively small game, had its full state space explored and had 2,577,052 log records created – at 12 bytes apiece, that is just under 31 MiB. As an offline process, reconstructing and printing the search solution from the logfile is not time-critical and is implemented in Python. It requires finding the shortest path in a directed graph, and consequently we use Dijkstra's algorithm with a Fibonacci heap (Cormen, Leiserson, Rivest, & Stein, 2002). When constructing the graph, all else being equal, we prefer edge labels (i.e., player commands) that are "shorter" and have fewer words. Once the shortest path is found, the word numbers are mapped to the shortest alias for them that was used in the game. Together, this means that a solution would prefer `W` over `WEST`, `GO WEST`, or `GO W`. We found that words in games could have the occasional incorrect mapping, and a correct solution could fail when tried in the real game, which is why GrACIAS' automated conversion program (described in the previous section) now issues a warning when that is detected.

There are some limitations to our automatic solution-finding. The obvious concern is that any nondeterminism in a game can make it unsolvable with our system. Imagine, for instance, a game where critical objects move around randomly, or are initially placed randomly, or where the player can teleport randomly from place to place. Meanwhile, we deliberately keep our search results repeatable by having our compiled version of GAC's `RAND`om function always return 0. Our exclusion of the turn counter from game state can also present problems for any games that have time-based events. More subtly, we assume in our input combination analysis that `IF` statements are independent, and they usually are, but if a series of conditional statements triggered one another, our system would not handle that correctly at present.

---

[6] This is not *quite* semantically correct in that the start of a turn alters some counters, as mentioned, but we are already excluding those from the game state.

We verified all solutions found by trying them in-emulator with the real GAC and the original game images. In total our system has correctly automatically solved 62 game images so far, including non-English ones. Some additional game images had solutions reported that could not be verified – in some cases this caught some semantic differences between our implemented GAC interpreter and the real one, and in other cases we conjecture that it is due to the limitations mentioned above. Regardless, the large number of correctly discovered solutions lends credence to our assertion that this type of exploration of a game collection is possible.

### *Related Work*

We are only aware of one piece of related work within game studies. A recent paper described automatically producing maps for NES games (Osborn, Summerville, & Mateas, 2017), although unlike our approach they require a playthrough to already exist. Apart from that work, we need to look farther afield.

Given that these games are being amassed into collections and are being made available via sites like the Internet Archive, it would be reasonable to look to work within digital libraries and archives; one would expect them to be at the vanguard of work on digital game collections. The reality is quite different. Games are an understudied area within digital libraries, especially considering games' cultural importance and the size of the games industry. There is work using games for learning (Guo, Goh, Muhamad, Ong, & Lei, 2016) and crowdsourcing (Goh, Razikan, Lee, & Chua, 2012; Goh, Pe-Than, & Lee, 2016), as opposed to exploring collection of existing games. Other work focuses on issues related to acquisition of games and associated artifacts (Lee, Jett, & Perti, 2015; Winget, 2009; Winget & Sampson, 2011) and their organization and cataloging (Clarke, Lee, Jett, & Sacchi, 2014; Donovan, Cho, Magnifico, & Lee, 2013; Dubin & Jett, 2015; Lee, Cho, Fox, & Perti, 2013; Rossi, Lee, & Clarke, 2014). These are complementary efforts but very distinct from the work we are doing.

The other major category of related work deals with analysis tools for collections. For example, collections of online texts (Cartright, Feild, & Allan, 2011; Matshall, 2008), and collections of text-based archives (Abbasi & Chen, 2007). The key underlying theme here is text, of course; we are not aware of similar efforts such as ours for exploring digital game collections. There is also work on exploration as an integration of disparate digital library resources (Shen, Vemuri, Fan, & Fox, 2008), but that is a different sense of exploration than what we are doing within a single collection.

More broadly, games are a frequent benchmark for AI-related research. For example, there is work employing machine learning to play classic Atari 2600 games (Kaplan, Sauer, & Sosa, 2017; Mnih et al., 2015), and their techniques may one day be adapted to collection exploration as well.

## Conclusion

The initial question we asked was how to make a transition from building game collections to doing something with them. The obvious answer is to make the games playable in some fashion, but this provides a limited view, because not everything in a game may be seen through gameplay, and not everyone may be able to play a game well enough anyway. Audiences of researchers and laypeople can both be served better.

Using a collection of games created using the *Graphic Adventure Creator*, we have demonstrated using our GrACIAS system some ways that a game collection can be analyzed and explored. Moreover, our system exposed the internals of all the games and permitted the game to be "read" not just at the level of gameplay, but at the implementation level. Our system went beyond that to automatically find solutions to 62 of the game images in the collection, permitting the games to be played and experienced even by non-experts, and even when the player does not understand the language the game uses. While our proof of concept with GrACIAS illustrates some of the ways the exploration of a game collection can be facilitated, there are doubtless more ways that this can manifest itself, and our work can be extended both in that direction and to games from other time periods and genres.

## Acknowledgments

Thanks to Stefan Vogt for first bringing GAC to the author's attention. The author's research is supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada. The image in Figure 1 is used under fair use and fair dealing guidelines for identification and commentary.

## References

Abbasi, A., & Chen, H. (2007). Categorization and analysis of text in computer mediated communication archives using visualization. In *Proceedings of the 7th ACM/IEEE-CS Joint Conference on Digital Libraries* (pp. 11-18). ACM, New York, NY, USA.

Altice, N. (2015). *I AM ERROR: The Nintendo family computer/entertainment system platform*. Cambridge, MA: MIT Press.

Aycock, J. (2014). *Strung out: Printable strings in Atari 2600 games* (Technical Report 2014-1062-13). Calgary, Canada: University of Calgary, Department of Computer Science. Retrieved from http://hdl.handle.net/1880/50203

Aycock, J. (2016a). *Interview with Sean Ellis Re: Graphic adventure creator* (Technical Report 2016-1086-05). Calgary, Canada: University of Calgary, Department of Computer Science. Retrieved from http://hdl.handle.net/1880/51523

Aycock, J. (2016b). *Retrogame archeology: Exploring old computer games*. New York, NY: Springer.

Brodie, L. (1987). *Starting forth* (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall.

Byte-Back column. (1984, July). Jet Set Willy solved! *Personal Computer Games*, 21.

Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (pp. 209-224). USENIX Association, Berkeley, CA, USA.

Cartright, M.-A, Feild, H.A., & Allan, J. (2011). Evidence finding using a collection of books. In *Proceedings of the 4th ACM Workshop on Online Books, Complementary Social Media and Crowdsourcing* (pp. 11-18). ACM, New York, NY, USA.

Clarke, R.I., Lee, J.H., Jett, J., & Sacchi, S. (2014). Exploring relationships among video games. In *Proceedings of the 14th ACM/IEEE-CS Joint Conference on Digital Libraries* (pp. 481-482). IEEE Press, Piscataway, NJ, USA.

Consalvo, M. (2007). *Cheating: Gaining advantage in videogames*. Cambridge, MA: MIT Press.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2002). *Introduction to algorithms* (2nd ed.). Cambridge, MA:MIT Press/McGraw-Hill.

Donovan, A., Cho, H., Magnifico, C., & Lee, J.H. (2013). Pretty as a pixel: Issues and challenges in developing a controlled vocabulary for video game visual styles. In *Proceedings of the 13th ACM/IEEE-CS Joint Conference on Digital Libraries* (pp. 413-414). ACM, New York, NY, USA.

Donovan, T. (2010). *Replay: The history of video games*. East Sussex, UK: Yellow Ant.

Dubin, D., & Jett, J. (2015). An ontological framework for describing games. In *Proceedings of the 15th ACM/IEEE-CS Joint Conference on Digital Libraries* (pp. 165-168). ACM, New York, NY, USA.

Fernández-Vara, C. (2015). *Introduction to game analysis*. New York, NY: Routledge.

Fuller, M. (Ed.). 2008. *Software studies: A lexicon*. Cambridge, MA: MIT Press.

Fuse. (n.d.). Fuse – the Free Unix Spectrum Emulator. Retrieved 5 June 2017 from http://fuse-emulator.sourceforge.net/

Gartenberg, C. (2016, November 7). Nintendo makes its NES emulator the same way everyone else does. *The Verge*. Retrieved from http://www.theverge.com/circuitbreaker/2016/11/7/13557134/nes-classic-nintendo-linux-emulator-rom-raspberry-pi

Goh, D.H., Razikin, K., Lee, C.S., & Chua, A. (2012). Investigating user perceptions of engagement and information quality in mobile human computation games. In *Proceedings of the 12th ACM/IEEE-CS Joint Conference on Digital Libraries* (pp. 391-392). ACM, New York, NY, USA.

Goh, D. H.-L., Pe-Than, E.P.P., & Lee, C.S. (2016). Games for crowdsourcing mobile content: An analysis of contribution patterns. In *Proceedings of the 16th ACM/IEEE-CS on Joint Conference on Digital Libraries* (pp. 249-250). ACM, New York, NY, USA.

Guo, Y.R., Goh, D.H.-L., Muhamad, H.B.H., Ong, B.K., & Lei, Z. (2016). Experimental evaluation of affective embodied agents in an information literacy game. In *Proceedings of the 16th ACM/IEEE-CS on Joint Conference on Digital Libraries* (pp. 119-128). ACM, New York, NY, USA.

Incentive Software Ltd. (1986a). *The GAC Adventure Writers Handbook* [Software manual].

Incentive Software Ltd. (1986b). *The Graphic Adventure Creator (Commodore 64)* [Software manual].

Internet Archive. (n.d.-a). Console Living Room. Retrieved 5 June 2017 from https://archive.org/details/consolelivingroom

Internet Archive. (n.d.-b). Internet Arcade. Retrieved 5 June 2017 from https://archive.org/details/internetarcade

Internet Archive. (n.d.-c). Software Library: MS-DOS Games. Retrieved 5 June 2017 from https://archive.org/details/softwarelibrary_msdos_games

Jerz, D.G. (2007). Somewhere nearby is Colossal Cave: Examining Will Crowther's original "adventure" in code and in Kentucky. *Digital Humanities Quarterly*, *1*(2). Retrieved from http://www.digitalhumanities.org/dhq/vol/1/2/000009/000009.html

Jones, S.E., & Thiruvathukal, G.K. (2012). *Codename revolution: The Nintendo Wii platform*. Cambridge, MA: MIT Press.

Kaplan, R., Saurer, C., & Sosa, A. (2017). Beating Atari with natural language guided reinforcement learning. Retrieved from https://arxiv.org/abs/1704.05539

Lee, J.H., Cho, H., Fox, V., & Perti, A. (2013). User-centered approach in creating a metadata schema for video games and interactive media. In *Proceedings of the 13th ACM/IEEE-CS Joint Conference on Digital Libraries* (pp. 229-238). ACM, New York, NY, USA.

Lee, J.H., Jett, J., & Perti, A. (2015). The problem of "additional content" in video games. In *Proceedings of the 15th ACM/IEEE-CS Joint Conference on Digital Libraries* (pp. 237-240). ACM, New York, NY, USA.

Marshall, C.C. (2008). Collection-level analysis tools for books online. In *Proceedings of the 2008 ACM Workshop on Research Advances in Large Digital Book Repositories* (pp. 41-44). ACM, New York, NY, USA.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., … Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*, 529–533.

Montfort, N. (2003). *Twisty little passages: An approach to interactive fiction*. Cambridge, MA: MIT Press.

Montfort, N., Baudoin, P., Bell, J., Bogost, I., Douglass, J., Marino, M.C., Mateas, M., Reas, C., Sample, M., & Vawter, N. (2013). *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*. Cambridge, MA: MIT Press.

Montfort, N., & Bogost, I. (2009). *Racing the beam: The Atari video computer system*. Cambridge, MA: MIT Press.

Morgan, R. (1998). *Building an optimizing compiler*. Boston, MA: Digital Press.

Osborn, J., Summerville, A., & Mateas, M. (2017). Automatic mapping of NES games with Mappy. In *Proceedings of the 12th International Conference on the Foundations of Digital Games* (Article 78). ACM, New York, NY, USA.

Reinhard, A. (forthcoming). *Archaeogaming: An introduction to archaeology in and of video games*. New York, NY: Berghahn Books.

Rossi, S., Lee, J.H., & Clarke, R.I. (2014). Mood metadata for video games and interactive media. In *Proceedings of the 14th ACM/IEEE-CS Joint Conference on Digital Libraries* (pp. 475-476). IEEE Press, Piscataway, NJ, USA.

Shen, R., Vemuri, N.S., Fan, W., & Fox, E.A. (2008). Integration of complex archeology digital libraries: An ETANA-DL experience. *Information Systems*, *33*(7–8), 699–723.

Sinclair, S., & Rockwell, G. (2016). Voyant tools. Retrieved 5 June 2017 from http://voyant-tools.org/

The cutting room floor. (n.d.). Retrieved 5 June 2017 from https://tcrf.net/The_Cutting_Room_Floor

Walsh, B., Maiers, C., Nally, G., & Boggs, J. (2014). Crowdsourcing individual interpretations: Between microtasking and macrotasking. *Literary and Linguistic Computing*, *29*(3), 379–386.

Wikipedia. (n.d.). Graphic adventure creator. Retrieved 5 June 2017 from https://en.wikipedia.org/w/index.php?title=Graphic_Adventure_Creator&oldid=719391655

Winget, M.A. (2009). Archiving the videogame industry: Collecting primary materials of new media artifacts. In *Proceedings of the 9th ACM/IEEE-CS Joint Conference on Digital Libraries* (pp. 459-460). ACM, New York, NY, USA.

Winget, M.A., & Sampson, W.W. (2011). Game development documentation and institutional collection development policy. In *Proceedings of the 11th Annual International ACM/IEEE Joint Conference on Digital Libraries* (pp. 29-38). ACM, New York, NY, USA.