Towards Automated Game Design

Mark J. Nelson[1] and Michael Mateas[2]

[1] College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA
mnelson@cc.gatech.edu
[2] Computer Science Department
University of California—Santa Cruz
Santa Cruz, California, USA
michaelm@cs.ucsc.edu

Abstract. Game generation systems perform automated, intelligent design of games (*i.e.* videogames, boardgames), reasoning about both the abstract rule system of the game and the visual realization of these rules. Although, as an instance of the problem of creative design, game generation shares some common research themes with other creative AI systems such as story and art generators, game generation extends such work by having to reason about *dynamic, playable* artifacts. Like AI work on creativity in other domains, work on game generation sheds light on the human game design process, offering opportunities to make explicit the tacit knowledge involved in game design and test game design theories. Finally, game generation enables new game genres which are radically customized to specific players or situations; notable examples are cell phone games customized for particular users and newsgames providing commentary on current events. We describe an approach to formalizing game mechanics and generating games using those mechanics, using WordNet and ConceptNet to assist in performing common-sense reasoning about game verbs and nouns. Finally, we demonstrate and describe in detail a prototype that designs micro-games in the style of Nintendo's *WarioWare* series.

1 Introduction

Game generation systems perform automated, intelligent design of games, reasoning about both the abstract rule system of the game and the visual realization of these rules. While procedural content generation focuses on the generation of assets such as textures, meshes, animations, sounds, and the physical layout of levels, game generation involves the entire game design process, including generating the game rules themselves, the game mechanics that describe how the game state evolves over time, how player action influences the game state, and the audio-visual realization of the game.

The goal of our research is not to replace human designers, but rather to: facilitate formal game analysis through the computational expression of game

rules, mechanics, and representations; enable new game mechanics and game genres where the game dynamically changes as a function of player interaction; move human design up the abstraction hierarchy to the meta-authorship of generative processes that generate games; and enable intelligent game design tools to support human game designers.

Game designers and scholars have discussed the need for a game design language, noting that there is no unified vocabulary for describing existing games and thinking through the design of new ones. While some semi-formal analysis languages are being developed [1], game design has not been described at the level of detail and formality necessary to support automatic generation. Automatic game generators can serve as highly detailed theories of both game structure and game design expressed operationally as a program. In the same way that AI-based story generators have, over the years, served as operational models of both narrative and the story generation process, and thus served to expose the strengths and weaknesses of different models of narrative, so too can game generation facilitate the development of a design science for games.

In addition to shedding light on the game design process, dynamic generation enables new game mechanics and game genres where the game dynamically changes (or is generated from scratch) as a function of player input or other exogenous events. Newsgames are one such category of game—micro-games that provide commentary on a news item, much like political cartoons. Unlike political cartoons, however, newsgames provide their commentary through gameplay: the point is made through interaction on the part of the player. Some well-known newsgames include *Madrid*,[3] a memorial game released shortly after the Madrid train bombings on March 11, 2004, and *Bacteria Salad*,[4] a game about the fall 2006 *E. coli* infections spread by spinach in the United States. To offer timely commentary on a news item, newsgames must be created rapidly, motivating the need for automatic (or at least AI-assisted) game design. Newsgames tend to be relatively small micro-games, making automated generation tractable, even in the short term.

In the rest of the paper, we describe our view of game design as a problem-solving activity comprising four major aspects of games, and describe a prototype system that generates games in the style of Nintendo's *WarioWare* games—short games that typically last several seconds, come in rapid sequence, and ask the player to do a single thing, such as dodging a car or shooting a duck.

2 Game Design as a Problem-Solving Activity

To understand game design as a problem-solving activity, we factor it into four interacting domains (or aspects): abstract game mechanics, concrete game representation, thematic content, and control mapping. A game design space (the space of possible games the design system can reason about) is defined by the

[3] http://www.newsgaming.com/games/madrid/
[4] http://www.persuasivegames.com/games/game.aspx?game=arcadewireecoli

knowledge given to the system for each of these domains. Though the domains interact, we hope to support the modular mixing and matching of different knowledge sources for each domain, thus supporting the rapid specification of new design spaces.

We'll use the 1983 arcade game *Tapper* as an example. *Tapper* is a nice example because is it well known within the game-design community, and small enough to allow formal analysis yet large enough to clearly exhibit all four knowledge domains. In *Tapper*, the player is a bartender who fills up mugs of beer and serves customers by sliding the beers down one of several bars. The customers move along the bars towards the bartender; serving a customer pushes him back towards the door. The goal is to push the customers out of the bar without letting any reach the bartender. Effective *Tapper* play is an exercise in rapid time management.

A game's *abstract game mechanics* specify an abstract game state and how this state evolves over time, both autonomously and in response to player interaction. In *Tapper*, the abstract mechanics are those of an order-fulfillment game: There are requesters (customers) who want certain items (beers) within time limits, sources for the player to get those items (taps), and a means for the player to ferry the items from the sources to the requesters (sliding them down the bar). The space of order-fulfillment mechanics is defined by general knowledge about requesters, sources, requested objects, the progression of time, and the relationships between each; the mechanics in *Tapper* are one instance, making commitments to specific design decisions in this space. Chess likewise makes concrete design commitments within the space of symmetric, 2D, tile-based games [2]. In *WarioWare* games, the abstract mechanics are usually a single rule, such as "avoid being hit for five seconds".

Concrete game representation specifies how the abstract mechanics are instantiated and represented to the player in a concrete game world, that is, the audio-visual representation of the abstract game state. In *Tapper*, the abstract time limit within which an order request must be serviced is represented concretely by the customer's position along the bar; in other games, an abstract time limit might be represented by a literal on-screen clock, through a slowly emptying bar graph, etc. In *WarioWare*, one concrete representation of the "avoid being hit for five seconds" abstract mechanic is a dodging game in which the player has to move around in a 2d top-down view and avoid getting hit. General knowledge about representational strategies for different types of abstract game states (and state transitions) constitutes a visual design space. Holding the abstract mechanics domain constant while changing the game representation domain results in a new overall design space, such as 3D, first-person order-fulfillment games (*Tapper* is a 2D, third-person game).

Thematic content comprises the real-world references expressed by the game. For example, *Tapper* takes place in a bar, with beer glasses, customers, and so on; *Diablo* takes place in a fantasy world with swords and monsters; *The Sims* takes place in a suburban house; and a *WarioWare* dodging game might have a person on a road dodging cars. The thematic knowledge domain comprises the

common-sense knowledge about the real-world domain being expressed in the game. Holding the other domains constant while changing the thematic content domain results in a new overall design space, such as the design space of 2D order-fulfillment games set in fast-food restaurants.

Finally, *control mapping* describes the relationships between the physical player input, such as button presses and joystick movement, and modification of abstract game state. In *Tapper*, pressing a button at the tap begins filling a mug, while releasing the button stops filling and, if the mug is full, automatically slides it down the bar. Possible alternative mappings for filling the beer include repeatedly pumping the joystick back and forth, repeatedly hitting a button, holding the joystick down for a specified period of time, etc. (to say nothing of alternate physical control mechanisms such as dancepads or gestural controllers).

It might be tempting to see these four game-design aspects as a pipelined process: Come up with an abstract game, represent it concretely, add a "skin" of thematic content, and finally set up control mappings. That approach may work for some types of games, but we feel that enabling non-pipelined interactions between these aspects is likely to lead to more interesting and creative game designs. Even something as seemingly straightforward as setting up the control mappings is not a one-way street; for example, a game written for a computer with keyboard and mouse might call for different game mechanics and representations than one written for the Nintendo Wii, with its physical gestural controller (indeed, the idea that an interesting control scheme can lead to interesting game design is one of the core market hypotheses of the Wii).

Thematic knowledge should also ideally be more than a "skin" chosen according to the constraints of an already-designed abstract mechanic; instead, it should suggest gameplay opportunities as well. For example, thematic knowledge from a bar theme might lead a system to reason that as people drink they become drunk and that drunk people move erratically, suggesting that drunk customers might serve as obstacles for the player as they serve drinks, but only if the player is a cocktail server moving between tables. This movement from theme to mechanics and representation is particularly important for the generation of newsgames and other games dealing with real-world events, where much of the rhetorical force of the game depends on the appropriate incorporation of thematic elements into the gameplay—rather than merely skinning an off-the-shelf game with the faces of politicians or something equally superficial.

Of course, it is not a requirement that all games equally emphasize all aspects. In chess the thematic content and concrete realization are of minimal importance; design of interesting chess variants would focus on the abstract game mechanics. A first-person shooter, on the other hand, puts large emphasis on the game's concrete representation (3D graphics and physics); design of first-person shooters would focus on this aspect. Our goal is to have all these aspects available for interesting interaction when desired.

Like any problem-solving activity, game design is driven by goal achievement. These may be internal goals such as "maximize the variety in a collection of generated games" or "design an interesting game", where internal notions of

variety and interestingness guide design; or external goals to create games with specific properties for one or more aspects of the design, or specific properties relative to a player model. For example, one might ask for a symmetric chess-like game with a certain locality of movement [2] (here abstract mechanics would be the guiding aspect of the design process), or an order-fulfillment game with a desired visual complexity in terms of the number of simultaneously moving objects on screen (here concrete representation would be the guiding aspect). For our *WarioWare*-style generator, we chose to focus on theme as the guiding aspect. Design goals are specified in terms of nouns and verbs; the generator's job is to create micro-games that are "about" the verb and/or noun. Since a micro-game instantiates a single, atomic game mechanic, and maps real-world referents onto this mechanic, such games are a nice vehicle for exploring the common-sense reasoning issues that arise in the thematic aspect of game design.

3 Generating *WarioWare*-style games

Although *WarioWare*-style micro-games are quite simple structurally, usually containing one or two pieces of abstract mechanics and lasting for no more than a few seconds, they are an example of a game style that actual game designers work on and release commercially (rather than an artificial toy domain), with a design space that touches on a wide cross-section of issues in game design [3]. Their abstract simplicity allows us to focus on the thematic elements of game generation, and the fact that they are lightweight games that are easy and quick to play meshes well with a number of potential applications, such as web-distributed casual games and customized cell-phone games.

3.1 Common-sense thematic content

Since our initial focus is on the thematic aspect of game generation, we have the user direct the system by specifying a verb and/or noun to describe a desired theme (e.g. a game about "shooting" and "ducks"). Generating a game that fulfills the request is now a combination of two common-sense problems: The game should "make sense" in terms of the roles its thematic elements are playing, and it should be "reasonably close" to what the user requested.

To address both problems, we use a combination of the ConceptNet [4] and WordNet [5] knowledge bases. ConceptNet is a graph-structured common-sense knowledge base mined from OpenMind [6], a collection of semi-structured English sentences expressing common-sense facts gathered from online volunteers. ConceptNet's nodes are English words or phrases, and links between them express semantic relationships such as `(CapableOf "person" "play video game")`. Compared to more formally specified common-sense knowledge bases such as Cyc [7] and ThoughtTreasure [8], ConceptNet uses natural language and informal semantics. This is nice for ease of use and interfacing with text, but has drawbacks when it comes to ambiguity and inability to usefully respond to complex queries; nonetheless, it has been useful for a number of applications [9], and we have found it useful as well.

A second drawback of ConceptNet is that its coverage is somewhat weak: it knows that a duck can be shot, but not that a pheasant can be shot. Fortunately, combining it with data from WordNet mitigates this problem to a very large extent. WordNet is a similarly graph-structured knowledge base (ConceptNet's structure was based on WordNet's), but it positions itself as a dictionary rather than as a semantic knowledge base. Nonetheless, it contains semantic information in the form of word hierarchies, where a word below another one in the hierarchy is a specialization of the higher-up one (the higher word is a "hypernym", and the lower one a "hyponym" if a noun, or "troponym" if a verb). This information allows us to add inheritance to ConceptNet queries: If something is true of animals in general, then it is true for specific kinds of animals as well. Since ConceptNet knows that an animal can be shot, WordNet-based inheritance lets us figure out that a pheasant, as a specific kind of animal, can also be shot.

In addition to using WordNet to extend ConceptNet's coverage, we use its hypernymy relationships to determine simple attributes, such as whether a noun is animate (if it is, it will be below "animate thing" in the hierarchy). This is only possible for attributes deemed primary by WordNet—although we might say "money" is conceptually a type of "valuable thing", that isn't a sufficiently primary attribute for WordNet. Finally, we use distances between nouns and verbs in WordNet and ConceptNet, respectively, as simple measures of similarity. We use WordNet for nouns due to its fairly comprehensive taxonomy, and ConceptNet for verbs since this captures more complex notions of similarity such as "can operate on the same object". While link distance is a simple notion of similarity, it has worked well as a first approximation; in the future we plan to explore more complex measures [10].

3.2 Decomposing *WarioWare*

To facilitate generation, we abstract a number of styles of *WarioWare* games into abstract game types; currently the system knows about three of them. An *Avoid* game is one in which one entity, the "avoider", must avoid (for the duration of the game) one or more other entities, the "attackers", which may attack the avoider either directly or via other objects. The player can play either role. An *Acquire* game is one in which the player must find an object within a time limit. A *Fill* game is one in which the player must fill a meter within a time limit. These abstract game types capture the abstract mechanics the system currently knows about.

These game types can be implemented via several stock sets of concrete game mechanics, represented in a mixable J2ME (Java mobile platform) class library. A *Dodger* game is a 2d top-down game in which one object, the "dodger", tries to avoid one or more other entities, the "attackers"; it is used to implement some Avoid games and some Acquire games. A *Shooter* game is a 2d side-view game in which objects move across the screen, and can be shot by aiming crosshairs and firing; it is used to implement some Avoid games. A *Pick-Up* game is a 2d top-down game with a player and an object for them to pick up, usually through

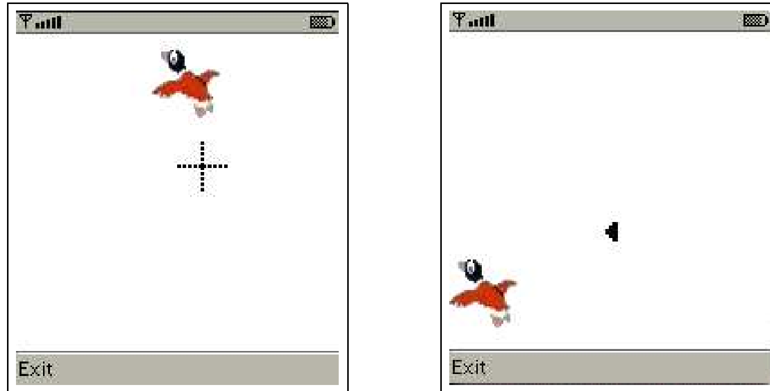

Fig. 1. Two games generated for the noun "pheasant" and verb "shoot". A duck is in both since it is the closest noun to "pheasant" for which the generator has a sprite. Both are Avoid games with Attacker-verb "shoot", Avoider-noun "duck", and Attacker-noun "bullet". The left game is implemented by the Shooter concrete mechanic and has the player trying to shoot the duck; the right game is implemented by the Dodger concrete mechanic and has the player, as the duck, trying to avoid bullets.

some obstacle such as a maze; it is used to implement some Acquire games. A *Pump* game has a reservoir of some sort that needs to be filled up; it is used to implement some Fill games. A *Move* game has a player moving some distance; it is used to implement some Fill games.

Each of the five types of game mechanics can then be matched with composable movement managers that determine how the non-player-controlled objects will move, based on some common-sense reasoning about the thematic representation they've been assigned (see next section). For example, attackers in a Dodger game where the player plays the dodging side might chase the player (if animate) or travel in a straight line (if not). The interface mappings are currently bundled with the objects in the concrete game mechanics: If the player plays the dodging side in a Dodger game, then the controls will be arrow-keys to move. These concrete game mechanics capture the concrete representations and control mappings the system currently knows about. Finally, we have a stock set of sprites, each attached to a noun describing them, that can be used as the graphical representation of any of the objects in any of the games.

The end result of the generation process is a recipe for building a game. The game is then built out of the composable classes corresponding to the five sets of stock concrete mechanics (plus input and movement-control classes) to produce a running game. Screenshots of the realizations of two generated games are shown in Figure 1.

3.3 Common-sense *WarioWare* generation

In response to a request, we generate a number of games meeting a set of constraints on what games "make sense" in each of the three types of abstract games, and score them according to a heuristic measure of how "reasonably close" they are to the player's actual request. These games are then built and fed to the player in rapid-fire sequence, much as in Nintendo's original *WarioWare*, with the games growing increasingly distant from the original request as the system has to stretch further to find games satisfying the thematic request that it hasn't yet used.

We start by defining prototype verbs that specify some canonical ways of describing the action in each game. For example, an Avoid game from the perspective of the attacker has prototype verbs "attack", "injure", "shoot", and several more. If the user requests a verb, we compare it with a set of prototype verbs via ConceptNet distance (with closer verbs making for higher-scored games), to determine if the verb can be mapped onto one of the abstract game types. The original verb, not the prototype, is then used to determine which nouns can be mapped into the game.

Filling in a game's nouns is where the meat of common-sense reasoning comes in; the process varies per game type, but can usually be done to reasonable accuracy with surprisingly simple constraints once the basic logic of the game type is teased apart. We choose nouns from those for which we have sprites that meet the constraints of the game and are close to what the player requested. The methods we use for each of the three game types are described below. In addition to selecting the abstract game type, the specifics of the semantic relationships discovered between the noun and verb are also used to select the concrete realization (*e.g.* deciding to implement *Acquire* using *Dodger* with the player playing the attacker).

Avoid The relationship of nouns and verbs in an Avoid game can follow two patterns: "Avoider-noun Avoids-verb being Attack-verbed by Attacker-noun"; or "Avoider-noun Avoids-verb an Attacker-noun Attacker-verbed by Instigator-noun". In the first case, an attacker directly attacks the avoider (*e.g.*, "person avoids being hit by car"), while in the second, an attacker is a projectile being used by some other noun (the instigator) to attack the avoider (*e.g.*, "pheasant avoids a bullet shot by gun"). Although both types of games make sense, it is important not to confuse one for the other, lest we end up with a gun shooting a pheasant by moving across the screen towards it instead of firing bullets at it. Since many of the same nouns could conceivably function in either type of game, we decide between these two phrases by testing the verb. We say that a game is of the projectile sort if the Attacker-verb is a type of verb that acts on devices; that is, whether for some noun that is a hyponym of "device", that noun is `CapableOfReceivingAction` the Attacker-verb.[5]

[5] We use "device" rather than "projectile" because some non-hyponyms of "projectile" can be used as projectiles, such as "baseball" and "hammer". There are also devices

In either version, the Avoider-noun must be `CapableOfReceivingAction` the Attacker-verb, and furthermore must be an “animate thing”. The Attacker-noun’s constraints depend on whether this is a projectile or non-projectile Avoid game. In a projectile game, the Attacker-noun is a device being acted upon by the Attacker-verb, so it must be `CapableOfReceivingAction` the Attacker-verb and must be a “device”. In a non-projectile game, the Attacker-noun is doing the attacking itself, so it must be `CapableOf` the Attacker-verb.

The player can be assigned to play either side. If assigned to play the Avoider-noun, the game is implemented by the Dodger concrete game mechanic. If the game is a projectile type game, then the Attacker-noun will be duplicated into multiple copies that move via a “move in a straight line” movement manager. If the game is a non-projectile type, then there will be a single Attacker-noun, controlled by a “chase the player” movement manager.

If the player is assigned to play the attacking side, the specifics depend on whether this is a projectile or non-projectile type game. In a projectile game, the player plays the Instigator-noun, and the game is implemented by the Shooter concrete game mechanic, with the player firing Attacker-noun projectiles. In a non-projectile game, the game is implemented by a Dodger game in which the player plays the Attacker-noun, and the Avoider-noun has a movement manager that runs away from the player.

Acquire In an Acquire game, the situation is somewhat simpler. There is an Acquirer-noun, which the player always plays, and it is trying to acquire a Target-noun. The Target-noun must be a `DesireOf` the Acquirer-noun (for example, a squirrel may want to acquire a nut). The game is usually implemented by the Pick-Up concrete game mechanic, but if the Target-noun is an “animate thing”, then it may also be implemented by the Dodger mechanic, with the player playing the attacker. In either case, a non-animate Target-noun will have a movement manager that causes it to sit still, while an animate Target-noun will run away from the player.

Fill In a Fill game, the player is a Filler-noun trying to fill an abstract reservoir, which can be represented as a literal reservoir via the Pump concrete mechanic, or metaphorically by distance across the screen with the Move mechanic; there are prototype verbs for both. Once we’ve chosen a verb, we require that the Filler-noun be `CapableOf` the verb. If the verb is a troponym of “move”, we generate a Move game; otherwise, we generate a Pump game with a Thing-To-Fill-noun that is `CapableOfReceivingAction` the verb.

---

that would be awkward to use as projectiles, but with the semantic information in ConceptNet and WordNet they are difficult to avoid, so for now we admit some—hopefully amusing—games with unlikely projectiles.

4 Related Work

The earliest automatic game-generation system we're aware of is the component of METAGAMER [2] that generates "symmetric chess-like" games. METAGAMER itself is a general game player for such games: It takes a formal description of a particular game written in a grammar that can represent a class of games, and tries to figure out how to play it well. Given a class of games specified by a grammar, new games can be generated at random by following production rules in the grammar. To generate games that would test specific aspects of METAGAMER, Pell parameterized the generator along four axes so it could, via some heuristics, generate games with greater or lesser rule complexity, decision complexity, search complexity, and movement locality. Although the primary purpose of the generator was to provide games with which to test METAGAMER, Pell notes that it nonetheless generated some interesting games. From of our view of game generation, METAGAMER tackles the portion of game design involving the abstract gameplay mechanics.

EGGG [11], on the other hand, takes a formally specified game as its *input*, and from that generates a graphical, playable game. The formal specification gives the abstract mechanics for a specific game from one of several classes of games (chess-like games, card games, etc.), and also partly specifies the concrete game representation (*e.g.* whether the abstract rules should be represented as operating on cards or board pieces). EGGG itself then tackles the process of fleshing out the on-screen representation, assigning input mappings, and "compiling" this all into a final product. For two-player games it also generates an AI opponent, which could be seen as generating abstract game mechanics to flesh out the one-human-player version of the game.

Another body of work, mostly in the game industry and graphics field, aims to procedurally generate graphics, animations, terrain, and levels [12–15]. Though only a subset of the game design problem, procedural content generation would be a useful component of an automated game-design system, allowing it to generate custom content on demand rather than relying on libraries of canned content.

Holm, Jukka, & Arrasvuori [16] propose games that customize themselves to music, for example by synchronizing movement in the game to the music's beat. This is similar to our goal of customizing games, but it is not game generation *per se*, since the games are programmed explicitly to respond to music as input, rather than adapted on the fly by a process that explicitly reasons about the game's design.

Insofar as we're exploring an expressive domain at least partly in order to better understand the potential for computer creativity within it, there are similarities with work in automated story generation [17, 18], art generation [19], music composition [20, 21], and film generation [22]. The main difference is that in story generation the product is an interactive artifact, rather than text, art, music, or film, and so involves mechanics and dynamics as well as content.[6] A

[6] It's worth noting that several of these systems [22, 21] are themselves highly interactive, but they don't *generate* interactive systems.

particularly interesting parallel is with MAKEBELIEVE [23], a demo that produces very short stories about a requested subject by querying OpenMind for plausible things the subject might do and stringing them together. Our prototype might be viewed as the interactive analog, generating micro-games that tell plausible stories about the noun and/or verb the user requests.

5 Conclusions and Future Work

Automated game generation is a little-explored area of research that we feel holds great potential, both as a technique that would enable the development of new types of games, and as a research agenda that tackles the problem of machine creativity from the perspective of the generation of highly interactive artifacts. We described a general framework for viewing this problem, and a prototype system that generates games in the style of *WarioWare* about user-requested subjects, while respecting common-sense expectations about the roles of verbs and nouns in those games.

There are a large number of avenues for future work on this subject. In the short term, we plan to scale up our prototype to generate a wider variety of *WarioWare*-style games and gain feedback from users on the perceived strengths and weaknesses of the system to guide future improvements. We will also explore the use of the system to generate newsgames, using biased summaries of news stories as a starting point for generation. In the longer term, we'd like a more iterative blackboard approach to generation, in which different components of the generator revise a design-in-progress, allowing design failures in various aspects of the design to force revision of other aspects. To allow such a system to perform complex analysis of its design-in-progress, we envision a more formal representation of a game, perhaps as a high-level game simulation defined by logical assertions; in such a system, a more formal database of common-sense knowledge along the lines of Cyc might be a good fit.

6 Acknowledgments

We would like to thank Nuri Amanatullah, Thib Guicherd-Callin, Jeremy Hay, and Ian Paris-Salb (UC Santa Cruz) for developing a package to implement *WarioWare*-like games in J2ME; Ian Bogost (Georgia Tech) for useful discussions on various aspects of the project; and Chaim Gingold (Maxis) for suggesting *WarioWare* as an interesting game-generation domain. Thanks also to support from Intel and from the National Science Foundation's Graduate Research Fellowship Program.

References

1. Zagal, J.P., Mateas, M., Fernández-Vara, C., Hochhalter, B., Lichti, N.: Towards an ontological language for game analysis. In: Proceedings of the 2005 Digital Games Research Association Conference (DiGRA). (2005)

2. Pell, B.: Metagame in symmetric, chess-like games. In Allis, L.V., van den Herik, H.J., eds.: Heuristic Programming in Artificial Intelligence 3: The Third Computer Olympiad. Ellis Horwood (1992)
3. Gingold, C.: What WarioWare can teach us about game design. Game Studies **5**(1) (2005)
4. Liu, H., Singh, P.: ConceptNet: A practical commonsense reasoning toolkit. BT Technology Journal **22**(4) (2004)
5. Fellbaum, C., ed.: WordNet: An Electronic Lexical Database. MIT Press (1998)
6. Singh, P.: The public acquisition of commonsense knowledge. In: Proceedings of the AAAI Spring Symposium on Acquiring (and Using) Linguistic (and World) Knowledge for Information Access. (2002)
7. Lenat, D.B.: CYC: A large-scale investment in knowledge infrastructure. Communications of the ACM **38**(11) (1995) 33–38
8. Mueller, E.T.: Natural Language Processing with ThoughtTreasure. Signiform (1998) Online: `http://www.signiform.com/tt/book/`.
9. Lieberman, H., Liu, H., Singh, P., Barry, B.: Beating common sense into interactive applications. AI Magazine **25**(4) (2004) 63–76
10. Pedersen, T., Patwardhan, S., Michelizzi, J.: WordNet::Similarity: Measuring the relatedness of concepts. In: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI). (2004)
11. Orwant, J.: EGGG: Automated programming for game generation. IBM Systems Journal **39**(3–4) (2000) 782–794
12. Wright, W.: The future of content. Keynote address, 2005 Game Developers Conference (2005) Recording: `http://www.gamasutra.com/features/20050525/wright_01.shtml`.
13. Roden, T., Parberry, I.: Procedural level generation. In Pallister, K., ed.: Game Programming Gems 5. Charles River Media (2005) 579–588
14. Compton, K., Mateas, M.: Procedural level design for platform games. In: Proceedings of the 2nd Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE). (2006)
15. Zhou, H., Sun, J., Turk, G., Rehg, J.M.: Terrain synthesis from digital elevation models. IEEE Transactions on Visualization and Computer Graphics **13**(4) (2007) 834–848
16. Holm, J., Arrasvuori, J., Havukainen, K.: Using MIDI to modify video game content. In: Proceedings of the 2006 International Conference on New Interfaces for Musical Expression (NIME). (2006) 65–70
17. Meehan, J.: TALE-SPIN. In Schank, R.C., Riesbeck, C., eds.: Inside Computer Understanding: Five Programs Plus Miniatures. Lawrence Erlbaum (1981) 197–258
18. Turner, S.R.: The Creative Process: A Computer Model of Storytelling. Lawrence Erlbaum (1994)
19. Cohen, H.: What is an image? In: Proceedings of the 6th International Joint Conference on Artificial Intelligence (IJCAI). (1979)
20. Cope, D.: Virtual Music: Computer Synthesis of Musical Style. MIT Press (2001)
21. Thom, B.: Interactive improvisational music companionship: A user-modeling approach. User Modeling and User-Adapted Interaction **13**(1–2) (2003) 133–177
22. Mateas, M., Vanouse, P., Domike, S.: Generation of ideologically-biased historical documentaries. In: Proceedings of the 17th National Conference on Artificial Intelligence (AAAI). (2000) 36–42

23. Liu, H., Singh, P.: MAKEBELIEVE: Using commonsense knowledge to generate stories. In: Proceedings of the 18th National Conference on Artificial Intelligence (AAAI). (2002) 957–958