An Interactive Game-Design Assistant

Mark J. Nelson
College of Computing
Georgia Institute of Technology
mnelson@cc.gatech.edu

Michael Mateas
Computer Science Department
University of California, Santa Cruz
michaelm@soe.ucsc.edu

ABSTRACT

Game-design novices increasingly hope to use game-like expression as a way to express content such as opinions and educational material. Existing game-design toolkits such as *Game Maker* ease the programming burden, bringing the design of small games within the technical reach of low-budget, non-expert groups. The design process itself remains a roadblock, however: It is not at all obvious how to present topics such as political viewpoints or bike safety in a way that makes use of the unique qualities of the interactive game medium. There are no tools to assist in that aspect of the game design process, and as a result virtually all expressive games come from a small number of game studios founded by experts in designing such games. We propose a game-design assistant that acts in a mixed-initiative fashion, helping the author understand the content of her design-in-progress, providing suggestions or automating the process where possible, and even offering the possibility for parts of the game to be dynamically generated at runtime in response to player interaction. We describe a prototype system that interactively helps authors define spaces of games in terms of common-sense constraints on their real-world references, provides support for them to understand and iteratively refine such spaces, and realizes specific games from the spaces as playable mobile-phone games in response to user input.

INTRODUCTION

As games have become a hot cultural phenomenon, an increasing number of game-design novices wish to express desired content or accomplish a goal using game-like expression. The UCSC Expressive Intelligence Studio alone receives several cold calls a month from advocacy and education groups who wish to build small serious games to teach specific material or express a point of view, typically with little to no budget. What is preventing these groups from being able to build their games is not access to game programming tools; there are now numerous game toolkits such as *Game Maker*,[1] as well as web technologies such as Flash, that greatly ease the programming burden for creating simple games.

Rather, the main roadblock is the game design process itself: the process of presenting desired content, such as bike safety, the policy implications of a proposed law, or high school algebra, in a way that makes use of the unique qualities of the game medium. These unique qualities include creating a strong sense of player agency, procedurally expressing the possible-worlds implications of player decisions, and enabling players to experientially explore rule systems.[2] Existing game toolkits such as *Game Maker* or *Alice*[3] provide no help with the design problem of mapping content into a system of game rules, but rather facilitate the programming task of implementing the game rules once design is already done. And indeed, we find that many games that deal with sophisticated real-world referents, such as political games and persuasive games, are produced by small studios studios run by academics who are experts in expressive game design.[4] We propose to build a tool to facilitate novice game designers in the initial, creative design phase.

Our major goal is to create a creativity enhancing tool that supports novice game designers in the creative practice of expressing a desired content area (theme) in game form. The purpose of such a tool is to help authors more quickly understand the current state of a partial game design, and to automatically suggest modifications or fill in design details. In addition, a system that understood how to partially automate the game-design process with input from the author could be used more radically to do partial game design online during actual gameplay. In that scenario, the author partially specifies how the game can respond, and the game adapts intelligently within those parameters to the person playing the game, allowing the author to deploy automated game design as an element in her expressive arsenal.

[1] `http://www.yoyogames.com/gamemaker/`

[2] Mateas [15] and Bogost [4] discuss in more detail the particular expressive affordances of procedural systems, and how they differ from those of other media.

[3] `http://www.alice.org/`

[4] Some well-known examples are *Madrid* (`http://www.newsgaming.com/games/madrid/`), a memorial game released shortly after the 2004 Madrid train bombings, from a team in which one author wrote his PhD thesis on the subject [8]; and *Food Import Folly* (`http://select.nytimes.com/2007/05/24/opinion/20070524_FOLLIES_GRAPHIC.html`), an editorial game for the *New York Times* online opinion pages about contaminated food imports, from a studio whose cofounder wrote a book on the subject [4].

Permission to make digital or hard copies of part or all 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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IUI'08, January 13–16, 2008, Maspalomas, Gran Canaria, Spain.
Copyright 2008 ACM 978-1-59593-987-6/ 08/ 0001 $5.00

Proceedings of the 2008 International Conference on Intelligent User Interfaces (IUI), pages 90–98.

In this paper we present a prototype game-design assistant as a first step towards that goal. This system helps an author interactively define spaces of games in terms of constraints on their real-world references. A space of games is something like, for example, "a game where an attacker chases a target and the target tries to get away"—our focus initially is on mapping thematic content into game spaces, not on reasoning about and generating abstract game rules. The author specifies the nouns and verbs in the game, and defines constraints over real-world referents using the ConceptNet [13] and WordNet [7] databases of common-sense knowledge. Since these knowledge bases are large, yet incomplete (often in unexpected ways), we help the author understand the implications of their current design (as the system understands it) as they add and modify constraints, and provide a way of efficiently browsing through the databases to suggest useful additions.

The author then provides mappings from the terms in their game space to slots in templated stock games that are implemented in literal executable code. Since the author-provided constraints define a space of games rather than a specific game, they allow for partial game-generation at runtime, constrained by the author's goals. In our current prototype, the author can either select a concrete game in which specific nouns and verbs that satisfy the constraints have been mapped to a stock game, or can deploy the generative space of games as specified by the constraints. In the latter case, the player asks for a game about a specific noun and verb, and games from the author-defined space of games are chosen based on their similarity to the player's request. The specific game is then mapped to the stock executable implementation by the author-defined mapping rules, and generated as a playable game. Of course in the long run we would like to support an author in designing the game mechanics as well, but for now this approach allows us to focus specifically on how to support an author in reasoning about a game's real-world references to produce a deployable game that customizes those references in response to the player.

Finally, we demonstrate some examples of using our game-design assistant to specify sensible game spaces that can be mapped on to micro-games in the style of Nintendo's *WarioWare* series. These are small games, typically lasting a few seconds, that involve one simple element of gameplay such as "dodge the car". We choose this domain as an example because its gameplay is simple enough to allow a focus specifically on the real-world references rather than the game mechanics, yet the space of possibilities is rich enough for the series to be interesting to game designers [10].

FACTORING THE GAME-DESIGN PROCESS

To assist with and partially automate the game-design process, we find it useful to factor it into several components.[5] We've previously proposed [17] a model of game design that uses four main interrelated components: the *abstract game mechanics* that specify a game's state and state evolution; the *concrete game representation* that specifies how these abstract mechanics are represented to the player; the *control mappings* that specify how the user interacts with the game state; and the *thematic content* that comprises the game's real-world references.

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 an typical arcade game, abstract state would include things such as time limits, health meters, player status, and so on. In a game like chess, the abstract game mechanics are specified by the rules of the game. Specific games make concrete design commitments within more general spaces of games; for example, chess is one particular set of rules for a symmetric, 2D, tile-based game. 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. A time limit, for example, might be represented as a literal clock on screen, by the position of some object on the screen, or even by the tempo of the music. 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 taking a 2D, third-person game and making a 3D, first-person version.

Thematic content comprises the real-world references expressed by the game. For example, the game *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 taking *Tapper*, a 2D resource-management game set in a bar, and changing its domain to a fast-food restaurant.

Finally, control mapping describes the relationship between physical player inputs, such as button presses and joystick movement, and modification of abstract game state. In *Tapper*, a bar-serving game, 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 might have included 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

[5]Note that while we'd like this to be a plausible model of game design, our focus is on a model useful for our purposes, not on analyzing how human game designers actually operate.

mechanisms such as dancepads or gestural controllers).

Existing systems

Two lines of existing research aim to automate or assist with videogame design in general. Game-design toolkits such as *Game Maker* and *Alice* allow users to interactively define objects that should appear in the game, their appearance, and how they should interact with each other. These tools generally provide "wizards" to automate certain common tasks, such as defining how collision detection between pairs of objects should work. Game-generation systems, meanwhile, completely automate game design within a predefined space of possibilities. For example, METAGAMER [20] generates chess-like games, parameterized by user-tunable characteristics such as locality of movement.

Understood in terms of our categories of game design, game-design toolkits generally collapse abstract game mechanics and concrete game representation, and then assist with specifying this unified mechanics/representation. The user specifies what objects will be on screen and how they'll interact with each other, which simultaneously specifies the mechanics and how they'll be represented. Control mappings are then assigned directly to objects of the concrete representation by mix and matching certain stock sets of controls, *e.g.* by assigning the arrow keys to control a particular object's position.

Existing automated game design systems hold some knowledge categories of game-design fixed and focus their generation effort on variations within one or two of the remaining categories. METAGAMER [20], for example, literally generates abstract game mechanics, producing as its output a set of logical statements that define the rules for a chess variant. The concrete representation, control mappings, and thematic content aren't specified, but are implicitly assumed to be those traditional for chess, with some variations to account for, *e.g.*, a different sized board or different kinds of pieces. The author controls the process by varying a few parameters that summarize some aspect of the abstract game mechanics, such as locality of movement. EGGG [18], meanwhile, takes a specification of abstract game mechanics as its input, and from that generates a graphical, playable game. It understands certain classes of games (chess-like games, card games, etc.), each of which has the thematic content hard-coded (*e.g.* the chess pieces), plus a set of general conventions for concrete representations and control mappings (*e.g.* how to represent a hidden hand of cards). The system's job is to adapt the set of representations and control mappings to the particular variation of abstract game mechanics being specified. The author does not really control the process in EGGG at all, except by what abstract mechanics she feeds into it.

Both of these approaches focus almost exclusively on helping the user define some or all of the abstract game mechanics, their concrete representation, and the control mappings, and provide no real assistance on how to design thematic content into the game. When designing expressive games, though, that's precisely the difficult part: not figuring out how to literally put some objects on screen, let the user click on them, and define how they react to clicks or bounce off each other, but thinking about how to make a game about, say, contaminated food imports, that makes sense and effectively conveys an opinion.

Our goal is to combine the incremental, interactive design supported by game design toolkits with the intelligent support of a game generation system. Game-generation systems automate parts of the game design process, but do so in a very non-interactive way: The user specifies some parameters, and then a game is generated. Revising a game consists of trying out some new parameters and looking at the result again. Game-design toolkits, on the other hand, provide much richer interaction to support incremental game design and revision, but rarely supply much (if any) intelligent automation or feedback; the "wizards" that automate certain common tasks generally do so just by allowing the author to mix and match a few common bundles of code, such as different kinds of collision detection.

An ideal system

We propose that a game-design assistant ought to support mixed-initiative interaction, in which the user carries out interactive, incremental game design and revision, but with the system using its understanding of the game-design process and the partial game design that's been specified so far to provide design suggestions and help the user understand the implications of various design decisions.

The system's interaction would serve two main purposes: First, to give feedback to the author on the state of their current design; and second, to suggest modifications, additions, and so on in an intelligent way. Like current game-design wizards it ought to include some stock resources, but from all the different components of game design (theme, abstract mechanics, etc.), rather than only stock elements for low-level concrete mechanics such as collision detection. For example, an author building a game to highlight some point about economic policy through a simulation could start with a set of generic resource-balancing game mechanics.

Feedback would be useful to short-circuit the usual code-compile-playtest cycle. If the design-support system had sufficient knowledge of the current design it could, for example, quickly play through with even a simple simulated player and spit out some screenshots of notable points in the game, allowing the author to glance and see if something was happening that they weren't expecting. More explicitly, the author could be warned of things such as unreachable parts of the game.

Our prototype

In our prototype system, we aim to combine generation and interactive design, and to focus on the problem of mapping thematic content onto game mechanics. The three main components of our system are *game spaces*, which define a space of possible games such as a game where one entity tries to avoid another entity; *stock mechanics*, which are literal game code with templated slots; and *mapping rules*, which spec-

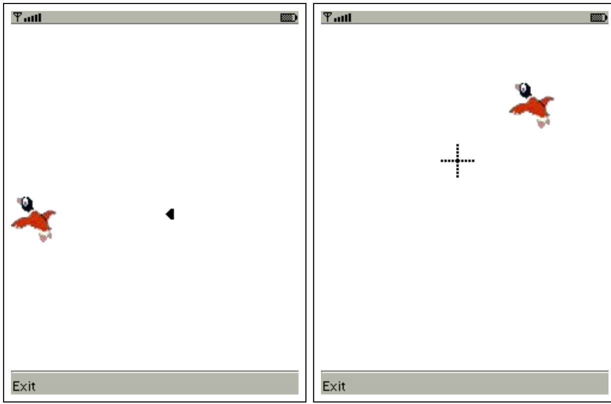

Figure 1. Two mappings of a game from the attacker-avoidance game space in which a duck avoids a bullet.

ify how a game space should map onto a stock mechanic to render a specific game within the space playable.

The game spaces are primarily thematic, but also include some assumptions about how the abstract mechanics would operate. For example, a game with one entity avoiding another has two entities (the avoider and the attacker), which must be chosen so that it "makes sense", in terms of common-sense knowledge, that one would be avoiding the other. This thematic content comes with an assumption that the game itself, however implemented, will involve a mechanic where the avoider tries to get away, and wins if successful, or loses if caught.

The stock mechanics are bundles of both abstract and concrete game mechanics that specify a fully playable game but with templated slots for some entities in the game. For example, one of our *WarioWare*-style mechanics (which we call *Dodger*) has one object moving across the screen, while another object that the player controls moves up or down quickly to get out of its way, as shown in Figure 1 (left).

Finally, the mapping rules ensure that the abstract-mechanic assumptions that are implicitly part of a game space are respected, by mapping the game space onto stock mechanics in such a way that the common-sense thematic constraints are preserved. For example, Figure 1 shows two different ways in which a game about a duck avoiding a bullet can be mapped onto two different stock mechanics. In the game shown on the left, the avoider (a duck) is mapped onto an avatar that the player moves up and down, while the attacker (a bullet) is mapped onto a computer-controlled sprite that moves left across the screen towards the player. In the game shown on the right, the the avoider (a duck again) is mapped onto a computer-controlled sprite that moves randomly, and the player controls crosshairs to play the role of the attacker (a bullet again), which isn't explicitly shown. These examples of mappings are straightforward one-to-one mappings, but more complex mappings can also be constructed, where for example animate and inanimate objects map differently, and other parts of the game are changed based on properties of the specific terms being filled into the slots.

In our example *WarioWare* domain we've so far defined three game spaces that map onto five sets of stock mechanics [17]. The three game spaces are an attacker-avoidance game, a reservoir- or meter-filling game, and an acquisition game. The five stock mechanics are *Dodger*, which shows a side view in which a player dodges an incoming object; *Shooter*, which shows a target crosshair the player uses to shoot an object; *Pick-Up*, in which the player runs through a maze to pick up an item; *Pump*, in which the player button-mashes to raise a meter; and *Move*, in which a player button-mashes to move across the screen. We'll use the attacker-avoidance game space as a running example.

Once a game space is defined, a constraint-solving backend finds possible specific games that fit in the space, guided by input from the player. In our prototype, the player is asked for a noun and verb, and then given a sequence of microgames where the terms are filled in based on similarity to their requested noun and verb (as measured by distance in the WordNet hierarchy). The specific game from the game space is then mapped onto one of the stock mechanics based on mapping rules specified by the author. In our current examples these are straightforward one-to-one mappings; for example, an attacker-avoidance game maps onto a Dodger stock mechanic by putting the avoider as the entity that's dodging (and controlled by the player) and the attacker as the thing to dodge. The filled in stock mechanic is then compiled and a custom playable game for the mobile Java platform is generated.

We focus in this paper on the authorial process of defining a game space. A full system would support the author in defining and mixing and matching interesting mappings to stock mechanics, and provide support for modifying and reasoning about the mechanics themselves, rather than restricting the author to a set of stock mechanics.

DEFINING GAME SPACES VIA CONSTRAINTS

The author in our system defines a game space by specifying variables, which are marked as nouns or verbs, and constraints on the variables. The constraints are either on the values that individual variables can take, or on how the values of multiple variables must relate to each other. The author may specify the constraints using relations in the ConceptNet [13] and WordNet [7] databases, as well as some combinations of the two and logical operators like *and* and *or*. Figure 2 shows the set of variables and constraints specifying an "avoid" game, where one noun avoids another noun, which we'll use as an example. Note that a user of the interactive tool need never see this raw game description; see Figure 3 for the graphical view.

ConceptNet and WordNet

ConceptNet is a graph-structured common-sense knowledge base extracted from OpenMind [22], a collection of semi-structured English sentences expressing common-sense facts gathered from online volunteers. ConceptNet's nodes are English words or phrases, and its links express semantic relationships such as *(CapableOf "person" "play video game")*. In our current prototype, *CapableOf*, *CapableOfReceivin-*

```
noun Avoider
noun Attacker
verb Attack_verb: shoot, attack, damage, chase, injure, hit
constraint: (ConceptNet CapableOfReceivingAction ?Avoider ?Attack_verb)
constraint: (WordNet hyponym ?Avoider "animate thing")
constraint: (or (and (WordNet hyponym ?Attacker "projectile")
                     (ConceptNet CapableOfReceivingAction ?Attacker ?Attack_verb))
                (ConceptNet CapableOf ?Attacker ?Attack_verb))
```

Figure 2. A definition of an example game space, specifying games where an Avoider avoids an Attacker. See text for explanation, and Figure 3 for the graphical view of this game space.

gAction, *PropertyOf*, and *UsedFor* are the most useful relations.

WordNet is a hierarchical dictionary of English words. 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 is a "hyponym" if a noun or "troponym" if a verb). This can be used to constrain variables in a game space to specific types of words; for example, constraining a noun to be a hyponym of "animate object" makes sure that inanimate objects aren't put into the slot of a stock mechanic where they wouldn't make sense. More generally, WordNet allows us to perform taxonomic generalizations over relations defined in ConceptNet.

Compared to more formally specified common-sense knowledge bases such as Cyc [11] and ThoughtTreasure [16], these databases use natural language and relatively loose semantics. This is nice because it allows an author to interact with the databases without having to learn how to navigate a particular formal ontological framework in order to define their game spaces. In addition, ConceptNet's natural-language approach to common-sense knowledge has been useful for a number of previous applications [12]. However, having the knowledge base in natural language does have some drawbacks when it comes to ambiguity and inability to usefully respond to complex queries, and indeed, helping an author navigate these difficulties is one of the main goals of our interactive approach to defining common-sense constraints.

ConceptNet has the more serious drawback of weak coverage: it knows that a duck can be shot, but not that a pheasant can be shot, for example. Fortunately, combining queries to ConceptNet with hierarchical information from WordNet mitigates this problem to a large extent. In specifying a particular ConceptNet constraint, the author can specify whether WordNet "inheritance" should be performed on any of the terms or variables in either direction (towards more general or towards more specific words). For example, the query about whether a pheasant can be shot should have hypernym (towards more general terms) inheritance enabled on "pheasant", and would therefore return true, because from WordNet we find that a pheasant is a type of animal, and ConceptNet knows that animals can be shot.

Example: An attacker-avoidance game space

Figure 2 shows an example game space specifying games where an *Avoider* tries to avoid an *Attacker*. The game space is defined by three variables—those two nouns plus a verb (*Attack_verb*)—and several constraints between them.

The nouns can implicitly range over any noun for which the system has graphics, subject to the constraints. The verb is specified to be one of five verbs that the author has chosen as representative of the type of action to take place in the game. The constraints specify how to bind these variables to specific terms from ConceptNet so as to maintain the common-sense semantics of "attack" and "avoid".

The first constraint specifies that the Avoider has to be capable of serving as the direct object of the Attack_verb (represented by the *CapableOfReceivingAction* relation in ConceptNet); hypernym inheritance is done on the Avoider (not shown in the figure for simplicity). The second constrains the Avoider to being an "animate thing", since it must move to avoid the Attacker. Some inanimate things could sensibly function as Avoiders, especially humorously (a piece of bread trying to avoid a toaster, say), but specifying which of them makes sense gets trickier, so in this example we as the authors have simply decided to limit the game space to consider only animate things as Avoiders.

The third constraint is somewhat more complex. The most obvious constraint to add is that the Attacker must be CapableOf the Attack_verb (with hypernym inheritance on the Attacker). After trying this, however, it turns out to preclude many games that we think of as canonical in the avoider game design space. Something trying to avoid a bullet, for example, is excluded because a bullet isn't itself CapableOf "shoot": A bullet doesn't shoot, but is shot, and therefore isn't *CapableOf*, but rather *CapableOfReceivingAction* shoot. We informally except it to serve in the role of attacker and perhaps anthropomorphize it as "chasing" the Avoider, but those notions are too subtle for the common-sense databases that currently exist to capture. To take these cases into account, we've added an alternate possibility for fulfilling the last constraint: If an Attacker is a projectile (according to WordNet), then we check whether the Attack_verb can sensibly act on *it*, rather than whether it can sensibly act out the Attack_verb.

Specific games in this game space are then mapped by author-specified rules to either of two sets of stock mechanics. Figure 1 shows two example mappings of a game where a duck avoids a bullet. In one case, the player plays the duck and avoids an incoming bullet in a side-scrolling mechanic; in

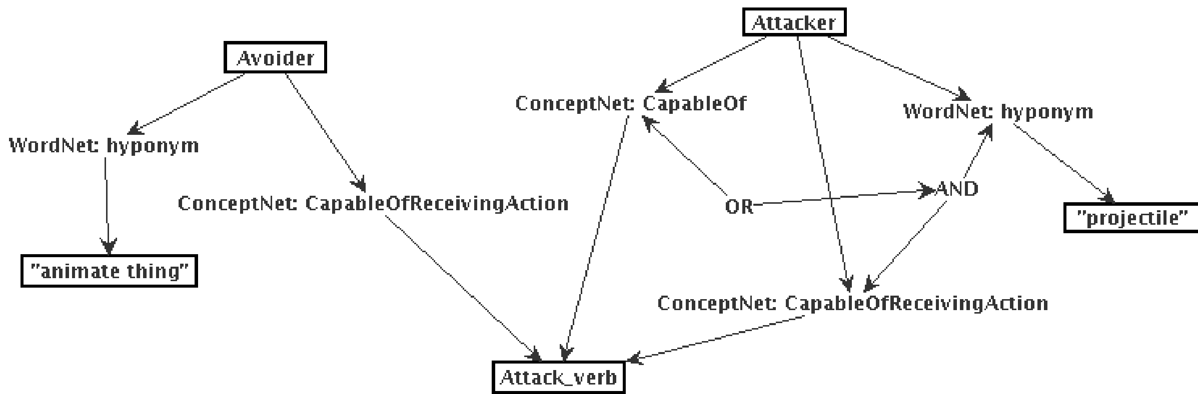

Figure 3. The graphical view of the attacker-avoidance game space specified in Figure 2.

the other, the player plays the implicit shooter of a bullet, and aims at the duck using a cross-hair targeting mechanic.

Providing a mechanism for specifying variables and constraints is not enough to support an author in the thematic mapping of game content. Given the vast size yet lack of completeness of common-sense knowledge bases, a game space defined by a given set of constraints will often produce counterintuitive results, including concrete games that the author doesn't expect, while excluding games the author might think are canonical in the game space. Thus an intelligent game authoring tool must support the author in dynamically exploring a game space, allowing her to ask the system why a given generated game is allowed by the current constraints and to iteratively modify constraints. In fact, it is this iterative exploration that is the heart of the leverage provided by an intelligent game design tool; the combination of human plus system can explore a much larger design space than an unaided human could, while the human provides heuristic search guidance that would be extremely difficult to encode in a fully automated system.

INTERACTIVELY DEFINING GAME SPACES

The interface for specifying game spaces centers around a graph-structured view of the variables and constraints defining a game space, with a number of tools for querying and modifying it. Figure 3 shows the constraint graph for the example attacker-avoidance game whose raw textual specification was shown in Figure 2. Variables and constraints can be added or removed interactively (the constraint-satisfaction backend re-runs in the background after modifications are made). Some filtering is done in the interface so that only relevant additions are shown; for example, an author selecting two nouns and clicking to add a constraint will only get a list of constraint types relevant for pairs of nouns, which excludes those such as ConceptNet's *CapableOf* that involve verbs.

An author can query a particular variable to get a list of terms that could be bound to that variable when generating a real game, under some binding of other nouns and verbs in the game space. More helpfully, she can click on a particular word to get an explanation of why it satisfied the constraints.

Figure 4 shows an example from an earlier iteration of our attacker-avoidance game space. Before we tried "projectile" in the third constraint (see Figure 2 and previous discussion), we had tried "device"; the idea was that it would be nice if objects other than traditional projectiles could be used as projectiles anyway, like a hammer being thrown at the avoider. Upon adding that option, though, we got a game with a piano flying across the room, which, without some sort of context that wasn't present in this simple game space (say, a gorilla shown as throwing the piano at the player), didn't make much sense. The view in Figure 4 quickly lets the author click "Why?" to find out why: a piano is a both a device (according to WordNet) and *CapableOfReceivingAction* "hit" (according to ConceptNet), so fits all the constraints. Presumably a piano is a device for making music, and its keys can be hit, which isn't really what we had intended those constraints to mean. It's possible that we could've refined the constraints further to allow some non-projectile devices while excluding others; we chose the alternate solution of limiting "device" further to "projectile". In an earlier, fully automated version of the system, we had to manually dig through debug output and trace through the ConceptNet and WordNet graphs in order to figure out why we were getting unexpected results such as this. This painful manual process was one of the motivations for moving towards an interactive game design assistant supporting visual exploration of game spaces.

To get a bigger-picture view of how all the variables and constraints in a game space interact, the author can get a list of some possible complete games; Figure 5 shows an example. The author can click on any particular term in this list to get an explanation as well. To provide a visual overview, we can also show the author a set of graphical screenshots of some of the concrete games in the game space, based on the separately defined game space to stock mechanic mappings, on which she can click to get textual description of the values filled in for the variables.

The overall purpose of the interface is to make it possible for an author to efficiently use specific examples of games that do or don't fit her goals for the game space (or make sense at all, for that matter) in order to reason about general rules specifying what games should be included in and excluded from the space. To deploy a game, the author has two choices. If she wishes to deploy one or more concretely instantiated games, she can select concretely instantiated games from the examples generated by the system. In this case, the authoring system is serving as a "brainstorming assistant", providing help in performing thematic mapping onto game mechanics, and helping her to explore a larger design space than she might have been able to unaided. If she wishes to deploy a generative design space, in which games are generated at runtime as the player plays them, she can deploy a fully-automated version of the system, where the game spaces (variables + constraints) have been iteratively refined during authoring. In this case, the authoring system is helping the author to visualize game design spaces, and to refine constraints until the game space(s) reliably produce the author-desired range of games.

People often find it more intuitive to define categories by specifying prototypes rather than logical rules [21]. In recognition of that fact, some systems try to infer constraints from examples demonstrated by users, rather than requiring the users to specify explicit constraints; for example the graphical editor GRACE infers constraints on the relationships between elements of the design (*e.g.* in a CAD diagram) by user demonstration and some inference heuristics [1]. Those approaches succeed in domains where the constraints are simpler and more concrete, however. In the game-design domain, the state of common-sense reasoning isn't sufficiently advanced for us to reliably infer something like "a game about avoiding an attacker (where the attacker and avoider and their relationships 'make sense')" from some examples of games that should and shouldn't be in that space.

Rather than inevitably getting inferences from prototypes wrong, we try to help the author understand the constraints they're in the process of specifying—and in particular to understand what the system thinks the constraints they're specifying mean—by frequent reference to prototypical examples.[6] The goal is to make failures of common-sense reasoning explicitly visible so the author can recognize and work around them, helping the author to define an explicit set of rules that can be applied concretely and that are congruent with her goals. We hope this incorporates some of the intuitive advantages of a prototype approach without relying on inferences that, in the common sense domain of thematic mapping onto game mechanics, would be unreliable and result in confusion.

OTHER RELATED WORK

While research on game generation and game-design wizards is most directly related to our work, there are several additional lines of related work.

[6]For the moment we don't explicitly choose examples to be prototypical; in future work it may be useful to identify examples that serve as particularly good prototypes.

Figure 4. Possible assignments to a particular variable, letting us look for unexpected results and query why they satisfied the constraints.

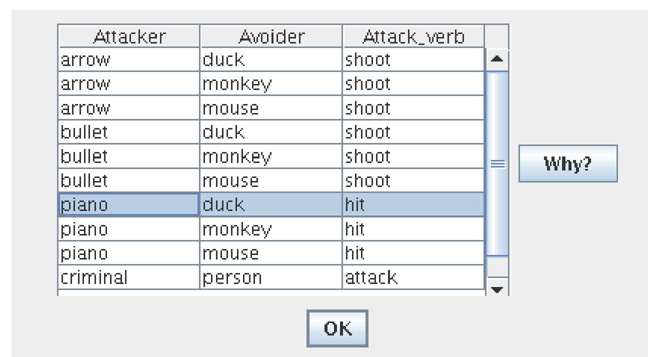

Figure 5. Possible assignments to all variables, showing a more contextual view of assignments than the single-variable view in Figure 4. The author can query particular assignments here as well.

Constraint-based approaches have been used to define "design spaces" for tasks such as designing buildings or user interfaces, or even composing contrapuntal music. Either an AI system or a person guided by the constraints then searches for solutions within these spaces [6, 1, 5, 19]. In these systems, however, both the constraints and the interactive exploration of the design space serve rather different purposes than in our system. In constraint-based design, the constraints are typically design parameters specifying what a system ought to do, such as load factors on structural components, required functionality of a user interface, or the rules of contrapuntal music. Interactive navigation in such systems is designed to help a designer explore within the space defined by those invariants; for example, to compose a particular piece of contrapuntal music while adhering to the constraints of that type of composition [19].

In our system, by contrast, constraints are a means to allow authorial expression while still retaining some degree

of freedom that the system can use online to customize the games to a player's input: The author is *expressing* some desired invariants, rather than working subject to them. The interactive design process is therefore intended to help the author reason about the constraints themselves in order to help her understand the current design space, and decide whether to add, remove, or revise constraints to modify it. This does have more similarity to the graphical editor GRACE, which infers constraints by demonstration and explains to users what constraints were inferred and why. Our focus is somewhat different, since we don't infer constraints automatically, but instead focus on explaining the effects of the existing set of constraints and how they might operate and interact in unexpected ways.

Allowing an author to partially specify a game which is then fleshed out into a specific complete game at runtime in response to player interaction shares some similarity with partial programming [2, 3]. However, partial programming is task-oriented, designed to allow an author to specify a partial solution to a problem while marking parts of the program for the system to adapt online in response to specific situations, generally using statistical machine learning techniques. Since our partial specifications are meant as a tool of authorial expression rather than as a way of making it easier to write programs that accomplish functional tasks, both the motivation and technical details of our approach differ [14].

There is some similarity between our authorial support tool and interactive knowledge acquisition tools. Helping an author specify a space of games can be seen as acquiring knowledge about game design, and so the design issues overlap somewhat with knowledge acquisition systems. We incorporate some suggestions from the literature on the use of tutoring-style interaction in knowledge acquisition [9]. Rather than merely passively allowing an author to specify their ideas about game design, we aim to provide frequent reflection back to the author of what the system thinks it's learned about the author's ideas thus far, in the form of example games generated with the current set of constraints, and explanations of why they fit those constraints. ComicKit [23] also uses ConceptNet to facilitate knowledge acquisition, though it does this in the form of one-shot suggestions that complete user-entered partial scripts, rather than offering iterative reflection over the consequences of the currently entered knowledge.

CONCLUSIONS AND FUTURE WORK

We propose a game-design assistant to help authors with the full game design process, not only with the implementation process that current tools assist with. We propose that such a system should support the following three tasks. First, it should help authors understand the state of a partial game design, reflecting back to them what the system understands about the current design so the authors can more easily find inconsistencies and design flaws. Second, it should suggest modifications or automatically fill in design details, both to help novice designers and to make the process less tedious. Finally, it should provide support for some automatic design to be pushed into the run-time game itself in ways specified by the author, for example to allow games to customize themselves to particular players.

Our prototype system focuses on a subset of this goal. It allows authors to define thematic "game spaces", that is, the space of real-world references a game can make, by specifying sets of nouns and verbs that comprise the abstract entities and action in the game, and constraints on how these entities must relate to each other to create a game that's both sensible and in the intended space. The constraints are specified in terms of the ConceptNet and WordNet common-sense databases. Our system focuses primarily on helping the authors to understand the current state of their design by automatically generating example games in the game space, and supporting the author in iteratively modifying the game space.

With respect to the full set of desiderata for a game-design assistant, our prototype can be seen in two ways, depending on how the game spaces are deployed to the player. A game space can be deployed as an author-defined space of possible variation, within which specific games are generated at run-time based on player interaction. In this view, our author-support tool works to help the author understand their design-in-progress, and to reason about how automatic design/customization will operate when it's pushed into the run-time game.

Alternately, a game space can encode knowledge about designing a particular type of game (something more specific than a game genre), within which specific games would be designed by novice authors. In that view, defining game spaces is part of building the knowledge that should go into a game-design assistant, and would be done by people with some knowledge of game design. A novice designer could then use a tool with many of these game spaces already predefined and would explore specific designs within those spaces. Should that prove too limiting past a starting point, they could then escape back into something like the the tool described in this paper to understand and edit the "meta-level" of game spaces rather than treating them as fixed.

Other future improvements can take place along a number of avenues. We currently have simple mappings from game spaces to stock game mechanics. An author-support interface to allow more complex mix-and-matching between game spaces and mechanics would allow for more easily defining interesting new types of games. More fundamentally, removing the "stock" in the stock mechanics and allowing iterative and interactive design of the game mechanics themselves would get us much closer to our ultimate goal of a game-design system that supports the author in all aspects of the game-design problem.

Our system also currently focuses on giving feedback to the author on their design so they understand how common-sense reasoning operates or fails to operate. There is plenty of room for more in the way of constructive suggestions. Expert game designers draw on a wide range of common game-design tropes; a system might help guide a novice

game-designer by helping them apply such tropes as necessary. Starting with a set of expert-defined game spaces is one way to encode such knowledge at the level of full, abstracted game designs that an author can start from, but it would also be useful for the system to have knowledge of finer-grained tropes, perhaps suggesting additions from a database that are filtered for applicability to the current partial design.

Finally, we've only scratched the surface of possible ways an author might want to push dynamic generation into the actual deployed game so that it can be customized by or react intelligently to a specific player.

ACKNOWLEDGMENTS

Thanks to Nuri Amanatullah, Thib Guicherd-Callin, Jeremy Hay, and Ian Paris-Salb (UC Santa Cruz) for developing a package to implement *WarioWare*-like games on the Java mobile platform; and to Chaim Gingold (Maxis) for suggesting *WarioWare*-like games as an interesting domain. This research was supported by grants from Intel and from the National Science Foundation's Graduate Research Fellowship Program.

REFERENCES

1. S. R. Alpert. Graceful interaction with graphical constraints. *IEEE Computer Graphics and Applications*, 13(2):82–91, 1993.
2. D. Andre and S. Russell. Programming reinforcement learning agents. In *Advances in Neural Information Processing Systems (NIPS)*, 2000.
3. S. Bhat, C. L. Isbell, Jr., and M. Mateas. On the difficulty of modular reinforcement learning for real-world partial programming. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, 2006.
4. I. Bogost. *Persuasive Games: The Expressive Power of Videogames*. MIT Press, 2007.
5. A. Borning and R. Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, 1986.
6. B. Chandrasekaran. Design problem solving: A task analysis. *AI Magazine*, 11(4):59–71, 1990.
7. C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
8. G. Frasca. *Play the Message: Play, Game and Videogame Rhetoric*. PhD thesis, IT University of Copenhagen, 2007.
9. Y. Gil and J. Kim. Interactive knowledge acquisition tools: A tutoring perspective. In *Proceedings of the 24th Annual Conference of the Cognitive Science Society*, 2002.
10. C. Gingold. What WarioWare can teach us about game design. *Game Studies*, 5(1), 2005.
11. D. B. Lenat. CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.
12. H. Lieberman, H. Liu, P. Singh, and B. Barry. Beating common sense into interactive applications. *AI Magazine*, 25(4):63–76, 2004.
13. H. Liu and P. Singh. ConceptNet: A practical commonsense reasoning toolkit. *BT Technology Journal*, 22(4), 2004.
14. M. Mateas. Expressive AI: A hybrid art and science practice. *Leonardo*, 34(2):147–153, 2001.
15. M. Mateas. Procedural literacy: Educating the new media practitioner. *On the Horizon*, 13(1), 2005.
16. E. T. Mueller. *Natural Language Processing with ThoughtTreasure*. Signiform, 1998. Online: `http://www.signiform.com/tt/book/`.
17. M. J. Nelson and M. Mateas. Towards automated game design. In *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, pages 626–637. Springer, 2007. Lecture Notes in Computer Science 4733.
18. J. Orwant. EGGG: Automated programming for game generation. *IBM Systems Journal*, 39(3–4):782–794, 2000.
19. R. Ovans and R. Davison. An interactive constraint-based expert assistant for music composition. In *Proceedings of the 9th Canadian Conference on Artificial Intelligence*, 1992.
20. B. Pell. Metagame in symmetric, chess-like games. In L. V. Allis and H. J. van den Herik, editors, *Heuristic Programming in Artificial Intelligence 3: The Third Computer Olympiad*. Ellis Horwood, 1992.
21. E. Rosch. Natural categories. *Cognitive Psychology*, 4:328–350, 1973.
22. P. Singh. 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.
23. R. Williams, B. Barry, and P. Singh. ComicKit: Acquiring story scripts using common sense feedback. In *Proceedings of the 10th International Conference on Intelligent User Interfaces (IUI)*, pages 302–304, 2005.