Towards Knowledge-Oriented Creativity Support in Game Design

Adam M. Smith and Michael Mateas
Expressive Intelligence Studio
University of California, Santa Cruz
{amsmith,michaelm}@soe.ucsc.edu

Abstract

This article reports on a work-in-progress system designed to support game designers in gaining knowledge about the implications of their design ideas on observable gameplay. Utilizing a convenient pattern language, evidence of the instantiation of many gameplay patterns can be gathered and organized, resulting in insight.

Introduction

In game design, practices such as *prototyping* and *playtesting* are integral parts of the iterative, exploratory process used to achieve the innovative gameplay sought by creative game designers (Fullerton 2008). These practices reveal concrete details about game design spaces, allowing designers to refine their personal store of design knowledge. This design knowledge is used to engineer the complete, polished products we recognize as popular games, but it most often comes from experience with crude or incomplete game artifacts.

In this paper, we describe a work-in-progress system based on the theory of *rational curiosity* (Smith and Mateas 2011). This theory suggests that, in order to support creativity in game design, *systems should directly support designers in gaining design knowledge*. This contrasts with Yeap's desideratum of *ideation* (2010), that a support system should generate new ideas on its own. Quickly extracting useful feedback from existing ideas, we claim, is an underappreciated bottleneck in creative design process.

In game design, knowledge-oriented creativity systems should systematically expose the relation between the concrete details in a game's definition, such as its mechanics and level design, and the implication of these details on gameplay.

Building on the LUDOCORE logical game engine (Smith, Nelson, and Mateas 2010), our support system is targeted at early-stage computational gameplay prototypes (functioning models of a game that permit a designer to ask and answer specific design questions). LUDOCORE models capture focused situations in gameplay, including any available knowledge about the ideal player in addition to the game's mechanics. By using a logic programming representation, the system is able to exploit model-finding techniques to automatically solve for gameplay traces which exhibit properties that a designer has requested via a query. Knowledge gained from machine playtesting with LUDOCORE can be validated with human playtesting using the interactive, graphical features of BIPED (Smith, Nelson, and Mateas 2009), a process which often inspires new formal queries to pose in iterative machine playtesting. Using these tools in the larger game design process requires an external, creative agent to spot interesting patterns in gameplay traces and translate these patterns into a language the logical reasoning tools can understand in subsequent exploration.

If LUDOCORE is about getting design feedback from prototypes, but it requires a designer to first specify formal queries, can we assist the designer by translating her high-level interests into such queries and informatively aggregating the results? Such a straightforward process could dramatically speed up the rate at which a designer learns about her designs, improving her ability to appreciate artifacts – *appreciation* being one leg of Colton's creative tripod of perceived creativity (2008).

In this paper, we report on a system capable of collecting and organizing evidence for a space of gameplay patterns which are described in a designer-friendly language. After reviewing our example game, we describe how a preliminary experiment with our support tool using simple, hand-authored patterns has resulted in design insight.

DrillBot 6000 in LUDOCORE

Our support system works using a LUDOCORE model as input. Our examples will use the game *DrillBot 6000* (the example game that comes with BIPED). A screenshot of *DrillBot* is shown in Figure 1. In the game, the player controls a mining robot that must explore underground caverns, drilling out rocks and treasures. Actions such as mining rocks and moving upwards cost the robot energy that can only be recovered by refueling at the base.

The logic program that defines the game model declares events that may occur (such as mining a rock, moving to a space, and trading or refueling) and elements of state that change over time (such as the robot's position, energy level, and the presence of the various rocks). Additionally, the

definition contains assertions about the configuration of the game world (including the existence and linkage of caverns and the treasure property of some of the rocks).

Performing either human or machine playtesting with *DrillBot* produces symbolic gameplay traces. Simple traces log the actions (events) selected by player at each logical timepoint. Often, however, understanding an interesting property of play requires understanding the context of a particular sequence of player actions with respect to both the dynamic state of the game and its static configuration. We modified LUDOCORE to produce complete execution traces, records of every logical fact that is true in the game world in both the static and dynamic sense. Such complete traces represent an accurate view of the knowledge available to the designer when she is looking for patterns during playtesting, but they are very tedious to explore manually.

Where a simple trace may state that the event `mine(dino_bones)` happened at timepoint `22`, a complete trace will assert that mining is a player-selectable game event, that the event was possible at that time and others and was mutually exclusive with the two available movement events, that `dino_bones` is a rock with the `treasure` property, and that it is located in the cavern designated `i` which is linked to caverns `g` and `h`. If there is something interesting to be said about mining this rock, it is likely to involve some of these contextual details.

Using LUDOCORE's query language (based on logical integrity constraints), it is possible to ask for gameplay traces that illustrate how a player might navigate the robot down, drill out `dino_bones`, and return it to the base without ever letting its energy level drop below `6`. The code for query is small (just four lines), however writing it requires careful reasoning about the scope of variable quantification and domain restriction as well as reasoning through double-negation.

A Language of Gameplay Properties

To ease the definition of gameplay patterns that may be of interest in to a designer, we created a new language inside of Prolog (the syntax also used to define LUDOCORE games). Pattern definitions are declarations of what evidence must be present in (or absent from) a complete execution trace to detect an instantiation of that pattern. An example pair of patterns is shown in Figure 2.

Syntax

The `<--` (or is-detected-when) operator binds the name of a pattern (which might be parameterized by logic variables) to its requirements. Requirements can refer to the presence of elements in a trace such as that a game includes some event, that the event happens, or that some element of game state holds at some time. All LUDOCORE games share the general concepts of events and state, but many interesting patterns will make reference to game-specific concepts (such as the action of mining or a particular rock in *DrillBot*). The `primitive` construct can be used to require (or forbid using the `\+` operator) any element of a trace, whether it is game specific or not.

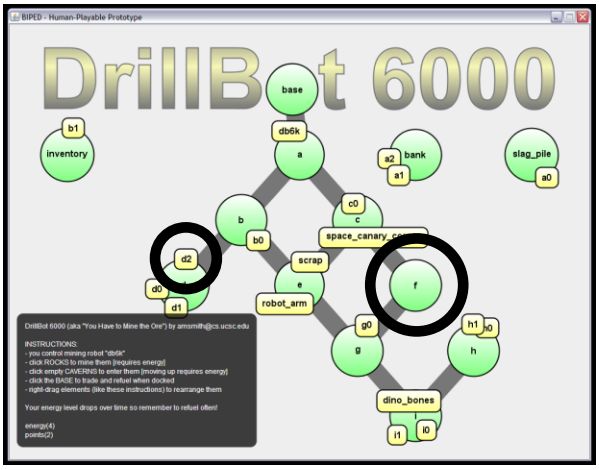

Figure 1. A screenshot of gameplay in the *DrillBot 6000* model. Black circles indicate game elements that our system automatically identified as ignored by players. The yellow token `d2` is a non-valuable rock in a dead-end cavern, and the space `f` is a linked cavern which provides no apparent navigation benefits.

To afford exploration of interesting patterns by seeing where they co-occur with other patterns and how their presence affects the conditional presence of other interesting patterns, requirements can also constrain the presence of any other pattern (using `pattern` construct).

A final construct of the language, `when`, can be used to describe additional constraints not present in the trace. A common use for this construct is to assert that two pattern variables should never be equal, or that (if they are timepoints) the enclosing pattern should only be detected when the values of the variables are strictly ordered.

Evidence Sets

Patterns in this language can be automatically translated into the more tedious query language supported by LUDOCORE. So far, we have only explored a fixed database of pre-collected traces. When asked to show evidence for the presence of a given pattern, our system finds all possible sets of evidence that, due to their presence in a trace, permit the detection of a pattern using some instantiation of its pattern variables.

Given a library of patterns, the system will produce a table of pattern names with concrete symbols substituted for variables scored by the number of distinct evidence sets which support each pattern. Given this table, a designer can then ask the system to display the detailed evidence sets for a particular instantiation. In many cases, it is the deeper examination of these evidence sets which suggests the definition of a new, composite pattern.

It is possible to use the compiled form of the pattern detector as a query in LUDOCORE. Thus, the designer can

use machine playtesting to directly search for more evidence of known patterns or ask about the existence of any possible traces that realize a freshly conjectured pattern.

Exploring Ignored Moves in *DrillBot*

To make our discussion of patterns and evidence sets more concrete, we will now consider the results of using our system to explore ignored moves in *DrillBot*. Figure 2 shows two pattern definitions in our library.

The `sometimes(E)` event captures the idea that some game event (bound to its pattern variable) happens at least once in a given trace. Building on this, the `ignored_move(E)` pattern describes the situation where an event that is supported by the game's rules is dynamically available to the player (possible) at least once in a trace while never observing the player selecting that action.

Running our system with *DrillBot* and these patterns yielded a report which described several instantiations of the ignored move pattern. The most commonly ignored moves involved the mining event, particularly non-treasure rocks at leaves of the map's navigation graph (such as the rock `d2` indicated in Figure 1).

A less common (but more interesting) instantiation of the ignored move parameter involved the `up_to(f)` event. What is so special about this move? It turns out the `f` cavern is an *emergent dead-end* when players leave the `c0` rock above it un-mined (because the robot cannot move into non-empty caverns). The nearby `e` cavern, despite being filled with rocks at the start of the game, is more often chosen by players as (1) it is filled with treasured rocks, (2) it is more connected to other caverns than `f`, and (3) it provides an equal length path to the deeper parts of map in comparison with the ignored `f` cavern.

Before having the system draw our attention to the `f` cavern's properties, we were previously un-aware of this type of emergent dead-end in *DrillBot*'s level design. In an iterative design process, we might intentionally create several such emergent dead-ends or even use a compiled pattern detector for these localized situations in conjunction with LUDOCORE's "structural query" feature to automatically solve for new level designs which include this pattern.

Future Work

Applying equally to humans and machines, the theory of rational curiosity suggests that we expand this creativity support system in two directions: further supporting human creativity and creating a software component that can be used in developing automated game design systems that are themselves creative.

Towards both of these goals, we would like to eliminate the need to directly formulate even these high-level pattern descriptors. Instead, we believe machine learning techniques can be adapted to translate a collection of manually assembled evidence sets into a most-likely pattern definition which can be used to collect and organize additional evidence sets or form a part of a higher-level pattern.

```
sometimes(E) <--
  primitive( event(E) ),
  primitive( timepoint(T) ),
  primitive( happens(E,T) ).

ignored_move(E) <--
  primitive( possible(E,T) ),
  pattern( \+ sometimes(E) ).
```

Figure 2. Two examples in our pattern definition language. Primitive terms refer to the presence of concrete elements in a complete gameplay trace and pattern terms refer to the presence (or absence in this case) of evidence for another design pattern.

Conclusion

Motivated by the theory of rational curiosity, this project has explored the idea that creativity support tools in game design should directly support gaining design knowledge. The system realized thus far has demonstrated the ability, in an automated manner, to direct a designer's attention to concrete instantiations of patterns of their interest and suggest subsequent patterns for future exploration.

Acknowledgements

This work was supported in part by the National Science Foundation, grant IIS-1048385. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Fullerton, T. 2008. *Game design workshop, 2nd Edition: a playcentric approach to creating innovative games,* Morgan Kaufmann.

Colton, S. 2008. Creativity versus the perception of creativity. Creative Intelligent Systems: Papers from the AAAI Spring Symposium, 14-20.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2009. Computational support for play testing game sketches. In *Proc. of the 5th Annual AI and Interactive Digital Entertainment Conference* (AIIDE2009).

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2010. LUDOCORE: a logical game engine for modeling videogames. In *Proc. of the 2010 IEEE Conference on Computational Intelligence and Games (CIG 2010)*.

Smith, A. M.; and Mateas, M. 2011. Knowledge-level creativity in game design. In *Proc. of the 2nd International Conference on Computational Creativity (ICCC 2011)*.

Yeap, Wai K.; Opas, Tommi; and Mahyar, Narges. 2010. On two desiderata for creativity support tools. In *Proc. of the Intl. Conference on Computational Creativity*, 180-189.