EGGG: Automated programming for game generation

by J. Orwant

EGGG, the Extensible Graphical Game Generator, is an experiment in automated programming. By concentrating on a particular domain—games—EGGG allows users to create applications with a minimum of programming effort. We codified the similarities among games and game programs into reusable software components that decouple the rules of a game from its implementation. As a consequence, users can create games merely by describing the rules to EGGG, which then generates a fully functioning game program. In this paper, we survey the design and implementation of EGGG and provide some examples of games that can be created with the system.

Programming is hard. But programming for a particular domain need not be—assumptions about the domain can be built into the language and into whatever system (compiler, translator, interpreter, or combination of all three) ultimately turns the program into an executable application.

Games have several advantages as a domain for automated program generation. They have the right amount of diversity (not too little, not too much); many games can be easily represented with a small set of rules; and the generated programs need not be perfect to be usable.

Classic games (poker, chess, tic-tac-toe, rock-paper-scissors, and so on) are much more alike than one might imagine, and these similarities were used to create a universal game engine called EGGG, the Extensible Graphical Game Generator. EGGG is a program that generates programs: designers provide it with the rules of a game, and the rules are rendered into an actual computer game ready for play.

The Atari 2600 system revolutionized the game industry in 1978 because of cartridges; previous systems (with the exception of the Fairchild Channel F) were able to play only a static set of games. The decoupling of hardware and software was made possible by simpler and more flexible hardware components. We take the decoupling further by creating simpler and more flexible software components. Instead of decoupling hardware from software, EGGG decouples a game's implementation (the "hard software") from the rules of play (the "soft software").

EGGG uses a high-level language that lets designers describe games in as few words as possible, while still retaining the precision that the underlying engine needs to render the language. The language is expressive (users can create almost any kind of graphical two-dimensional game) and concise (statements are short and powerful; debugging is easy because users can see the entire game on one page).

The design criteria for the EGGG language and engine follow, in decreasing order of importance:

- Game descriptions should be brief.
- Easy games should be easy to generate, and hard games should be possible to generate.
- The EGGG engine should contain as little *a priori* information about particular games as possible.

0018-8670/00/$5.00 © 2000 IBM

- It should be easy to create variations.
- The EGGG engine and the games that it generates should be portable across platforms.
- The games generated by EGGG should be easy to modify.
- EGGG should not take a long time to generate games, and the games that it does generate should not run so slowly that playability is affected.

Game categories and descriptions

A taxonomy of games was developed for EGGG and is described in detail elsewhere.[1] Unlike most popular game categorizations that focus on the structure of the game, or game theoretic categorizations that focus on information and probability, we focus on what matters to a game developer: *process*. We classify games according to the following attributes:

1. Frenetics—whether the game requires quick action, or is timed (used to determine how [and whether] EGGG generates a pause feature for the game)
2. History—how many player actions need to be stored to play the game
3. Synchrony—whether players move all at once, or in order, and if so what that order is
4. Movement—what players do (divided into moves, phases, turns, rounds, and steps)
5. Topology—the shape of the game (e.g., rectangle, sphere, or "zero-dimensional" games like interactive text-based games)
6. Board—the playing surface (grid, graph, canvas, or none at all)
7. Pieces—the attributes of items on the board, whether they have state, color, or other artistic meaning, and how they are grouped
8. Compartments—whether there are occluding barriers that prevent players from seeing information possessed by other players, or by the game itself
9. Genre—whether the game has a theme, and how important that theme is to the play of the game
10. Information—what communication occurs between players during game play, whether the communication is intentional or inadvertent, and whether the communication is intended to help or confuse
11. Referees—whether the game has an umpire, or just competitors
12. Endings—what determines when the game is over (not necessarily the same as a player's goal)

An example game description: Poker. To use EGGG, the first step is to create a game description—typically a page or half-page of text provided as a file with an .egg extension. Here is the game description for poker:

```
game is poker
turns alternate clockwise
Discard means player removes 0..3 cards or 4 cards
  if Ace
Fold means player loses
2..6 players
game is Shuffle(deck) and Deal(cards, 5) and (bet(money)
  or Fold) and Discard(hand, N) and Deal(cards, 5−N)
  and (bet(money) or Fold) and compare(cards)
StraightFlush is (R, S) and (R−1, S) and (R−2, S) and
  (R−3, S) and (R−4, S)
FourKind is (R, s) and (R, s) and (R, s) and (R, s)
FullHouse is (R, s) and (R, s) and (R, s) and (Q, s) and
  (Q, s)
Flush is (r, S) and (r, S) and (r, S) and (r, S) and (r, S)
Straight is (R, s) and (R−1, s) and (R−2, s) and (R−3,
  s) and (R−4, s)
ThreeKind is (R, s) and (R, s) and (R, s)
TwoPair is (R, s) and (R, s) and (Q, s) and (Q, s)
Pair is (R, s) and (R, s)
HighCard is (R, s)
hands are [StraightFlush, FourKind, FullHouse, Flush,
  Straight, ThreeKind, TwoPair, Pair, HighCard]
hand is five cards
goal is highest(hand)
```

Game descriptions are a series of statements, which can appear in any order. The EGGG language intentionally has no flow control; it is a description and not a program. Statements can be split across multiple lines, as long as lines after the first are indented. Comments can appear anywhere, and begin with #.

These 17 lines are all that is required for EGGG to produce a fully functional poker program. Tic-tac-toe and rock-paper-scissors each require only eight; chess requires 85. A screenshot of our poker game is shown in Figure 1.

A sample heuristic: How does EGGG know that poker is a game of rounds? Participants play many rounds of rummy at a sitting, but not many rounds of chess. EGGG needs to know whether poker is more like rummy or chess; in particular, it needs to know whether to maintain state between games. Without knowing that poker consists of rounds, EGGG would display GAME OVER at the end of a poker game, and

Figure 1 A poker game generated by EGGG

empty the player's coffers if the player chose to play again.

When EGGG creates a game, it employs several dozen heuristics that embody the similarities between games. As a sample heuristic, we examine the rule that EGGG uses to decide that poker is a game of rounds. The heuristic is that a game is *not* played in rounds if it fulfills any of these conditions:

- The game has levels (as in arcade games where a succession of foes need to be defeated).
- The game has a particular solution (as in crosswords, logic puzzles, or number puzzles).
- The game has no hands.
- The game has hands, but the hands contain more than 13 pieces per side.
- The game is played on a grid with more than 25 squares.

A new game: Deducto. In this section, we describe an entirely new game created with EGGG. The new game is called *Deducto*, and is a logic game played on a 5×5 square grid. The game is described in 42 lines of EGGG, and a sample game board is shown in Figure 2.

Each of the 25 squares is either black or white. We can see from the label above and to the right of the squares that this game has levels; each level has a secret rule, and the player's goal is to determine the rule. The rule for Level 1 is simple: at least half the squares in the grid must be white.

When the player presses the Example button, the game generates a random grid satisfying the rule. If a player then selects one of the grid squares, it changes color (from black to white, or from white to black). Thus, a player can take a compliant grid—one that satisfies the current rule—and make it non-compliant, or vice versa. The player can then test whether the board satisfied the rule by pressing the Test button.

When the player thinks he or she understands the rule, the player presses Understand. The game then generates either a board that satisfies the rule or a board that does not, each with probability 50 percent. The board is displayed, and the player is prompted to Vote Yes if the grid fits the player's understanding of the pattern, or Vote No otherwise. When the player guesses correctly five times in a row, he or she moves on to the next level.

Deducto highlights one limitation of EGGG: the language required to express the rule behind each level. The rule might be simple, like counting the white squares at each level, or it might be complex. Maybe the number of white squares at each level has to be a prime; maybe the white squares have to form a letter, or a face; maybe the squares have to encode the current time. Here is the rule that tests whether a grid satisfies Level 1. This demonstrates how EGGG lets game designers burrow down into its implementation language: Perl.

```
Tester(1) is Tot++ if grid[x][y]; return Tot > 12
```

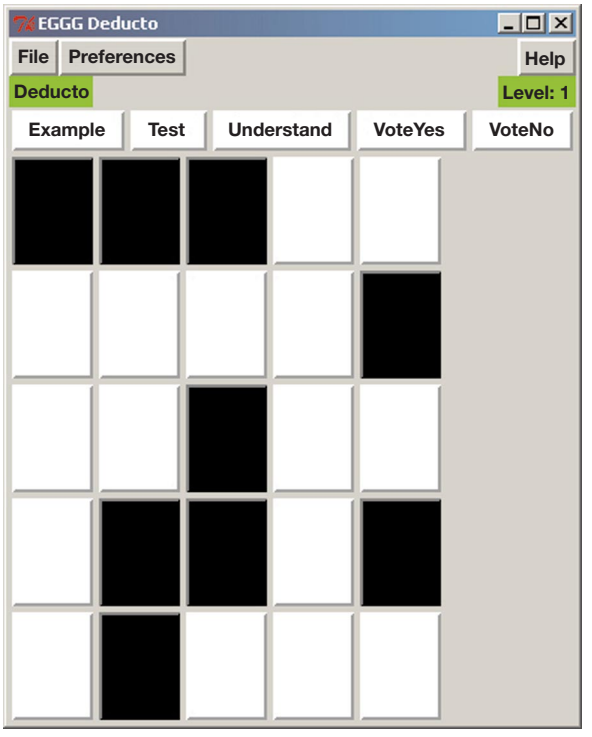
Most EGGG statements are simple declarations like goal is maximize(level), but statements like Tester(1) show how designers can express the programmatic behavior that some games require. Here are two more statements for Deducto:

```
assert VoteYes: lastmove("Understand")
assert VoteNo: lastmove("Understand")
```

EGGG allows game designers to create actions that will be triggered at particular times. These two assertions are conditions that must be satisfied for the VoteYes and VoteNo subroutines to be invoked. If the player's last move was not to press the Understand button, nothing will happen when the user presses either VoteYes or VoteNo. That is to prevent users from cheating; otherwise, they could use the Test button to learn whether the grid matches before voting yes or no.

Variations. EGGG makes it easy to create variations on games. Game designers can copy game descriptions from EGGG's central repository and modify them, or they can take the game description and create a new game description with a rule like RandomChess is like Chess, which fetches the rules for chess from EGGG's repository and uses them as a starting point for modifications.

Figure 2 Deducto, a new game created with EGGG



For the October 1998 News in the Future consortium meeting at the MIT Media Laboratory, a sequence of five variations on the Tetris** arcade game was demonstrated, each building on the last.

The game was turned on its side, so that pieces went from left to right instead of from top to bottom:

```
SideTetris is like Tetris
```

The game was changed from one player to two, so that the "serving" player could choose the piece:

```
TwoPlayerTetris is like SideTetris
```

Players were then allowed to bounce pieces back to the other side:

```
BounceTris is like TwoPlayerTetris
```

The game was made faster:

```
BounceFast is like BounceTris
```

The pieces were all made square:

```
SquareBounce is like BounceFast
```

The top and bottom edges were made less sticky, so that pieces bounced off:

```
Pong is like SquareBounce
```

The result is Pong, the classic video game.

Other work

In this section, we describe two systems that inspired EGGG: the Programmer's Apprentice and METAGAME.

The Programmer's Apprentice. The Programmer's Apprentice is an automated programming effort headed by Charles Rich and Richard Waters of the MIT Artificial Intelligence Laboratory.[2] The project yielded a series of intelligent assistants for software engineers: programs that helped programmers formalize requirements, create and edit programs, and analyze the programs they created.

EGGG differs from the Programmer's Apprentice in that its goal is simultaneously both more and less ambitious. EGGG is more ambitious in the sense that it is an attempt to create something that people with a minimum of programming skill can use: its target audience is not programmers, but people who want to create computer games—people who may not have much (or any) programming experience. EGGG is less ambitious in that it drastically constrains what can be created. EGGG does not do requirements analysis or program verification; it simply generates games.

METAGAME. METAGAME[3] is similar in spirit to EGGG, but arose from a different premise. For a long time, chess was considered a formidable problem for AI (artificial intelligence). The reasoning was that any program that could beat an expert player, or even a competent one, would possess some of the intellectual capacity of a human. Yet IBM's Deep Blue and other successful chess programs demonstrate that this is not necessarily the case: chess programs have gotten better largely through improvements in computer power and in "hardwiring" knowledge of chess into the architecture. Put another way, chess is not AI-complete: just because a program is an intelligent chess player does not mean that it is intelligent.

Barney Pell's METAGAME strives to bring AI into computer game playing. He suggests that the test of a computer's game-playing prowess should not be chess, or even Go, but the ability to play a game with no *a priori* knowledge. To this end, Pell develops a formalism for representing "symmetric chess-like" (SCL) games, including a proof that all finite two-player games of perfect information (that is, games that can be represented as game trees) can be represented using his formalism. A computer program that can play arbitrary SCL games is called a *metagame player*.

Pell goes on to develop an evaluation mechanism for metagame players and a program that generates games that are similar to chess, but with random rules. He then tests his METAGAMER program against several such games and reports the results.

METAGAME is a mathematical framework for developing programs that can play games well, in contrast to EGGG, which is a system for developing games. Nevertheless, METAGAME has influenced the design of EGGG in a few ways. In particular, EGGG uses logical predicates that represent changes in the state of game play. Quoting Pell,[4] page 104:

> In addition, these predicates are all *logical*, in that state is represented as a relation between two variables, StateIn and StateOut, instead of a global structure which is changed by side effects (as in a *current board* array used in many traditional playing programs). This enables a program to use the predicates in the domain theory in both directions. For example, by constraining SOut in Figure 12.2 instead of SIn, a program can determine possible predecessor states, thus using the rules "in reverse" to find all the positions which would have been legal before a given move.

EGGG has global structures that are changed by side effects, but it also has "actions" that are passed around, composed, concatenated, and hypothesized about, just like METAGAME.

Implementation

The games that EGGG generates are Perl programs, and EGGG itself is written in Perl. Perl is ideal for several reasons: its portability means that the games will work on all major platforms without needing to change or compile either the engine or its generated games. It is fast enough for arcade games; its support for regular expressions makes parsing the game

descriptions efficient; its built-in hash tables are ideal for representing nested data structures; and it has powerful and free mechanisms for networking, persistent storage, and database support.

Data structures. Every game generated by EGGG has two umbrella data structures: the %game and %state hash tables. (In Perl, a % denotes a hash table.) The first contains the permanent aspects of game play. The contents of %game vary from game to game, typically including:

- $game{type}, the type of the board. (In Perl, a $ denotes a single value.)
- $game{sides}, the dimensions of the board
- $game{board_start}, the initial board configuration
- $game{num_players}, the minimum and maximum number of players
- $game{synchrony}, the synchronization of the game

These are all attributes that remain constant regardless of who is playing the game or how it is being played. For instance, in a crossword puzzle, the arrangement of the black and white squares is part of the %game data structure ($game{board}, in particular).

In contrast, the actual letters that the player has written are stored in $state{board}. The %state data structure complements the %game data structure; it is a hash table containing the ephemeral aspects of game play such as these:

- $state{board}, what is on the board at the moment
- $state{player1}{name}, $state{player2}{name}, . . . , the names of the current players
- $state{turn}, whose turn it is

Documentation. While EGGG has no deep understanding of the games it generates, it is able to document them. EGGG generates both documentation for the developer (comments in the Perl programs that it creates), and documentation for the player (how the game is played).

When EGGG parses a game description, it ignores case and number wherever possible. For instance, players are white and black is interpreted identically to players are black and white. This is ostensibly to make designing games easier, but it has another advantage: by letting game designers express their rules grammatically, EGGG can transform the rules into instructions without a deep understanding of what the rules mean. For instance, this line would be translated into the English sentence "The two players are black and white," which is one of many sentences players will read when they select Instructions from the game's pull-down menu.

Computer opponents

Computer programs that play two-player deterministic games (such as chess, tic-tac-toe, Reversi, or Go) typically consist of three components: a minimax procedure, a static evaluator, and a library of precalculated moves. For such games, EGGG generates a computer opponent for humans to play.

A generic minimax procedure. EGGG has a generalized minimax procedure with no *a priori* knowledge about a particular game. It requires only the %game and %state data structures, and returns the optimal move for the current player. It includes an enumerate_moves( ) subroutine, which iterates through all of the player's pieces, identifying all the possible moves for each, and eliminating any moves deemed illegal by the game description.

A generic static evaluator. Using the following procedure, EGGG generates a score_ board( ) subroutine (the static evaluator):

- Set the board score to zero.
- If the goal is *binary* or *trinary*—that is, players either win, lose, or draw—loop through all the pieces on the board as follows:

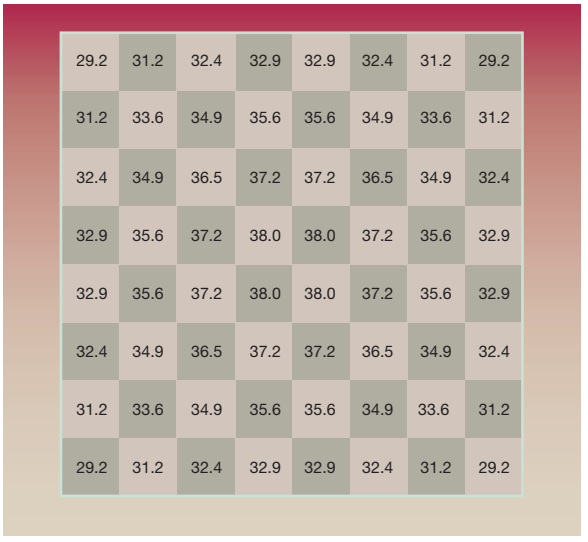
  1. Set piece_score equal to the *power* of the piece. (Power is defined in the next section.)

     This step ensures that pieces with no moves and no surrounding empty squares are not ignored. Because we are looping through all the pieces, this rule rewards boards in which the player has more pieces and the opponent has less, so capturing is favored.

  2. Increment piece_score by the power of the piece times the nth root of the number of moves it has available, where n is the dimension of the board, or the density of the board if it is represented as a graph.

  3. For each location on the board, increment piece_score by the power of the piece divided by the nth root of the distance of that location from the piece.

Figure 3 Weights assigned to chessboard squares



This rewards pieces played in the center of the board, which is good for chess but bad for Go. On a standard 8×8 chessboard, this weights the squares as shown in Figure 3 (assuming unit power).

- If the goal is *comparative*—that is, the player's goal is to maximize (or minimize) some quantity, like points or dollars—the game description might supply a direct means of determining how many points a board is worth. If so, that is used; otherwise, the piece power is replaced by its expected value (described below) if it can be calculated. The pieces are then looped through as above.

This static evaluator will perform quite badly in comparison to one hand-tuned for a particular game. However, it can be dropped into any game, and in most games it will be better than nothing.

Estimating piece power. To score a board, the static evaluator uses an estimate of the value of a piece, called the *power*. The power of a piece is computed as *motility* times *rank* times *rarity*.

Estimating piece motility. The static evaluator heuristic uses the number of moves available to a particular piece in some situation—that is, the piece in a particular %state. Here, we consider the number of moves available to a piece in *all* situations.

EGGG examines the rules involving the piece motion in the game description file. For each piece, the rank is calculated by setting $game{*piece*}{rank} to 0, then extracting all the rules involving motion of that piece and looping through them:

1. Increment $game{*piece*}{rank} by the length of the rule.

   This step might seem arbitrary at first glance, since longer rules count more toward motility, regardless of what the rule means. A very long rule—say, one that applies only in unusual conditions and hence needs many expressions to describe—will add linearly to the piece motility in spite of the rarity of the situation it describes. This is compensated in part by the next step, which has much greater potential to affect the rank.

   In a game like Stratego, this first step has the effect of weighting immobile pieces like flags and bombs more heavily than mobile pieces; this heuristic is really more about estimating the *importance* of a piece than estimating its motility. For chess, this step weights the king more heavily because of the rules describing checkmate and castling. One can make the argument that if there are lengthy rules describing what a piece can do, it is more likely that the piece is important. Whether this is a good assumption for games is very much open for debate, although it does work well for the subset of games used to test EGGG.

2. Increment $game{*piece*}{rank} by the dimension of the board (or density of the graph) raised to the power of itself, minus the board dimension (or density) raised to number of dimensions that are constrained in the rule describing piece motion.

3. Divide $game{*piece*}{rank} by the square of the number of times the piece occurs in the starting board configuration.

Finally, the ranks are normalized so that the lowest ranking piece has rank 1. The result, applied to the chess.egg game description, follows:

```
$game{King}{rank}   = 23.6277551725428;
$game{Queen}{rank}  = 15.2191074823911;
$game{Rook}{rank}   = 3.95340184744318;
$game{Knight}{rank} = 3.11256637988997;
$game{Bishop}{rank} = 2.72902739469025;
$game{Pawn}{rank}   = 1;
```

In comparison, chess books for beginners typically rank the pieces as follows:

```
King   = infinity
Queen  = 9
Rook   = 5
Knight = 3
Bishop = 3
Pawn   = 1
```

Estimating piece rank. A piece has *rank* if it can capture pieces that cannot capture it, or if it has a higher point value than another piece (such as in playing cards: a jack is higher than a ten, a ten is higher than a nine, and so on).

The rank of a piece may be hard or impossible to calculate if the piecewise comparisons are intransitive. Rank is impossible to calculate in rock-paper-scissors, since the three "pieces" form a cycle: rock beats scissors, scissors beats paper, and paper beats rock; here EGGG gives each piece rank 1. In contrast, consider Stratego, in which the pieces form a strict military hierarchy: the marshal beats the general, the general beats the colonel, and so on down to the scouts, which beat the lowly spy. But the spy can beat the marshal. It might seem that no ranking makes sense. However, there is only one spy and one marshal, but there are eight scouts, each of which can capture the spy. The spy has eight ways to be captured, and the marshal only has one, so EGGG ranks the marshal higher than the spy—and from there, the rest of the piece rankings follow.

Estimating piece rarity. When the pieces are part of a bag (in the mathematical sense: a set, but allowing duplicates), the rarity is the size of the bag divided by the number of times the given piece occurs in the bag. In Scrabble, letters J and Q and Z occur only once; they each have a rarity of 100, because there are 100 tiles in a Scrabble set. In chess, the king and queen have rarities of 16; rooks, bishops, and knights have rarities of 8; and pawns have rarities of 2.

Estimating piece expected value. In some comparative games, the power of a piece can be calculated directly. The possible outcomes are enumerated and the ranking of each established; the expected value of the piece is then the sum of the scores where the piece is on the board, divided by the sum of the scores of all possible boards.

For instance, it is clear that a king is worth more than a queen in poker, but it is not clear *how much more* it is worth. Similarly, an ace is worth more than a king in poker—and the difference between an ace and a king is greater than the difference between a king and a queen, because the ace can be used in more hands (low straights as well as high). When all possible outcomes of the game can be enumerated, as they can be in poker, EGGG can directly calculate how much each piece is worth without relying on the more *ad hoc* heuristic used to estimate piece motility.

A generic library of opening moves. When run in a "public mode," EGGG games communicate game results back to the central repository. EGGG uses this *global history* (the game history of all sessions played by anyone on the Internet, as opposed to the *local history*, the series of moves during one game session) to amass a library of opening moves. Every night, EGGG follows these steps:

1. Store all of the game names in a @games array.
2. Remove a game from @games.
3. Extract the opening moves of the game into @array.
4. Sort @array by frequency of occurrence.
5. Pop the first sorted move off @array.
6. If there are no more moves, move down the game tree one level (from the opening moves to second moves, or from second moves to third moves), extract the moves into @array, and go to Step 4.
7. Calculate the chi-square value for two hypotheses: that the move wins more often than it loses, and that it loses more often than it wins.
8. If either move is statistically significant at the 0.05 level, store it.
9. If 1024 significant moves have been found for the game, go to Step 2.
10. If the entire game has been searched, go to Step 2.
11. If the amount of time spent evaluating the results exceeds the number of games in EGGG's repository, divided into six hours, go to Step 2.

Strategies. In this section, we discuss techniques that EGGG employs to predict what human players will do, or what they are thinking. Some of this work derives from the techniques used in DOPPELGÄNGER[5] to identify an individual's taste in news. In that domain, the problem is one of incomplete information: given only a few bits of information about a reader of known gender (say, which articles he read, or when he last read his newspaper), how can the system deduce why he read those articles, or when he will next

Figure 4 Rock-paper-scissors, a seemingly trivial game

read his newspaper? This can be viewed as a game of incomplete information, and EGGG uses the same model to make its assertions about players as DOPPELGÄNGER did about newspaper readers.

Analyzing strategies with hidden Markov models. Given a discrete series of symbols, and a series of mathematical models that generate symbols, a hidden Markov model technique allows a system to determine what model is most likely to be generating those symbols. For our purposes, the "symbols" are player moves, and the "models" are player strategies. EGGG's goal is to determine the player's strategy given his or her moves, and it uses a hidden Markov model algorithm[6] (in particular, the Viterbi lattice algorithm) to make that determination.

About a dozen simple strategies have been included in EGGG: simple functions that, given a history of player moves, return what the next move should be for a given player. EGGG's implementation of the Viterbi algorithm accepts an arbitrarily large set of these functions along with the game history, and then ranks the functions according to their success in predicting those histories. That is, given the first move, how well did each strategy predict the second move? Given the first two moves, how well did each strategy predict the third move? The results allow EGGG to infer what (if any) strategy a player is using, without requiring that the player's moves rigidly adhere to the strategy. The Viterbi algorithm is also tolerant of players who switch strategies midgame.

Consider rock-paper-scissors, which is a simple game with only three kinds of moves. There would seem to be *no possible successful strategy* for winning at what appears to be a game of pure chance.

Here is the game description for rock-paper-scissors:

```
move is choose
pieces are Rock and Paper and Scissors
board starts [[Rock, Paper, Scissors]]
turns synchronize
Beat means player(Rock) and opponent(Scissors) or
   player(Scissors) and opponent(Paper) or player(Paper)
   and opponent(Rock)
goal is Beat
score increments
3 × 1 grid
```

A screen image of the game is shown in Figure 4. The player presses either rock (the fist), paper (the open hand), or scissors (the two fingers), and the computer opponent reveals what it picked. The trick to winning this game is predicting what your opponent is going to do, and choosing the one symbol that beats it. In a session of multiple rounds, EGGG records the move histories and attempts to infer the player's strategy. The longer the history, the more information EGGG has to reveal the strategy, and the better its predictions will be.

The most common strategy in rock-paper-scissors is to aim for a draw over the long term: choosing ran-

domly so as to reveal no information to EGGG. Were players to *actually* choose randomly, this strategy would be successful at accomplishing its modest goal. Unfortunately (for the players) guessing randomly is harder than it might first appear. As an exercise, write down a sequence of ten instances of R, P, or S, simulating how you would play EGGG if you were trying to guess randomly. The strings generated probably look like this:

RSPPRSRPSSP
SPRPRSRPPRS
SSRPPRSRPSR
RPSSRPRPPSS

Those look pretty random. Here are some strings generated at random with the help of a computer.

PRRSRSPRPRS
RRPRRRPRPRS
PSSPRRSSSRR
PRSPPRPRRPP
RPSPPSSPRSP
RPPPSSRSSPS
PSSPSPRPRRR
SRRPRPSRSSR
SSRRRPRSPSR
RPSRPRPSSSP
SSPRSPSRSRP

(This was the first and only run of the program.) Note that six of the ten truly random strings have runs of three: RRR, or PPP, or SSS; one would expect 56.4 percent of randomly generated strings of ten symbols to have a run of three. None of the seemingly random strings has runs of three. This is the key to the strategy: when players try to guess randomly, they usually do so in a predictable way, and EGGG's computer-generated opponent exploits this as it plays.

Generating hypotheses with the chi-square test and beta distribution. In a game like rock-paper-scissors, it might be the case that a player has an inherent bias toward a certain piece. Perhaps he or she is trying to be random but really is not, or perhaps he or she is following some strategy that is more likely to choose one piece than another. If someone plays rock-paper-scissors for nine rounds and chooses rock five times, paper twice, and scissors twice, is it fair to assume that the player favors rock? Or are nine rounds too little to make that conclusion? Nine rounds *is* too little; we know this because of the chi-square test from statistics. EGGG makes simple hypotheses of this sort and tests them with Perl's Statistics::ChiSquare module (written by the author) to establish a confidence level for the hypothesis. (If a 5:2:2 ratio between the three choices is observed, the player would need to play 43 rounds for the hypothesis to be statistically significant at the $p = 0.5$ level.)

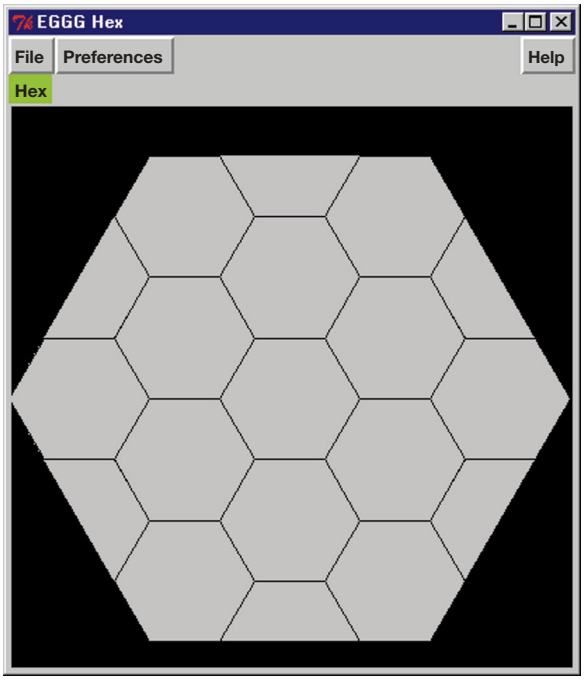
If a player beats EGGG at chess, that suggests that the player is better than EGGG at chess. But what does that tell us about the player's innate ability? Less than it would seem. The commonly accepted Elo[7] chess ranking system assumes that a player's performance can be described by a normal distribution, but as Beasley[8] points out, this is almost certainly not the case, and the research supporting this contention is specious.

Some EGGG strategies rely on an assumption that the player has less than (or more than) a particular degree of skill. It is one thing to say that our best estimate for the player's skill is 0.7; how can we estimate the likelihood that the player's skill is less than 0.8? The *beta distribution* is used to estimate the player's skill, allowing EGGG to establish confidence levels for estimates other than the most likely. The system maintains the entire play history, and weights more recent outcomes more heavily to account for improvements in the play of both the player and EGGG. The beta distribution yields both an assertion about strength and a confidence that the strength is accurate. The strength is easy to calculate: just the mean of the distribution, which is trivially the number of wins divided by the number of wins plus the number of losses. The confidence is defined as the inverse of the variance. For instance, if a player plays EGGG ten times and wins seven times, the strength is $7/10 = 0.7$, and the confidence is $1100/21 = 52.381$.

Making interesting moves and probabilistic bluffing. When the confidence levels for all its strategies is below a particular threshold, EGGG has little information about how to play; every opportunity seems equally attractive. EGGG sorts the available moves from most to least attractive based on the static evaluator output. Early versions of its computer opponents would simply choose the first move on that list, without regard to whether subsequent moves were equally attractive. The result was dull play: given a game state, EGGG would always make the same exact move. So what EGGG does is to choose randomly, but to make the probability of each move proportional to its attractiveness.

The proportionality is not linear. If two moves are available, one with a score of 10 and the other with a score of 1, it would be folly to choose the worse

Figure 5 A hexagonal game board



move one-tenth as often as the better move. So in the absence of other information, EGGG squares the distribution when choosing its moves. The move with the score of 10 will then be chosen 100 times as frequently as the move with the score of 1.

This framework for making interesting moves helps computer opponents play *more interesting* chess, but it can also help them play *better* poker. We generalized the notion of applying a weighted distribution to our move choice, and developed it into a generic bluffing strategy. Bluffing is only possible in games of hard-to-calculate known odds, as opposed to games of unknown odds (e.g., sporting contests) and easy-to-calculate known odds (e.g., blackjack).

Bluffing is tantamount to flattening the weighted distribution. This lessens the disparities between the relative attractiveness of different moves, making high-scoring moves less attractive, and low-scoring moves more attractive. This makes it less likely that EGGG will bet the "ideal" amount.

The extent to which EGGG considers nonoptimal betting amounts is determined by a state variable, $state{bluff}. The higher the value, the more EGGG's bets will vary. The procedure is as follows: EGGG takes the squared mass function described in the previous section, and raises each value to the power of 1/$state{bluff}. In betting games with no knowledge about the players, $state{bluff} begins at two, so the dampening is exactly equivalent to a square root, and we end up undoing the squaring that the last section described. EGGG maintains a separate $state{bluff} for each player (the actual values are stored as $state{*player*}{bluff}) and uses a gradient descent algorithm to adjust that value over the course of the game, and from game to game, to choose the bluffing strategy most successful against the player.

Garnering trust. In a game of complete information like chess or tic-tac-toe, cheating is impossible. In games of partial information, like poker or Scrabble**, cheating is possible, but human players tend to trust that computer opponents will not cheat. In games of zero information, like rock-paper-scissors, the player presses a symbol and is immediately greeted with "You lost!" or "You won!" or "Draw." How can the player verify that the computer did not cheat?

EGGG computes a "message digest" of a few random words and numbers concatenated with the message text.[9] The result computed for the rock-paper-scissors screen image shown in Figure 3 is:

```
Rocky has moved. Proof:
  6ec22e8d46e98df39a08e3dd1afe2917
```

The string of hexadecimal digits is the message digest. The message digest algorithm is one way; given that string of digits, it is computationally intractable to uncover the message that was digested. If the player selects Verify, EGGG will reveal the message, ensuring that it had already made its selection before the player selected his or her move.

Graphic layout

EGGG stores the display-dependent aspects of the game in an object named %display, which contains attributes such as $display{paused}, $display{iconified}, $display{height}, $display{color_depth}, and $display{granularity}. This object can be saved to disk for persistence: most games can be stored and reloaded at a later time.

The displayed board can be a variety of shapes, including any equilateral polygon. EGGG determines the vertices of the polygon, and by default the ver-

tices are aligned so that an edge is parallel with the bottom of the screen, as shown in Figure 5. This was done because many games that have polygonal boards expect each player to "own" a side.

A grid statement can be used to generate a variety of patterns on the board. Figure 4 shows the board created by these three statements:

```
board is hexagon
squares are hexagons
5 × 5 grid
```

Note that the "5 × 5" does not refer to *x* and *y* axes as it does in rectangular boards; new axes are chosen parallel to two adjacent sides of the polygon. This result is sometimes surprising, because a 5 × 5 grid will not have 25 squares when the board shape is a hexagon. As you can see from the diagram, there are actually 13 squares (and six half-squares).

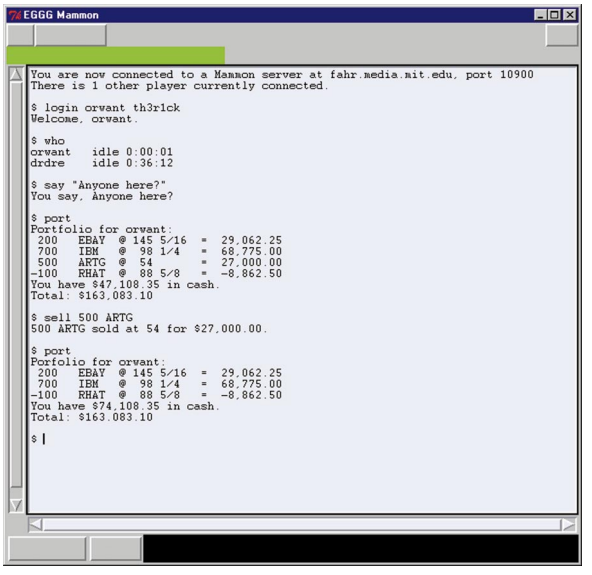
EGGG uses heuristics[1] to determine when to "checker" game boards (so that alternating squares are different colors) and to thicken particular grid lines to better delineate regions of the board.

Multiplayer games. EGGG is able to generate three types of multiplayer games: those with no spatial meaning, like networked text games; games where each player has his or her own side of a grid or canvas (chess, Chinese checkers); and games where the orientation is fixed (Scrabble, Monopoly**). If the game is designed for a particular fixed number of players, EGGG will usually be able to render it, generating all the necessary networking code and identifying how to rotate the board (if there is one) from client to client.

EGGG can also generate TCP/IP networking code to create games that can act as both a server and a client, as in a networked multiplayer text game. Typical interactive chat servers take thousands of lines of code, but most of that code is the same no matter what the server does. EGGG requires game designers to specify only what makes their server different from the most basic server; all the rest is generated automatically.

To illustrate a networked multiplayer text game, we examine Mammon, a stock-picking game where players log in to a dedicated server, buy and sell stocks using play money, and chat with one another. Mammon (in its non-EGGG incarnation) was the first Internet stock-picking game, developed by the author in July 1994 and at one time supporting over six thousand users.

Figure 6 Mammon, a networked chat game



The first statements of mammon.egg are the following:

```
networked multiplayer text game
port 10900
players are 0..20
```

Communication between the server and client will take place over port 10900 (port numbers are recorded in EGGG's central repository) and the server can accept up to 20 simultaneous connections. These three statements generate a chat server where players can send messages to one another. A screen image of Mammon (which depends upon a 63-line game description) is shown in Figure 6.

Conclusion

By exploiting the similarities among games and game programs, we were able to create EGGG, a system that decouples the rules of a game from its implementation. A designer provides a brief description of a game, and EGGG renders it into a graphical Perl program, making it possible for novice programmers and game designers to create computer games with a minimum of effort. A designer need specify only

the rules that make his or her game different from more generic examples of the genre; EGGG's reusable components supply the rest of the game logic. EGGG has been used to generate over 40 games, including chess, poker, rock-paper-scissors, crossword puzzles, marbles, Mammon, a foreign language learning game, a news game, Tetris, and new games like Deducto and variants used as instructional tools for conveying color relationships.

EGGG makes game design easier by shortening the game development cycle from months to minutes. This enables game designers to implement a game quickly, and then play it to discover any design problems. Game design thus becomes an iterative process of trial and error, an inherently easier way to create programs than to rely on perfect planning and meticulous execution.

This is an unusual approach to automated programming. Automated programming efforts always embody a compromise between the scope of the automated programs and the degree of automation. The Programmer's Apprentice[2] had a broad scope—it helped programmers create any sort of program. It was a suite of tools that helped expert programmers program better. In contrast, some currently available game generation systems allow users to "program" with nothing more than a mouse—but the domain is extremely narrow, restricted to a particular game subgenre. We can think of the Programmer's Apprentice as being *top-down*—full of deep abstractions, discoveries about the programming experience, and domain-independent idioms used by expert programmers. We can think of the more commercial systems (whether game-related or not) as being *bottom-up*, driven by a particular task and full of domain-dependent behaviors. EGGG, then, is *middle-out*, combining the lofty aims of a generic solution with the realities of a domain-specific real-world software project and its attending desiderata—efficiency, speed, portability, and modifiability.

**Trademark or registered trademark of The Tetris Company, Hasbro, Inc., or J. W. Spear & Sons Ltd.

Cited references and note

1. J. Orwant, *EGGG: The Extensible Graphical Game Generator*, Ph.D. thesis, MIT, Cambridge, MA (December 1999).
2. C. Rich and R. C. Waters, *The Programmer's Apprentice*, ACM Press, New York (1990).
3. B. Pell, "METAGAME: A New Challenge for Games and Learning," *Heuristic Programming in Artificial Intelligence 3—The Third Computer Olympiad*, H. J. van den Herik and L. V. Allis, Editors, Ellis Horwood Ltd., Chichester, West Sussex, UK (1992).
4. B. Pell, *Strategy Generation and Evaluation for Meta-Game Playing*, Ph.D. thesis, University of Cambridge, Cambridge, UK (August 1993).
5. J. Orwant, "For Want of a Bit the User Was Lost: Cheap User Modeling," *IBM Systems Journal* **35**, Nos. 3&4, 398–416 (1996).
6. L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proceedings of the IEEE* **77**, No. 2, 257–285 (1989).
7. A. E. Elo, *The Rating of Chessplayers, Past and Present*, Batsford (1978).
8. J. Beasley, *The Mathematics of Games*, Oxford University Press, Oxford, UK (1989).
9. The algorithm used is MD5. See http://theory.lcs.mit.edu/~rivest/Rivest-MD5.txt for more information.

Accepted for publication June 9, 2000.

Jon Orwant *O'Reilly & Associates, 90 Sherman Street, Cambridge, Massachusetts 02140 (electronic mail: orwant@oreilly.com).* Dr. Orwant is Chief Technology Officer of O'Reilly & Associates and Editor-in-Chief of *The Perl Journal*. He is the coauthor of *Programming Perl* and *Mastering Algorithms with Perl* (both published by O'Reilly) and author of the *Perl 5 Interactive Course* (published by Macmillan). Before joining O'Reilly, he was a member of the Electronic Publishing Group at the MIT Media Lab, where he received his Ph.D. degree for research involving the prediction of user behavior, the automation of game programming, and computer-generated personalized news and entertainment. Dr. Orwant also serves on the advisory boards of VerticalSearch.com, Focalex, Inc., and YourCompass, Inc. He is a frequent speaker at conferences, speaking to such diverse gatherings as (most recently) programmers, journalists, and lottery executives.