Automating the Design of Game Visualizations

Abhijeet Mohapatra Michael Genesereth
{abhijeet, genesereth}@stanford.edu
Stanford University

1 Introduction

In general game playing [5], game rules are communicated at *runtime*, and the players must be able to read and understand the descriptions they are given in order to play legally and effectively. Unlike specialized game players, such as Deep Blue, general game players cannot rely on algorithms designed in advance for specific games. General game playing expertise depends on intelligence on the part of the game player and not merely intelligence of the programmer of the game player.

In general game playing, games are defined in a formal language known as Game Description Language (GDL), which is similar to the logic programming languages PROLOG and Datalog. At runtime, the GDL description of a game is communicated to the game players. Since general game players are primarily computer programs which are adept at reading GDL description of games and playing them effectively, the general game playing community has not placed much emphasis on the game visualizations. However, recent General Game Playing competitions [4] feature a *Carbon versus Silicon* contest where a human player plays against the best general game playing program in the competition. Humans, unlike computer programs, need a visual medium to understand, strategize and play games. Hence, human players must be provided with a game visualization in addition to supplying them with the game description in GDL.

Currently, the need for a game visualization is being served by game authors having to specify a style sheet [1] along with the game rules. However, the visualization created by an author may not be ideal for a gameplayer. For example, suppose that the game of checkers in its standard visualization with red and black pieces is presented to color-blind people. In this case the game visualization is ineffective since the players cannot distinguish between red and black pieces and cannot, there-

fore, play the game. In addition, different visualizations may offer different perspectives to a player leading to potential development of new and unanticipated strategies.

In this article, we present a novel system, called *Merlin* that automates the design of an arbitrary game's visualization from its description. Merlin discovers underlying game concepts such as boards and pieces by computing *invariant projections* of game states, and automatically generates visualizations for these concepts. Merlin also allows different visualizations for the game concepts to be composed, thereby generating different, potentially new visualizations for games. This composition can be leveraged by game artists to incrementally improve their existing visualizations.

The organization of this articles is as follows. In Section 2, we present a general overview of Merlin, and present the technique used by Merlin to identify and characterize game concepts such as boards and pieces. In Section 3, we present different techniques to visualize the game concepts identified by Merlin, and compose these visualizations to generate different, potentially new visualizations. In Section 4, we present a case study on the application of Merlin to generate visualizations from the GDL description of Tic-Tac-Toe. In Section 5, we discuss related work and explore future directions.

2 Merlin: Identification of Game Concepts

In general game playing, games are finite state machines, with an initial state and one or more final (or terminal) states. Transitions between game states are specified as GDL rules. We assume that the reader is familiar with general game playing, Game Description Language (GDL), and propositional nets (propnets) [5].

As mentioned in the previous section, Merlin is a system that automatically generates designs for visualizing games. The GDL description of a game is supplied as an *input* to Merlin. Merlin generates visualizations for the supplied game in a two-step approach. The first step or the **extract phase** consists of identifying the *game concepts* i.e. boards and pieces and their properties (*board graph* and *mobility*). In the final step called the **render phase**, different canonical visualizations are generated for boards and pieces. This framework is depicted in Figure 1.

We now describe the extract phase in which game concepts (i.e. boards and pieces) are identified for a supplied game. In the next section, we discuss how canonical visualizations are generated for these game concepts.

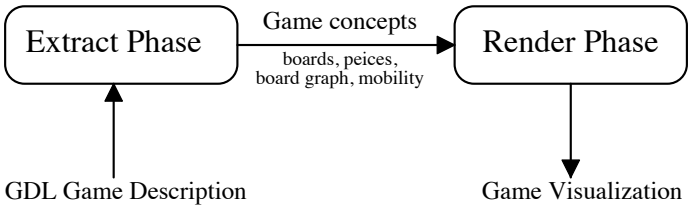


Figure 1: Merlin's Framework for Designing Game Visualizations

In general game playing, a game state is essentially a database relation. Therefore, a game could be conceptualized as active database. We use a remarkably simple yet powerful characterization over game states to identify game boards. Informally, a board is any component of the game that does not change with gameplay. We capture this informal characterization of boards formally using *invariant projections*.

Invariant Projections: Suppose that in a supplied game, that a k-ary database relation r is used to model the game states. We use $\pi_S(r)$ to denote the projection of the relation r on the set of attribute ordinals S. Suppose that the structure $\langle r, S\rangle$ is an invariant projection *iff* the following conditions hold.

1. S is a non-empty subset of $\{1,2,\ldots,k\}$.
2. Suppose that $I = \pi_S(r)$ in the initial state of the game. Then, for every reachable state of the game $\pi_S(r) = I$.

In our characterization, any invariant projection of a game is a board. We illustrate using the game called *Alquerque*. The GDL description of the game can be found in [1].

Example 2.1. In Alquerque, the game state is described using a ternary relation called *cell*. From the *init* and the *next* rules, the following invariant holds in every game state: for every combination of the first two arguments of cell, the value of the third argument is either blank, red or black. This result can be easily proved using induction, since it is true in the initial state of the game, and by reasoning over the possible legal actions over any intermediate state. Since $\langle \text{cell}, \{1,2\}$ is an invariant projection, the first two arguments of the relation cell constitute the game board.

We note that since $\langle \text{cell}, \{1,2\}$ is an invariant projection, $\langle \text{cell}, \{1\}$ and $\langle \text{cell}, \{2\}$ are also invariant projection, and therefore, boards as per our definition. Merlin's framework allows multiple

boards to be visualized differently leading to new, unanticipated game visualizations.

We also note that, as per our characterization of boards, there could be games without any invariant projections. In such games, either a board truly does not exist or a board is incrementally built with gameplay. In the latter case, the board positions, which are incrementally added, could alternatively be viewed as piece; this is the case with our characterization.

Pieces: Informally, pieces are dual of game boards. In formal terms, pieces are projections of game states that change with game play. If $\langle r, S \rangle$ is a board for a k-ary relation r, then $\langle r, T \rangle$ is a piece where $T = \{1, 2, \ldots, k\} \setminus S$. As with boards, there can be multiple possible sets of pieces or no pieces for a given game.

In addition to computing boards and pieces, Merlin computes two additional properties called *board graphs* and *mobility* in the extract phase.

Board graph: A board graph captures the relationship between board positions is captured in a game. Suppose that $B = \langle r, S \rangle$ is the game board. The board positions are tuples $\in$ B. . The board positions constitute the vertices of the board graph. If in the propnet representation of the game, the tuples corresponding the board positions serve as inputs of some view proposition, then these two board positions are connected by an edge. This edge is annotated by the relation constant that is used to denote the view. For example in the game of Alquerque, the views connecting the board positions are *doublet*, *triplet*, *horizontal*, *vertical*, *slash*, and *backslash*.

Mobility: The mobility of a piece models its degrees of freedom on the game board. Consider a snapshot of a game in the state say s. In this snapshot, suppose that m is the maximum number of board positions that a piece say p may occupy in the next state. The state-mobility of p for the state s is m. The mobility of p is defined as the maximum state mobility of p. For example, in the game of chess, the mobility of a pawn, king and rook is 3, 8, and 14 respectively.

The outputs of the extract phase i.e. boards, pieces, board graph, and piece mobility are fed into the render phase.

3 Visualizing Game Concepts

In its render phase, Merlin independently generates different canonical visualizations for boards and pieces. In this section, we first discuss the visualization of game boards, and then turn our attention to the visualization of pieces.

Visualization of game boards: Merlin visualizes a game board by generating different renderings of the board graph. The renderings of the game boards can be categorized into the following three categories.

- **Node-based**: This canonical class of renderings is generated by applying different layout algorithms [8] to render the board graph, e.g. spectral layout, grid layout. In addition, the number of crossings in the graph are minimized. In node-based renderings, the pieces are placed on the nodes of the graphs, and the edges of the graph represent relationship between board positions. This type of rendering is common in games such as Chinese Checkers, and Alquerque.

- **Face-based**: This canonical class of renderings is generated by constructing the dual graph of node-based renderings. Face-based renderings are common for games such as Chess, Skirmish, and Tic-Tac-Toe, and in these renderings the pieces are placed on the faces induced by the rendering of the board graph.

- **Card-based**: This is a new canonical class of renderings that we have introduced. Card-based renderings of graphs are based on the presentation of graphs as adjacency lists i.e. as individual nodes with their lists of neighbors. Suppose that a board graph has n vertices. Then, Merlin generates different card-based renderings of the supplied board graph using n cards, each representing vertex (or equivalently, a board position). The list of neighbors are represented in the cards, by assigning similar or dissimilar properties to different cards analogous to the rules used in the game of Set. We present an example of a card-based rendering in our case study in Section 4.

Visualization of game pieces: Merlin currently visualizes pieces as different regular polygons. For a supplied piece, the number of edges in its polygon rendering are determined by the piece's mobility. Suppose there are 3 pieces (p_1, p_2, and p_3) in increasing orders of mobility. Then Merlin visualizes piece p_1 as an equilateral triangle, p_2 as a square, and p_3 as a regular pentagon. Since, it is hard for humans to distinguish different n-sided polygons from each other where $n > 8$, we use a combination of shapes, colors, and size to represent pieces. We note that this is similar to the presentation technique used in [7]

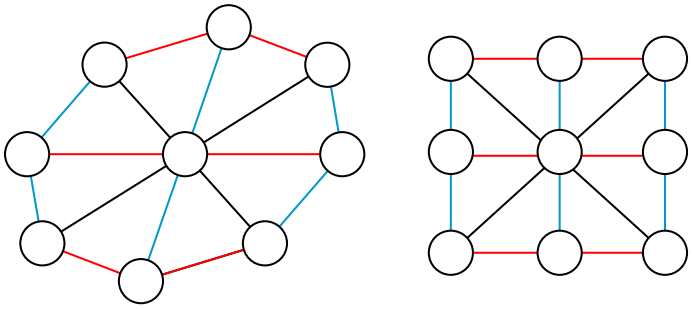
4 Case Study: Tic-Tac-Toe

In this section we evaluate Merlin by supplying as input the GDL description of Tic-Tac-Toe that is presented at [1]. In the GDL description of Tic-Tac-Toe, the game state is described using a ternary relation called *cell*.

Extract Phase: To compute the boards and pieces, Merlin first computed the invariant projections of Tic-Tac-Toe. Merlin determined that $\langle \text{cell}, \{1,2\} \rangle$ is the only maximal invariant projection and hence, the board. There are 9 different board positions: (1, 1), (1, 2), (1,3), (2,1), (2, 2), (2, 3), (3,1), (3, 2), and (3, 3). The propnet for Tic-Tac-Toe was then analyzed by Merlin to generate the board graph for the 9 different board positions. There are three different relation constants: *row*, *column*, and *diagonal* that correspond to views in Tic-Tac-Toe. These constants were used to annotate the edges connecting the 9 different board positions.

Merlin determined that there are three different pieces in Tic-Tac-Toe i.e. blank, *x* and *o* each with a mobility of 1.

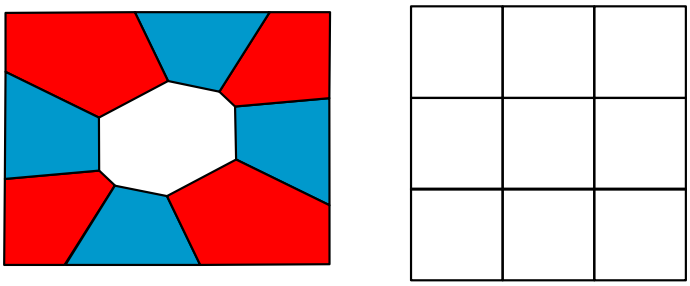
Render Phase: In the render phase, Merlin generated two node based and face-based renderings, and one card-based rendering. The two node-based renderings and their corresponding dual graphs i.e. the face-renderings were generated by Merlin by using the grid and the spectral layout algorithms respectively. The node-based and face-based renderings are shown in Figures 2 and 3 respectively.



(a) spectral layout (b) grid layout

Figure 2: Node Based Renderings of Tic-Tac-Toe's Board. The different edge annotations are color-coded: black, blue, and red for diagonal, col, and row respectively.

To generate the card-based rendering Merlin used common symbols to indicate the differ-



(b) spectral layout (b) grid layout

Figure 3: Face Based Renderings of Tic-Tac-Toe's Board. In the spectral layout, a player wins by placing 3 markers across any two consecutive red positions along the circumference, or across any of the 4 diagonal positions - either red or blue.

ent edges between board positions. Each symbol in this rendering has four properties: (a) shape $\in \{\heartsuit, \spadesuit, \clubsuit, \diamondsuit\}$, (b) number or quantity $\in \{1,2,3,4,5,\ldots,9\}$, (c) foreground color and (d) background color. This rendering is shown in Figure 4. In this card-based rendering, the players pick one card per turn. The act of picking up a card corresponds to marking a board position in the standard node-based or face-based rendering. A player wins the game by collecting three cards that have any of the 4 properties in common. We have published an online version of the card-based rendering of Tic-Tac-Toe as the game Trifecta [3].

5 Discussion of Related Work and Future Directions

The problem of visualization of static, database tables has been well studied in the database and artificial intelligence communities. The seminal work on the topic of visualizing database tables and queries is presented in [7]. However, the problem of automatically generating visualizations for discrete dynamic systems, notably games, has not received much attention in the artificial intelligence community. Prior work presented in [6] has proposed a technique to visualize games through abstract cell complexes. However, the technique presented in [6] relies on the use of relations with a specific schema to describe game states, e.g. a ternary relation *cell* is to describe the game state, where the first two arguments are ordered by a successor relation and constitute the board of the game. Furthermore, the technique presented in [6] assumes that there is at most one piece per board

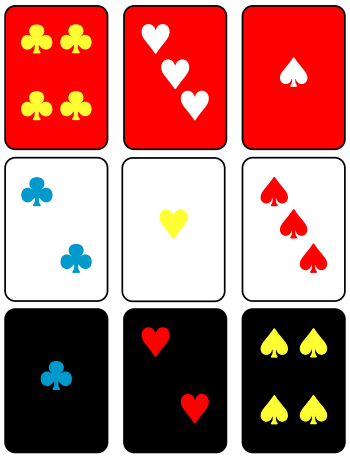


Figure 4: Card based rendering of Tic-Tac-Toe.

position which clearly does not hold in games such as Parcheesi [2]. In contrast, Merlin's visualization techniques neither makes any assumptions on degree of the relationship between boards and pieces, nor imposes any limitations on the schema of game states.

Customizing Visualizations: A game designer may customize the visualizations generated by Merlin by tweaking the following parameters.

- **Board Graph**: Currently Merlin, uses all view propositions to construct the board graph. However, a game designer may instruct Merlin to ignore certain visualizations, thereby generating different board visualizations. For example in Tic-Tac-Toe, a game designer may advice Merlin to ignore the view *diagonal*.

- **Visualization Parameters for Pieces and Cards**: Merlin uses three dimensions to represent pieces: shape, color, and size. The values of these dimensions default to n-sided regular polygons for shape, the 6 primary and secondary colors, white and black for color and a number between 1 to 10 to denote the relative size of pieces. Different values of these dimensions may be specified by a game designer to generate customized pieces.

- **Symbols used in Cards**: In the card based renderings, the set of neighbors of a card are encoded by assigning a symbol for each type of edge between two cards (or equivalently,

vertices in the board graph). A game designer may instruct Merlin to use different set of values for the four dimensions: shape, number, foreground and background color to generate different looking cards.

Visualizing Gameplay: Currently, Merlin uses only the game description to visualize the game. However, the game play could also be taken into account to visualize the dynamics of an instance of a game. For example, in chess, if the white player captures multiple black pieces using the white queen, the white queen could be made fatter to indicate the number of pieces capture. Alternatively, if the black player makes a move that reduces its odds for winning the game (perhaps, indicated by some heuristic), then the black pieces may be made more transparent to indicate that are in fact fading away.

References

[1] Gamemaster. `http://gamemaster.stanford.edu/`. Accessed: 2016-01-26.

[2] Parcheesi. `http://www.hasbro.com/common/instruct/Parcheesi.PDF`. Accessed: 2016-01-19.

[3] Trifecta. `http://stanford.edu/~abhijeet/trifecta/`. Accessed: 2016-01-26.

[4] M. R. Genesereth, N. Love, and B. Pell. General game playing: Overview of the aaai competition. *AI Magazine*, 26:62–72, 2005.

[5] M. R. Genesereth and M. Thielscher. *General Game Playing*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2014.

[6] I. Keller. Automatic generation of game component visualization. Master's thesis, Technische Universität Dresden, June 2010.

[7] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5:110–141, 1986.

[8] R. Tamassia. *Handbook of Graph Drawing and Visualization*. Discrete Mathematics and Its Applications. Chapman and Hall/CRC, 2013.