EGGG: The Extensible Graphical Game Generator

by

Jon Orwant

Sc.B. Computer Science and Engineering, MIT (1991)
Sc.B. Cognitive Science, MIT (1991)
S.M. Media Arts and Sciences, MIT (1993)

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2000

Author _____

Program in Media Arts and Sciences, School of Architecture and Planning
November 19, 1999

Certified by _____

Walter Bender
Senior Research Scientist
Program in Media Arts and Sciences
Thesis Supervisor

Accepted by _____

Stephen A. Benton
Chairman, Department Committee on Graduate Students
Program in Media Arts and Sciences

Thesis Committee

Thesis Supervisor _____

Walter Bender
Senior Research Scientist
Program in Media Arts and Sciences

Reader _____

Mitchel Resnick
Associate Professor
Program in Media Arts and Sciences

Reader _____

Murray Campbell
Research Scientist
International Business Machines

EGGG: The Extensible Graphical Game Generator

by

Jon Orwant

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning

on November 19, 1999 in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy in Media Arts and Sciences

ABSTRACT

An ontology of games was developed, and the similarities between games were analyzed and codified into reusable software components in a system called EGGG, the Extensible Graphical Game Generator. By exploiting the similarities between games, EGGG makes it possible for someone to create a fully functional computer game with a minimum of programming effort. The thesis behind the dissertation is that there exist sufficient commonalities between games that such a software system can be constructed.

In plain English, the thesis is that games are really a lot more alike than most people imagine, and that these similarities can be used to create a generic game engine: you tell it the rules of your game, and the engine renders it into an actual computer game that everyone can play.

Thesis Supervisor: Walter Bender

Title: Senior Research Scientist

Table of Contents

Preface

About This Dissertation

This dissertation is part of an ongoing project called EGGG, the Extensible Graphical Game Generator. As its name suggests, EGGG is a system that creates computer games, and the thesis behind the dissertation is that there exist sufficient commonalities between games that such a software system can be constructed.

In plain English, the thesis is that games are really a lot more alike than most people imagine, and that these similarities can be used to create a generic game engine: you tell it the rules of your game, and the engine renders it into an actual computer game that everyone can play. You can play with EGGG if you want; details are given in Appendix A.

In the *Introduction*, I talk about what drove me to create EGGG, and how my work differs from what others have done.

Anatomy of a Game discusses the common components of games that made this dissertation possible.

In the *Physiology of EGGG*, I explain the design of the EGGG parser and engine. I also describe the user experience of creating EGGG games: what novices have to know about programming, and what they don't.

Enemy of the Game State explores how EGGG builds computer opponents that can play games designed by users — including how they can bet and bluff.

Beauty on the Inside: Graphic Layout explains how EGGG generates the graphical elements of computer games.

Connect the Bots: Networking examines the networking components that EGGG uses to link computer games together.

Conclusion and Analysis concludes the dissertation and suggests future research directions.

The *Bibliography* lists relevant sources for games, computer gaming, game variations, and game creation.

Appendix A: EGGG Installation Instructions describes how to install and use EGGG.

Appendix B: Sample EGGG Games contains some EGGG files for some popular games to demonstrate EGGG rulesets.

Finally, *Appendix C: The EGGG Grammar* contains a Backus-Naur specification of the EGGG language.

Acknowledgments

I've spent a substantial portion of my life at the MIT Media Laboratory, and above all, I would like to thank Walter Bender for the environment he's somehow managed to maintain during all of that time. I defected from the MIT AI Lab in 1988 because Walter gave me the chance to hack image processing on the Connection Machine, and soon became enthralled by the freedom and intellectual camaraderie of the Terminal Garden and now the Cube. Since then, I've worked on news, user modeling, a multitude of other projects, and finally games, the subject of this dissertation. Walter combines a love of ambitious ideas with a laid-back attitude — an unusual combination, and I cannot imagine a better advisor.

Thanks to the other members of my thesis committee, Mitchel Resnick and Murray Campbell, for their helpful insights and suggestions.

Thanks to my friends who contributed, knowingly or not, to some of the ideas in this dissertation: Sandy Aronson, Alan Blount, Amy Bruckman, Richard Christie, Diego Garcia, Carolyn Grantham, Kyle Pope, Kimberly Scearce, and Sekhar Tatikonda.

I'd like to thank the many members of the Perl community that have provided me with innumerable distractions during my Ph.D.: David Blank-Edelman, Tom Christiansen, Mark-Jason Dominus, Jarkko Hietaniemi, Tuomas J. Lukka, John Macdonald, Linda Mui, Chris Nandor, Andy Oram, Tim O'Reilly, Madeline Schnapp, Mike Stok, Nathan Torkington, and Larry Wall. In particular, Nathan Torkington and Tom Christiansen were a continual source of intellectual amusement.

Thanks to the officemates who've put up with my folding bike and my ten pounds of silly putty: Judith Donath, Doug Koen, and Marko Turpeinen. Other members of the Media Lab provided camaraderie, inspiration, and help: Nathan Abramson, Bill Butera, Pascal Chesnais, Klee Dienes, Scott Fullam, Dan Gruhl, Roger Kermode, Jill Kliger, Hakon Lie, Michelle McDonald, Chris Metcalfe, Warren Sack, Laura Teodosio, Sunil Vemuri, and Chris Verplaetse. Thanks to Nicholas Negroponte for creating the Media Lab, and to Felice Gardner and Linda Peterson for making it function so smoothly.

Finally, thanks to Robin Lucas, and to my parents, Jack and Carol.

Chapter 1: Introduction

> *I think the thing I take the most pride in about Spacewar is that it got so many people hooked on computer programming. It caught a lot of eyes and got a lot of interesting people asking, "How do you do that?"*
>
> Steve Russell, author of *Spacewar*, the first video game.

There are obviously similarities between games. Are those similarities sufficiently numerous and deep that a system can be constructed which turns a written description of a game into a playable program? This dissertation is an attempt to answer that question in the affirmative. In support of this thesis, a system that translates the rules of a game into a program was built: EGGG, the Extensible Graphical Game Generator.

In this chapter, I'll describe what motivated me to create such a system, and survey related work. First, we'll look at the breadth of computer games that exist today.

What's In A Computer Game?

Frenetic shoot-em-up games like Doom and Quake sell millions of copies and tax the abilities of today's most powerful personal computers. Considerably less demanding games like the ubiquitous Windows Solitaire occupy thousands of hours of idle (and not-so-idle) time; Napoleon played the old-fashion version while in exile on Elba. Supercomputers like IBM's Deep Blue defeat the world's greatest human chess player. Internet users circumvent U.S. gambling laws by wagering in Antigua.

Each of these four experiences is considered "computer gaming", even though they differ in who plays them, and why, and how. Games like Doom are graphic-intensive, often violent, and players move from location to location in a virtual world. (In spite of what some detractors say, some of these games have challenging puzzles and intricate narratives.) The various Windows Solitaire games are played with picture-for-picture identical 52-card decks. The heart of Deep Blue is the same as every other chess-playing program: a minimax engine coupled with a static evaluator and a library of opening moves. Gambling games, whether online or off, involve unpredictable outcomes.

In an attempting to categorize computer games, we could treat each of these four scenarios as representative of an entire genre of games:

- "First-person-shooter" games
- Card games

- Chess-like games
- Gambling games

Is this a good categorization? No; many games are missing. Where's tic tac toe, or Tetris? Here's another categorization:

- Games requiring hand-eye coordination
- Card games
- Games played on a grid
- Gambling games

This is still not up to snuff. Some of these categories overlap: is poker a card game or a gambling game? And who cares about a taxonomy of games, anyway?

Programmers do. Once you arrive at a comprehensive categorization of games, you can begin to make the leap from merely categorizing games to drawing inferences about them — and ultimately, to helping people create them. Here's an elaboration of those categories:

- Games requiring hand-eye coordination

  ...have a main loop and need callbacks

- Card games

  ...use the same images and share some actions, like shuffling.

- Games played on a grid

  ...need a two-dimensional array to contain the game state.

- Gambling games

  ...require security and a "Money" abstraction.

Now we're getting somewhere. This categorization is still quite unclean — the first category has to do with process, the middle two with structure, and the final category with function — but it serves to illustrate what I'm talking about when I say that there are similarities between games. These similarities are all obvious, but in *Anatomy of a Game* (Chapter 2) we'll develop a more comprehensive taxonomy of games, arriving at some far less obvious (but just as useful) observations. For the remainder of this chapter, let's just assume that such a taxonomy exists.

Decoupling Hard from Soft

Before the Atari 2600 debuted in 1978 (and its obscure predecessor, the Fairchild Channel F in 1976), videogames were embedded in chips inside chunks of wires and plastic called game consoles. When you tired of whatever games were burned onto the chip, you stopped using the system entirely.

The 2600 (also called the "Atari VCS") revolutionized the game industry because of *cartridges* — no longer were chunks of plastic and wires only able to play a fixed set of games. When you got tired of a cartridge, you'd just buy a new one, for a fraction of the console price.

This decoupling of hardware and software was made possible by simpler and more flexible hardware components. The decoupling is what made home video game systems a commercial success, because someone plunking down $250 wouldn't be restricted to a small fixed number of games. He knew that as long as tens of thousands of fellow gamers had consoles just like his, that new games would be developed for it.

So what's the next step? We can take the decoupling further, thanks to simpler and more flexible software components. Instead of decoupling hardware from software, what we can do is decouple a game's implementation (the "hard software") from the rules of play (the "soft software"). This is what EGGG does, and it does it by incorporating assumptions about game play into a program that translates the rules of play into a fully-functioning game.

The Tradeoff

Programming is hard. But programming for a particular domain needn't be — when you can build assumptions about the domain into the language. And more to the point, those assumptions can be built into whatever system (compiler, translator, interpreter, or combination of all three) that ultimately turns the program into something you can run.

Let's say your domain is widgets. If you know your clientele will be designing widgets, you could create a software package called a Widget Designer that lets them specify a recipe for building their widget. The recipe can be viewed as a program, and the Widget Designer is a translator that transforms the program into something more complex.

Designing a widget might not seem like programming for two reasons. First, the Widget Designer might be easy to use, and people are unfortunately conditioned to think that programming has to be difficult. Second, the range of behaviors permitted by the Widget Designer will be severely limited: you won't be running spreadsheets or calculating differential equations with it. You just design widgets. There's a tradeoff between expressivity and convenience; the Widget Designer chooses

convenience.

Another example: drawing pictures on a screen used to be hard. In the old days, graphics libraries gave you ways of drawing points and lines and circles, and not much more. Creating usable applications with buttons and pictures and scrollbars was tough — but you had enough low-level control over what the electron gun shot to the CRT that you could do anything. You could be expressive, but it wasn't very convenient. Later, graphics toolkits took away the drudgery, but they also took away some of the low-level control. Applications were developed faster, but they all had a certain similarities of appearance. In Xlib, you could do anything. In Motif, you can do almost anything, and people would look at the result and say, "Oh, you must have used Motif." The same is true of other visual programming environments, like Visual Basic and NeXTSTEP's Interface Builder.

You can see the same phenomenon with HTML. People used to speak about "HTML programming", but that phrase is becoming more and more rare as programs like ColdFusion are used to generate HTML automatically. You don't hear people saying "Oh, you must have used ColdFusion" when they see those web pages, but that's only because web browsers affect how web pages appear.

Every system that helps you build things has to make a tradeoff between the variety of things it lets you build, and the ease with which you can build them. EGGG is an attempt to help people create programs, but it is not a generic programming assistant like the Programmer's Apprentice, described later. Nor is it a visual development environment that lets people drag and drop elements into place, which would permit only a small variety of games.

Circumventing the Tradeoff: Patching and Burrowing

NeXTSTEP's Interface Builder lets you construct applications by dragging and dropping components (buttons, frames, text boxes, and the like), and by establishing links between them. As you would expect, the applications you construct all have the same look and feel; there's only so much customization you can do. But under the hood, what the Interface Builder does is generate Objective C code. Programmers who want to specify the behavior of the component do so in code. The look and feel is still constrained, but the system lets you *patch* the generic output by writing your own code.

Other systems let you *burrow* down into internals. For instance, languages like Perl and some versions of LISP let you implement chunks of code in another lower-level language when speed is paramount. Browsers like Netscape, web servers like Apache, and paint programs like the GIMP and PhotoShop provide hooks so that developers can write code for features not provided by the prepackaged application.

EGGG lets you both patch and burrow. details are provided in Chapter 3, the *Physiology of EGGG.*

Games As A Domain For Automated Program Generation

What is the right size domain for creating an automated program generator? If the domain is too broad, the assistant won't be able to help much, because it won't be able to infer the intentions of the would-be programmers. If the domain is too narrow, the assistant won't be necessary. No one needs help creating calculator programs, because there aren't that many different programs to create.

Games have several advantages as a domain:

- Games have the right amount of diversity. The similarities between games like chess and tic tac toe are obvious; the similarities between poker and chess and Tetris, less so. By building a system that can create all of those games, we support to our thesis that the similarities between games are broadly applicable and useful to an automated program generator.

- There are many game programs. This allows us to identify common techniques in their design, and exploit those techniques in the design of EGGG.

- Many games can be easily represented with a small set of rules. It is possible to create systems that generate stories, or financial applications, or music — but the systems would require much more input from the user.

- A game generator needn't be perfect. Because games are so popular, they'll still be used even if they lack the polish and speed that only dedicated programmers can provide.

- Games benefit from large populations of users. One way to improve a system as large and sprawling as EGGG is to release it and see whether (or how) people use it. Furthermore, games appeal to novices and experts alike, so a system that solicits feedback from users will benefit. Novice programmers will have different comments and suggestions about the system than experts, and EGGG can use feedback from both camps. This feedback won't be evidence for or against the thesis, but it will help make EGGG more useful and fun for everyone.

So how can users communicate their intentions to EGGG? They do so by codifying the rules of their game in a pseudo-English language. We call this the EGGG language, and turn to it in the next section.

Design Principles: Languages, Overengineering, and

Brevity

It's an uncertain world out there, and classically trained programmers know this. They know to design for scalability: if your program operates on a thousand pieces of data today, it might be called upon to manipulate a million pieces of data next year. And designers decide that since they can't know what protocols or standards or data formats will be around in a year, they create abstractions for those protocols and standards and data formats.

The unfortunate result is that many designers *overdesign*, abstracting out components too much, and in so doing slow down their programs (that's bad) and make them harder for others to understand (that's worse). The abstractions that seem so clear to a designer are opaque to the next person who has to use the program. When designers assume that their users share their tenacity and attention to detail, the result is bad software.

But this is the wrong approach. The right approach is to be able to program quickly, so when that new protocol arrives, you can throw away your old code and begin anew. The right approach is to create lots of tiny tools that do one thing well, so you can combine them in whatever pipeline you need.

Consider the Berkeley socket library used for networking computers together. The library doesn't do very much: it just creates a connection between two computers on the Internet, or between two processes on the same Unix system. Yet because the designers wanted a uniform interface to both, they ended up with a system that requires you to specify which connection you want. Some of the arguments you have to provide make no sense if you're connecting two computers, and others make no sense if you're connecting two processes. Nowadays Berkeley sockets are used almost exclusively for Internet connections, because there are better ways to communicate between processes on the same computer, like SysV IPC and threads.

What is the right level of abstraction for games? As with the Berkeley socket library, there's a danger here. Abstractions are all well and good, but designers shouldn't force users to cope with their particular organization of the world unless it's *intuitive*. EGGG disguises its abstractions behind terms with common usages, providing a simple language through which game designers can communicate their intentions to the system.

So "EGGG the engine" and "EGGG the dissertation" include "EGGG the language": a high level language that lets users describe games in as few words as possible while still retaining the precision that the EGGG engine needs to render the language. The language is expressive (you can create nearly any kind of graphical two-dimensional game) and concise (statements are short and powerful, so that debugging is easy).

The importance of brevity shouldn't be underestimated. It's good to be able to see all of the game on a single page — you never have to visit different files, or even scroll up and down. For instance, here's how one tells EGGG that players alternate turns in

chess:

```
turns alternate
```

Here's how one stipulates that the deal moves clockwise in poker:

```
deal moves clockwise
```

Every attempt was made to mimic language that people would normally use to describe games. That's not always possible, of course, and there is a particular danger involved with systems that accept natural language as input: users tend to assume that the parser is much more sophisticated than it actually is [Brennan 90]. Here's the description of a stalemate in chess:

```
Stalemate means turn(P) and no moves(P) => tie
```

This can be read as "A stalemate is a situation where it's P's turn and he doesn't have any moves. If that happens, the result is a tie." This single statement triggers EGGG to do a number of tasks:

1. Create a subroutine called `Stalemate`.

   When EGGG encounters `Stalemate means`, it sees that `Stalemate` is capitalized. Lowercased words have special meaning to EGGG; uppercased words are terms that have a per-game or per-rule meaning.

2. Execute the subroutine at appropriate times.

   Because the subroutine mentions a turn, it will be executed at the beginning of each turn. Checkmate is checked for at the end of each turn; that's why the Stalemate definition needn't verify that the player isn't in check.. If the game designer preferred precision to brevity, the rule could have been written this way:

   ```
   Stalemate means turn(P) and no moves(P)
             and not Attacking(Q, King) => tie
   ```

3. Have the subroutine set a variable, `P`, to whoever's move it is.

   Different single letters mean different things. `P` always means a player, and because it's capitalized, it means a particular player. Had it been `p` instead, EGGG would have generated a loop that cycled through all players. Of course, in chess there are only two players, but in a game like poker, the rule we saw earlier, `deal moves clockwise` would indicate how the loop would proceed.

4. Have the subroutine check to see if `P` has any available moves.

   EGGG will already have generated a subroutine that enumerates all of the moves available for each player. `no moves(P)` causes that subroutine to be invoked, and immediately returns zero if any moves are found.

5. If there are no available moves, invoke the `tie()` subroutine.

EGGG will already have created a `tie()` subroutine on a previous pass through the game description; any mention of "tie" triggers that behavior. If the subroutine gets to this point, it must be player `P`'s turn, he must have no available moves, and he must not be in check, so a stalemate occurs and the game is tied.

These game descriptions are stored in single files. The description is very concise: the rules of poker require 19 lines, and even chess requires only 81 lines. The ordering of the lines within the file is immaterial.

There is plenty more to the EGGG language than what is shown here, and we'll see more examples throughout this document. The EGGG language strives to approach the brevity and simplicity of human writing.

When there is ambiguity in the language (in poker, "draw" means to receive cards and "rank" means the strength of a hand, while in chess, "draw" means to tie the game and "rank" means the *y*-coordinate) EGGG makes its best guess. If it's really stumped, it queries the user.

It is assumed that most users will create their own games by copying game description files from EGGG's central repository and modifying them. As one would expect, variations on traditional games are easier to create than entirely new games.

The design criteria for the EGGG language and engine are the following, in approximately 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.
- It should be easy to create variations.
- The EGGG engine and the games that it generates should be maximally portable.
- The games generated by EGGG should be easy to modify.
- EGGG shouldn't take a long time to generate games, and the games that it does generate shouldn't run so slowly that playability is affected.

What This Dissertation Is And Is Not

EGGG is not about the "who" or the "why" of games, only about the "what" and "how". It is an exploration of the similarities between traditional games like board games and card games, and how those games are realized in computer programs.

The thesis underlying the dissertation is that those similarities are sweeping enough that it is possible to construct a system that generates a computer game from its distilled essence: the rules of play.

This dissertation is not about children, society, gender, culture, education, or any particular community of game players. While there are many interesting research topics in these areas, EGGG's focus is on the similarities between games and the resulting implications for the mechanics of computer game creation. This dissertation offers no opinion about what sort of games should be created, nor about the effects of any game — good or bad — on the people who play them. However, we do note that Monopoly sends a rather brutal message to would-be industrialists, and no game generated by EGGG can be as dangerous as the fluoroscopes found in the earliest penny arcades, where medically uninformed amusement seekers paid for the ability to X-ray the bones in their hand.

The games that EGGG creates are those that lend themselves to concise descriptions: the simpler the game, the better. It is best-suited for creating games involving pieces and boards and cards and icons, and not well-suited at all for games like Mortal Kombat, or Doom, or sports simulations. The graphical sheen of games like Mortal Kombat and Doom is intrinsic to their appeal, and EGGG can't supply artistry. Sports simulations might not require so much polish (although the successful ones certainly do), but the sheer number of rules involved in any sport makes it difficult to write the description in such a way that EGGG could know how to render it. If you're creating a baseball game, you can't just tell EGGG to create bases. You have to tell it how far apart to make the bases, and the pitcher's mound, and the outfield wall. You'd have to describe — algorithmically — the paths of the different types of pitches. You'd have to express the speeds of the pitches, and maybe even of the runners and the bat. And so on. It is impossible to describe baseball concisely to EGGG, because it is impossible to describe baseball concisely.

Theoretically, EGGG *could* create any game, since you can always burrow down into the underlying Perl programming language. However, in this dissertation we will restrict ourselves to games that are graphical, but not graphics-intensive or overly complex.

One could argue that these are the best games to study because they are the most timeless: card and dice and board games have been around for centuries, so they best epitomize gameness. I don't agree with this; while arcade games have eclipsed "classic" games, it's unclear whether this is because of changing tastes of youth exposed to incessant marketing or because those games really are more fun to play. It's quite possible that if the ancient Romans had had Nintendo, the empire would have collapsed a lot sooner. I make no claims about the superiority of some games over others, nor do I claim that game play is an intrinsic part of intellectual life. Gaming is less universal than many suspect; despite the best efforts of anthropologists, no one has been able to stake a claim for the universality of games across cultures [Avedon 71].

However, game play is nearly universal among computer scientists, and in the next

section we turn to other systems pertaining to computer game creation and automated programming assistance.

Related Systems

The Programmer's Apprentice

Before I came to the MIT Media Laboratory, I worked at the MIT Artificial Intelligence Laboratory on the Programmer's Apprentice, an automated programming effort headed by Charles Rich and Richard Waters. The Programmer's Apprentice 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 type of systems you can create. EGGG does not do requirements analysis or program verification; it simply generates games.

METAGAME

METAGAME [Pell 92] is similar in spirit to EGGG, but arose from a different premise. For a long time, chess was considered a formidable problem for AI. The reasoning was that any program that could beat an expert player, or even a competent one, would surely possess some of the intellectual capacity of a human. Yet 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 doesn't mean that it's intelligent.

Barney Pell's METAGAME strives to bring the AI back to computer game playing. He suggests that the test of a computer's game-playing prowess shouldn't 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 aribtrary 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, cf. [Pell 1993], p. 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.

Commercial Products

There have been several commercial "game construction kits", but they are typically not very expressive: the range of the games you can generate is quite limited. Bally's Pinball Construction Kit, for instance, let you adjust the number and placement of flippers, and it let you change gravity, but at the end of the day all you could create were slightly different variations on the same pinball game. Many game constructions kits are like this.

In this section, we will briefly discuss two commercial products that have broader approaches to game construction: GURPS and Klik & Play.

GURPS

Steve Jackson's GURPS (Generalized Universal Role Playing System) is an ambitious attempt to generalize role-playing games into a single set of all-encompassing rules. It focuses on dice-and-paper simulations and therefore is only tangentially related to computer games, but faces the same problem EGGG does: how do you partition the universe of games, and what are the proper levels of abstraction? For instance, GURPS has a single unifying architecture for all combat, but overrides that architecture when certain conditions occur (for instance, when the

combatants are in water). Parallels exist for EGGG: it makes guesses about games and then overrides those guesses as more specific rules are found.

GURPS is a reference manual; it is essentially a compendium of information to help you create your own simulation. It does not create the simulation for you, nor can it supply the creativity that every successful role-playing game requires. (EGGG can't supply the creativity either, of course.)

Nevertheless, GURPS is an ambitious work, and it is a philosophical ancestor of EGGG. Reading it, you can sense Steve Jackson's frustration as an avid scenario designer who recognized the similarities between the different milieus: protecting your body with chain mail against an arrow is not unlike protecting your body with a Kevlar vest against a bullet. In each case you have to establish probabilities that the target will be able to dodge the projectile (the bulkier the protection, the slower the target can react), the probability that the projectile will penetrate the target (which depends on the speed and force of the projectile), and the probability that the wounds caused will be fatal. That, in turn, depends on the health of the target, and so on. Nor is GURPS entirely about combat; it provides guidelines for a scenarios as obscure as loading cargo onto an aircraft without the aircraft having to land.

Implicit in GURPS's design are decisions about when to generalize (social traits that are universal, and combat rules that apply to all milieus) and when not to (a character's skill at xenology — the knowledge of alien races — which is unique to space-age milieus). However, GURPS' generalizations are all obvious: you can fight in any game, but laser weapons have no place in Westerns.

Klik & Play

Klik & Play, originally created by Europress Software and distributed in the U.S. by Maxis, is a game construction kit that has met with some commercial success. In contrast to GURPS, Klik & Play is more than a reference guide: like EGGG, it creates computer games.

Klik & Play is aimed at children. It's a visual application builder that provides you with predefined objects — a running man, walls, monsters, and the like — and lets kids assemble them into video games. There is an "event editor" that lets you define complex actions — but these actions are all essentially simple conditionals that depend on the other objects. Klik & Play games aren't Turing complete.

The video games all look pretty much the same: in J.C. Herz's categorization of games (see *The Game Space* in Chapter 2), they are all "shooters", a subgenre of action games. Herz's genres, in turn, are all subgenres of video games. You can't make crosswords, or poker, or chess, with Klik & Play.

The system touts its object orientation, which only means that its monsters and walls and objects have independent picture, animation, and movement properties. Klik & Play is good, but it is not a universal game generation engine. In the tradeoff

between expressivity and ease, it favors ease to a much higher degree than EGGG.

Related Fields

Game Theory

EGGG is about games, and EGGG is about theory, but EGGG is only tangentially about game theory. Game theory is a branch of mathematics (some would say economics) that deals with human interactions where the outcomes depend on the interactive strategies of two or more people with opposing motives. Game theory began with a simple betting card game called *le Her*, in which two players each drew a card and could optionally exchange the card for another based on simple rules. An optimal strategy for playing le Her was discovered in 1713, and that optimal strategy was applied to all two-player deterministic games with the Generalized Minimax Theorem by John von Neumann in 1928.

One well-studied game in game theory is the Prisoner's Dilemma, which has this scenario: Two criminals, in cahoots, are arrested for the same crime. They are placed in separate interrogation booths where they can't communicate with one another, and each is encouraged to confess to the crime. If one criminal confesses and indicts the other, the confessor gets a light sentence while the holdout does heavy time. If both confess, they both get a heavy sentence, but not as heavy as a single holdout. If both hold out, they both get a light sentence — but not as light as a single confessor.

What should a criminal do if he wants to minimize his jail time? Tough question, and this is the bread and butter of game theory. Active research topics include variants of this scenario. What if there are repeated trials? What if they are able to communicate? What if they aren't rational? What if there are three criminals instead of two? What if the criminals aren't sure of the penalties? What if one criminal is less averse to jail than the other? And so on, applied to many other scenarios: arms control, marriage, patents, college applications.

Game theory has nothing to do with actual computer game creation; it is only about strategies for making the best decision, in games where it's possible to reason about what a best decision is (as opposed to arcade games or crossword puzzles). EGGG uses the results of game theory when it generates computer opponents for a game, but the strategies it uses are not the cutting-edge game theory research. The strategies are discussed in Chapter 4, *Enemy Of The Game State*.

Complexity Theory and Game Automata

It's possible to view games as problems to be solved. Can the first player guarantee a win in tic tac toe? No. Can the first player guarantee a win in chess? No one knows;

it's a problem yet to be solved. You can try to solve this problem with brute force, but you'll fail. There are thought to be 10^{120} chess games, and there are thought to be about 10^{70} protons in the universe. If every proton in the universe were evaluating a trillion chess boards per second, you still wouldn't be able to answer the problem before the universe ended.

Complexity theory is the branch of computer science that analyzes how hard problems are to solve. If a problem can be solved in polynomial time, it is said to belong to *P*. (More precisely, *P* is the class of languages that are decidable by some Turing machine in a number of steps that is bounded by a polynomial.) If a problem can be solved by a nondeterministic Turing machine in polynomial time, it belongs to *NP*. Every problem in *P* is obviously in *NP* as well. It has not yet been proven that there are problems in *NP* that aren't in *P* as well, although it's a good bet. Much of complexity theory involves proving that a given problem can be "reduced" to a known problem in *P* or *NP*, where "reduced" means, roughly, "translated into, if you change how you represent the problem in a nonintrusive way."

Garey and Johnson [Gar 79] list the known complexities of hundreds of problems, including crossword puzzle construction (which is *NP*-complete), checkers (*PSPACE*-hard), and Go (*PSPACE*-hard). This translates roughly into:

- Given a set of words, finding a crossword grid that contains those words will take too long if you have enough words.
- Given checkers on an *N*x*N* grid, determining whether a player can always win will take too much memory if *N* is high enough.
- Given Go on an *N*x*N* grid, determining whether a player can always win will take too much memory if *N* is high enough.

These results may seem far removed from the practical concerns involved in determining a game designer's intentions and converting them into working programs, and for the most part they are. When a typical game opponent is created, its designer has a rough idea of how complex the problem is. Perhaps he can't determine it to the satisfaction of a theoretical computer scientist, but he knows whether a brute-force approach is feasible or not, and he can choose an appropriate strategy accordingly. EGGG doesn't have that luxury.

Condon's Probabilistic Game Automaton

Condon's *Computational Models of Games* [Con 89] notes that "Traditional models of computation, such as Turing machines, do not reflect the game-like properties of many problems of interest to computer scientists. On the other hand, the traditional approach of mathematicians to game theory did not focus on questions regarding the computational complexity of the game."

To rectify this, Condon defines a theoretical construct called a *probabilistic game automaton* that combines elements of several game-theoretic models. The probabilistic game automaton is able to model features of games that traditional game theory can't accommodate: randomness, secrecy, and limited resources of the players.

The different types of probabilistic game automatons can be classified along three dimensions:

Universal steps	Coin-tossing steps	Degree of information
∀	B (bounded)	Z (zero)
	U (unbounded)	P (partial)
		C (complete)

In Condon's analysis, every probabilistic game automaton has at most one symbol from each of the left and center columns, and exactly one symbol from the right column. The left column, *Universal steps*, has only a single symbol, the "for all" operator. Universal steps are situations where a player has a move available to him without any randomization involved. The middle column, *Coin-tossing steps*, are situations where what a player does is determined randomly. *B* or *U* are chosen depending on whether the automaton needed to model the game is finite or infinite. The right column, *Degree of information*, indicate how much information a player reveals. A completely secretive game reveals zero information, *Z*; a game in which the player reveal some information is *P*, and a game where no information is concealed is *C*.

These games are all between two players. Multiplayer games like poker, or single-person games like crosswords, cannot be modeled by probabilistic game automatons. And games that depend on hand-eye coordination can't be easily modeled by any automaton at all.

The Eight Games

Computer science has a narrower view of games than the broader population, and so for exploring the similarities between games and describing EGGG's capabilities, we'll focus on a representative core of eight games that, taken together, portray the versatility of EGGG:

- Tic Tac Toe

  A two person game of complete information, with no randomness and simple rules.

- Chess

A two person game of complete information, with no randomness and complex rules.

- Poker

  A two to six person game of partial information, with randomness and complex rules.

- Crosswords

  A one person game of complete information, with no randomness and simple rules.

- Tetris

  A one person game of partial information, with randomness and simple rules.

- Rock Paper Scissors

  A two person game of zero information, with no randomness and simple rules.

- Deducto

  A one person game of partial information, with randomness and complex rules. Deducto is the author's creation, and will be discussed in Chapter 3, *The Physiology of EGGG.*

- Mammon

  A multiple-person game of partial information, with no randomness and complex rules. Mammon is the author's creation, and will be discussed in Chapter 6, *Connect the Bots: Networking.*

EGGG can generate an infinite number of games, and it can generate games that are substantially different from these eight, but in the interests of having concrete examples that can be continued throughout the discussion of EGGG, we will restrict ourselves to these.

In the next chapter, we'll take a deeper look at the similarities between games that made EGGG possible.

Chapter 2: Anatomy of a Game

> *[Game] genres do seem to hold together in the middle, weathering revolutions in chip speed and licensing. It's like the proverbial fourteen novels that have been endlessly rewritten throughout history. The costumes change, but the basic matrices remain. There are certain things that people want to see on a video screen. There are certain strategies that are inherently satisfying. There are certain ways of organizing obstacles that are hard to improve upon.*
>
> J.C. Herz, in *Joystick Nation* (page 25).

In this chapter, we will develop a taxonomy of games, laying the foundation for Chapter 3, *The Physiology of EGGG*, where we discuss the actual mechanics of game generation.

How are games organized? Is there some Platonic essence of gameness, an ur-game from which all other games have evolved over millennia? A romantic thought, but the answer is no: even if Australopithecus took turns seeing how far they could throw a rock, plotting a gradual evolution from that to Myst would require shoehorning cricket, logic puzzles, Pac-man, card tricks, and professional wrestling all into the same family tree. With games, as with most other collectible things, categorizing is messier than it would first appear.

If we're going to make sweeping conclusions, we first need to identify what it is that we're making conclusions about. Is baseball a game? What about hide and go seek? Anthropological literature typically divides leisure activity into game, play, and sport. Play has no explicit goal, sport involves a test of physical ability, and everything else falls under the catchall category: games. This has the curious corollary that a baseball game played on a field is a sport, while a baseball game played on a computer is a game, even though the two differ only in which muscles are being strained. Nevertheless, it is this definition that we will use when we categorize games and game elements. Game strategies are deferred until Chapter 4, *Enemy of the Game State*.

A Structural Categorization Of Video Games

We showed some overly simplistic categorizations in the Introduction. For another, deeper categorization, we can examine a particular genre of computer games: the arcade game. J.C. Herz, in *Joystick Nation*, divides up the space of arcade games as follows:

- Action games.

These are also known as "twitch" games, and are the most popular subgenre of arcade game. They also have commercial opportunities that few other subgenres do, because they can have character development, and are therefore ripe for cross-licensing deals with other entertainment industries. Sometimes the games lead to television shows (Pac-Man) or movies (e.g. Mortal Kombat), and sometimes television shows (The Simpsons) or movies (Star Wars) lead to games. In Herz's words, these relationships are good for "hatching a slew of games based on movies that are mostly special effects anyway." Herz continues, "These include some of the worst cartridges and arcade cabinets ever produced."

Herz divides up action games into what she calls "structural subcategories":

- Horizontal scrolling games. In these games, ships move horizontally across dangerous terrain. Examples: Scramble, Defender.
- Maze chase games. The player navigates around the screen, eluding opponents. Examples Pac Man, Rally X.
- Platform climbing games. The player tries to navigate obstacles by moving between areas of the screen. Examples: Donkey Kong, Lode Runner.
- Shooters. Enemies are attacking you; shoot them all. Examples: Doom, Robotron.
- Raining games. Missiles are falling down at you; avoid them (or prevent them from hitting the innocents below you). Examples: Missile Command, Kaboom.
- Breakout. Break down a wall with many repeated attacks. Examples: Breakout, Arkanoid.

- Adventure games

  Adventure games include some of the earliest computer games: Zork and Adventure, which were text-based; and the graphics- and sound-intensive commercial success Myst. Common to all of them is that you wander about accumulating items which are used to solve puzzles.

- Fighting games

  In Herz's words, "comic books that move." An enemy is attacking you; use the right combination of moves to defeat him. Examples: Mortal Kombat, Tekken.

- Puzzle games

  Adventure games have an ultimate goal, a Holy Grail; these don't. For

instance, Tetris has no ending. The play just gets harder and harder.

- Role-playing games

  In these games, the player chooses or invents a character and behaves accordingly: you can't just hack and slash everything in sight. Examples: Wizardry, Ultima.

- Simulations

  Simulations strive for realism over frenetic button pushing; frequently, there is military funding somewhere in the development pipeline. In some simulations, the player needs to manipulate a complex vehicle; in others, he needs to manage limited resources to develop something. Examples: Lunar Lander, SimEarth, and all flight simulators.

- Sports

  Sports games are a combination of action games and simulations, and try to be as realistic as possible so that they can cater to real-life sports aficionados. Examples: NBA Jam, NFL Quarterback Club 99.

- Strategy

  These are games where you have to plan long-term strategies, or foster temporary alliances with enemies. Frequently, the theme is consolidation of power, and the games are often multiplayer — either between humans or between a human and "intelligent" computer opponents. Examples: Civilization, Populous.

Herz is only attempting to categorize video games here, but note how she does it: she categorizes them by the player's experience. That makes sense, because her audience is players.

Burns categorizes non-video games into the following categories [Burns 98]:

- Card games
  - Patience Games
  - Gambling Games
  - Non-Trick Games
  - Trick Games
  - Children's Games
- Board Games
  - Family Board Games

    - Race Games
    - War Games
    - Territorial Games
- Domino & Dice Games
    - Domino Games
    - Dice Games
- Family Games
    - Parlor Games
    - Paper & Pencil Games
    - Word Games & Spoken Games
    - Written Games
- Sporting & Active Games
    - Games of Skill
    - Outdoor Games

However, if we try to categorize games from the developer's perspective, we need different criteria. When you're programming, the difference between a shooter and a raining game is far less than the difference between either of those games and a card game. Likewise, the difference between the software architecture of a Go game and a rendition of Capture The Flag is tremendous, even though Burns would classify them both a territorial games.

A taxonomy of games from the player's perspective focuses on structure. A taxonomy from a mathematician's perspective focuses on information and probability, as Condon's probabilistic game automata suggest. A taxonomy of games from the designer's perspective focuses on *process*.

A Designer's Taxonomy of Games

In the rest of this chapter, we will explore the similarities of games by creating a *game taxonomy*: an organization of games. Each game will be described by a *categorization string* that describes the process of game play. The categorization strings don't contain the rules of the game, and knowing the categorization doesn't

enable one to reconstruct the game, or even visualize it. But if you know the categorization string for a game, you can generate the entire software infrastructure of the game. Everything else is just frosting.

In our taxonomy, we attempt to use familiar English words wherever possible. While the taxonomy would sound loftier if we used precise terms (`OccludingBarrier`, `PlayingSurface`, `IntrinsicAttribute`) we have chosen to use imprecise terms instead (`Hand`, `Board`, `Color`) in the interest of making our study of games a little more accessible.

Frenetics

The first criterion of how a game is to be programmed is whether it is frenetic — whether it requires "quick" action. We can divide those games into "twitch" games that require near-continuous quick action (most arcade games) and those that are simply time-based, like chess when it is played with a chess clock.

Any game that requires moves in a fixed period of time requires a program that can record how much time has elapsed, which in turn requires that any sort of pause feature not provide a player with an undue advantage.

This provides us with the first category of our taxonomy, which we will depict with a table:

Frenetic and fast	ff
Frenetic, but merely timed	ft

If an `ff` is present in the categorization string for a game, it means that the game is frenetic and fast; if an `ft` is present, it means that game is merely timed. If there's nothing at all, the game isn't time-critical at all. These categorization strings were part of the design of EGGG, but are not part of its implementation; they serve only to help us identify the similarities between games.

Some Tetris implementations blank out the screen when you pause, so that you can't analyze where a piece would best fit, unpause, and play the perfect game. There are right ways to pause and wrong ways to pause; that's a right way to pause. However, any pause feature would defeat the purpose of timed chess; since chess has relatively little state (any reasonably good chess player will be able to reconstruct the board from memory), blanking out the screen isn't sufficient.

Thus, a universal game generator that renders games from descriptions first needs to determine whether the game is time-based, and then needs to determine what to do when pausing: the more frenetic the game is, the more sufficient it will be to simply blank out the screen.

This provides the second category of our taxonomy:

Frenetic?	History-critical?
f	hx

Let's classify the eight games described in the Introduction according to these three bits:

	Frenetic	History
Tic tac toe		
Chess (with chess clock)	ft	h
Poker		h
Crosswords (untimed)		
Tetris	ff	
Rock Paper Scissors	ft	h
Deducto		h
Mammon	ft	

Throughout this chapter, we'll add new categories to our classification scheme as we define the other components of our taxonomy. Before we add more bits to our classification scheme, we'll examine history in greater depth.

History

In this section, we turn to the notion of a game's *history*. This has nothing to do with how or when the game itself was developed; instead, it refers to the succession of moves made in a game, and their importance to deciding how future moves should be played.

In particular, we turn to what we call *local history*, which is what Condon calls simply "history": the sequence of moves in a particular game between particular players.

Condon [Condon 89] defines *Markov games* as games where history is irrelevant. If you look at a tic tac toe game in progress, you know everything you need to know just from looking at the board and identifying whose turn it is. Tic tac toe is a Markov game; so are crosswords. To say that something is a Markov game is the same as saying that you can choose your move without remembering earlier moves.

Text adventure games are not Markov games, because what happens to the player at

a given point depends on actions he took earlier.

Chess, to the surprise of some, is not a Markov game. You cannot capture a pawn *en passant* unless the opponent moved that pawn forward two squares *in his last turn* — so you need to remember the last turn: a history of one move. Furthermore, you cannot castle unless you have never moved your king. Therefore, a chess program has to remember all of the moves back to move 2, which is the earliest that a king could be moved. You need to remember almost the entire history. Go also requires history: even though Go books teach readers by depicting boards and asking what the player should do, which suggests that the board contents are all you need to know to render a decision, Go is not a Markov game, due to the rule of *ko*, which prevents a board from having the exact same arrangement of stones on successive turns.

When you peer closer at what it means to be a Markov game, the distinction between Markov games and non-Markov games gets even more slippery. For instance, consider Rock Paper Scissors. Is it a Markov game? It might seem so, because the previous rounds don't affect the play of the current round: the rules are all still the same. Yet the *strategy* of the game involves analyzing the history of your opponent.

When the history is itself part of the board, the distinction blurs even more. Consider Mastermind, where a player inserts pegs into the lowest level of the board on turn 1, level 2 on turn 2, and so on up to the top of the board. On turn N, the pegs in levels 1 to N-1 are there *only to provide a record of the game history*. Yet you can look at the board and immediately have all the information you need to decide the perfect move — clearly a Markov game. The play of Mastermind would be exactly the same if the player had only one level available to him, which he would fill with pegs and then remove when his next turn began. He'd just have to remember all of his previous moves — clearly a non-Markov game.

Classifying games into Markov games and non-Markov games seems clear-cut when you consider the idealized games of theoretical computer science, but it falls down upon closer scrutiny.

Condon's probabilistic game automata view history as something that a particular game possesses. But EGGG takes a broader view with a *global history*, defined in Chapter 6, *Connect the Bots: Networking*.

The Six Types Of Synchrony

In this section, we continue our emphasis on the process of the game, rather than its structure. Once we know whether the game is time critical, whether it is frenetic, and whether it has to remember everything it's done, we can turn to the actual game construction.

In a game like bridge or chess or tic tac toe, the turns alternate. If you assume that

the game is not networked (networked games will be explored in Chapter 6), you could envision players taking turns playing the game on a single computer, with one player using the keyboard and then handing it off to the next player; the game program then has to rotate the board so that it is presented from the appropriate viewpoint for each player.

But there are some games for which this doesn't make sense. Consider Diplomacy, in which everyone decides where to move their armies and navies, and submit their moves simultaneously. Or consider Spit, a card game where two players are racing to place their cards on top of a central pile. Even one-person games like Doom, which have no discrete moves at all, aren't without synchrony: the monsters move at the same time you do.

We classify the synchrony of a game as follows: If all players move simultaneously, we call that *complete synchronization.* If only a subset of the players (which we'll call a team) move simultaneously, that is *partial synchronization.* And if the game is sequential, we call that *zero synchronization.*

If only one thing happens at a time, we call that a sequential game. Poker is sequential, because only one person at a time can move. In sequential games, the turns might alternate between two players, as they do in chess, or they might circulate in a particular order, as in the clockwise betting rotation of poker. They might be arbitrary but fixed, as they are in Monopoly. They might have no order at all, as in quiz shows or Charades: you "move" whenever you like.

Here, then, are the different types of synchrony. We have no categorization for simple alternation, since alternation between two players can be viewed as a degenerate form of clockwise or counterclockwise rotation. (Below, we refer to counterclockwise rotation by its more esoteric and evocative name, "widdershins", so that we can have a unique one-letter descriptor for the six types of synchrony.

	Synchronization
Total synchronization	st
Partial synchronization	sp
Sequential movement (rotation clockwise)	sc
Sequential movement (rotation widdershins)	sw
Sequential movement (other ordering)	so
Sequential movement (random)	sr

Games that are not sequential usually require multiple strands of execution: either the threads provided by many operating systems, or distributed computing, or a fork/exec computation model. At the very least, they will require *emulating* multiple

threads of execution.

Games that are sequential can use the ordering to loop through the players in the appropriate order.

Now we can classify the eight games of our dissertation according to their synchrony as well as the previous time and history criteria.

	Frenetic	History	Synchrony
Tic tac toe			sc
Chess	ft	h	sc
Poker		h	sc
Crosswords			sr
Tetris	ff		sr
Rock Paper Scissors	ft	h	st
Deducto		h	sr
Mammon	ft		sp

Movement

Now that we've classified the sequence of play in the game, we can turn to what is being synchronized: the moves. Our taxonomy defines several components to movement: the move, the phase, the turn, the round, and the step. Few games have all five components. (Magic: The Gathering is one such game.) Each of these components is denoted with a single letter following m: A game with moves is denoted mm. A game with moves and phases is denoted mmp.

To a first approxmation, these components form a hierarchy: a round can consist of multiple turns; a turn can consist of several moves; a move can consist of multiple phases; and a phase can consist of multiple steps. However, there is not strictly adhered to: in Nine-Men's Morris, phases consist of multiple moves, and is denoted mpm to indicate that moves constitute a phase instead of the other way around.

We now discuss each of the five components of movement.

The Move

In games like tic tac toe and chess, the notion of a move is intuitive, and we hear it in "X should move here", or "Black moved his king's pawn forward two squares." Yet even these moves are slightly different. In tic tac toe, the movement is really *placement*: a player chooses which square to occupy, and by so doing occupies it. In chess, choosing a square is not enough. You have to choose two squares: a source and a destination. In Diplomacy, even the source and destination of a move aren't enough.

In a crossword, a move is writing a letter in a square. But a crossword move can also be *erasing* a letter in a square. Thus, moves aren't always steps toward a goal, and they don't always add information to the game state.

At this point, the astute reader will note that we've corrupted the intuitive definition of move — casual users would never call filling in a crossword grid as a series of moves. That's okay, because here we are only talking about the abstractions inside EGGG. The casual user never sees these abstractions, because they can use the more familiar terminology when describing games. EGGG maps the familiar terms to the internal abstractions described in this chapter. Multiple words in the EGGG vocabulary trigger the same abstraction; the choice of abstraction depends on their context.

The Phase

The picture becomes further complicated when we consider the moves of other games. People don't speak of making "moves" in poker, but it's not difficult to broaden the concept of moves to include what it is that players do in poker: ante, bet, call, raise, fold, and discard. Now we have several different types of moves, and each is valid only at particular times during the game. Likewise, deciding who goes first in Monopoly or backgammon or billiards implies a sequence of actions entirely different from the regular play of the game.

We call these distinct times *phases*, and if a game has phases, it is represented as `mp`. The movement letters can be concatenated; a game with both moves and phases is denoted `mmp` if the phases are part of the move (as in poker) or `mpm` if the moves are part of the phase (as in backgammon).

The Turn

Where does a game like Progressive Chess fit into this categorization? In Progressive Chess, the first player takes one move; the second player takes two moves; the first player take three moves, and so on. Games typically last about seven or eight turns [Burns 98].

When a sequence of moves are made at a time, we call that a *turn*, and designate it `mt`. In Progressive Chess, EGGG needs to know not just that moves are combined into turns; it also needs to know how. Unfortunately, this requires that you burrow down into the underlying representation and specify how: with code. When a component of the taxonomy requires writing code, we designate it with an additional opening curly brace, `{`.

In games like that have tricks, like bridge and pinochle, the moves are individuals playing cards, and the turns are the tricks.

The Round

When a game consists of a series of short, independent games, each with its own winner, it is said to consist of *rounds*. Poker has rounds, as does Rock Paper Scissors. In both of these games, there is no state kept between independent rounds. You can play one round of poker, but in practice many rounds are played at a sitting. A game with rounds consisting of turns would be represented `mrt`.

The Step

Some games have complicated moves that require compound actions. If the actions are integral to the play of the game, they are the *moves* and *phases* that we discussed earlier. If they are merely customary or superficial, they are called *steps*. Rapping your knuckle on the table to signify the end of a phase in Magic, or sorting the cards in your hand, or saying "Check" in chess: these are all inconsequential actions that are nevertheless important for any faithful rendition of the game by a computer.

Here is a summary of the five components of movement:

	Movement
Move	mm
Phase	mp
Turn	mt
Round	mr
Step	ms

And here are the categorization strings for our eight games:

	Frenetic	History	Synchrony	Movement
Tic tac toe			sc	mm
Chess	ft	h	sc	mms
Poker		h	sc	mrmp
Crosswords			sr	mm
Tetris	ff		sr	mtm
Rock Paper Scissors	ft	h	st	mrm
Deducto		h	sr	mrpms
Mammon	ft		sp	mmp

We've now classified when players act, and the sequence of actions. But we haven't talked about what it is the players are acting upon. We turn to that in the next section, *Tangibles*.

Tangibles

Descriptions of games from the player's perspective typically begin with the look and feel of the tangible objects: board, pieces, cards, and so on. Or they begin with what the user first sees in the game. "You have a sideline view of a basketball court, and you control one character at a time in a game of two-on-two."

This is precisely the way you want to describe a game to a person, because it helps them visualize the game. Pictures first, rules and buttons and controls later. However, we have delayed introducing tangible objects into our taxonomy for a reason: they actually aren't that important to the design of a game. You can play chess with cards instead of solid pieces; you can play poker with chess pieces instead of cards. They are not intrinsic to the game play.

Some tangible objects are important to render properly, however. You can play tic tac toe on a chessboard, but you can't play chess on a tic tac toe board: appearance matters. In this section we discuss the different tangible objects that a game designer needs to sprinkle on the screen; in Chapter 5 we will discuss how EGGG renders them. We call the spatial objects "tangibles" to avoid confusion with the "objects" of object-oriented programming.

First, we turn to the board.

The Board

In our taxonomy we use the term "board" to mean any playing surface. It might be a Scrabble board, or a poker table, or a virtual baseball field. In our taxonomy, we use `b` to denote a board.

In our taxonomy, every game has a board, but the board might be invisible. For instance, text adventures and role playing games have an invisible board. Why not just say that these games have no board? Because all computer games inevitably have to be rendered on a screen, and having a board abstraction is necessary so that the game generator knows how to depict whatever text it needs to show. Invisible boards are denoted `bi`.

There is a large variety of visible boards, and we define the major subtypes now.

The Grid

[Burns 98] classifies chess as a "board game" and tic tac toe as a "pencil and paper game". But this is clearly the wrong way to organize games from the perspective of a computer programmer, because both chess and tic tac toe are board games. More specifically, they are grid games, and we denote them with `bg`.

Chess, checkers, tic tac toe, and crosswords all played on square grids of squares. That is, the grid itself is square, and the constitutent elements are square as well. We call the constituent elements "squares" no matter what their shape.

There are other types of grids; the mathematician Piet Hein invented Hex in the 1940's, which is played on either a hexagonal or a triangular grid. Like the familiar square grids, the grids of Hex are tessellated: the constituent shapes completely fill the grid. Only three equilateral shapes can tessellate a grid: squares, triangles, and hexagons.

The board of Chinese checkers is a series of tessellated triangles, but the overall shape is a star. For our purposes, we consider that a grid where many of the triangles are off limits.

Games like Scrabble are played on grids, but the individual squares have different meanings. That's okay; we're just concerned with the architecture of the spaces on the board, and not their meaning.

Monopoly (still banned in Cuba, China, and North Korea, even though the original version was developed as a tool of *anti*-capitalist propaganda) is played on the border of a grid. We consider that a grid too.

So we have three different shapes of grids (square, hexagonal, and triangular) and one bit to set if the border of the grid is used. The meanings of the squares isn't part of our taxonomy, so Scrabble is just a `bgs` game, for board-grid-square. If Scrabble

were just played on the border, we'd add a `b` to the end: `bgsb`.

We call the component elements of a grid *squares*, even if they are triangles or hexagons.

The Graph

In games like chess and hex and Chinese checkers and Go, distances are proportional. That is, adjacency on the grid implies adjacency in game play; the geometry of the grid allows a computer to infer how far apart squares must be. However, in games like Chutes and Ladders, certain squares can "teleport" you to other squares. Even though the board is shaped and packed like a grid, it's not really a grid, because distances aren't linear. The board is best viewed as a series of nodes with arbitrary connections between them — what computer scientists call a *graph*.

Chutes and Ladders is a *directed graph*, because the connections are one-way. We can model games like Diplomacy and Risk as *undirected graphs*, because the connections are two-way. In general, most war games can be modeled as undirected graphs, since geography is two way: if you can cross from Austria into Germany, you can cross from Germany to Austria. In mathematical terms, adjacency is symmetric in games like Risk but asymmetric in games like Chutes and Ladders. Once you slide down a chute, you can't climb back up.

If a game has a board that is best represented as a directed graph, we call that `bdg`. Undirected graphs are `bugs`.

The Canvas

To play a card game, all you need is the cards and a flat surface. The surface can be blank, or mottled, or a grid. It doesn't matter what the surface looks like. We call that a *canvas*.

A canvas is not the same as an invisible board. An invisible board is used when a board has no physical meaning, as in a text adventure or role playing game. A canvas is used when there must be a playing surface, but it doesn't matter what the playing surface is.

Topology

Most board games are two-dimensional. We call text-based games zero-dimensional, because they have no spatial meaning. Text adventures and MUDs (Multi-User Dungeons) might seem to be a counterexample, because there is a spatial architecture that the players inhabit. However, our taxonomy is from the perspective of the game developer, not the game player.

Some games are nominally three dimensional, but the third dimension adds nothing substantive. In card games, some cards will be on top of others, but that can be easily represented with two dimensions. Connect Four is a three dimensional game that relies on gravity to slide checkers into position, but it can easily be represented with a straightforward two-dimensional grid.

There are some true three-dimensional games, like 3D tic tac toe and Jenga. And there are some games in which the topology of the board is important. Defender and Pitfall are situated in spaces that are topological cylinders: your up and down movement is bounded, but your left and right movement is not; we call that a *tube*, and a game with the topology of a two-dimensional tube is denoted `t2t`. Asteroids is a two-dimensional doughnut (torus), and we denote that `t2d`. Games that are played in or on spheres use `b`, for ball. A `t3b` game is played on a sphere; a `t2b` is played on a two-dimensional sphere, otherwise known as a circle.

Board Summary

Here are the types of boards recognized by our taxonomy:

	Board
Invisible	bi
Grid of squares	bgs
Grid of triangles	bgt
Grid of hexagons	bgh
Directed graph	bdg
Undirected graph	bug
Canvas	bc

Here are some sample topologies:

	Topology

Text-based games	t0
Conventional games	t2
Three-dimensional games	t3
Two-dimensional tube	t2t
Two-dimensional doughnut	t2d
Four-dimensional ball	t4b

We can categorize our eight games (and a few extras) as follows:

	Frenetic	History	Synchrony	Movement	Board	Topology
Tic tac toe			sc	mm	bgs	t2
Chess	ft	h	sc	mms	bgs	t2
Poker		h	sc	mrmp	bc	t2
Crosswords			sr	mm	bgs	t2
Tetris	ff		sr	mtm	bgs	t2
Rock Paper Scissors	ft	h	st	mrm	bc	t1
Deducto		h	sr	mrpms	bgs	t2
Mammon	t		sp	mmp	bi	t0
Chinese checkers			sc	mm	bgt	t2
Chutes and Ladders			sc	mm	bdg	t2
Diplomacy	ft		st	mptm	bug	t2

Now that we know what the board looks like, we examine what's placed on it: the pieces.

The Piece

Most classic games have small objects that are picked up and moved around. We call these *pieces*, whether they are checkers, chess pieces, cards, stones, or tiles.

In arcade games like Tetris, you control the piece. In an arcade game like Pac Man or Doom, you *are* the piece. When the player projects himself onto the screen, identifying with the protagonist and cringing when he dies, a much more immersive game experience results. (Immersion isn't always desirable; when you're playing solitaire, you don't want to *be* the cards, perishing in screams when you end an unsuccessful game.)

For our taxonomy, this is all irrelevant anyway. Whether the player is the piece, or merely controls it, doesn't matter to how the game is programmed.

Sometimes the pieces themselves have state. In Othello and Shogi, a piece has two sides, and which side is up has a great deal of meaning. Othello and Shogi pieces have one bit of state. A Trivial Pursuit piece has six bits of state, corresponding to which of the six wedges it contains. A piece with state is denoted `ps`.

Some pieces have artistic meaning that is essential to the look and feel of the game but unimportant to the play; computer versions of the games need to give them the appropriate appearance. Monopoly is the canonical example; the game could refer to the players by number or color, but instead uses the tycoon, the thimble, and so on. A piece which is best represented with art rather than a simple geometric shape is denoted `pa`.

Some games have two very different kind of pieces. Poker has both cards and, sometimes, chips. The chips could just as well be real money, but a faithful rendition of the game must treat them as tangible objects that can be stacked and moved around. When a game has multiple pieces, our categorization strings separate them with semicolons. Poker's piece categorization string is `pac;nc`. The `ac` refers to the cards, which require art and color. The `nc` refers to the chips, which stand for a number (the value of the chip) and depend on color as well.

Even a game like Rock Paper Scissors has "pieces" when it's rendered on a computer screen — they just don't move, simply appearing and disappearing as the game is played. Rock Paper Scissors is a `pa` game. The pieces of a crossword are the letters, a `pl` game.

In games like chess and Scrabble, no more than one piece can be on a square. In Monopoly, multiple pieces can be on a square. In Tetris, the active piece constitutes many squares. To denote whether pieces and board locations have a one-to-one, many-to-one, or one-to-many relationship, we use the mathematical terms for these mappings: *bijection*, *surjection*, and *injection*. The relationship is assumed to be a bijection unless the categorization string implies otherwise.

Here are some examples of piece categorization strings:

	Piece
Plain piece	p
Piece, colored	pc
Piece, letter	pl
Piece, artistic	pa
Piece, artistic and colored	pac
Piece, with state and color	psc
Piece, many of which can occupy a board location	pb
Two pieces, one with state and number, another with art and color	psn;ac

And here are the piece strings of the twelve games shown earlier, plus Trivial Pursuit:

	Frenetic	History	Synchrony	Movement	Board	Topology	Piece
Tic tac toe			sc	mm	bgs	t2	pc
Chess	ft	h	sc	mms	bgs	t2	pac
Poker		h	sc	mrmp	bc	t2	pac;nc
Crosswords			sr	mm	bgs	t2	pl
Tetris	ff		sr	mtm	bgs	t2	pi
Rock Paper Scissors	ft	h	st	mrm	bc	t1	pa
Deducto		h	sr	mrpms	bgs	t2	pc
Mammon	ft		sp	mmp	bi	t0	
Chinese checkers			sc	mm	bgt	t2	pc
Chutes and Ladders			sc	mm	bdg	t2	pc
Diplomacy	ft		st	mptm	bug	t2	pac
Trivial Pursuit			sc	mm	bug	t2	psc

More About Color

In all of the abstractions EGGG uses, color is the most inappropriately named. That's because color is almost exclusively used as an identifying attribute, to distinguish between players. The red and black of checkers or backgammon could be replaced by `1` and `2` or `X` and `O`. Similarly, the letters in tic tac toe or the numbers in Minesweeper could just as easily be represented by colors.

We considered calling any attribute that serves merely to identify a player a *name*, and decided against it, because "name" has too many other meanings. It would be strange to say that a "black knight" is a named piece, not because it is a knight, but because it is black. The word "color" is much more evocative, and even if it sometimes evokes the wrong concept, we have chosen to retain it. That's why tic tac toe is classified as a `pc` game above.

In developing EGGG, two games were designed that made use of color as a meaningful attribute instead of just an identifier. The games are RGB Deducto and Color Deducto, and are described in the next chapter.

Compartments

Not all pieces are on the board at the same time, and in this section, we discuss *compartments*: containers for pieces that aren't on the board: hands and bags. When rendering a game, the computer needs to know what to show to every player, what to show only to some players, and what to conceal from everyone.

The Hand

Poker is a game of partial information: you reveal some information to the other players, and you keep some to yourself by concealing them in your *hand.* Scrabble operates the same way, even though the tiles aren't physically in your hand, and aren't called a hand: the wooden barrier that hides your tiles from prying eyes is a hand as well, in our taxonomy. A hand is a compartment, `ch`, that only a single player can see.

Hands can be finite, `chf`, as in the five cards that constitute a poker hand. Or they can be infinite, `chi`. The particular size of the hand is not part of our taxonomy, although obviously the game generator needs to know it, just like the generator needs to know the colors of pieces or the artwork on cards. Those are part of the game description; they just aren't part of the vital information needed to classify games.

Furthermore, a player's hand can conceal information from himself as well. In the

trading card game Magic, the hand has two components. The first component is a deck of cards that the player has chosen, but shuffled so that he doesn't know the order. The second component are the cards that he actually holds in his hand. His opponents have no knowledge of either component; he has total knowledge of one component and only partial knowledge of the other.

The Bag

In Scrabble, there is a *bag* of pieces (tiles) from which players replenish their hands after every play. The bag is a concealed set of pieces just like the hand, but it is unordered and concealed from everyone.

Poker has a bag, too: the deck. But poker's deck has one subtle difference from Scrabble's bag: in a deck, the cards have a fixed but unknown order. Drawing from the deck doesn't change the order, whereas every draw from the bag randomizes the bag.

If this seems like a pedantic distinction, consider the passed cards of Anaconda or competitive Tetris: in each of these games, there is a set of pieces, with the order determined by another player as part of the strategy of the game.

Incorporating this distinction also allows EGGG to model the inefficiencies of various shuffling methods. There is a shuffle method that performs a true random shuffling, but also "riffle shuffle" and "cut" methods that more accurately model how real decks of cards are perturbed by human hands.

Bags are types of hands, so infinite bags are denoted `cbi` and finite bags are denoted `cbf`.

	Frenetic	History	Synch	Move	Board	Topo.	Piece	Compart.
Tic tac toe			sc	mm	bgs	t2	pc	
Chess	ft	h	sc	mm	bgs	t2	pac	
Poker		h	sc	mrmp	bc	t2	pac;nc	chf;bf
Crosswords			sr	mm	bgs	t2	pl	
Tetris	ff		sr	mtm	bgs	t2	pi	
Rock Paper Scissors	ft	h	st	mrm	bc	t1	pa	chf
Deducto		h	sr	mrpms	bgs	t2	pc	
Mammon	ft		sp	mmp	bi	t0		chi

Chinese checkers			sc	mm	bgt	t2	pc	
Chutes and Ladders			sc	mm	bdg	t2	pc	
Diplomacy	ft		st	mptm	bug	t2	pac	
Trivial Pursuit			sc	mm	bug	t2	psc	

Abstract Facets of Play

In this section, we turn from the tangible objects of boards and pieces to four intangible aspects of game play: genre, ending, and communication.

The Genre

Some games have themes, like war games, or role playing games, or the futuristic alien battles of Doom and Quake. These are richly-expressed genres, and the themes pervade the artistic expression of the game. You could take the play of a game like Diplomacy or Doom, or the family-oriented utopia of Life (the one by Hasbro, not the cellular automata game by Conway), and render it with bland geometric shapes, but people wouldn't recognize it. These games have a genre that is expressed richly, and so we denote it `gr`.

Many games have themes, but they're stripped down. Chess is a game of war, but only nominally. A chess game rendered with pieces that bleed, or a variation that involved the more familiar aspects of war games like ambushes or resource management, would seem strange indeed. [Barnes 98] categorizes backgammon and Chinese checkers as race games, but the trappings of actual flesh-and-blood (or engine-and-gas) races are missing. These are *weakly* expressed genres, and we denote that `gw`. (However, unusual variations of a game can impart a richly-expressed theme where there was none previously, such as the Jews versus Catholics chess game on display at the Corning Glass Museum in Corning, New York.)

Other games are divorced from any theme whatsoever, or are so far removed from whatever genre they might have once had that renderings of the game can safely ignore the genre. All card games fit this category; even the children's card game of War is so far removed from what is traditionally called a war game that we can say that no genre exists. Gambling games typically have no genre as well. We call these `gn` games.

Information

Information communicated between players can take many forms. Here, we refer to messages passed between players that is over and above the information conveyed by moves. Chess has no communication; players speak with their pieces. Simple bridge has no communication, because bids are moves. Precision bidding and the Blackwood Convention, on the other hand, are out-of-band communication, because the bids are meant to tell your partner what cards you have rather than what tricks you plan to take. Games like Pictionary and Netrek and Dungeons & Dragons involve cooperative communication between players; we denote this `ic`.

In poker, the communication isn't friendly. Poker players aim to reveal as little information as possible; or if they do reveal information, they want it to be misleading. We denote this unfriendly communication `iu`.

Referees

Games like Kriegspiel, Dungeons and Dragons, Black Box, and Mastermind have referees: one player who knows more than the others — and because of his knowledge acts as an umpire rather than a competitor. If a game has a referee, we denote that `r`.

Endings

Finally, we turn to the ending of the game. It's tempting to call this a *goal*; artificial intelligence and common parlance suggest that would be the best term. We chose *ending* instead. Goals are something you strive for — a win condition. But consider a game like Tetris. You cannot win; you just play until you lose. You could say the goal is to maximize the number of points, or to get to the highest level, but those goals are independent of when the game ends. The programmer needs to know when to end the game: the end condition. We denote endings with `e`.

If the end condition is synonymous with the player's goal, as it is in chess, we call that an `es` game. If the end condition is antonymous with the player's goal, that's an `ea` game. And if the end condition has nothing to do with player goals, we denote that `en`. Note that a game can have several end conditions; when there are, we separate them with semicolons in the category string.

We now have a reasonably complete taxonomy of games:

	Fren	Hist	Synch	Move	Board	Topo	Piece	Compart	Genre	Info	Ref	End
Tic tac toe			sc	mm	bgs	t2	pc		gn			es

Chess	ft	h	sc	mm	bgs	t2	pac		gw			es;n
Poker		h	sc	mrmp	bc	t2	pac;nc	chf;bf	gn	iu		es
Crosswords			sr	mm	bgs	t2	pl		gn			es
Tetris	ff		sr	mtm	bgs	t2	pi		gn			ea
Rock Paper Scissors	ft	h	st	mrm	bc	t1	pa	chf	gn	iu		es
Deducto		h	sr	mrpms	bgs	t2	pc		gn			es
Mammon	ft		sp	mmp	bi	t0		chi	gr	if;u	r	en
Chinese checkers			sc	mm	bgt	t2	pc		gn			es
Chutes and Ladders			sc	mm	bdg	t2	pc		gw			es
Diplomacy	ft		st	mptm	bug	t2	pac		gr	ic;u		es
Trivial Pursuit			sc	mm	bug	t2	psc		gn			es

In the next chapter, we turn to how these categorizations are used by the EGGG engine to generate games.

Chapter 3: The Physiology of EGGG

> *Programs...that write programs...are the happiest programs in the world.*
>
> Andrew Hume

We now turn to a demonstration of EGGG, a system that exploits the similarities between games to translate high-level game descriptions into fully functional game programs. In this chapter we discuss how EGGG accomplishes this transformation.

EGGG also generates computer opponents that can play games with users; the strategies that EGGG uses are discussed in the next chapter. Similarly, the graphic appearance of the games and networking aspects of game programs each merit their own chapters, and will be deferred until later.

First, we turn to the EGGG language, which is what game designers use to specify the rules of their game to EGGG. Next, we cover the software architecture of the EGGG engine, and then turn to some of EGGG's features, such as the automatic generation of documentation and instructions for game players.

Game Descriptions

To use EGGG, the first step is to create a game description—typically a page or half-page of ASCII text provided as a file with an `.egg` extension. In this section, we'll examine a game description line by line as a way of showing you what the EGGG language is (and isn't) capable of expressing.

Here is the game description for poker:

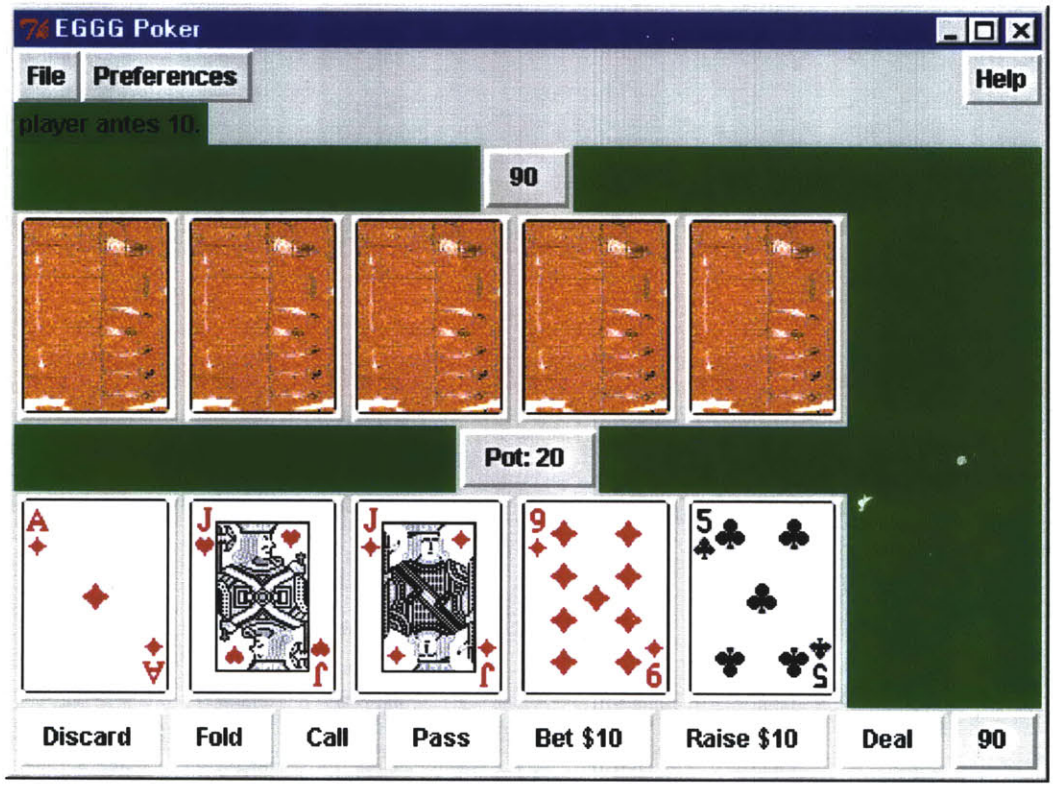
```
game is poker

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)
```

As you can see, game descriptions are a series of statements. The statements can appear in any order. The EGGG language intentionally has no flow control; that's why we call it a description and not a program.

A statement can be split along multiple lines, as long as lines after the first are indented with any whitespace. Comments can appear anywhere, and begin with #.

These seventeen 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.) Our poker game is shown below:



We'll now proceed through the poker description line by line.

```
game is poker
```

This serves only to name the game. It is optional; if this line isn't present, EGGG assigns a name based on the filename of the game description.

```
turns alternate clockwise
```

This tells EGGG that if there are more than two people playing, that the action of the game (dealing and betting, but EGGG doesn't know that yet) passes from player to player, clockwise.

When someone runs this program from the command line (on either Windows or Unix), EGGG assumes that there's only a single player. That's why it generates the screen you see above; the opponent is played by the computer.

```
Discard means player removes 0..3 cards or 4 cards if Ace
Fold means player loses
```

EGGG refrains from assigning meaning to any capitalized word; relying on the game designer to do that instead. That's what the game designer does in these two lines, defining the `Discard` and `Fold` actions. EGGG turns these definitions into subroutines in the generated game program.

Later on in the game description, we see `Discard(hand, N)`; because of the capitalization, EGGG treats this as a reference to the action defined above. `Discard(hand, N)` is then translated into a subroutine invocation.

What will the `Discard()` subroutine do? When it is invoked, it insists that the current `player` remove between zero and three cards, or four cards if the player has an Ace. Similarly, a `Fold()` subroutine is generated; when invoked with a player's name, that player is removed from further play this round.

`player` is lower case, which means that it has a special meaning for EGGG. EGGG knows what a player is; for instance, it assumes that when `turns alternate clockwise`, that the things being alternated are players.

`removes` is also lower case, and EGGG assumes that it is pieces that are being removed from the player's collection unless the designer specifies otherwise. Note that we say "collection" and not "hand" — as was described in the last chapter, "hand" has a particular meaning to EGGG, and EGGG can't yet infer that poker is a game that has hands.

That inference becomes possible as soon as EGGG encounters "cards", which triggers a number of behaviors by EGGG later.

There are two possible interpretations of the expression `removes 0..3 cards or 4 cards if Ace`, depending on the relative precedence of the `or` and `if` operators. Without knowledge of EGGG's precedence, it could mean `(removes 0..3 cards if Ace) or (remove 4 cards if Ace)`, or it could mean `(removes 0..3 cards) or (remove 4 cards if Ace)`. It means the latter; `if` binds tightly in EGGG.

How does EGGG know what an Ace is? It doesn't. `Ace` is capitalized, so EGGG assumes that it will either be defined elsewhere in the game description (it isn't) or that it will be defined in a *module*. Modules link names like "Ace" to images (a picture of a particular Ace) and ranks (asserting that it can be both below a deuce and above a King). None of this is known to EGGG as the game description is being parsed; the capitalization is enough to tell EGGG what code to generate for the `Discard()` subroutine.

EGGG does not distinguish between singular and plural cases. `player` and `players` are interpreted identically, as are `remove` and `removes` and `card` and `cards`. This tolerance allows users to write in grammatical English without requiring EGGG to verify grammatical correctness.

```
2..6 players
```

This statement tells EGGG how many players the game can support. Again, the range operator `..` is used. All EGGG games contain code that looks for the number of players on the command line, and networked games look for connections from other players. The sole effect of this line is as a check that terminates command-line programs if too many players are specified, and refuses additional connections if the maximum number of networked players are already present.

```
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)
```

The earlier `game is poker` named the game; this defines the game play. (EGGG determines how to interpret a `game is` statement by the number of words in the statement: if only one or two words follow `game is`, it's treated as the name of the game.) Here, the game play is a shuffle followed by a deal, a bet or a fold, a discard, another deal, another bet or fold, and a final comparison. Every word preceding a left parenthesis turns into a subroutine invocation, and the subroutine definition can originate from three places. It can originate from the game description itself, as in the `Discard()` subroutine shown earlier. It can be something that EGGG has particular knowledge about, such as the `bet()` and `compare()` subroutines. Finally, it can be something defined in a module bundled with EGGG. The `Deck` module defines subroutines for shuffling and dealing; it defines a `Deck` object with `shuffle()` and `deal()` methods, as well as the ranking that defined Ace. In the Deck module, `Ace()` is actually a subroutine; when `Ace()` is called with no arguments, it returns true if the current player has an ace.

```
StraightFlush is (R, S) and (R-1, S) and (R-2, S)
      and (R-3, S) and (R-4, S)
```

This line defines the condition that a set of cards must meet to contain a stright flush. EGGG doesn't yet know that a straight flush is a good hand; that comes later. `StraightFlush is` triggers a subroutine definition, just like `game is` earlier. That subroutine loops through all of the cards in the player's hand.

In each iteration of the loop, it sets a variable, `R`, to the rank of the current card. It sets `S` to the suit of the hand. Subsequent loops generated by EGGG test subsequent

cards. If the rank of the second card isn't one less than the rank of the first card and with the same suit, the loop exits prematurely, and no straight flush will be found.

To be precise, the loop will begin another iteration through the cards, starting with the second card, as soon as that first noncompliant card is found. The subroutine doesn't know that a `StraightFlush` requires that all the cards be part of the straight flush; it only knows that *five* cards must be part of the straight flush. So on the next loop iteration it will look for a straight flush starting with the second card and ending with the sixth. It will fail, move on to the third card, and so on until there are no cards left in the hand.

This is exactly the sort of inefficiency that you don't want in a program dedicated to playing poker, but that you *do* want in a generic game-playing program. Perhaps someone will devise a variation that involves more than five cards. Someone might even devise a variation where different players have different numbers of cards at different times during the game, and in those cases it's important to distinguish between a straight flush of five cards and a straight flush of *all* cards.

```
FourKind is (R, s) and (R, s) and (R, s) and (R, s)
Flush is (r, S) and (r, S) and (r, S) and (r, S) and (r, S)
```

These lines are similar; in each, the ordered pair representing each hand has one lower case letter and one upper case letter. As we saw, upper case letters have to match particular piece attributes. Lower case letters do not. The `s` and `r` letters above are mere placeholders.

```
FullHouse is (R, s) and (R, s) and (R, s) and (Q, s)
          and (Q, s)
TwoPair is (R, s) and (R, s) and (Q, s) and (Q, s)
```

`Q` is like `R`, only different; in the definition for `FullHouse`, `Q` is used to distinguish the second rank (the rank of the pair of cards) from the first rank.

As you can see, each piece (card) is represented as a coordinate pair. EGGG actually represents all pieces, not just cards, as tuples (arrays).

The values of each coordinate can be a variable (`R`, `Q`, or `S` above), a placeholder (`r` or `s`), or a literal value. Traditional poker has no literal values in its rankings — that is, there aren't any hands that require you to have some particular card. Pinochle attributes special meaning to a Queen of Spades / Jack of Diamonds combination, which would be represented like this:

```
Pinochle is ("Queen", "Spades") and ("Jack", "Diamonds")
```

The remaining hand definitions should be self-explanatory.

```
hands are [StraightFlush, FourKind, FullHouse, Flush,
           Straight, ThreeKind, TwoPair, Pair, HighCard]
```

This tells EGGG what the possible hands are — and more importantly, their order. `StraightFlush` is the highest, because it is first.

Note that poker hands overlap: a straight flush is both a straight and a flush, three of a kind is also a pair, and every hand also has a high card. How does EGGG know to identify four aces as a `FourKind` and not as a `TwoPair`? After all, any hand with four of a kind also satisfies our definition of two pair. Because EGGG searches for the hands in order, the `FourKind()` subroutine will be invoked first, return a true value, and preclude `TwoPair()` from ever being called.

This is another of the similarities behind games built into the design of EGGG: in games, rare piece combinations are worth more than common combinations. Code generated by EGGG will, by default, check for the best outcomes first. This assumption won't work for all games, but it does for the vast majority of them. Game designers can always override EGGG's assumptions in the game description by specifying the play of the game with greater precision. This is one of the design principles underlying EGGG: with apologies to Alan Kay, our goal is to make easy games easy, and hard games possible.

```
hand is five cards
```

The first two words of this statement are parsed exactly the same as the first two words of the previous statement: since EGGG ignores pluralizations, `hand is` is interpreted identically to `hands are`. However, the words that follow trigger different behaviors. In the `hands are` statement discussed previously, the game designer provided a list of hands, each with a definition elsewhere in the game description. `hand is five cards` tells EGGG that the game has hands (which makes EGGG conceal the computer's hand from the player) and how many cards each should contain. To specify that a game has hands without specifying how many cards the hand should contain, the designer could have written `game has hands`.

Numbers can be either spelled out ("five"), or provided as numerals ("5"); this statement could also have been written `hand is 5 cards`.

```
goal is highest(hand)
```

This final line of the game description tells EGGG what each player should be attempting to do. This serves two purposes: it serves to generate a strategy for the computer opponent (discussed in the next chapter) and it tells EGGG how to decide the winner once the round is over.

How Does EGGG Know That Poker Is A Game Of Rounds?

People 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 rounds. Without knowing that

poker consists of rounds, EGGG would display `GAME OVER` at the end of a poker game, and empty the player's coffers if the player chose to play again.

This is a rather abstract concept to explain to a game designer, so EGGG infers whether a game has rounds without the designer having to specify it explicitly. Games with concealment usually have rounds, so the presence of `hand` in the game description file suggests that the game may be played multiple times at a sitting. But this rule is not ironclad. Scrabble, Stratego, and Battleship are games of concealment, but typically aren't played over and over.

The best metric for determining whether a game has rounds is the average length of time that it takes to play. The shorter the game, the more likely that it's played more than once at a time. However, EGGG has no way to infer how long a game takes merely from the game description.

Looking closer, we can see that these are really one and the same rule. The reason that games with concealment (or games of *incomplete information*, to use Condon's classification scheme) are more likely to employ rounds is because they take less time, and the reason they take less time is that they require less analysis than games of perfect information.

EGGG's heuristic is that a game is not played in rounds if it fulfills any of these conditions:

- The game has levels (as in Deducto or Sokoban).
- 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 thirteen pieces per side.
- The game is played on a grid with more than 25 squares.

For games that are played in rounds, the score (or money tally) is kept from round to round.

We considered including another trait in this heuristic: whether the game has unequal roles for players. In games like poker, the deal shifts from player to player; in games like MasterMind or Black Box or Charades, there is one person who knows a secret, and other players try to guess. In such games, there's usually a notion of rounds so that other players can take turns knowing the secret. However, games like Dungeons and Dragons or Kriegspiel have referees even though the individual games take too long for rounds. MasterMind and Black Box and Charades all have concealment (hands), so the above heuristic works for them.

Furthermore, note that the presence of random elements in the game is not used to

determine whether a game has rounds. It's not whether you know what's going to happen, but how much you can predict what will happen from the information provided to you. Another way to say this is that the distinction between "games of skill" and "games of chance" is artificial. Yahtzee is both; being good at Yahtzee means being good at probability theory, whether the player is conscious of his knowledge or not.

How Can EGGG Assume That The Cards Are In Descending Order?

Here is poker's definition for a high card (the lowest possible hand, where the highest ranking card determines its strength).

```
HighCard is (R, s)
```

How does EGGG know *which* card is the high card? The `HighCard` subroutine will verify that any hand satisfies this definition as soon as it inspects the first card, without regard to whether there is a higher card elsewhere in the hand. Since the ranking subroutine return not just a true or false value, but a score indicating how good the hand is, it's important for `Highcard` to identify the highest card of the hand for tie breakers.

Earlier, we saw that EGGG searches for hands from the best to the worst; this is an example of a general principle: EGGG does *everything* from best to worst. After hands are dealt, EGGG sorts them from highest to lowest. When captured chess pieces are shown off to the side, EGGG shows the queen first and the pawns last.

EGGG has a generic sorting algorithm for pieces (remember, cards are pieces too) that orders rare, valuable, or powerful pieces first. When it doesn't know the rarity or power, it estimates them. Those estimates are used for the generic static evaluator described in the next chapter, and the estimation process is described further there.

This is important for graphic layout, too: when there is a limited amount of screen real estate, you want to make sure that the players can see the most significant pieces. Even if the pieces are displayed in some area with scroll bars, you want to most important information to be not front and center, but top and left. (This is gaming's answer to the inverted pyramid scheme of news articles.)

How Did EGGG Know To Generate All Those Buttons?

Notice the seven buttons generated by EGGG: `Discard`, `Fold`, `Call`, `Pass`, `Bet $10`, `Raise $10`, `Deal`. Only `Discard` and `Fold` are defined explicitly in the game description. `Deal` is mentioned, and defined in EGGG's `Deck` module, described later in this chapter.

As for the other four buttons (`Call`, `Pass`, `Bet $10`, and `Raise $10`), EGGG scanned the game description and found the expression `bet(money)`. The `bet` triggers five options for players: offering a bet, declining an offer, declining to offer, meeting an offer, and trumping an offer. These correspond to the four buttons, and `Fold`. EGGG chose the names of the buttons because of hardcoded rules about card games; for non-card games it calls those actions `Challenge`, `Reject`, `Pass`, `Accept`, and `Raise`.

The presence of `money` adds the dollar signs, and nothing more. Without `money`, the currency of betting is treated as abstract points. EGGG uses `$10` as a default betting amount.

There are also three labels: in the picture shown earlier, they read `90`, `Pot: 20`, and `90`. In any betting game, EGGG reserves a central area of the game canvas (or one side of a grid game) as an escrow area, or *pot*, where the wagered dollars, points, or pieces are displayed before players win them. (Incidentally, rummy acquired its name because players used to gamble for rum while playing.)

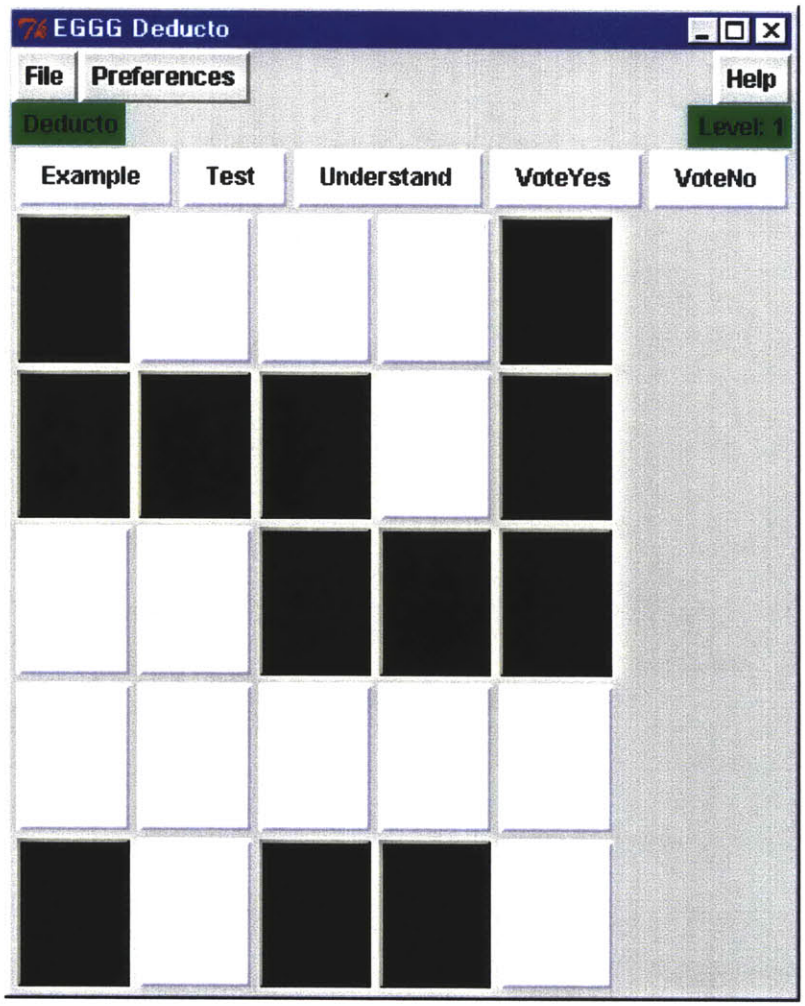
In the next section, we'll examine Deducto, a game where EGGG can't deduce what buttons to make.

A New Game: Deducto

EGGG is an attempt to showcase the similarities between games, and if the EGGG engine were only able to generate known games, that would still be ample evidence for the existence of those similarities.

But that wouldn't be very useful. EGGG is also good for generating entirely new games, and variations on games. In this section, we will develop a new game using EGGG, and later in the chapter we will create a variation on it.

The new game is called *Deducto*; it is a logic game played on a 5x5 grid. Here is what the game board looks like:

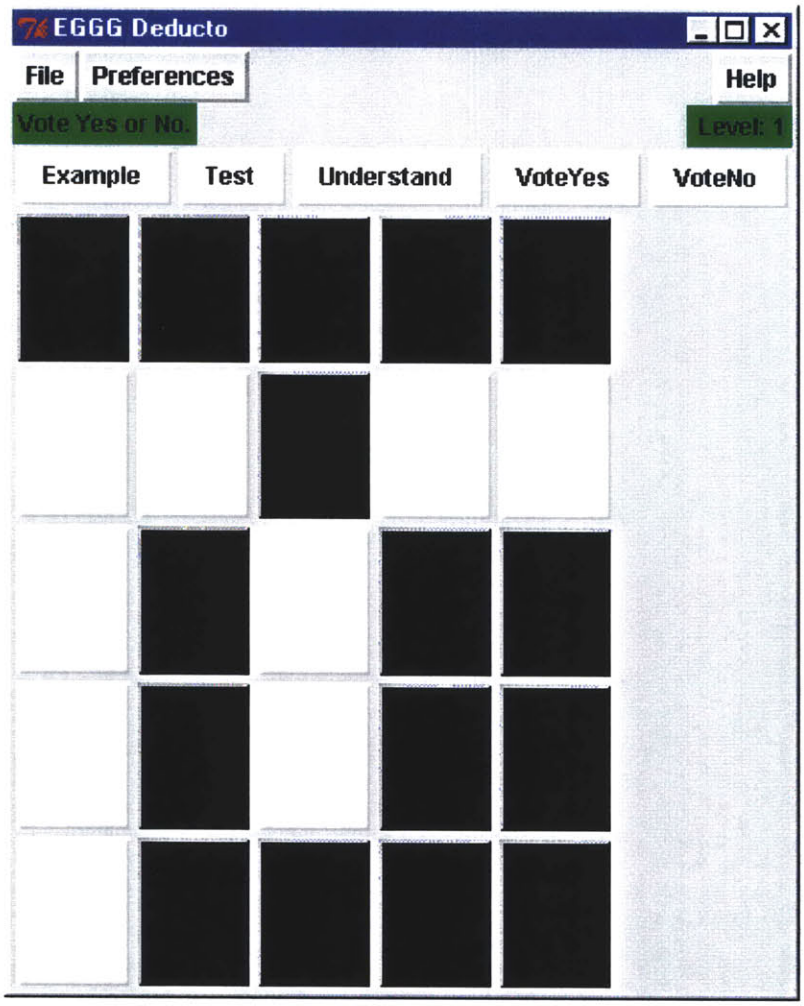


Each of the twenty-five squares is always black or white. You can see from the green label above and to the right of the buttons that this game has levels; we start with level 1.

Each level has a secret rule, and your goal is to determine the rule. The rule for level 1 is simple: the grid of squares must be at least half white. When the player presses the `Example` button, it generates a random grid satisfying the rule.

There are actually thirty buttons in this game — not five, as it might appear. The other twenty-five buttons are the grid squares themselves. When a player presses one of the grid squares, it toggles from black to white and back. Thus, a player can take a compliant grid — one that satisfies the current level's rule — and make it non-compliant, or vice versa. He can then test whether the board satisfied the rule by pressing the `Test` button.

When the player thinks he understands the rule, he presses `Understand`. The game then generates either a board that satisfies the rule or a board that does not, each with probability 50%. The board is displayed, and the player is prompted to `Vote Yes` (if the grid fits the pattern) or `Vote No` (if it doesn't):



If the player guesses correctly, his score increments. If he gets five in a row, he is deemed to have understood the level. The level increments, and the process repeats with a new rule.

In the words of one Media Laboratory ex-graduate student who would probably prefer to remain anonymous, it's "like you're a neural net." The former student explained that he didn't really understand the rule behind each level, but could still tell when a grid satisfied the rule or not.

We use Deducto to demonstrate some of the features — and more importantly, some of the limitations — of automated game generation. The limitation that Deducto highlights best is in the language necessary to express the rule behind each level. The rule might be simple, like counting the white squares on each level, or it might be complex. Maybe the number of squares on each level has to be prime; maybe the white squares have to form a letter, or a face; maybe the squares have to encode the current time. EGGG could restrict the patterns that appear, but that would constrain the game too much and run counter to the game's intention of testing players' analytical skills.

Obviously the EGGG language needs the expressiveness of a fully-featured programming language. Rather than develop yet another programming language, EGGG lets users burrow down into the language that it's implemented in: Perl. The loop constructs and operators are easy to learn for anyone experienced with programming, but it is at this point that creating a game stops being a matter of describing and becomes a matter of programming. Again, the underlying tenet of EGGG as an application is to make easy games easy, and hard games possible.

As with poker, we will go through the game description line by line, explaining EGGG's features along the way. Here is the game description:

```
game has Example button
game has Test button
game has Understand button
game has VoteYes button
game has VoteNo button
Example    means grid becomes Generate(level)
Test       means display(Tester(level))
Understand means display("Vote Yes or No.");
VoteYes    means Tester(level) and ++CORRECT or CORRECT = 0
VoteNo     means not Tester(level) and ++CORRECT or CORRECT = 0

goal is level(highest)
one player
5x5 grid
squares are white and black
click makes square toggle
board starts empty
Generate(1) is while (!Tester(1)) { grid[x][y] = flip ? 1 : 0 }
Generate(2) is { grid[x][y] = flip ? 1 : 0 }; grid[3][3] = 1
Generate(3) is while (!Tester(3)) { grid[x][y] = flip ? 1 : 0 }
Generate(4) is while (!Tester(4)) { grid[x][y] = flip ? 1 : 0 }
Generate(5) is while (!Tester(5)) { grid[x][y] = flip ? 1 : 0 }

AntiGenerate(1) is while (Tester(1)) { grid[x][y] = flip ?1:0 }
AntiGenerate(2) is { grid[x][y] = flip ?1:0 }; grid[3][3] = 0
AntiGenerate(3) is while (Tester(3)) { grid[x][y] = flip ?1:0 }
AntiGenerate(4) is while (Tester(4)) { grid[x][y] = flip ?1:0 }
AntiGenerate(5) is while (Tester(5)) { grid[x][y] = flip ?1:0 }

Tester(1) is Tot++ if grid[x][y]; return Tot > 12
Tester(2) is return grid[3][3]
Tester(3) is return 1 if grid[3][y] && (grid[1][y]
          && grid[2][y] or grid[4][y] && grid[5][y]); return 0

# This one is tricky.  Tester(4) is true if there's an "on" square
in each row.
Tester(4) is Tot |= (2**y) if grid[x][y]; return Tot == 62 ?1:0
Tester(5) is Tot++ if grid[x][y]; return Tot % 2

assert VoteYes: lastmove("Understand")
assert VoteNo:  lastmove("Understand")
assert after move: CORRECT == 5 => level = level+1 => CORRECT=0
assert after Understand: flip ? Generate(level) :
AntiGenerate(level); update()
```

We will tackle the game description in chunks.

```
game has Example button
game has Test button
game has Understand button
game has VoteYes button
game has VoteNo button
```

When EGGG sees these lines, it creates five buttons, each with the specified text on it. It also creates a callback for each button — a subroutine that will be invoked when the player presses the button. It doesn't know what the buttons do yet, and if no part of the game description specifies what the button does, EGGG warns the user that the button does nothing. It still generates a callback; all it does is display "You pressed the *<name of button>* button." Luckily, the game description defines the buttons:

```
Example    means grid becomes Generate(level)
Test       means display(Tester(level))
Understand means display("Vote Yes or No.");
VoteYes    means     Tester(level) and ++CORRECT or CORRECT = 0
VoteNo     means not Tester(level) and ++CORRECT or CORRECT = 0
```

(Note that extra whitespace is ignored by the EGGG parser; that's why we can align these rules on `means` to enhance readability.)

Each line generates the callback for a button. The `Example` callback changes the state of the entire board: `grid becomes`. The grid becomes whatever `Generate(level)` returns; that's a function invocation (which is actually translated into `Generate($state{level})` when the game program is created). The `Generate` function (not yet defined) is passed whatever the current level is (an integer from 1 to 5), and returns a data structure representing the entire board, which then becomes the grid that the player sees. `becomes` is used to change the state of a board, piece, or hand in a sweeping way.

The `display` function (lower case, so it's known to EGGG) shows a message to the user. `display` decides which message area is appropriate; in EGGG games, the left area is used for informative messages ("Vote Yes or No") by default, and the right area is used for stats, such as scores and levels.

The definitions of the `VoteYes` and `VoteNo` buttons are the first examples we've seen of burrowing down into the language underlying EGGG: Perl. `VoteYes` invokes the `Tester` function with one parameter, the current level. `CORRECT`, because it is capitalized, is a variable that EGGG creates — and remembers even after the subroutine terminates. The single letters `P` and `Q` and `S` that we saw with poker all disappeared when the subroutine ended; `CORRECT` is what we call a *state variable*, and lasts for the entire game. Here's the actual `VoteYes()` subroutine generated by EGGG:

```
sub VoteYes {
    Tester($state{level} and ++$state{CORRECT} or $CORRECT = 0;
}
```

EGGG's logical operators *short-circuit.* In the expression *A* `and` *B*, *B* is only evaluated if *A* is true. In the expression *A* `or` *B*, *B* is only evaluated if *A* is false. That's short-circuiting. And ++ just means to add one to something, so this subroutine is equivalent to:

```
sub VoteYes {
    if ( Tester($state{level} ) {
        $state{CORRECT} = $state{CORRECT} + 1;
    } else {
        $state{CORRECT} = 0;
    }
}
```

The next lines are all straightforward:

```
goal is level(highest)
one player
5x5 grid
```

Players try to attain the highest level, it's a one-player game, and the board is a grid with five squares on each side.

```
squares are black and white
click makes square toggle
```

The first line indicates the two colors that a square can be. That's important, because the second line indicates what happens when a user clicks on one: it toggles between the two colors. Had the lines been:

```
squares are black and white and red
click makes square rotate
```

...then clicking on a square would cycle from black to white to red and back to black again.

Board Initialization

```
board starts empty
```

This tells EGGG that the initial state of the Deducto grid is empty. The twenty-five

squares won't be black or white; they'll be gray. In a game like chess, the board doesn't start empty. To demonstrate how players can specify the initial state of a board, we'll talk about chess instead of Deducto briefly.

How would you describe the initial state of a chessboard? You'd say that it's an eight by eight grid, and you'd say which pieces belong on which squares. Specifying the tabula rasa state of a chess game is pretty easy, because it's the same each time. In EGGG, the initial board state looks like this:

```
board starts [[black Rook, black Knight, black Bishop, black Queen,
               black King, black Bishop, black Knight, black Rook],
              [black Pawn, black Pawn, black Pawn, black Pawn,
               black Pawn, black Pawn, black Pawn, black Pawn],
              [empty, empty, empty, empty,
               empty, empty, empty, empty],
              [empty, empty, empty, empty,
               empty, empty, empty, empty],
              [empty, empty, empty, empty,
               empty, empty, empty, empty],
              [empty, empty, empty, empty,
               empty, empty, empty, empty],
              [white Pawn, white Pawn, white Pawn, white Pawn,
               white Pawn, white Pawn, white Pawn, white Pawn],
              [white Rook, white Knight, white Bishop, white Queen,
               white King, white Bishop, white Knight, white Rook]]
```

Note that there is no syntactic distinction between squares which have pieces on them and those that don't. How does EGGG know that `empty` isn't a piece? Because empty is one of words in EGGG's vocabulary. It's used for the pieceless triangles in backgammon and for the empty squares in a crossword grid. (If `empty` hadn't been a known word, it would have been considered an unknown attribute instead of a piece because it's not capitalized, and "empty" would have been printed on each of the 32 initially empty squares.)

Where does a game like Random Chess fit into this categorization? Bobby Fischer, the eccentric chess champion, once claimed that he'd never consider playing any form of chess again other than Random Chess. In Random Chess, the play of the game is the same as regular chess, but the pieces on each player's back row are randomized. Both black and white get the same pieces in the same order, so (for instance) the kings and queens of each side will be on the same file; and the only constraint is that the bishops must be on squares of different color [Burns 98, p. 145]. It would be incorrect to say this:

```
BigBlackPiece is [black Rook, black Knight, black Bishop,
                  black Queen, black King]
board starts [[rand BigBlackPiece, rand BigBlackPiece,
               rand BigBlackPiece, ...]
```

These are legal EGGG statements; the generated game will choose a `BigBlackPiece` at random, place it on the first square of the back row, and repeat that seven more times. However, that could result in two queens, or five bishops. That's not Random Chess.

Instead, the game designer has to treat the collection of black pieces as a set:

```
BigBlackPiece is [black Rook, black Knight,
                  black Bishop, black Queen,
                  black King, black Bishop,
                  black Knight, black Rook]
shuffle BigBlackPieces
board starts [[BigBlackPieces], ...
```

Note that the earlier invocation of `BigBlackPiece` treated the collection of pieces as an enumerated type, while the second treated the collection as what computer scientists call a *bag* (like a set, but allowing duplicate elements). EGGG chooses the internal representation based on whether or not duplicates occur.

Functions

Now, back to Deducto. The game description mentions three functions: `Generate()`, `AntiGenerate()`, and `Tester()`. The first two are functions that generate grids that match, and fail to match, the rule of the current level. `Tester()` is a function that returns true if the current grid matches, and false if it doesn't.

In traditional programming languages, such functions might be written like this:

```
sub Generate {
    if    (level == 1) { ... }
    elsif (level == 2) { ... }
    elsif (level == 3) { ... }
}
```

You can write functions this way with EGGG. But the line-oriented EGGG language also allows designers to specify functions simply by identifying what happens for each input. In the fifteen lines below, each line indicates one input-output pair.

```
Generate(1) is while (!Tester(1)) { grid[x][y] = flip ? 1 : 0 }
Generate(2) is { grid[x][y] = flip ? 1 : 0 }; grid[3][3] = 1
Generate(3) is while (!Tester(3)) { grid[x][y] = flip ? 1 : 0 }
Generate(4) is while (!Tester(4)) { grid[x][y] = flip ? 1 : 0 }
Generate(5) is while (!Tester(5)) { grid[x][y] = flip ? 1 : 0 }
AntiGenerate(1) is while (Tester(1)) { grid[x][y] = flip ?1:0 }
AntiGenerate(2) is { grid[x][y] = flip ?1:0 }; grid[3][3] = 0
AntiGenerate(3) is while (Tester(3)) { grid[x][y] = flip ?1:0 }
AntiGenerate(4) is while (Tester(4)) { grid[x][y] = flip ?1:0 }
AntiGenerate(5) is while (Tester(5)) { grid[x][y] = flip ?1:0 }

Tester(1) is Tot++ if grid[x][y]; return Tot > 12
Tester(2) is return grid[3][3]
Tester(3) is return 1 if grid[3][y] && (grid[1][y] &&
             grid[2][y] or grid[4][y] && grid[5][y]); return 0
```

```
# This one is tricky.  Tester(4) is true if there's an "on"
# square in each row.
Tester(4) is Tot |= (2**y) if grid[x][y]; return Tot == 62 ?1:0
Tester(5) is Tot++ if grid[x][y]; return Tot % 2
```

These functions burrow down into Perl, and are the core of the Deducto game. We won't discuss how they all work, but note that many of them look like this:

```
Generate(3) is while (!Tester(3)) { grid[x][y] = flip ? 1 : 0 }
```

This indicates what the `Generate()` subroutine should do when the level is 3 and a matching board needs to be generated. A while loop is entered, and continues for as long as `Tester(3)` is false; that is, as long as the grid generated doesn't match the current rule. Inside the while loop, EGGG sees `grid[x][y]`, and so generates two implicit loops through the possible values of x and y — the lower left square being `grid[1][1]` and the upper right being `grid[5][5]`. For each square, the value of the square is set to `flip ? 1 : 0` — EGGG flips a virtual coin, and if the coin comes up heads, EGGG sets the square to white, otherwise black. (As a value, EGGG treats `1` and `+` and `on` the same as `white`.) Here is relevant part of the `Generate()` subroutine, with the rules for levels other than 3 omitted:

```
# Generate($level) returns a new grid in which the squares
# match the current level $level.
#
# The Generate() subroutine is called by the Example() button
# and by the Understand() button.
sub Generate {
    my ($x, $y);
    my $board;
    if ($_[0] == 1) {   ...
    if ($_[0] == 2) {   ...
    if ($_[0] == 3) {         # If the level is 3
        do {
          for ($x = 1; $x <= $SIDES[0]; $x++) { # Loop horizontally
             for ($y = 1; $y <= $SIDES[1]; $y++) {  # Loop vertically
                 { $board[$x][$y] = (rand() < 0.5) ? 1 : 0 } ;
             }
          }
        } while !Tester(3);   # as long as Tester(3) is false
        return $board;        # Return the generated board
    }
    if ($_[0] == 4) {   ...
    if ($_[0] == 5) {   ...
}
```

Assertions

Here is the remainder of Deducto's game description.

```
assert VoteYes: lastmove("Understand")
assert VoteNo:  lastmove("Understand")
assert after move: CORRECT == 5 => level = level+1 => CORRECT = 0
assert after Understand: flip ? Generate(level)
                         : AntiGenerate(level); update()
```

These lines are the first assertions we've seen. Assertions allow game designers to create actions that will be triggered at particular times. The first two assertions are conditions that must be satisfied for the `VoteYes()` and `VoteNo()` subroutines to be invoked. If the player's last move wasn't pressing the `Understand` button, nothing will happen when the user presses either `VoteYes` or `VoteNo`. That's to prevent users from cheating; otherwise, they could use the `Test` button to learn whether the grid matches before voting yes or no.

The other two assertions begin `assert after`, and are not so much assertions as actions to be taken. After every action by a player, EGGG checks to see whether the number of correct guesses has reached 5; if so, it increments the level and sets the `CORRECT` variable to zero. And whenever the player presses `Understand`, Deducto generates a new grid — a grid that matches or a grid that doesn't, each with a 50% probability. This behavior could just as well have been part of the `Understand means` statement; we include it as an assertion for variety.

Variations

EGGG makes it easy to create variations on games. Game designers can copy game descriptions from EGGG's central repository (described in Chapter 6) and modify them, or they can take the game description and create a new game description with a `like` rule. For instance, the Random Chess game described earlier can be created with this game description file:

```
RandomChess is like Chess
BigBlackPiece is [black Rook, black Knight,
                  black Bishop, black Queen,
                  black King, black Bishop,
                  black Knight, black Rook]
shuffle BigBlackPieces
BigWhitePiece is [white Rook, white Knight,
                  white Bishop, white Queen,
                  white King, white Bishop,
                  white Knight, white Rook]
shuffle BigWhitePieces
board starts [[BigBlackPieces], ..., [BigWhitePieces]]
```

The complete board description has been omitted to save space, but other than that,

this is the entire game description. The game designer only has to include the statements that differ between the two games; the rules in common will be read in from the description of chess in `chess.egg`.

For the October 1998 News in the Future consortium meeting at the MIT Media Laboratory, I demonstrated a sequence of five variations, each building on the last. I started with the `tetris.egg` file in Appendix B, and made the following changes.

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

```
SideTetris is like Tetris
```

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

```
TwoPlayerTetris is like SideTetris
```

I changed the game so that players could bounce a piece back to the other side:

```
BounceTris is like TwoPlayerTetris
```

I sped the game up by shortening the delay between each piece motion:

```
BounceFast is like BounceTris
```

I made all the pieces square:

```
SquareBounce is like BounceFast
```

I removed the stickiness of the left and right edges:

```
Pong is like SquareBounce
```

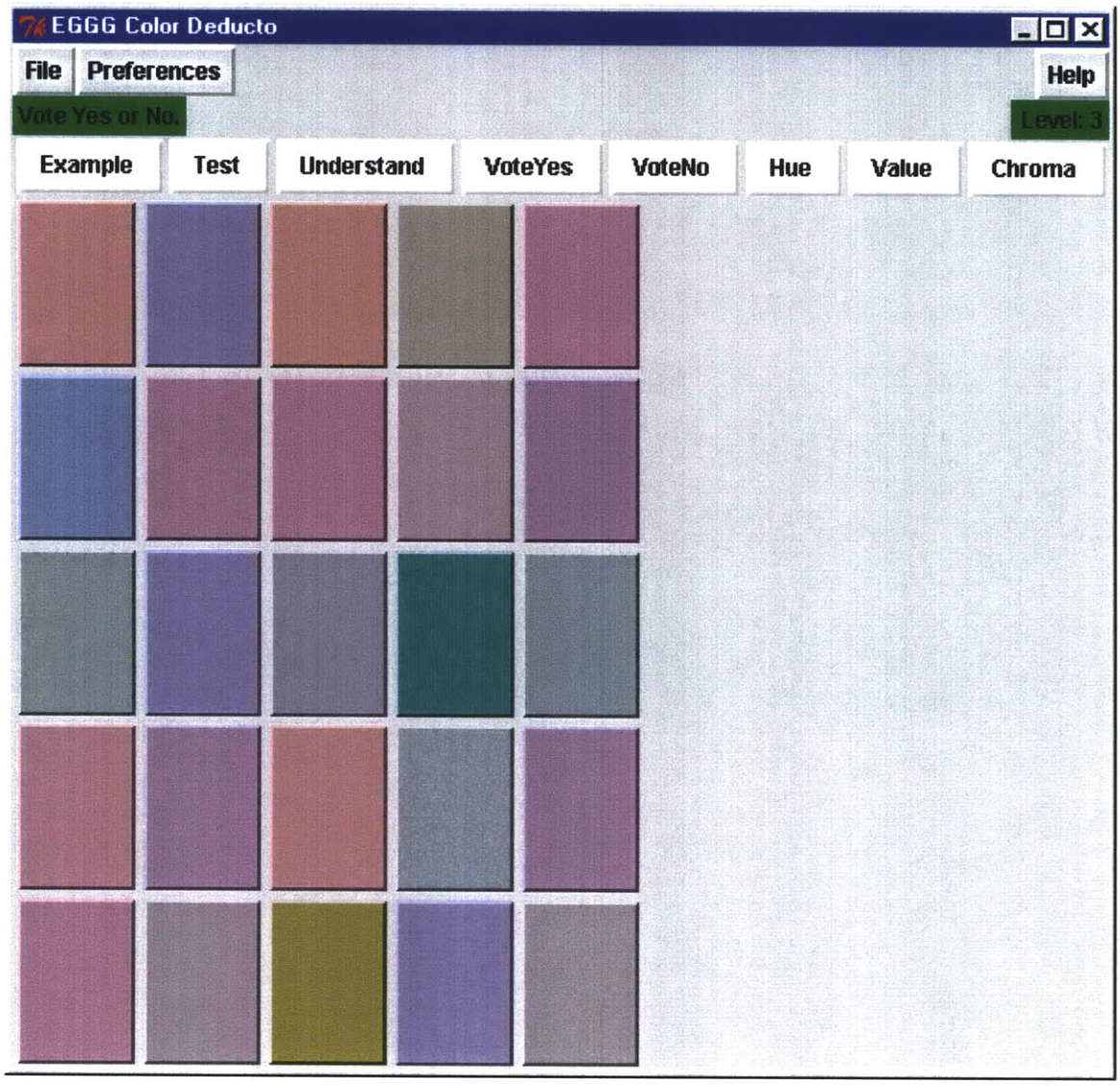
The result is the classic game Pong. With another ten steps, you could probably turn Tetris into tic tac toe, and then you're another ten steps from chess or Rubik's cube. But poker is a long way off.

A Sample Variation: Color Deducto

We'll now create a variation of Deducto, called Color Deducto. This variation was used as a teaching exercise in Color in Media Technology, a graduate course at MIT.

Color Deducto is like Deducto, except that the squares are colored instead of black and white, and the rules have to do with the colors. for instance, the rule for the first

level tests whether enough of the squares are bright; the second level tests whether the grid has saturated colors at the top of the grid, and pale colors at the bottom. EGGG has a Color module (more about modules later in the chapter) that provides functions such as `red()`, `green()`, and `blue()`, which extract the red, green, and blue components of a color. `value()` computes the brightness of the color ("value" is the color term for brightness), `hue()` returns the Munsell hue of the color, `opponent_hue()` returns the opponent Munsell hue (red for green, blue for yellow), and `chroma()` returns the saturation. It also provides functions that generate random colors and color components (`randmunsell()`, `randvalue()`, `randhue()`, and `randchroma()`), and functions for incrementing and decrementing the color components (`munsell_hue_add()`, `munsell_value_add()`, `munsell_chroma_add()`). A screenshot is shown below.



Here is the game description for Color Deducto.

```
ColorDeducto is like Deducto
Generate(1) is while (!Tester(1)) { grid[x][y] =
   munsell_value_add(munsell_chroma_add( randcolor(), -1, 1 ),
   2, 1) }
Generate(2) is while (!Tester(2)) { grid[x][y] =
   randmunsell("10PB", undef, "==1.5*int(2**(y-2))") }
Generate(3) is { grid[x][y] = randhue() . " 5/" . randchroma(1,3) };
Generate(4) is while (!Tester(4)) { grid[x][y] = ( (x%2) ?
  (randmunsell("10G", "<9", ">2")):(randmunsell("10RP","<9",">2")))}
Generate(5) is { grid[x][y] = randcolor() }; grid[3][3] = "5Y 9/0"
AntiGenerate(1) is while (Tester(1)) { grid[x][y] = randcolor() }
AntiGenerate(2) is while (Tester(2)) {
  grid[x][y] = randmunsell("10PB", undef,
                           "==1.5*int(2**((6-y)-2))") }
AntiGenerate(3) is { grid[x][y] = randhue() . " " . randvalue(0,2) .
  "/" . randchroma(1,2) };
AntiGenerate(4) is while (Tester(4)) { grid[x][y] = randcolor() }
AntiGenerate(5) is { grid[x][y] = randcolor() };
                   grid[3][3] = "5Y 1/0"

Tester(1) is Tot++ if value(grid[x][y]) > 6; return Tot > 15
Tester(2) is Tot++ if chroma(grid[x][y]) > chroma(grid[x][y-1]);
  return Tot > 19
Tester(3) is Tot++ if value(grid[x][y]) == 5; return Tot > 24
Tester(4) is Tot++ if x>1 && hue(grid[x][y]) eq
  opponent_hue(hue(grid[x-1][y])); return Tot > 19
Tester(5) is return value(grid[3][3]) > 8
game has Hue button
game has Value button
game has Chroma button
Hue means MODE = "hue"
Value means MODE = "value"
Chroma means MODE = "chroma"

squares are colors
click makes square increment mode
```

Color Deducto replaces the `Tester()`, `Generate()`, and `AntiGenerate()` functions of regular Deducto, as one would expect. Three additional buttons are provided: `Hue`, `Value`, and `Chroma`, which set a "mode" variable to `"hue"`, `"value"`, or `"chroma"`; when the player presses on a square, the `click makes square increment mode` looks for a way to add something to the hue, value, or chroma, triggering the appropriate functions from the EGGG `Color` module.

Software Architecture

In this section, we'll discuss how EGGG was built, and the reasons for our implementation decisions.

Why Perl?

The games that EGGG generates are Perl programs, and EGGG itself is written in Perl. Perl is ideal for several reasons:

- Portability.

  EGGG games work on all Windows and Unix platforms without any modification whatsoever. No recompilation is needed.

- Speed

  1. Regular expressions. Speedy regular expressions allow some game data structures to be represented as simple strings for ease of programming. Typically, the disadvantage of representing data as strings is speed — but Perl's text manipulations are fast enough that they can be performed between player actions even in frenetic games like Tetris.

  2. Hash tables. Most of EGGG's data structures are represented internally as hashes, providing a time- and space-efficient way to encapsulate the implementation of a game and the state of a currently playing game. Efficient storage is especially important for EGGG because of all the redundancy in the programs it generates — it's not unusual to see vestigial code and data structures in the programs that it generates.

     Perl supports persistent storage of its data structures, so these hash tables can be dumped out to disk and read back in again. EGGG uses this to save and recreate the complete state of a game (even arcade games), and to record the successes and failures of the strategies that its computer opponents use.

- Introspection. A program that generates programs is necessarily a complex beast. Perl has several features that make automated programming easier:

  1. Closures. Closures are dynamically generated subroutines, and are used to implement the actions triggered when a player performs an action like pressing a button.

  2. Self-examining source code. EGGG verifies the syntactic correctness of the programs that it produces as it generates them. However,

players and game designers might well use EGGG merely as a starting point, using it to create a game and then manually applying their own modifications. Or, the vagaries of different environments might somehow cause EGGG to fail.

When an EGGG game runs, it reads in its own source code and verifies that it's a syntactically correct Perl program. If it isn't, EGGG can spot the errors before the program aborts, and sends diagnostic mail to EGGG central (the "Henhouse"). The mail includes the game description, the generated game, and a copy of the EGGG engine used to create the game.

3. Symbolic references. Symbolic references let you manipulate a data structure by name; for instance, you can modify `%game`, the game data structure, given only the string `"game"`. This is used for games in which players or pieces are added while the game is in play — in such cases, the data structures have to be created dynamically, and symbolic references give EGGG a way to use meaningful names for the data structures created by the additions.

4. `eval`. Perl's built-in `eval` function lets EGGG create and execute little Perl programs inside itself. When game descriptions burrow down into Perl code (as Deducto does), `eval` provides a way for Perl to execute that code and trap both compile-time and run-time errors.

- Graphics. The Perl/Tk library makes it easy for Perl to create fully-functioning graphical applications. Perl/Tk restricts EGGG portability more than Perl itself does, since the Tk library has been ported to fewer architectures than Perl. That's why EGGG games won't work on Macs until someone ports Tk to that platform.

- Database support. Perl has free APIs for all major database engines, and provides its own simple database management system. EGGG uses it to store the results of game play, high score files, e-mail addresses, and other configuration information.

- Networking. Perl has built-in support for creating TCP/IP sockets and sending data across them. When EGGG creates multiplayer games meant to be used across the Internet, each game implementation is able to behave as both server and client.

- Mail filtering. The abundance of mail tools available for Perl makes the Henhouse possible. The Henhouse, described in Chapter 6, retrieves mail sent to `eggg@media.mit.edu`. This mail includes the bug reports mentioned above, as well as which games are being played, and the success or failure of strategies employed by EGGG's autogenerated computer opponents.

- Popularity. There are over a million Perl programmers as of this writing, and over a thousand free Perl utilities on the Comprehensive Perl Archive Network.

Perl is an object-oriented language, but only the EGGG modules employ object-oriented features — a small part of the EGGG system. Modules are described later in this chapter.

Running EGGG

To run EGGG, game designers create the game description in a single text file using the editor of their choice. They then run the `eggg` program, providing the game description:

```
$ eggg poker.egg
This is a card game.
Creating a game called poker.
Boardtype is canvas.
Main loop written.
State subroutines written.
User subroutines written.
Display subroutines written.
Looking for assertions.
Goal written.
Strategies created.
poker created.
Eggg::Poker module found.
```

This creates an executable program called `poker`.

Systems that don't recognize the `#!` syntax may be unable to identify `eggg` as a Perl program. They need to invoke the Perl interpreter explicitly:

```
$ perl eggg poker.egg
```

Users can specify a different name for the generated game with the `-o` option:

```
$ eggg -o mypoker poker.egg
```

The Parser

Early implementations of EGGG used a top-down recursive descent parser: Perl's `Parse::RecDescent` module. The grammar used is shown in Appendix C, and consists of 71 words that have special meaning to EGGG. 15 of these words were non-game-specific words like "means" and "are". The other 56 were game-specific

words like "piece", "board", and "button".

However, this parser was abandoned for speed reasons; it took about twenty minutes to generate a chess game on a 233 MHz Pentium II. And because EGGG was intended to make the process of game design demand as little knowledge from the user as possible, the parsing process was made interactive. (Examples of this interactivity are shown in *Interactive Parsing*, below.)

The decision to forgo the rigor of a true parser in favor of a more haphazard "look at the game description and see what you can see" loosely parallels what Perl itself does: instead of a strict division between lexical analysis and parsing, EGGG performs both simultaneously. Traditional compilers make a small number of passes through the code — usually no more than two. EGGG, in a sense, makes hundreds of passes, using Perl's regular expressions to scan for one clue at a time. Most computer scientists wouldn't view this as compilation at all, but the effect is the same: transforming a higher-level language (EGGG) into a lower-level one (Perl). In the current implementation of EGGG, the parsing of a game description happens as the engine is running. Most of the parsing takes the form of ad hoc statements like this:

```
$eggg->{colormatters} = 1 if $game =~ /\b(red|chroma|hue)\b/m;
```

The `eggg` data structure contains information gleaned from the parse; here, the `colormatters` attribute is set to `1` if the words "red", "chroma", or "hue" appear anywhere in the game description. This tells EGGG to make the generated game use the EGGG Color module, and it tells EGGG not to make its other color choices arbitrarily. For instance, if `$eggg->{colormatters}` is true, then EGGG won't choose green as the default canvas color for betting games, because that green might make it hard to discern the other colors in the game.

Similar rules determine whether a game is played on a grid, whether a game has betting, and so on. Instead of making the game designer explicitly categorize the game, EGGG attempts to infer it from the presence of words in the game description. The EGGG language isn't elegant, for the same reason that English isn't elegant: there are messy rules, and messy exceptions to the messy rules, and messy exceptions to the messy exceptions.

Ambiguity in the EGGG language

A certain degree of ambiguity in game vocabularies is inevitable. For instance, the "height" words in EGGG descriptions (like "over" and "above" and "higher") have very different meanings in games played on a grid (chess, tic tac toe, Go) than in "accumulation games" (poker, blackjack, Monopoly, rock paper scissors). In the former, "height" words refer to position. In the latter, they refer to ranking in a set — usually having to do with the score (the size of your bank account in Monopoly, the ranking of your hand in poker). (And the set needn't be well-ordered; consider rock-paper-scissors, where hand ranking is intransitive.) The same is true for a word like

"rank", which refers to the *y*-coordinate on a chessboard but the strength of one's hand in poker.

EGGG uses the categorization string of the game (described in Chapter 2) to disambiguate words with multiple meanings.

Interactive Parsing

When I first demoed EGGG, I decided to demonstrate creating a variation on the fly. I took the game description for poker and defined a new hand. The new hand was just like a straight, but you only had to have four cards in succession: a baby straight. I copied the line for `Straight`:

```
Straight is (R, s) and (R-1, s) and (R-2, s) and (R-3, s) and (R-4, s)
```

and made it into a `BabyStraight`:

```
BabyStraight is (R, s) and (R-1, s) and (R-2, s) and (R-3, s)
```

Unfortunately, I forgot to paste the `Straight` line back into the game description. EGGG generated a syntactically correct program, and it even generated a `Straight()` subroutine because `Straight` was mentioned in the `hands are` rule, but the subroutine did nothing.

As a result, EGGG was modified to query the user during the parse. If EGGG sees that it is about to generate an empty subroutine, it allows the game designer to correct the error:

```
$ eggg poker-baby.egg
This is a card game.
Creating poker-baby.
Boardtype is canvas.
Main loop written.
State subroutines written.
User subroutines written.
Display subroutines written.
Eggg::Poker module found.
No rule found for Straight.
    Enter rule: (R, s) and (R+1, s) and (R+2, s) and
                (R+3, s) and (R+4, s)
Looking for assertions.
Goal written.
Strategies created.
poker-baby created.
```

The Engine

The `eggg` data structure described earlier exists only while the game is being created. The generated game makes use of similar structures; the two most important are `game` and `state`.

The Game

The `%game` data structure is a hash table containing the permanent aspects of game play. The contents of `%game` vary from game to game, typically including:

- `$game{type}`, the type of the board
- `$game{sides}`, the dimensions of the board
- `$game{board_start}`, the initial configuration of the board
- `$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 are part of the `%game` data structure (`$game{board}`, in particular).

The State

Whereas the grid arrangement of a crossword is stored in `$game{board}`, the actual letters that the player has written in are stored in `$state{board}`. The `state` data structure complements the `game` data structure; it's a hash table containing the ephemeral aspects of game play such as these:

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

EGGG games perform some rudimentary *memoizing*, or caching of function results. When a function takes a long time to execute, and depends only on the `%state`, EGGG remembers the result. Then, if the function is called again with the same `%state`, the result can be returned immediately. There is a *dirty bit*, `$state{dirty}`, that indicates whether any component of the `%state` data

structure has changed, and this is used as the game is running to determine whether memoized results are still valid.

Modules

In Perl, a *module* is a collection of variables and subroutines occupying their own namespace and their own file. Modules are the fundamental unit of code reuse in Perl; when people make their programs available to others on the Comprehensive Perl Archive Network, they typically do so by extracting the reusable components of their programs and building a module out of them.

EGGG uses Perl modules to provide behaviors that are common to many games. It does not use them for behaviors that are common to nearly all games; if the behavior is that common, it's integrated into EGGG itself. There are a few modules currently included with EGGG:

- Eggg::Deck

  When EGGG deduces that a game is a card game (by the presence of "card words" like "deal" or "ace" or "cards" in the game description), it creates code that invokes `Eggg::Deck`, the Deck module. This module contains an object-oriented deck implementation. EGGG games create `Deck` objects, and invoke `Deck` methods that deal cards from the deck, and use `Deck` functions that, given a card name, returns a filename containing an image of the card. `Eggg::Deck` can't shuffle the deck, however.

- Eggg::Random

  Games that require randomness beyond the simple pseudorandom number generator that Perl provides use the `Eggg::Random` module. The `Deck` objects created by `Eggg::Deck` are shuffled with the `shuffle()` function provided by `Eggg::Random`.

  `shuffle()` generates a perfect shuffle. However, when real fingers shuffle real cards, the shuffling is never perfect. The riffle shuffle that most dealers use don't randomize the cards very well, even if two are performed in immediate succession, followed by a cut of the deck. This module provides `riffle_shuffle()` and `cut()` methods so that the deal in an EGGG card game can more closely approximate the deal in a tangible card game. You can change the dealing behavior by changing this line:

  ```
  game is shuffle(deck) and ...
  ```

  to

  ```
  game is riffle_shuffle(deck) and riffle_shuffle(deck)
       and cut(deck) and ...
  ```

`Eggg::Random` also provides functions for generating random numbers with weighted distributions (so you can make the long bars of Tetris twice as rare as other pieces, for instance).

When games rely on the randomness of a pseudorandom number generator, the initial state can be recreated from one number, called the *seed.* The `save_seed()` subroutine stores it, allowing those games (notably, card games) to be recreated later.

- Eggg::Color

  EGGG games have to represent colors as red-green-blue triplets for the sake of the graphics libraries it uses, but RGB is the wrong colorspace for most meaningful color manipulations. For instance, it's tough to make an object brighter or paler with RGB, but it's trivial with other colorspaces.

  RGB is an especially bad colorspace for Color Deducto, which has rules involving the brightness or paleness of colors. To make these manipulations easier, the `Eggg::Color` module was created. `Eggg::Color` provides functions that give designers greater control over the colors in their application, by letting them manipulate a game's colors in the Munsell colorspace instead of RGB. The module makes use of routines from Perl's `Image::Colorimetry` module, by the author of this dissertation.

- Eggg::Chess and Eggg::Crossword

  This was the first EGGG module created. EGGG can figure out everything it needs to know about how chess is played from the game description in `chess.egg`, except for one thing — what the pieces look like. The `Eggg::Chess` module provides a single function, `piece_image()`, that takes the name of a piece as input and returns a filename that contains a GIF image of that piece. EGGG games then use that image whenever the piece is placed within view. `Eggg::Crossword` does the same for grid squares, providing one image for every letter on squares labeled from 1 to 200. Both `piece_image()` functions can accept either a piece name, or grid coordinates. If there's no piece on the grid at the specified coordinates, `piece_image()` returns an "empty" piece — a black square for crosswords, or a beige or brown square for chess, depending on whether the sum of the coordinates is odd or even. (That results in the checkerboard pattern common to all chessboards.)

  The complete EGGG system comes with an `images` directory containing the chess, crossword, and card GIF files.

In addition, EGGG generates a module for each game as the game is being created. If the game is a variation, then the module will inherit the subroutines and variables from the game it's derived from. Most of these game-specific modules have little else; they serve mostly as repositories for recording the outcomes of play: who

played what when, and which strategies worked. For instance, the `Eggg::Chess` module slowly builds up a library of opening moves — not because it has any knowledge of how the game is played, but because as people play EGGG Chess, their opening moves and eventual outcomes (win/lose/draw) are recorded and used by EGGG's computer opponent to model both individual players and to model a "universal" player. This will be discussed more in the next chapter.

Sorting

Game-specific modules can also provide a `piece_sort()` subroutine; if one exists, it will be used to override EGGG's default piece sorting algorithm. By default, EGGG sorts pieces from most valuable to least. Sometimes, that won't be what's desired. In a bridge hand, aces are still the most valuable cards, but players typically sort them by suit (the primary sort) and then by value (the secondary sort). (Some competitive players avoid this, because the physical act of moving cards around — or the patterns of eye movement resulting from a sorted hand — can reveal information about their hand to opponents.)

Furthermore, some games impose twists on the default orderings: in pinochle and skat, the ten is between the ace and the king. A `piece_sort()` subroutine for these games might look like this:

```
my %card_values = (A => 6, 10 => 5, K => 4, Q => 3, J => 2, 9 => 1);
sub piece_sort {
    return $card_values{$b} <=> $card_values{$a};
}
```

EGGG uses the `piece_sort()` routine as an input to Perl's quicksort algorithm. Why are the piece values provided as a hash table (the `my %card_values` line above) instead of an ordered array where the index implies rank? Because there are games in which two different pieces are worth exactly the same.

Timing

In the taxonomy of games developed in Chapter 2, games were classified as frenetic-fast, frenetic-timed, or not frenetic at all. If the game is frenetic-fast or frenetic-timed, the program generated by EGGG needs to keep track of how much time has elapsed.

In frenetic-fast games, this is accomplished with Tk's `after()` method for graphical games, and Perl's built-in `select()` function for non-graphical games. Each lets the designer specify a minimum delay between player actions, in milliseconds. EGGG chooses a default delay based on how frenetic the game seems; it uses the granularity of the board as an estimate. Tetris has a coarse 10x20 grid, and

EGGG starts the delay at 1000 milliseconds. Shooter games use canvases instead of grids, and EGGG starts the delay for those games at 100 milliseconds. If there are levels in the game, EGGG reduces the delay by ten percent when the level increments.

Frenetic-timed games require speed but not hand-eye coordination; chess with a chess clock, or Minesweeper, are examples. These games don't require sub-second accuracy. While frenetic-fast games typically require continual actions, some frenetic-timed games can get by with POSIX alarms. Once an alarm is set, the game progresses as normal, until the alarm triggers and passes control elsewhere. However, not all platforms support alarms — in particular, Windows doesn't. If the game is graphical, there's no problem: Tk's `after()` method works fine. For non-graphical games, `select()` will do, as long as the platform also supports a separate thread of execution (either through `fork`, threads, or some other interprocess communication). If all of those fail, EGGG uses Perl's `time()` function to count seconds, and checks how many seconds have elapsed after every player action.

Documentation

```
Avoid Missing Ball For High Score

Deposit Quarter
```

The two instructions for Pong.

One criticism of EGGG might be that it works by exploiting the similarities between games, rather than through achieving a "deep understanding" of the game, using natural language parsers and formal planning techniques from AI. One of the touchstones, then, would be how well EGGG can use its educated guesses about games to explain how a particular game works.

EGGG generates documentation for the Perl code that it creates, and it also generates instructions for the players to read. In this section, we'll explore both of these features.

Documenting The Program

Writing documentation is tedious. The programmer typically knows all too well what his code does, sometimes too well to explain to an audience of indeterminate ability. And often the programmer is unsure whether anyone will even bother to read the documentation.

EGGG doesn't mind the tedium, and generates thorough documentation. Generated programs start with headers like this:

```
## This game was automatically generated by EGGG
## at Tue Oct  5 10:13:33 1999
## For more information about EGGG, visit
## http://orwant.www.media.mit.edu/eggg/eggg.
```

EGGG documents most standalone lines of code:

```
## Included modules:

use Eggg;           # Primary EGGG routines
use Data::Dumper;   # For deep copies
use Tk;             # Perl/Tk graphical display routines
use Sys::Hostname; # So that EGGG can identify who's playing

# During compile-time, search through the lib directory
# for Eggg:: modules.
# We don't know what delimiter the filesystem uses, so we check
# $^O to see if we're on Windows.

BEGIN { if ($^O =~ /win32/i) { push @INC, '.\lib\ ' }
        else { push @INC, './lib' } }

# Load the game-dependent Chess module (for pictures of pieces)
use Eggg::Chess;
```

It also documents every subroutine, and some of the lines inside:

```
# moves($board, $x, $y, $player, $piece) returns a
# list of the moves available for the piece at $x, $y.
sub moves {
    my ($board, $x, $y, $player, $piece) = @_;
    my (@results);          # The array of moves to be returned.
    if (ref $x) { ($x, $y) = @{$x->[0]}; }
    # If $x is a reference, this contains both coordinates.
```

EGGG also uses the documentation that it generates in one part of the game to generate other parts of the program. This happens when EGGG generates a chess game. The game description for chess has rules like these:

```
Bishop moves (x-1..7, y-1..7) if empty(x-1..7, y-1..7)
Bishop moves (x+1..7, y+1..7) if empty(x+1..7, y+1..7)
Bishop moves (x-1..7, y+1..7) if empty(x-1..7, y+1..7)
Bishop moves (x+1..7, y-1..7) if empty(x+1..7, y-1..7)
Bishop captures as it moves
```

The first four lines describe how a bishop moves when there are no intervening pieces; that is, when a bishop moves without capturing. The first line describes a move in the direction of the lower left corner. Here's the code that EGGG generates for that line:

```
### Code chunk 25
# One type of move for Bishop
# Bishop moves ($x-1..7, $y-1..7) if empty($board, $x-1..7, $y-1..7)
# Loop through possible moves for Bishop
MOVE: for (my $i = 1; $i <= 7; $i++) {     # Loop horizontally
  last unless $piece eq "Bishop";
     for (my $j = $i-1; $j >= 0; $j--) {    # Loop vertically
          next MOVE unless empty($board, $x-$i+$j, $y-$i+$j);
     }
     push @results, [$x,$y,$x-$i,$y-$i];  # Found a valid move.
  }
### End of code chunk 25
```

The `### Code chunk 25` and `### End of code chunk 25` are markers that EGGG sprinkles throughout the programs that it generates. EGGG generates code for "plain" moves before it generates code for captures; when it sees a line like Bishop captures as it moves in the game description, it searches through the already generated code and duplicates it, modifying it to search for moves where the final square is occupied by another piece (the `other($board, $player, $x, $y, $x-$i, $y-$i);` below).

```
### Code chunk 61
### (Autogenerated from chunk 25)
# One type of move for Bishop
# Bishop moves ($x-1..7, $y-1..7) if empty($board, $x-1..7, $y-1..7)
# Loop through possible moves for Bishop
MOVE: for (my $i = 1; $i <= 7; $i++) {     # Loop horizontally
    last unless $piece eq "Bishop";
    next unless other($board, $player, $x, $y, $x-$i, $y-$i);
    # Occupied by opponent
    for (my $j = $i-1; $j >= 1; $j--) {    # Loop vertically
         next MOVE unless empty($board, $x-$i+$j, $y-$i+$j);
    }
    push @results, [$x, $y, $x-$i, $y-$i];  # Found a valid capture.
}
### End of Code chunk 61
```

Documenting The Game

Earlier in the chapter, we noted that game designers were allowed to use whatever plurals they like, and in general that EGGG ignores grammar wherever possible. 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, a line like `players are black and white` is transformed into "The two players are black and white", which isn't much of a transformation.

Here are the instructions that EGGG generates for chess. These instructions are generated by a standalone program named `egggdescribe`.

EGGG Chess is a two-player game played on an eight by eight board. The two players are called white and black; white moves first. In each turn, the player moves one piece. The pieces are King, Queen, Rook, Bishop, Knight, and Pawn.

Each player tries to checkmate, which means that the opponent king has no moves and a piece P is Attacking the opponent King and no piece Q captures P. If a player has a turn and has no moves, the game is a tie.

It is illegal for a player's king to be attacked after a move.

The king can move left one, down one, down and to the left one, right one, up one, up and to the right one, down and to the right one, or up and to the left one. The king also captures as it moves.

The queen can move down from one to seven, up from one to seven, left from one to seven, right from one to seven, down and to the left from one to seven, up and to the right from one to seven, up and to the left from one to seven, or down and to the right from one to seven. The queen also captures as it moves.

The rook can move down from one to seven, up from one to seven, left from one to seven, or right from one to seven. The rook also captures as it moves.

The bishop can move down and to the left from one to seven, up and to the right from one to seven, up and to the left from one to seven, or down and to the right from one to seven. The bishop also captures as it moves.

The knight can move up one and right two, down one and right two, up one and left two, down one and left two, up two and right one, down two and right one, up two and left one, or down two and left one. The knight also captures as it moves.

This is a reasonably coherent and concise explanation of chess. However, the rules for castling and *en passant* captures lack the same literary punch:

The white king can move to (7, 1) if it is on (5, 1) and a white rook is on (8, 1) and (6, 1) is empty and (7, 1) is empty and no opponent piece attacks (5, 1) and no opponent piece attacks (6, 1) and no opponent piece attacks (7, 1) and the white king has not moved and the white rook on (8, 1) has not moved. If so, then the piece on (8, 1) moves to (6, 1).

The white king can move to (3, 1) if it is on (5, 1) and a white rook is on (1, 1) and (2, 1) is empty and (3, 1) is empty and (4, 1) is empty and no opponent piece attacks (3, 1) and no opponent piece attacks (4, 1) and no opponent piece attacks (5, 1) and the white king has not moved and the white rook on (1, 1) has not moved. If so, then the piece on (1, 1) moves to (4, 1).

The black king can move to (7, 8) if it is on (5, 8) and a black rook is on (8, 8) and (6, 8) is empty and (7, 8) is empty and no

```
opponent piece attacks (5, 8) and no opponent piece attacks (6, 8)
and no opponent piece attacks (7, 8) and the black king has not
moved and the black rook on (8, 8) has not moved.  If so, then the
piece on (8, 8) moves to (6, 8).

The white king can move to (3, 8) if it is on (5, 8) and a black
rook is on (1, 8) and (2, 8) is empty and (3, 8) is empty and (4, 8)
is empty and no opponent piece attacks (3, 8) and no opponent piece
attacks (4, 8) and no opponent piece attacks (5, 8) and the black
king has not moved and the black rook on (1, 8) has not moved.  If
so, then the piece on (1, 8) moves to (4, 8).

The white pawn can move to y=4 if it is on y=2 and y=3 is empty and
y=4 is empty, or up one if up one is empty.  It can capture up one
and to the right, up one and to the left, from (x, 5) to (x+1, 6) if
the last move was a black pawn from (x+1, 4) to (x+1, 5), or from
(x, 5) to (x-1, 6) if the last move was a black pawn from (x-1, 4)
to (x-1, 5).

The black pawn can move to y=5 if it is on y=7 and y=6 is empty and
y=5 is empty, or down one if down one is empty.  It can capture down
one and to the left, down one and to the right, from (x, 4) to (x+1,
3) if the last move was a white pawn from (x+1, 2) to (x+1, 4), or
from (x, 4) to (x-1, 3) if the last move was a white pawn from (x-1,
2) to (x-1, 4).
```

Note that the instructions for chess omit the starting board. Any array of data with over 25 elements is omitted from the instructions to minimize verbosity.

Perl's `Text::Wrap` module is used to format the generated instructions to 80 columns.

Naming

Most games have some way of naming the forces that are competing (or cooperating). The distinction might be by color (chess), symbol (tic tac toe), direction (bridge), or country (Diplomacy and other war games). Role playing games let players choose their names. Some games name the players after their pieces (chess and Monopoly) and other on more abstract qualities or roles (bridge and craps).

In any multiplayer game where the players are named in the game description file (`players are ...`), EGGG has it easy, because the game designer has just provided the names, knowingly or not. In unsynchronized games, the players are assumed to take their turns in the order in which they're mentioned. That's why the game description file for chess says `players are white and black` instead of `players are black and white`: because white always moves first.

When the players aren't named, EGGG assigns names to the human players. If there is only one human player, EGGG imaginatively names him "Player". If there are

two, four, six, or eight players, EGGG names them after the appropriate compass points. If there is some other number of players, EGGG calls them simply "Player 1", "Player 2", and so on.

Computer opponents deserve names too, and EGGG generates names based on the vocabulary of the game description. It first looks for piece names, goal states, or rule definitions — anything that begins with a capital letter followed by lower case letters. It then picks one of those words at random and transforms it into a name via an ad hoc "cutening" rule, which can add `r`, `er`, `ster`, `ie`, or `y` to the end of the word depending on the length and vowel-consonant pattern.

For instance, the names chosen at random for chess computer opponents are:

```
$state{names}{computer} = ["Kingy", "Stalemateer", "Rookie",
     "Checkmateer", "Queenie", "Knightie", "Bishopy",
     "Pawny",  "Attacker"];
```

The possible names for poker opponents are:

```
$state{names}{computer} = ["Ace", "Cardster", "Straightie",
   "Pairy", "Discardie", "Highster", "Flushie", "Housey",
   "Fully", "Kindy"];
```

Abbreviations

EGGG can sometimes generate ASCII versions of games. Translating graphical games into a seven-bit character set necessarily means making compromises in the representations of the pieces — images have to be replaced by abbreviations of a few characters. The game specific modules can provide a `piece_abbreviation()` function that map piece names to abbreviations. In the absence of a `piece_abbreviation()`, EGGG uses Perl's `Text::Abbrev` module to find the shortest unique abbreviation for each piece. "king" becomes `k` in EGGG poker, but would become `ki` in chess since "knight" also begins with a "k". (The `piece_abbreviation()` function in `Eggg::Chess` overrides this, abbreviating the king and knight to the more conventional `k` and `n`.)

When there are two players, EGGG lowercases one and capitalizes the other, so that the white queen becomes `q` and the black queen becomes `Q`. When there are more than two players, EGGG appends an abbreviation for the player name to the end of the piece abbreviation in grid games. In canvas games, the ownership of the piece is determined by the placement on the canvas. For instance, a king of hearts (`kh`) in poker doesn't need to be named `khs` if it's owned by the South player; the fact that it's in front of the South player is enough.

Error Recovery

There are several ways in which the act of playing a game with EGGG might fail.

The first type of error occurs when parsing the game description. That can happen in two ways: First, the game designer might create an ungrammatical ruleset for a game. If that happens, EGGG stops and refers the designer to the EGGG grammar. Second, the EGGG engine might need information not present in the game description, as with the poker example earlier. EGGG sees that it there is a hand called `Straight`, but it can't find a definition of what that means, either in the game description or in the `Eggg::Poker` module. In that case it queries the user if it was invoked from the command line, and generates an empty subroutine instead:

```
if (!$rule) {
    if (-t STDIN) {
      print "No rule found for $ranks[$i].\n\tEnter rule: ";
      chomp($rule = <STDIN>);
    } else {
      print STDERR "No rule found for $ranks[$i]. Creating a null rule.\n";
      print O " "x$in, "return 0;\n";
      $in-=4; print O "}\n\n";
      next;
    }
}
```

The second type of error occurs when the game parses correctly, but EGGG generates a syntactically incorrect game. This is hard to guard against; one of the prices paid for allowing people to include Perl code in an EGGG description is that there is no way to verify the syntactic correctness of that code without invoking the Perl interpreter. So immediately after hatching a game, EGGG invokes the Perl interpreter to verify that game is syntactically valid. If it isn't, it sends mail to `eggg@media.mit.edu` with the EGGG engine and the game description for analysis.

Third, the game might be generated correctly, but fail to run on a particular platform. For instance, if EGGG was installed incorrectly, or some of the modules it depends upon aren't present, Perl will abort the program as it is compiled into bytecode. It is also possible that the vagaries of different Perl distributions will cause other errors, either during compile-time or run-time. In these cases, Perl's `$SIG{__DIE__}` handler is used to intercept the error message and send it to `eggg@media.mit.edu` along with the engine, generated game, and game description (if available) for later analysis.

Fourth, the game might generate and run without error, but still fail to behave as the designer wanted. There's nothing EGGG can do about this.

In the next chapter, we explore the common strategies of computer opponents and describe how EGGG integrates them into the games it generates.

Chapter 4: Enemy of the Game State (Computer Opponents)

> *A habit sometimes adopted by children who are faced with inevitable loss at chess is to prolong the game by "spite checks": sacrificial attacks on the winner's king, which merely waste time and have no effect on the ultimate result of the game. (Children usually grow out of this habit; computers, in my experience, do not.)*
>
> John Beasley, in *The Mathematics of Games*, p. 131

The wealth of similarities between games includes similarities between the strategies used to play them. Humans have strategies, computers have strategies, and on occsasion the two blend in interesting ways. In this chapter, we will discuss how EGGG imparts strategies to the computer opponents that it generates.

We will first consider what Pell calls SCL (Symmetric Chess-Like) games: game like chess, checkers, tic tac toe, and Go, two-player alternating games that rely on neither chance nor concealment. Computer programs that play these games typically consist of three components:

- A minimax procedure
- A static evaluator
- A "library" of precalculated moves

We will turn to the first two now. EGGG's technique for amassing a library of moves will be deferred to Chapter 6.

A Generic Minimax Procedure

In the literature of computer gaming, the most-analyzed games are typically games of perfect information: games like chess or checkers or Go, where one can "prove" what the best move is in a given situation. Such analyses almost always depend on the assumption that one's opponent thinks just like you do; only recently have researchers thought to model the opponent's strategy in a generalized way [Gao 99].

In particular, most analyses have to do with *game trees*. A game tree is a data structure that helps an entity (either a computer program or a human) figure out what move to make. It works like this: you enumerate all of the moves available, and consider your opponent's responses, and your counterresponses to those responses, and so on. Eventually, when you can't search any further, you compute a score for

each board, and work your way back up the tree, assuming that at each opportunity the player will choose the best available move. The number of boards that you have to consider increases exponentially with how far into the future you look.

There are many variations on the procedure just described, most of which have to do with pruning the tree so that you don't have to consider as many moves. Taken together, they are often called *minimax* algorithms. The first known minimax procedure was described in a 1713 letter by James Waldegrave. He wrote about the card game "le Her" to Pierre-Remond de Montmort, who then wrote about it to the famous mathematician Nicolas Bernoulli, who popularized it.

Obviously, minimax algorithms apply only to certain types of games. The games have to have moves that can be enumerated; you can't use a minimax algorithm to play tennis or Pictionary, because there an infinite number of "moves" available. Minimax assumes that the play is one against one, and not one against one against one, or one against many, or many against many. It also assumes that the game is deterministic.

You'll find a minimax procedure in just about every chess program. The minimax procedure will make use of a *static evaluator* — a function that computes the "score" of a given board and is used to rank the moves available to a player. The basic algorithm is fine-tuned for chess.

EGGG has a generalized minimax procedure that works for any game that permits conventional minimax analysis. The procedure makes no assumptions about the particular game being played; one provides it with the `%game` and `%state` data structures, and two subroutines: `enumerate_moves()` and `score_board()`, and it returns the optimal move for the current player. EGGG-generated opponents thus qualify as metagame players, according to Pell's definition [Pell 1993].

EGGG generates computer opponents only for asynchronized games — games in which players alternate turns. That includes grid games like chess and tic tac toe, games where the design of computer opponents has been well-studied. But it also includes games like poker, which have been analyzed far less due to the psychological elements of successful play.

EGGG generates the `enumerate_moves()` subroutine by iterating through all of the player's pieces, identifying all the possible moves (including captures) for each, and eliminating any moves deemed illegal by assertions included in the game description.

A Generic Static Evaluator

EGGG generates the `score_board()` subroutine (the static evaluator) with the following procedure (given a board and the player whose turn it is):

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

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

       This step ensures that pieces with no moves and no surrounding empty squares are not ignored.

       Because we're looping through all the pieces, this rule rewards boards in which the player has more pieces and his opponent has less, so capturing is favored.

    2. Increment `piece_score` by the power of the piece times the *n*th root of the number of moves it has available. *n* is the dimension of the board (usually 2 for grid games) 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 *n*th root of the distance of that location from the piece, where *n* is the dimension of the board or density of the graph, as before.

       This rewards pieces played in the center of the board, which is good for chess but bad for Go. On a standard 8x8 chessboard, this weights the squares as follows (assuming unit power):

```
29.2 31.2 32.4 32.9 32.9 32.4 31.2 29.2
31.2 33.6 34.9 35.6 35.6 34.9 33.6 31.2
32.4 34.9 36.5 37.2 37.2 36.5 34.9 32.4
32.9 35.6 37.2 38.0 38.0 37.2 35.6 32.9
32.9 35.6 37.2 38.0 38.0 37.2 35.6 32.9
32.4 34.9 36.5 37.2 37.2 36.5 34.9 32.4
31.2 33.6 34.9 35.6 35.6 34.9 33.6 31.2
29.2 31.2 32.4 32.9 32.9 32.4 31.2 29.2
```

- If the goal is *comparative* — that is, your 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 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. On the other hand, it can be dropped into any game, and in most games it will be better than nothing. We make no claims that this is how static evaluators should be built; only that in the absence of deeper information about game strategy, EGGG has to make assumptions about game play that may well turn

out to be misguided.

Estimating Piece Power

To score a board, the static evaluator uses an estimate of the value of a piece, called the *power*. (We call it "power" instead of "value" to avoid confusion with the "values" that variables contain, and the "value" of a Munsell color.)

The power of a piece is computed as *motility* times *rank* times *rarity*.

Estimating Piece Motility

The static evaluator heuristic described above 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.

One way to do this would be to place the piece in the center of an empty board and calculate how many moves it has. However, that wouldn't allow for moves involving other pieces — castling or *en passant* moves in chess, or the moves of the cannon in Chinese chess (Xiangqi).

Instead, EGGG examines the rules involving the piece motion in the game description file. For each piece, the rank is calculated as follows:

- Set `$game{piece}{rank}` to 0.
- Extract all the rules involving motion of that piece, and loop through them:
    1. Increment `$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 step has the effect of weighting immobile pieces like flags and bombs more heavily than mobile pieces; really, this heuristic is 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, that it's 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 small 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.

   This rule may seem complex, and is best explained using chess as an example. Here is a rule for rook motion:

   ```
   Rook moves (x, y+1..7) if empty(x, y+1..7)
   ```

   And here is one of the rules for a bishop:

   ```
   Bishop moves (x+1..7, y+1..7) if empty(x+1..7, y+1..7)
   ```

   The bishop rule is longer, but only because it has a constraint that the rook rule doesn't have. This rule weights the rook (2^2-2^1) more highly than the bishop (2^2-2^2).

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

- Normalize the ranks so that the lowest ranking piece has rank 1.

The result, applied to the `chess.egg` game description:

```
$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 can't 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, Paper beats Rock. However, consider Stratego. In Stratego, 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 — but that's not the case. 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. Rock Paper Scissors has no such disparities, so each piece is given rank 1.

Chess has no ranks, because every piece can capture every other piece. (That isn't quite true — a king can't capture a king — but since that is a result of the prohibition against moving into check, it can't be deduced from such a cursory analysis of the game description.) So every chess piece has a rank of 1.

Estimating Piece Rarity

The third component of the power is the *rarity* of the piece.

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, 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. In conventional card games, all cards have the same rarity.

If the pieces are part of an enumerated type, as they are in Tetris, Diplomacy, and crossword puzzles, they are all considered to have the same rarity unless EGGG finds a rule that invokes the `random_weighted()` function from the `Eggg::Random` module. In that case, the proportional weights supplied to that function are used to establish the rarity.

The one exception to this heuristic is when the pieces are letters — say, for Hangman or crossword puzzles. In that case, EGGG uses a frequency table of English letters: E is less rare than T, which is less rare than A, and all the way on down to the most infrequent letter, Z.

Estimating Piece Expected Value

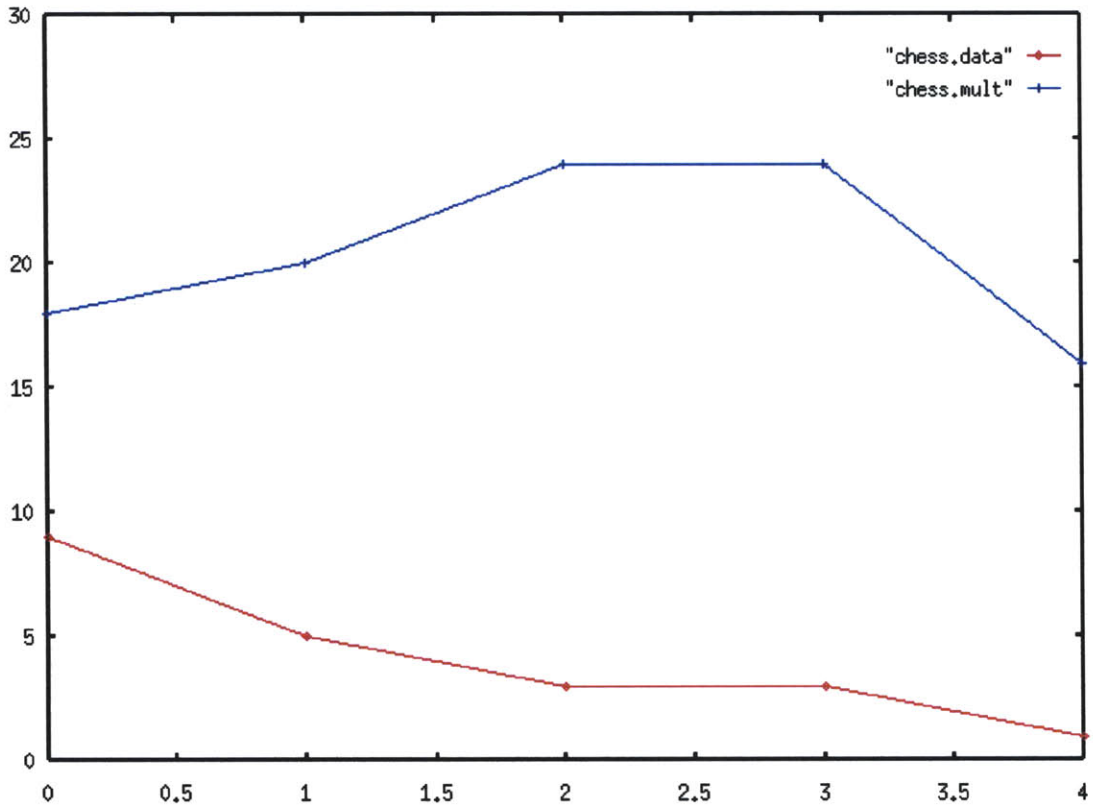
In some comparative games, the power of a piece can be calculated directly. First, we assume that each outcome is equally likely. Then, the possible outcomes have to be enumerated and the ranking of each established; the expected value of the piece is then the sum of the scores where the piece is on 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 — and it's hard to find out because the number of poker outcomes is so large. Similarly, an ace is worth more than a king in poker — and the difference between and ace and a king will be 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).

To our knowledge, no one has done this for regular draw poker; the analysis for draw poker is much more difficult than for stud poker, since it depends on the player's strategy. The probabilities for five-card stud poker are easily calculated.

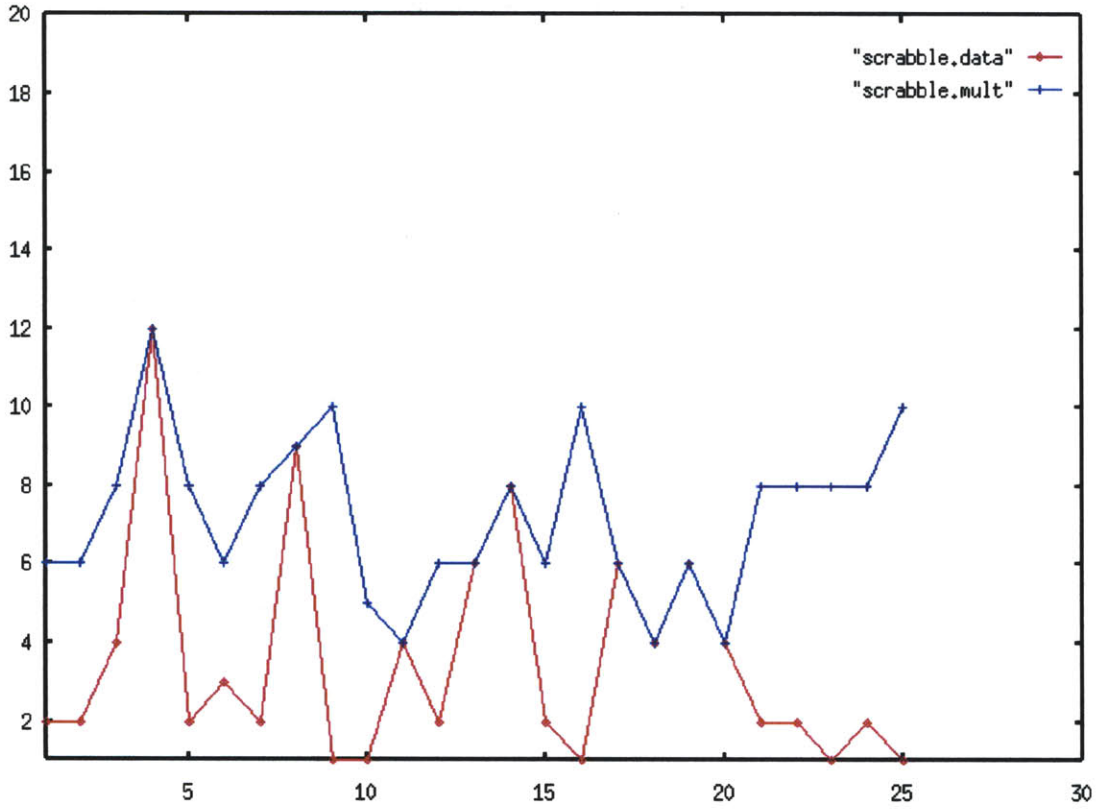
Now that we have a way to estimate the power of a piece, we've finished our generic static evaluator — and we can use it even for games of incomplete information, such as poker. In poker, the moves taken by players aren't responses and counterresponses, but they are moves. Unknown pieces are assumed to have the mean of the expected values of the set of all pieces, less the pieces that are known to the player.

As an aside, we conjecture that when pieces have numeric point values, their frequency of occurrence is often approximated by Zipf's law: the value multiplied by the frequency is a constant. (Zipf was actually interested about the ranked occurrence of words in a corpus.) Data is sparse, however; few games have pieces with universally agreed-upon point values. However, we will examine the applicability of Zipf's law for chess and Scrabble to illustrate what Zipf's law means. If we assume the 9-5-3-3-1 point values in chess (for the queen, rook, bishop, knight, and pawn), we can multiply those values by the number of times that each piece occurs in the starting configuration:



The lower line contains the point values, and the upper line contains the point values multiplied by the number of occurrences. The closer the upper line is to horizontal, the more Zipf's law is obeyed.

Here are similar calculations for Scrabble, where the point values are fixed by the rules of the game:



Next, we turn to an unconventional use of our generic minimax procedure — for helping players, and not computer opponents, solve single-person puzzle games.

Competition versus Cooperation

> *There is no general way of solving puzzles, of course; if there were, they would cease to be puzzling.*
>
> John Beasley, in *The Mathematics of Games.*

So far, we have discussed a generalization of a standard minimax procedure to handle arbitrary grid games, and a further generalization to any unsynchronized game in which the turns alternate from one player to another. Can we generalize our procedure further, to incorporate puzzles as well?

Minimax was designed for games in which players compete: moves at one level of the game tree, responses at the level beneath, counterresponses beneath that, and so on. EGGG identifies the player to consider for a given level by calling `player_next()`, which rotates clockwise in poker and alternates between players in chess. In single-player games, `player_next($state{turn})` simply returns `$state{turn}` — our game tree can be applied to single-player games. Our minimax becomes "maximax."

In puzzles, the computer opponent might be able to provide hints to the player. If the solution is part of the game description, this is trivial; a far more interesting question is whether the computer can help even when it has no access to the answer. [Hofstadter 95] attacks this problem for number sequence puzzles; with a much shallower analysis we might be able to help players solve letter puzzles like crosswords. Letter puzzles offer greater opportunities for easy help, because EGGG could look in word lists to help players solve crosswords, and use letter frequency lists to help people play Hangman.

(This has not yet been implemented in EGGG.)

Common Strategies

> *Any truly comprehensive game theory would therefore have to include a thorough understanding of the human psyche.*
>
> M. Eigen and R. Winkler, *Laws of the Game*, p. 16

In this section, we discuss *analytic* strategies — 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 ([Orwant 91] and [Orwant 93]) 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 (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'll next read his newspaper? That can be viewed as a game of incomplete information, and EGGG uses the same framework to make its assertions about players as Doppelganger did about newspaper readers.

We call these strategies "analytic" because their goal is to uncover the reasons behind the player's actions. In the following section, we discuss *synthetic* strategies — proactive decisions that EGGG makes to win games. These include generic rules of thumb for choosing moves apart from a minimax model, and decisions that EGGG makes to intentionally mislead players. In a word, bluffing.

Analyzing Strategies With Hidden Markov Models

Given a discrete series of observations, and a series of mathematical models that generate observations, a Hidden Markov Model technique allows a system to determine what model is most likely to be generating those observations. For our purposes, the "observations" are player moves, and the "models" are player strategies. EGGG's goal is to have EGGG determine the player's strategy given his moves, and it uses a Hidden Markov Model algorithm (in particular, the Viterbi lattice algorithm, described in [Rabiner 89]) to make that determination.

A few 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 and the game history. It 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? And so on. The result allows 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 during the game.

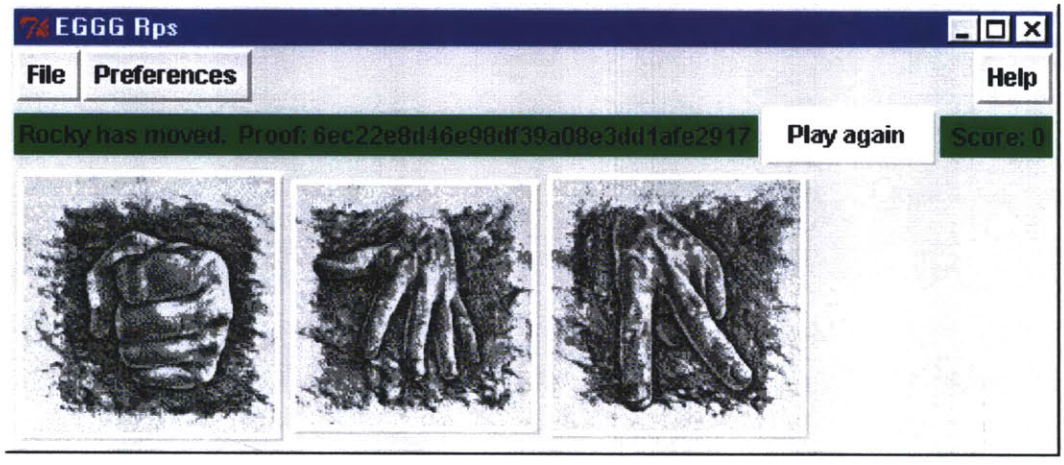
Rock Paper Scissors

The Hidden Markov Model approach is best explained with an extensive example. We choose Rock Paper Scissors, for two reasons. First, it's a very simple game, with only three kinds of moves. Second, 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
3x1 grid
```

A screenshot of the generated game:



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 move that beats it. In a session of one round, it is hard (but not impossible) for EGGG to have a better than 1/3 chance of beating a human opponent. In a session of multiple rounds, EGGG records the move histories and attempts to infer the player's strategy. Obviously, the longer the history, the more information EGGG has to reveal the strategy, and the better its predictions will be.

Predicting what your opponent is going to do is a special case of predicting what your opponent's strategy is: the Hidden Markov Model problem. We now enumerate the strategies that EGGG considers.

Nothing Beats Rock

This is the simplest of all strategies: the player picks one symbol and chooses it again and again. Some of the people that have played EGGG's Rock Paper Scissors adopted this strategy because they figured that it's what EGGG would least expect. Unfortunately for the players, it's easy to deduce when this strategy is being used.

In games like poker, the "Nothing Beats Rock" strategy maps to making the same sequence of decisions at each phase: perhaps always betting ten and then calling, or always calling and then raising ten, without regard to what opponents do.

Randomness

The most common player strategy in Rock Paper Scissors is to aim for a draw over the long term: choosing randomly 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.

In the Rock Paper Scissors trials at the 1997 News in the Future Consortium, players played ten rounds of Rock Paper Scissors with EGGG. As an exercise, write down a sequence of ten R's, P's, or S's, simulating how you would play EGGG if you were trying to guess randomly. The strings you generate 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. Note that none of the seemingly random strings have runs of three. This is the key to the strategy: when players try to guess randomly, they usually do so in a predictable way.

As it turns out, one would expect 56.4% of randomly generated strings of ten symbols to have a run of three. That is, it's more likely than not to find an RRR, PPP, or SSS, then to find none of them. And obviously when more than ten rounds are played, the likelihood of a run of three will increase.

If someone following this strategy makes the same move twice in a row, they are unlikely to choose it a third time. In a game like Rock Paper Scissors, the computer opponent can then choose the piece that the move would have beaten, and be assured of either a draw or a win.

Switcheroo

A less patient player might simply decide never to do the same thing twice in a row. This strategy is even easier to identify and exploit, since it requires looking at only the last move, instead of the last two moves (to decide what response to make) or the entire history (to determine the confidence that the player is using this strategy).

Generalized Tit For Tat: The Vendetta

"Tit for tat" is a general strategy that can be summarized as "If you're nice to me, I'll be nice to you." It was found to be a good strategy in a famous iterated Prisoner's Dilemma contest many years ago, and people routinely follow this strategy in Diplomacy, Dungeons and Dragons, Risk, Monopoly, multiplayer Magic, and even in casual poker, where the desire to beat someone who beat you earlier can sometimes override a more logical analysis of the game — especially when the stakes are low. The greater the number of players in a game, the more likely vendettas will materialize, for there is greater opportunity to make, and therefore break, alliances.

Sacrifice

Many games involve the notion of *sacrifice* — accepting a short-term loss for a long-term gain. Games that people bet on typically don't involve sacrifices. Games that do involve sacrifices can be divided into three categories:

- Games where you forgo a good move now (or forgo avoiding a bad move) for a great move later (gin, Scrabble)
- Games where you have to go two steps back to go three steps forward (Rubik's cube, Othello)
- Games where you concentrate resources in one area to the detriment of another (Go, Risk, chess).

EGGG currently has no way to identify sacrificial moves other than searching the game tree to a sufficiently great depth.

Synthesizing Strategies

In the previous section, we discussed a few patterns that EGGG looks for in player moves. In this section, we discuss a few strategies that EGGG can generate itself.

Freedom As A Universal Strategy

In nearly every grid and graph game, it's better to have more opportunities than fewer. Chess is the only common example where this maxim fails, because a stalemate results in a draw rather than a loss. This suggests a simple and often surprisingly effective strategy: moving to maximize the ratio of the moves you have available to the moves your opponents have available.

This strategy is part of the generic static evaluator discussed earlier.

You Cut, I'll Choose

The notion of freedom as a universal strategy is similar to the children's algorithm for dividing up dessert, or other divisible resources. One child divides the resource, and the other child chooses which division to consume.

What these two strategies have in common is the notion of *equalizing an opponent's options*. [Beasley 89] notes that this is the ideal bluffing strategy as well: ensuring that the opponent has no reason to favor one move over another.

Beasley notes that this strategy can be applied to any contest where the probabilities of success for different players are unequal. As an example, he describes a

construction worker who bets another that he can demolish a building faster. The problem is that there's only one building to demolish.

> *If he believes the abilities to be unequal, he should stipulate a target at which he estimates that his own chance of success equals his opponent's chance of failure.* Note that this rule applies whether or not he regards himself as the stronger. *If he is indeed the stronger, it guarantees a certain chance of success (assuming that his estimate is indeed correct); if he is the weaker, it minimizes the amount that his opponent can gain.* (p. 70)

Hypotheses

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's trying to be random but really isn't, or perhaps he is following some strategy that is more likely to choose one piece than another. EGGG has the player's move history available to it, and it's easy to tally the different moves made and identify which was the most common.

The problem is in interpreting these results. 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. When EGGG makes hypotheses of this sort, it uses Perl's `Statistics::ChiSquare` module, by the author of this dissertation, 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.)

Estimating Player Ability

In a game like Go, both human and computer players usually try to play the most rational game possible. It doesn't matter who the opponent is; there is a single course of action that is "provably" best. This assumes that the opponent will play the best possible game, when in fact he is far more likely to play merely to the best of his ability— it assumes that players don't model the psychological state of their opponent. If you know that your opponent has a particular weakness, it should obviously be exploited to maximize the chances of winning. You can never know for sure whether you've uncovered a true weakness, or are being hustled. You have to consider likelihoods and confidence levels, and that is why game theorists prefer the comfort of game models in which the players are rational and skilled. The paradox is that if players actually were equally skilled and maximally rational, most gaming competitions would be moot.

This is true even for games of incomplete knowledge, like poker. Part of what makes casual poker enjoyable is that players can continuously model one another: the actions and gestures that reveal, or fail to reveal, or merely *seem* to reveal information beyond the simple play of the game.

The notion of modeling opponents thus includes ascertaining not just how skilled they are, but what particular foibles they have. Apart from the strategy analysis described in the previous section, EGGG has no way to identify foibles; all it can do is try to guess a player's skill. In a game like chess, that means determining how good the player is; in a game like poker it means measuring both the player's success rate and his skill at bluffing. A better understanding of games would enable EGGG to make more profound inferences, such as "this player is weak in the endgame" or "this player is more likely to bluff when this other player remains in the game", but such an understanding is beyond EGGG's capabilities at the moment.

If a player beats EGGG at chess, that tells us that the player is probably better than EGGG at chess. But what does that tell us about the player's innate ability? Less than it would seem. The chess ranking system assumes that a player's performance can be described by a normal distribution, but as [Beasley 1989] points out, this is almost certainly not the case, and the research supporting this contention is specious. Beasley is clearly correct; the tendency to see the normal distribution everywhere unfortunately manifests itself in Arpad Elo's research, used as a justification for chess rankings. Elo wrote *The Rating Of Chessplayers, Past And Present*, and the Elo rating system has been used to rate chess players since 1970; today, it's used to rate dozens of different board games. Elo calculates his standard deviations without allowing for draws, and he does not take into account the circular nature of his system: the ratings that his system generates are used to estimate the strengths of players, which are then used to justify the ratings that his system generates. Beasley notes that other sports publish only rankings and not claims about not absolute strength. Such statistics are less precise than Elo's rankings, but undoubtedly more accurate.

It is unfortunate how much emphasis U.S. researchers place on creating better chess programs, because chess games tell us so little about the strengths of the players. Unlike Go, chess has no handicap system, so unevenly matched players must play an unevenly matched game. Game outcomes in chess are trinary (win/lose/draw); in Go, there are no draws, and a player wins by a certain number of points (stones). The higher the number, the greater the disparity between the player abilities. A Go game doesn't just reveal who the winner is, it reveals *how much* stronger the winner is than his opponent. The result of a chess game has just over 1.5 bits of information; the result of a Go game, nearly 8.5.

Sometimes winning 99% of your games is better than winning all of them. Consider someone who plays Windows Solitaire instead of working on his dissertation. Because of the length of his dissertation, he plays a lot of Windows Solitaire, and becomes pretty good at it. Windows Solitaire records the number of games you've played, and displays that as well as your winning percentage. If player A wins five games, his winning percentage is 100%. If player B plays one hundred games and

wins 99, his percentage is 99%. Who is the better player?

In binary (win/lose) games, EGGG estimates the strength of the player (relative to EGGG) with the beta distribution. This is how the Doppelgänger user modeling system [Orwant 93] estimated a reader's interest in particular topics. 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. The greater the number of trials, the lower the variance, and the higher the confidence. The variance of the beta distribution for a and b (a wins, b losses) is:

$$\frac{a * b}{(a + b) * (a + b) * (a + b + 1)}$$

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. Strengths range from 0 to 1, and confidences can be any non-negative number. Note that the confidence for both unbeaten and winless players will be zero. This makes sense, because until *some* upper and lower bound on a player's strength has been established, assertions about the player's ability cannot be definitive. For our Solitaire example, the strength of someone who wins five games and loses none is unbounded — but the confidence is zero. In contrast, someone who wins 99 times and loses once has a strength of 0.99 and a confidence of just over 10,202.

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 that 0.8? This is why the beta distribution was chosen; it allows EGGG to establish confidence levels for estimates other than the most likely.

In trinary (win/lose/draw) games, a draw is treated as two separate events: a half-win and a half-loss. A player who plays ten times, winning four, drawing three, and losing three, would have a calculated strength of 0.55 and a confidence of 4.44. It is unclear whether this is the best approach.

In comparative (win by a certain number of points) games, a different approach is required due to the extra information available. Implicit in our use of the beta distribution was that the individual trials could be modeled with a Bernoulli distribution, but Bernoulli will no longer do because each sample is an element of a range rather than simple success or failure. We have to choose another distribution to base our analysis on. We choose the normal distribution, and hope that we're not committing the same error that Elo did when he established rankings for chess players. However, we feel that using the normal distribution to model the variation in one player's play against another is a much lesser sin than using the normal distribution to model play against a population of players.

Note that in these analyses we used a Bayesian approach: we chose a probability distribution (strictly speaking, a probability *mass function*) and then attempted to estimate the distribution parameters. There is an ongoing controversy between Bayesians and classical statisticians. The classical approach is more conservative, involving no assumptions about the underlying distribution. The price you pay for this risk avoidance is a less-sweeping result: classical statistics give the best results only when the data set is large. Since EGGG needs to make decisions based on sparse data, the Bayesian approach was selected.

Bluffing

> *An inexperienced player thinks of a bluff as a manoeuvre whose objective is to persuade his opponent to play wrongly on a particular occasion. There are indeed many occasions on which such bluffs are successful, but their success depends on the opponent's lack of sophistication.*
>
> John Beasley, *The Mathematics of Games*, p.71

In addition to trying to determine what the player is thinking so that it can choose the best move, EGGG can try to fool the player as well. In this section, we turn to how computer opponents can bluff players.

Bluffing brings to mind two things: poker and negotiations. But bluffing is far more general than this, cropping up in almost every game. I once played a game of Go with someone far better than myself. He awarded me a nine-stone handicap — large, but we both knew it wasn't large enough to equalize the disparity between our abilities. A game of Go consists of many parallel battles; much of the skill of the game comes from realizing when a battle is lost or won, and concentrating your efforts elsewhere. The player who realizes that one corner is lost (or won) can play elsewhere on the board, and seize initiative.

This initiative is a critical part of Go play; the player that possesses it is said to have *sente*, and his opponent is said to have *gote*. When a player has *sente*, he is the protagonist. He acts, and his opponent reacts. His opponent has *gote*. You always want *sente*, and never *gote*. Whenever my opponent switched battles, I followed, because I knew that he could see farther down the game tree than I could. I willingly accepted *gote* — and essentially, willingly accepted a loss — in the hope that I would lose by fewer stones. Of course, this gave my opponent a prime opportunity to bluff. He says he didn't, and I believe him: Go players pride themselves on ignoring psychological evidence and being as rational as possible. Tricking someone into a bad play is dishonorable. Nevertheless, the option was available.

Because EGGG is a computer program, it has a valuable advantage. Players willingly accept *gote*, for the same reason that I accepted it when playing Go against the stronger player: players assume that EGGG can see farther than they can. We see

it in Rock Paper Scissors, where people give up trying to second-guess the computer and bet randomly instead; we see it in chess, where people try unusual opening moves in an attempt to avoid the well-trod paths of the game tree.

So we disagree with Beasley, who claim that the best bluffs are simply those which make the opponent's actions irrelevant in the long term. This is a good strategy for a static evaluator, but there are other strategies — the *psychological* bluffing strategies familiar to every casual poker player — that can be generated computationally even when they cannot be treated with the same mathematical precision of the more predictable games that Beasley analyzes.

In Beasley's defense, he hedges his claim with the assumption that opponents play with what he calls "sophistication". Yet Beasley's sophistication is quite rare in real games played by real people; it requires that people use every bit of the information available to them, and in the most rational manner possible. Yet psychological strategies clearly work for poker; they even work for Go. Only for games of perfect knowledge and low-to-moderate complexity, like Go, will psychological strategies nearly always fail.

Future versions of EGGG might take advantage of the fact that people are primed to perceive patterns where there are none. EGGG would present just enough behavioral cues to make it seem like it is following a particular course of action in order to mislead the player.

Let's categorize the different types of probabilistic games. There are two types of probabilistic games: games where the true odds are unknown (e.g. sports), and games where the odds are known. Games where the odds are known can be further divided into games where the odds are easy to calculate (e.g. roulette), and games where they aren't (e.g. poker). So we have three types of probabilistic games: unknown odds, easy known odds, and hard known odds. Note that it is important to distinguish the *odds of winning* from the *payout*. A 20:1 bet on the Cleveland Browns winning a football game falls into the category of unknown odds (as all sporting contests do), since it is impossible to calculate the actual chances of winning. Bluffing is only possible in games of hard known odds.

As our Go example showed earlier, bluffing is also possible in non-probabilistic games, and since EGGG maintains an estimate of a player's skill relative to itself, it could in theory try to mislead its opponents when the player is likely to perceive EGGG as being far superior. It does not do this at the moment, EGGG models players, and it models player strategies, but it does not model the player's estimate of the skill disparity. EGGG might believe that it is stronger than the player, but it will never hypothesize that the player believes EGGG to be stronger when it is in fact weaker, so it is unable to attempt *sente/gote* bluffing.

So we are left with bluffing for games of hard known odds. We will turn to EGGG's technique for bluffing soon, but to set the stage we will first talk about making interesting moves.

Making Interesting Moves

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 attractive to least, and early versions of computer opponent 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. It always opened chess games by moving its king pawn forward two squares. So what EGGG does is to choose randomly, but to make the probability of each move proportional to its attractiveness. So if five moves are equally attractive, EGGG is as likely to choose the first as the last.

We don't want our proportionality to be linear, however. 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 move one-tenth as often as the better move. So in the absence of other information, EGGG squares the linear distribution (a mass function, actually) and then uses the `weighted_distribution()` function in the `Eggg::Random` module to choose its moves. The move with the score of 10 will then be chosen one-hundred times as frequently as the move with the score of 1.

Probabilistic Bluffing

This framework for interesting moves helps computer opponents play *more interesting* chess, but it can also help them play *better* poker. We can generalize the notion of applying a weighted distribution to our move choice, developing it into a generic bluffing strategy. No strategy is provably optimal, and we make no claims about the effectiveness of what we describe here.

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, and more likely that it will choose an amount that is either higher or lower.

The extent to which EGGG considers non-optimal betting amounts is determined by a state variable, `$state{bluff}`. The higher the value, the more random EGGG's bets will be. The procedure is as follows: EGGG takes the squared mass function described in the previous section, and raises each value to power of `1/$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. When an EGGG opponent finds that a bluff fails, `$state{player}{bluff}` decreases; when a bluff succeeds, it increases.

In poker, bluffs can go undetected; a player who scares his opponents out of the round doesn't have to reveal his cards. Since the computer-generated opponent is part of the game program, EGGG could orchestrate omniscience easily. It does not. Some players will trust EGGG not to cheat, and others need to be convinced.

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 won't cheat. In games of zero information, like Rock Paper Scissors, the player presses an icon and is immediately greeted with "You lost!" or "You won!" or "Draw." How can the player verify that the computer didn't cheat?

EGGG uses Perl's MD5 module to compute a cryptographic signature — a hexadecimal message digest of a few random words and numbers concatenated with the text of EGGG's choice. The result was visible near the top of the Rock Paper Scissors screenshot shown earlier:

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

The `6ec22e8d46e98df39a08e3dd1afe2917` is the message digest. The MD5 digest algorithm is one-way; given that string of hexadecimal digits, it is computationally intractable to uncover the message that was digested. But if the player selects `Verify`, EGGG will reveal the message, ensuring that it had already made its selection when the screen was displayed.

Chapter 5: Beauty on the Inside (Graphic Layout)

Drunkard: "Will I ever, ever get home again?"

Polya: "You can't miss, just keep going and stay out of 3-D!"

G. Adam and M. Delbrueck, *Reduction of Dimensionality in Biological Diffusion Processes*

When I was a child, one of my favorite toys was a ten dollar handheld electronic game. It consisted of fifteen diodes. You'd press one button, and a series of diodes leading to your thumb would illuminate. When the diode closest to you lit up, you had to press another button. If your timing was right, other diodes would light up. That was it. That was the whole game.

But because the diodes were arranged in a particular sort of diamond, and the areas of the game field without diodes were painted green and brown, I was playing baseball.

How a game appears to the player has a profound impact on the player's experience. When the game is a simulation of a real-life activity like my electronic baseball game, it fosters a mental model of the gaming activity that would not otherwise be present. The play of the game is more easily learned as a result, and the strategies that a player chooses are affected as well.

The similarities between games extends to their appearance. Some of these similarities are irrelevant from a developer's perspective. For instance, the theme of a game — say, the medieval theme of games like Dungeons and Dragons or Magic — is important to someone who's marketing the game or designing artwork, but it is not critical information for developer concentrating on the core of the game's design. All he needs to know is that the game has a theme and that it is important, because if so he must render the game with an eye toward detail. EGGG doesn't generate artwork, and its games do not have the level of detail that such themed games require for popularity.

In Chapter 2, there were three graphical components included in our categorization of games: topology, board, and piece. In this chapter, we will explore how the different types of boards and pieces are rendered.

Topology merits a quick discussion. Most games are two-dimensional, and EGGG is best suited to generate two-dimensional games. One-dimensional games like our visual rendition of Rock Paper Scissors are treated as degenerate two-dimensional games: that's why the game description for Rock Paper Scissors includes the statement `3x1 grid`. Zero-dimensional games are just games with no spatial

meaning, and that gives EGGG no clues about how to render the game. Typically, EGGG won't be able to render the game; only MUD-like games such as Mammon are possible at the moment. EGGG is also not capable of handling games in three dimensions, for the most part because the Tk graphical library on which it relies does not support three-dimensional rendering.

So in this chapter, we will discuss only the rendering of two-dimensional elements: the elements that we called boards, squares, and pieces in our ontology.

EGGG stores the display-dependent aspects of the game in an object named `%display`. This object can be saved to disk for persistence of gaming: players can store and recreate any game, and it contains attributes such as `$display{paused}`, `$display{iconified}`, `$display{height}`, `$display{color_depth}`, and `$display{granularity}`.

Before we turn to boards, we'll briefly discuss what we'll call the *frame* of a computer game. The frame includes the essential elements of computer games not present in traditional games: graphical elements like the titlebar, window, and menubars.

The Frame

When a traditional game is "ported" to a computer screen, certain decisions that need to be made as a consequence. For instance, there needs to be a way for players to quit the game. In traditional game play, there's no commonly accepted way for players to stop playing; they just agree to do so. On the other hand, most windowing systems provide a uniform way to let users quit their applications, typically via a button on the upper right-hand corner of the titlebar. The appearance of the titlebar is often customizable by the user or fixed by the windowing system, so it's out of EGGG's control.

What we call the frame has three components: the window itself, the menu bar, and the status line.

The Window

An early version of EGGG generated four windows for each game. There was a window for the board, a "control" window that contained the menu bar, an "opinion" window that conveyed EGGG's assessment of how well the computer opponent was performing, and a picture of a face that conveyed the performance graphically.

The current version of EGGG has only one window, ever. This was an aesthetic decision — since many window managers sometimes force people to cycle through all the windows on the screen to arrive at a particular desired window, we decided to minimize the number of windows that players had to cope with. The disadvantage is

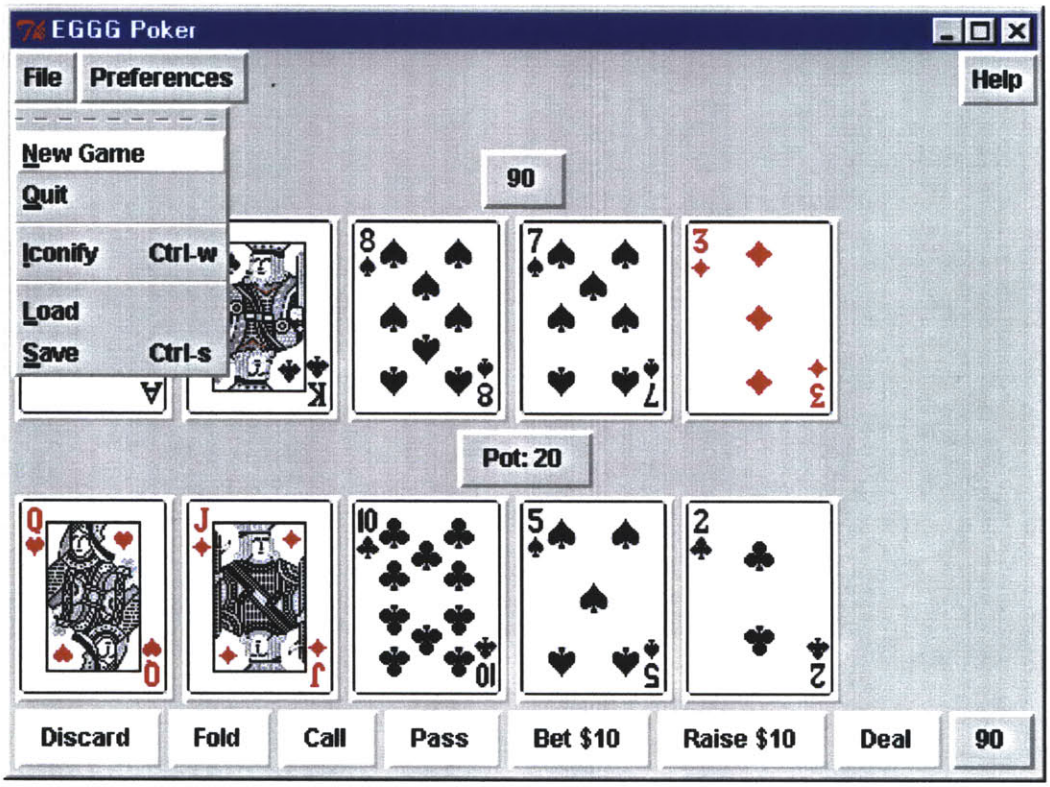
that now EGGG has to place all of that information onto the screen somehow. We discuss how this happens throughout the chapter.

The Menu Bar

EGGG generates the pulldown menus familiar to most computer users. Every game has a "File" menu, a "Preferences" menu, and a "Help" menu. These provide application-oriented features that are common to all computer games.

The File Menu

In the figure below, we see the five File options: `New Game`, `Quit`, `Iconify`, `Load`, and `Save`.



`New Game` restarts the game, running the entire application again. The current game window is destroyed and a new one takes its place. This is different from the `Play Again` button you see on games with rounds, such as Rock Paper Scissors and poker.

`Quit` quits the game, appending a notation that the game was terminated prematurely to the game's global history. (Global histories are discussed in the next chapter.)

`Iconify` replaces the game window with a much smaller version, or makes it disappear entirely and adds an entry to the icon manager. Precisely what happens depends on the window manager and any user customizations to the manager.

Players can also save the state of a game (`Save`), which dumps the `%state` structure to disk using the Perl `Data::Dumper` module. The saved game can then be loaded in later with `Load`. The state of the game is exactly duplicated, with one difference: `$state{loaded}` is set to the time value at which the game was loaded. If the game is timed (that is, `$game{timed}` is true), then the results are assumed to be corrupt and the game result is not appended to the game's global history. This lets players cheat, but it doesn't let them impress EGGG.

The Preferences Menu

The Preferences menu allows players to change a few elements of the `%state` data structure. By default, players can change only one element: their player name. EGGG uses the login name of the player if the operating system supports login names, and "player" otherwise. The player names are used to identify the player to others in multiplayer games, and to identify the player to EGGG in the global history.

If EGGG is able to generate a computer opponent for the game, the Preferences menu will include the `Computer opponent` option you see pictured above.

If the game description includes a line that begins with `player can set`, an entry is automatically added to the Preferences menu. This statement might look like this:

```
player can set level to 1..8
```

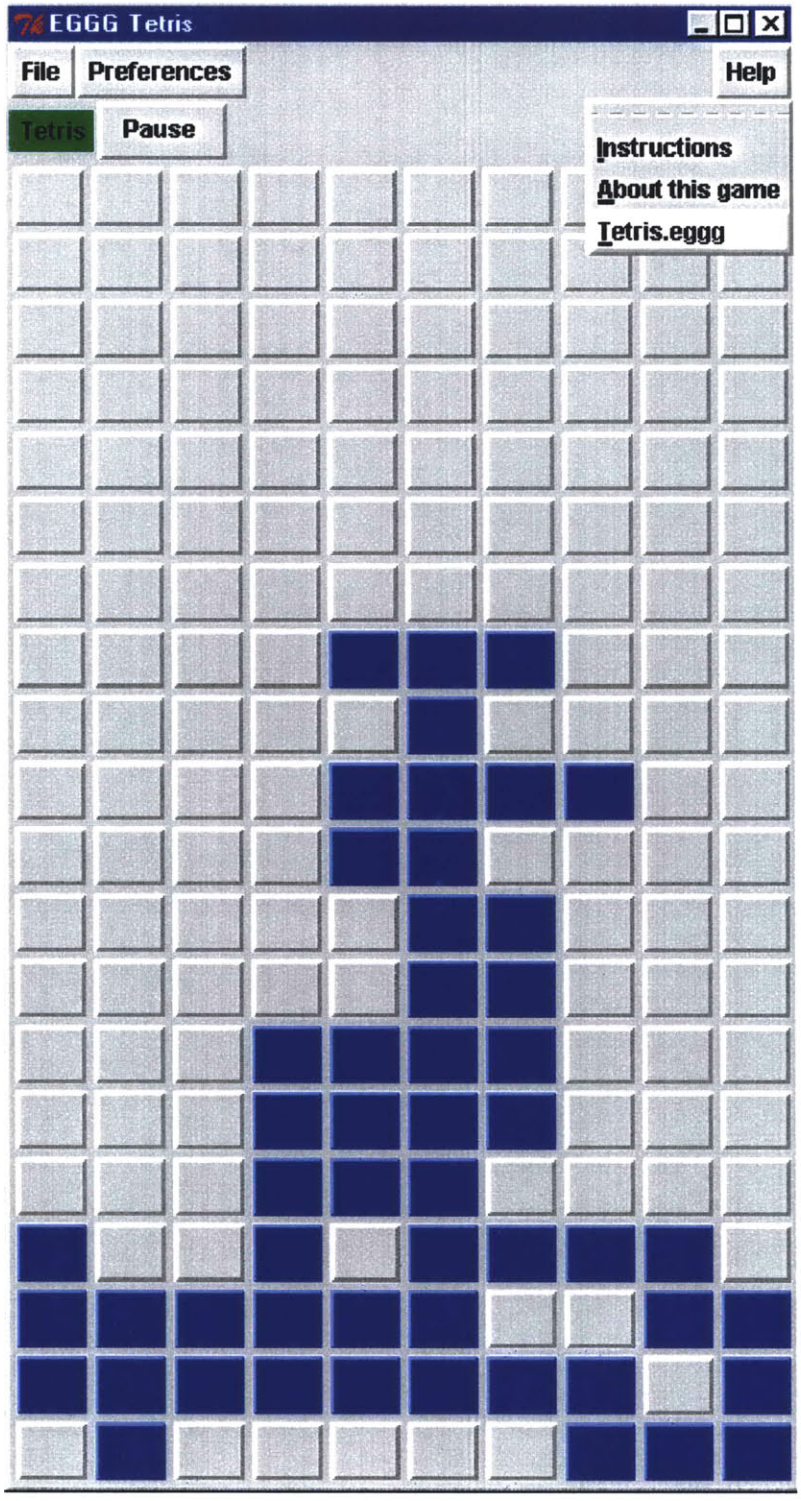
Such a statement causes EGGG to add a `Set level` option to the Preferences menu; selecting that option would then let the player set `$state{level}` to an integer from one to eight. If the line had been this, players would be allowed to change the size of the board while the game was running:

```
player can set board size to 2..8
```

The board would then be cropped along the top and right edges, since EGGG's coordinate system places (0, 0) at the lower left.

The Help Menu

Every Help menu has an `Instructions` option that displays (in a new window, regrettably) the game's instructions, generated automatically with the `egggdescribe` utility mentioned in Chapter 3.



To encourage players to learn about EGGG game descriptions, the Help menu allows players to see the game description for the game they're playing: that's the `Tetris.eggg` selection shown. Puzzle games such as crosswords have the solutions embedded in the game description; if EGGG finds a `solution` line in the game description, it omits this option.

The Status Line

Below the menu bars is what we call the *status line*, which can contain up to three elements: the message field, a `Play Again` button, and a score field.

The Message Field

Every game has a *message field*, which is used for information that EGGG wants to convey to players. Announcing that someone has just won or lost, or stating whose turn it is, or complaining about illegal moves: these are all messages that appear in the message field.

EGGG has an internal subroutine called `display()` that paints the message on the screen. Most regular buttons invoke `display()`. (By "regular", we mean buttons that aren't themselves grid squares.) The message field is always on the left side of the game window.

Play Again

Games with rounds have a `Play Again` button immediately to the right of the message field. An earlier version of EGGG assumed the player would always want to play another round, and began a new round as soon as the old one ended. However, the new round overwrote the message field from the old round, making it difficult to tell who won the old round. That's why the `Play Again` button was added.

The Score Field

For games where the player has a score, EGGG creates a score field to the left of the `Play Again` button if there is one, and to the left of the message field otherwise. Nothing is ever placed to the right of the score field, so it ends up defining the right edge of the game window.

Whenever the player's score (`$state{score}`) changes, the score field is updated appropriately. In some games, lower scores are better that higher ones, but this has no effect on how the score field is displayed: when the window updates, it merely dumps whatever is in `$state{score}` into the field and prepends `Score:`.

If there is a statement beginning `score starts` in the game description, EGGG initializes `$state{score}` to that value. The values needn't be strings, either. Statements like this let the score increment through an array of values:

```
score is [Acolyte, Prestidigitator, Thaumaturge, Magician, Wizard]
```

If a line like this is present in the game description, `$state{score}++` will advance `$state{score}` along this array, terminating at `Wizard` (and not wrapping around back to `Acolyte`).

The Board

> *"Ah yes," he says. "Hunt the Wumpus. Hunt the Wumpus was another one of those fifty-line BASIC programs, although this one was more like two hundred lines. It was a network of tunnels and nodes. And I believe the actual geometry of the network was a dodecahedron. So there were twenty vertices with three tunnels coming to each node."*
>
> J.C. Herz, in *Joystick Nation*, p. 10, quoting Walt Freitag

We now turn to the primary element in the window: the board. Recall that by our definition, it's not just board games that have boards. *Any* game with spatial meaning has a board. Poker has a board, and that board is a blank canvas onto which cards are dealt. Rock Paper Scissors has a board: the display on which the choices are made available. Only Mammon (described in the next chapter) has no board, because it is a purely text-based game.

The Geometry Manager: Packing, Placing, and Gridding

The first stop in our survey of board architecture has more to do with the window manager than with the board itself. Here, we talk about the geometry manager used by EGGG games. The geometry manager creates new graphical elements on the screen and decides where to place them, whether to make them "stick" to each other or to the borders of the screen, and how the graphical elements should adjust if the window dimensions change.

The Tk library provides three managers: the *packer*, the *gridder*, and the *placer*.

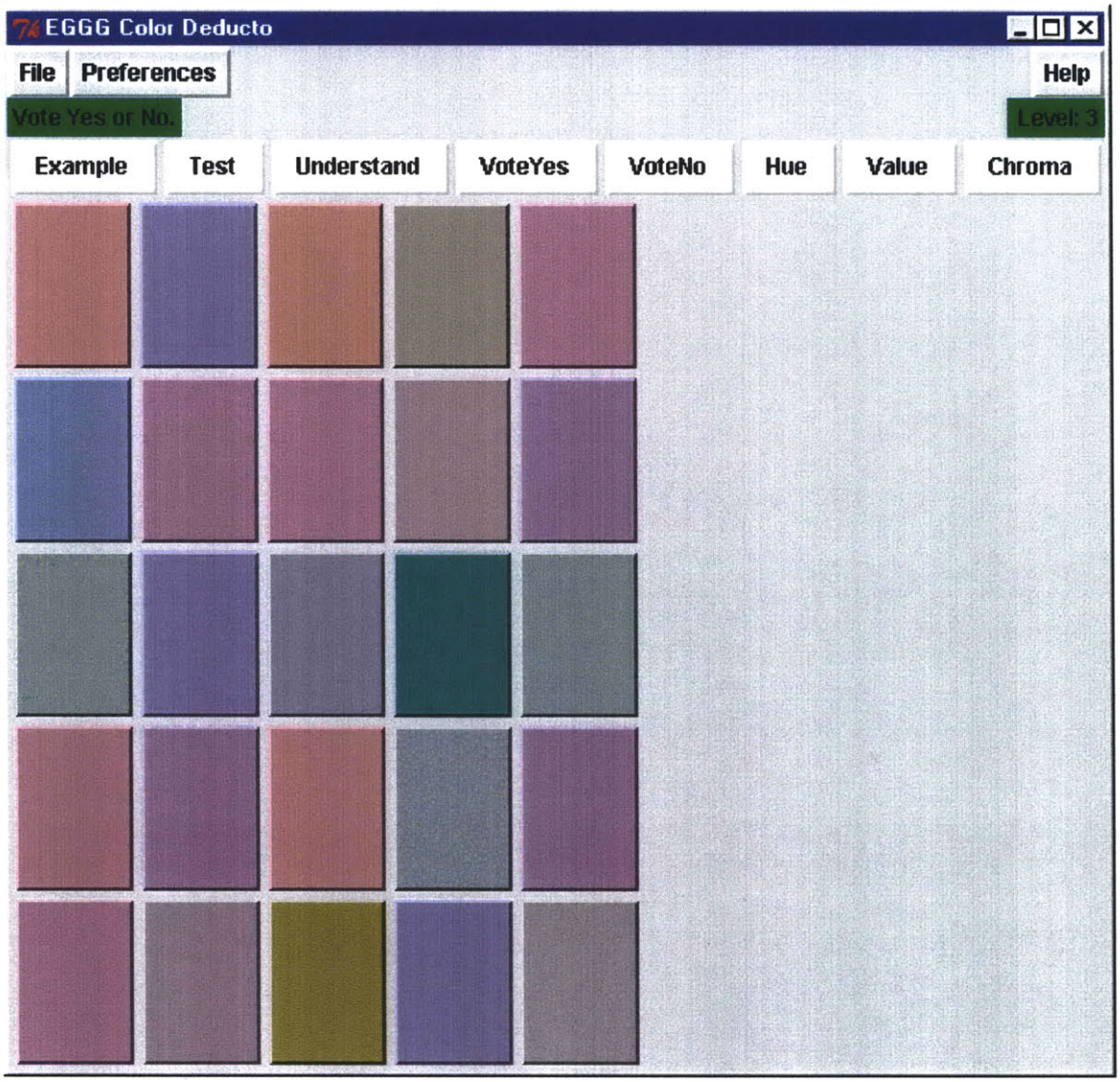
- The gridder lays down an invisible grid over the screen, letting programs place elements onto the grid given *x* and *y* coordinates along the grid axes.
- The placer lets a program position a graphical element anywhere on the screen, without regard to the positions of other elements.
- The packer "packs" the graphical elements onto the screen. When an element is rendered, the geometry manager locates it in, say, the leftmost available

space (`$element->pack(-side => 'left')`), or the bottommost space (`$element->pack(-side => 'bottom')`). The element can be packed in one direction but anchored (attached) in another: `$element->pack(-side => 'bottom', -anchor => 'n'))` With the packer, it is difficult to stack elements into a grid without creating an intermediate graphical abstraction called a frame.

Each geometry manager is good for certain classes of games. The packer is generally the easiest manager to use, because it deals in relationships between the graphical elements. So if the user resizes a window and the elements need to be repositioned, the relationships between them are maintained automatically.

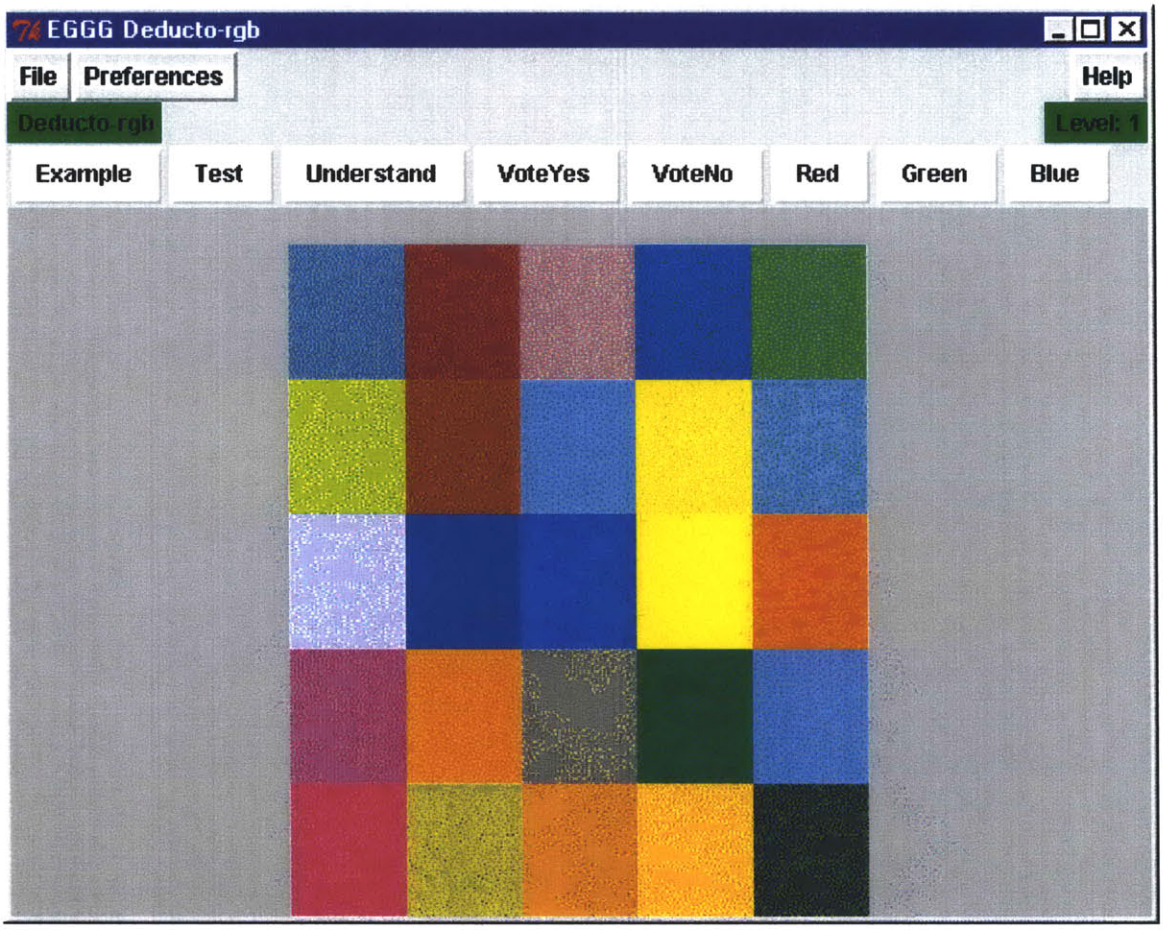
It's awkward for an applications to use more than one geometry manager, and early on in the implementation of EGGG we settled on one. The gridder clearly was not appropriate; it would be fine for some grid games, but extremely awkward for other games. The placer was attractive because of its pixel-by-pixel control, but we settled on the packer because of the ability to express relationships between graphical elements. This is especially important; since EGGG can't know what elements it will have to generate before reading the game description (and sometimes not even then, if the board structure changes during the course of the game), it made sense to choose the manager that represented positioning via relationships instead of pixels.

This was the wrong choice. In the tradeoff between expressivity and convenience, the packer was too far toward the right. An example of the packer's inadequacy was revealed in the rendering of Color Deducto.



The mullions (gaps between the squares) are undesirable, since they affect how the color relationships are perceived by the player, which is the heart of the game. However, the mullions are an unavoidable consequence of using the packer.

Ideally, the squares in Color Deducto would be flush against one another, like this:



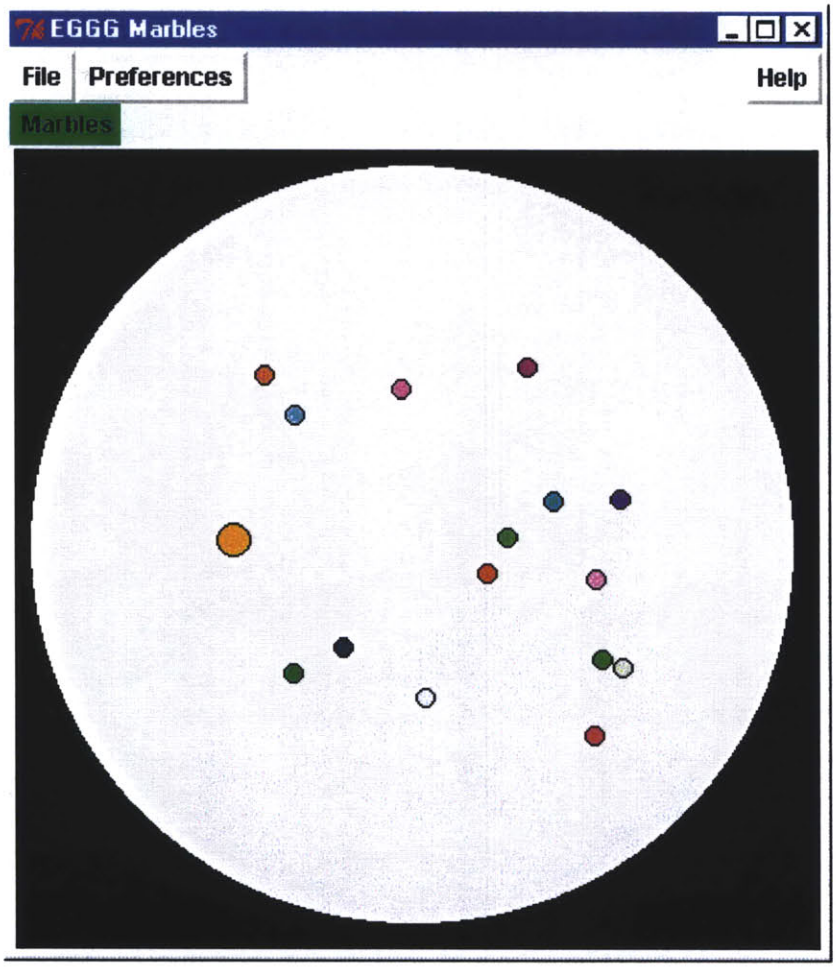
This is not an actual game created by EGGG, but rather a Color Deducto game that was patched after generation to use the placer instead of the packer.

Board Shape

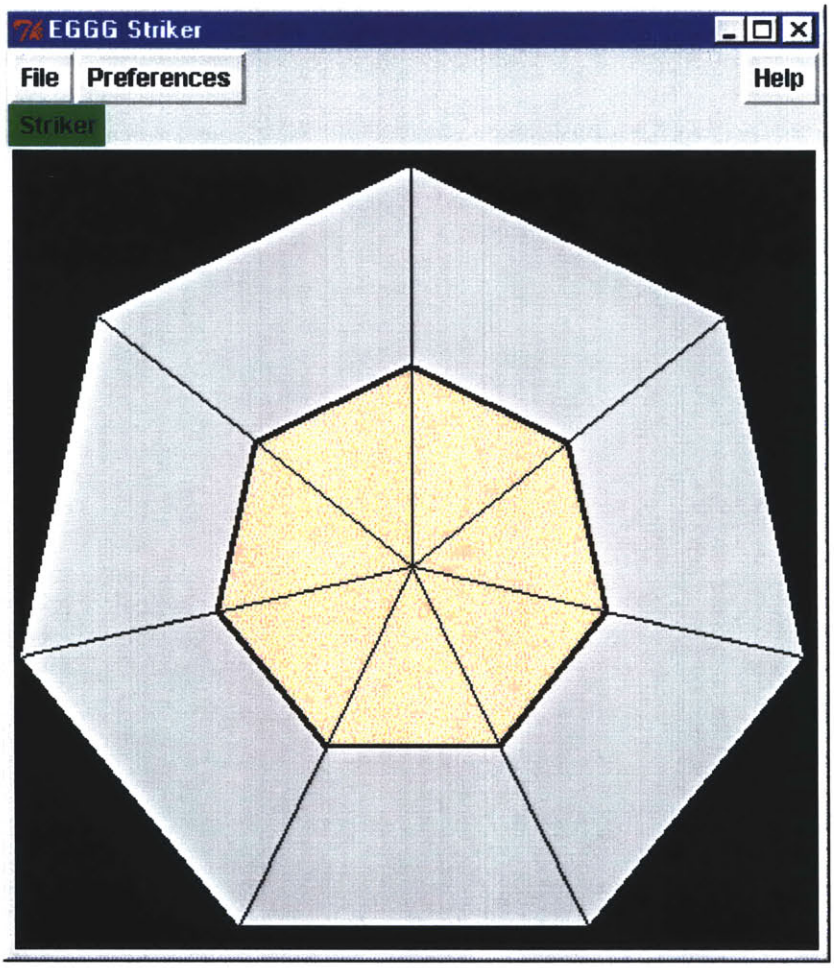
In this section, we turn to the shape of the overall board. Here, we are unfortunately constrained by the fact that many window managers are unable to display non-rectangular windows. Game descriptions can include lines like this:

```
board is circle
```

The result will be a circle inscribed inside a square:



EGGG recognizes any equilateral polygonal board shape. A game with the statement `board is heptagon` yielded this:



Polar coordinates are used to determine the vertices of the polygon, and by default the vertices are aligned so that an edge is parallel with the bottom of the screen, as shown above. This was done because many games that have polygonal boards expect that each player will "own" a side. Since players playing an EGGG game aren't likely to tilt their heads, it makes sense to ensure that the bottom of the board is an edge and not a point.

Squares

Our Soldiers and Lowest Classes of Workmen are Triangles...

Our Middle Class consists of Equilateral or Equal-Sided Triangles.

Our Professional Men and Gentlemen are Squares (to which class I myself belong) and Five-Sided Figures or Pentagons.

Next above these come the Nobility, of whom there are several degrees, beginning at Six-Sided Figures, or Hexagons, and from thence rising in the number of their sides till they receive the honourable title of Polygonal, or

many-sided. Finally when the number of the sides becomes so numerous, and the sides themselves so small, that the figure cannot be distinguished from a circle, he is included in the Circular or Priestly order; and this is the highest class of all.

Edwin A. Abbott, *Flatland*, p. 8-9

Many games can be represented as grids even when there's nothing truly gridlike about the game. For instance, Rock Paper Scissors is represented as a 3x1 grid. These are not the grids of the Tk library; the grid abstraction is entirely within EGGG.

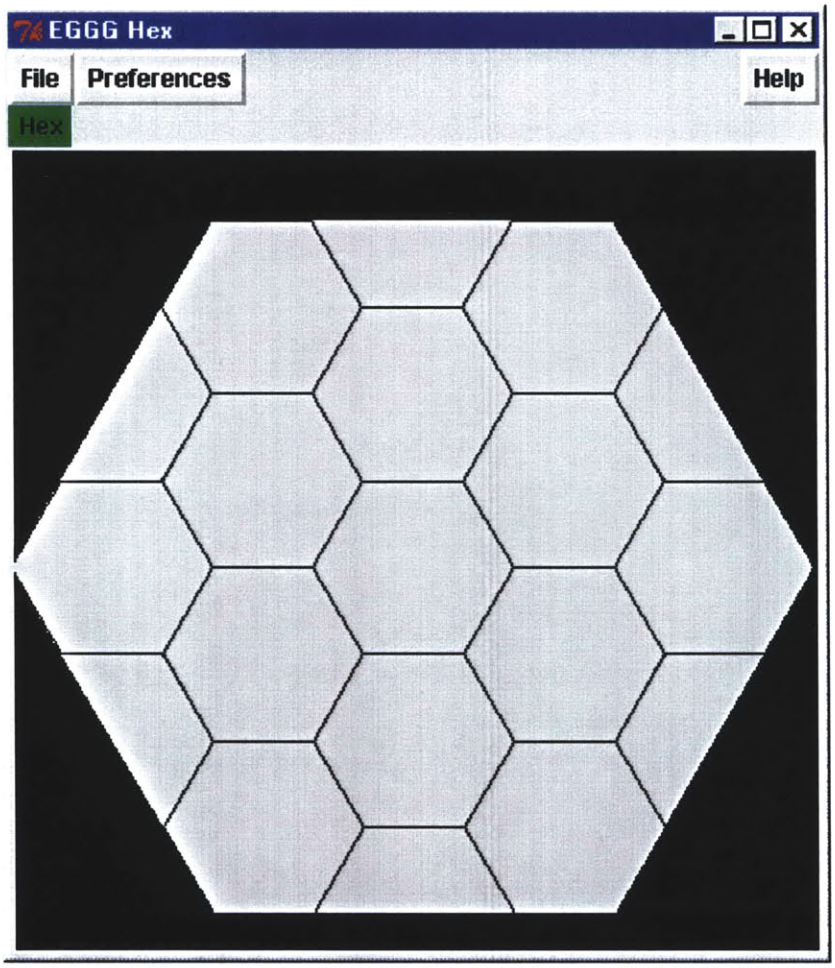
From the ontology developed in Chapter 2, every grid is a two-dimensional array of elements, and we call those elements `squares` no matter what their shape. Game designers can specify different designs with riddlesome statements like these:

```
squares are triangles
squares are hexagons
squares are circles
```

These can be combined with `board` and `grid` statements to generate an infinite variety of boards:

```
board is hexagon
squares are hexagons
5x5 grid
```

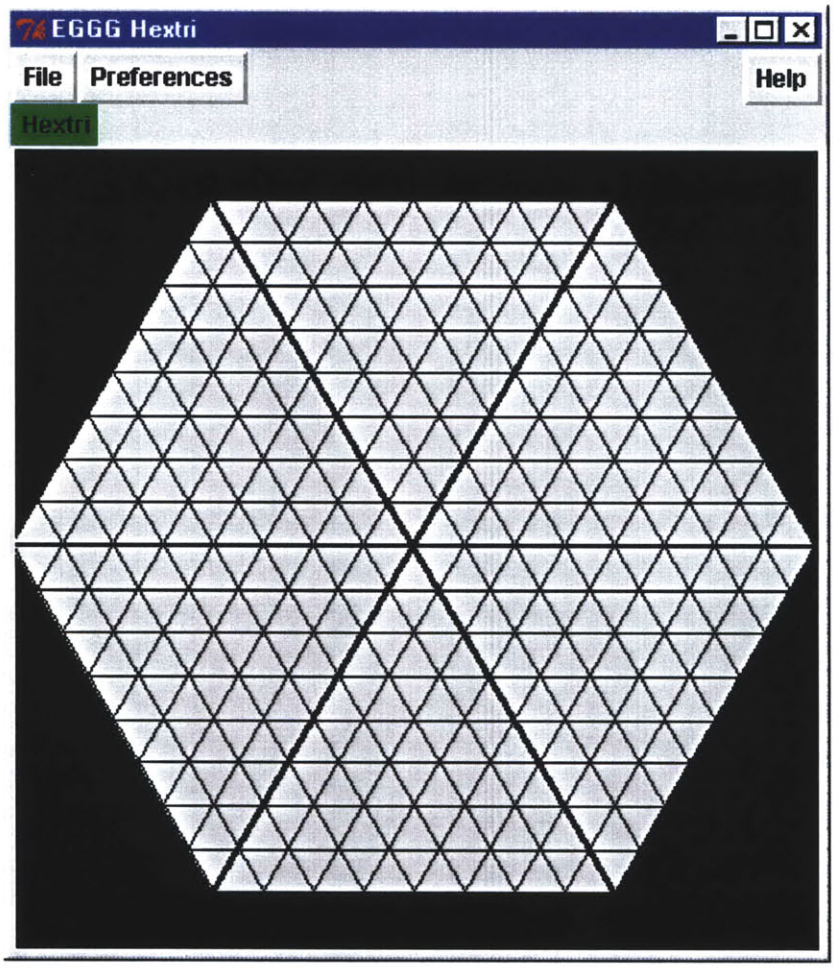
Here is the board for that game description.



Note that the "5x5" no longer refers to *x*- and *y*-axes; new axes are chosen parallel to the oblique sides of the polygon. This result is sometimes surprising, because a 5x5 grid won't have 25 squares when the board shape is a hexagon. As you can see from the above diagram, there are actually 13 squares (and six half-squares).

The squares needn't be the same shape as the board:

```
board is hexagon
squares are triangles
16x16 grid
```



Our algorithm for tessellating the grid is recursive; generating a 16x16 grid involves generating a 2x2 grid, and subdividing those grids into a 4x4, an 8x8, and finally the complete 16x16 grid. When there are more than 100 squares, the initial subdivision is drawn with thick lines, as shown above.

Note that these boards have clean lines and intersect at exact points only because we chose our board and square shapes carefully. No equilateral polygons other than triangles, squares, and hexagons can tessellate a plane; `board is pentagon` and `squares are triangles` generates a mess.

Checkerings

Checkerboards and chessboards alternate light and dark squares. This alternation makes it easier to convey the rules of the game: "checkers go on all the red squares" or "a bishop on a white square will never be able to move to a black square" or "a queen starts on the square of its own color." The colors don't affect the play of the game; the squares on a chessboard could just as well alternate between red, green, and blue, or be chosen at random, or be all black. The square color is a *superficial* aspect of the game appearance.

We seem the same phenomenon in tables of data: tables are often displayed with alternating rows or columns shaded. As with checkerboards, the alternation helps people view the grid more easily.

	Fren	Hist	Synch	Move	Board	Topo	Piece	Compart	Genre	Info	Ref	End
Tic tac toe			**sc**	**mm**	**bgs**	**t2**	**pc**		**gn**			**es**
Chess	ft	h	sc	mm	bgs	t2	pac		gw			es;n
Poker		**h**	**sc**	**mrmp**	**bc**	**t2**	**pac;nc**	**chf;bf**	**gn**	**iu**		**es**
Crosswords			sr	mm	bgs	t2	pl		gn			es
Tetris	**ff**		**sr**	**mtm**	**bgs**	**t2**	**pi**		**gn**			**ea**
Rock Paper Scissors	ft	h	st	mrm	bc	t1	pa	chf	gn	iu		es
Deducto		**h**	**sr**	**mrpms**	**bgs**	**t2**	**pc**		**gn**			**es**
Mammon	ft		sp	mmp	bi	t0		chi	gr	if;u	r	en
Chinese checkers			**sc**	**mm**	**bgt**	**t2**	**pc**		**gn**			**es**
Chutes and Ladders			sc	mm	bdg	t2	pc		gw			es
Diplomacy	**ft**		**st**	**mptm**	**bug**	**t2**	**pac**		**gr**	**ic;u**		**es**
Trivial Pursuit			sc	mm	bug	t2	psc		gn			es

EGGG alternates square colors when the following conditions hold:

- The grid is a square, and the squares are truly square. We don't want squares sharing an edge to share the same color, because then the boundary between the squares disappears. Only grids of squares and triangles can be colored with two colors; other shapes need more than two, and we decided that the coloring might confuse as much as it helps. We also elected not to color triangular grids.

- The grid is 5x5 or larger. Small grids are easy to see all at once, so they aren't colored.

- Pieces fit inside a square. In games like Tetris, pieces consist of multiple squares; the motion of a Tetris piece on an alternating field of colors means that the piece would change color as it moved.

  EGGG does not color a Go board, because in Go one plays on the intersections of squares and not inside the square. That is, the squares are

actually crosses so large that they touch.

The Pieces

Pieces can be things placed on top of the squares, or they might be attributes of the squares themselves. They can even *be* the squares. We now consider the eight games of EGGG mentioned in Chapter 1.

- Pieces placed on top of squares. Chess is the best example of placing pieces on top of squares. The piece images are placed by the window manager on top of the squares beneath, and both piece and square are rendered as buttons. In the example shown earlier, the images have no transparency layer; that's why their backgrounds obscure the colors of the squares beneath.

  Players need to be able to click either the pieces or the squares; that's why they're both buttons. Players need to be able to select the pieces to identify which piece they want to move, and they need to be able to select the square to move to. If the move is a capture, the square will already have a piece on it (with the exception of *en passant* capture), so EGGG has to distinguish between the phases of the move: the first phase is picking a piece up, and the second phase is putting it down again.

  Poker is also a game where the pieces are laid on top of the surface below, but since the surface is a canvas and therefore unresponsive to user actions, this is uninteresting.

- Pieces that are attributes of squares. In many games, the pieces can simply be treated as the text or color of the square. When the squares are truly square, they can then be implemented as Tk buttons.

  In deducto, the "piece" is simply the color of the square: black or white in regular deducto, and the hue/value/chroma or RGB triplet in the color variants. Clicking on the piece is therefore the same action as clicking on the square.

  In tic tac toe, the pieces are simply X and O symbols: simple text strings. When the pieces are strings, EGGG renders them as the text attributes of the buttons representing the squares.

  In Tetris, each piece is a set of several adjacent squares, colored differently from their background. Since color is all that is necessary to distinguish "piece" from "not-piece", the squares of a Tetris grid can be represented as simple attributes and not as the squares themselves.

- Pieces that are squares themselves. In EGGG's rendition of Rock Paper Scissors, the squares don't move, and no attributes of the squares ever change. The squares are the pieces themselves; the act of selecting the piece

is the move.

The identification of pieces that are squares themselves is tough to deduce from the game descripton alone. One might assume that in the crosswords that EGGG generates, the squares are simply white or black, and the pieces are the letters, placed on the squares by setting the text of the corresponding Tk button to the appropriate letter. If that were all there was to a crossword grid, that's what EGGG would do. The problem is that some of the white squares need a visible clue number in the upper left, and the text attribute of a Tk button can't have text in two places. The solution is inelegant: the `Eggg::Crossword` module contains links to an image directory with a pre-rendered image of every possible square. There's a black square, a white square, a square with a 1 and an A, a square with a 1 and a B, and so on throughout the alphabet and up to 200: 5202 images in all. When the user types a letter on a square, he may think that he's typing into a text field; in fact, he is deleting a button, reading in a new image from disk, and creating another button in its place. Luckily, the images are small, and players don't perceive the computation happening behind the scenes.

This odd treatment of pieces-as-squares highlights the discrepancy between a gamer's perspective of game classifications and a developer's perspective. Crosswords are an example where players might expect pieces and squares to be implemented as separate abstractions, but they are not. The "15-puzzle" is an example of the opposite.

In the 15-puzzle, a 4x4 grid of fifteen tiles is numbered one through fifteen. Initially, the tiles are in some random order, and the player's goal is to order them numerically. You can slide any tile in any direction as long as it satisfies the intuitive physical constraint: the space you slide it into must be empty. Players would assume that the fifteen puzzle is a perfect example of a game where the pieces are the squares, because they are in the tangible, real-life version. However, not in EGGG. Here is how EGGG renders the game:

The buttons never actually move; the act of clicking on one of the squares simply renumbers all of the squares *as if* they had moved.

- Games without pieces. Mammon (discussed in the next chapter) has no pieces at all.

Whenever a game has pieces, EGGG always places it in the center of the square if there are squares, and half the distance from the center of the canvas if there are no squares. If the piece overlaps the square or canvas boundaries, it is cropped to fit.

Color

> *The agitation for the Universal Colour Bill continued for three years; and up to the last moment of that period it seemed as though Anarchy were destined to triumph.*
>
> Edwin A. Abbott, *Flatland*, p. 38

EGGG divides games into those in which color has intrinsic meaning, and games for which color is ignored or is only a superficial attribute. In the games rendered by EGGG, only the color variants of Deducto treat color as an attribute with intrinsic meaning — that is, using color as a multidimensional value in the play of the game, rather than merely as something to identify players or pieces. In games where color matters, EGGG represents colors as Munsell values, and only converts to RGB triplets when Tk requires it.

EGGG also makes a few aesthetic choices for games in which color doesn't matter. For instance, the canvas for any sort of betting game is colored green. EGGG also

sets the white point of the game's window (affectionately called the "EGGG white") using Tk's `setPalette()` method to enhance legibility of the text or grid when necessary. At some point, EGGG will be modified to represent all colors as Munsell values, in part so that both regular and highlighted text can be guaranteed legible regardless of the palette. The color choices will be made according to the alignment and amplitude model of color juxtapositions described in [Jacobson 91].

Note: this has not yet been implemented in EGGG.

Get Your Game Face On

> *This type of illusion is generally known as the "Eliza effect", which could be defined as the susceptibility of people to read far more understanding than is warranted into strings of symbols — especially words — strung together by computers. A trivial example of this effect might be someone thinking that an automatic teller machine really was* grateful *for receiving a deposit slip, simply because it printed out "THANK YOU" on its little screen.*
>
> Douglas Hofstadter, in *Fluid Concepts and Creative Analogies*, p. 157.

What Hofstadter says is as true of faces as it is of words. An early version of EGGG emphasized conveying the state of the computer opponent to the player. As it searched the game tree, it would display a one-phrase opinion of how well it was doing in a window. Another window displayed the opinion in the form of a crude face:

The length and arc of the smile were proportional to how well the computer opponent thought it was doing. As EGGG played, the smile would subtly deepen as it did better and better, or frown more and more as its condition worsened. When the game began, the face would begin with a slight smirk to convey its confidence and intimidate the player.

The eyes drifted randomly. Some players assumed that the EGGG face was looking at the part of the board that it was considering. This made it, inadvertently, a cheap form of bluffing — by conveying illusory intentions, it misled players into believing that EGGG was following a different strategy that it actually was.

Note: this has not yet been reimplemented in the current version of EGGG.

Chapter 6: Connect the Bots (Networking)

The multitude is always in the wrong.

Wentworth Dillon, Earl of Roscommon

The Internet has rendered solitaire obsolete. If you're looking for a human opponent for a popular game, you can always find someone at a server that lets participants play at any hour of the day or night. Most computer games today are sold as standalone applications, but it's likely that not long from now, it will be the exception rather than the norm for computer games not to make *some* use of the Internet.

To a large extent, the networking needed for games is no different than the networking needed for other computer applications. But this is true only up to a point; the networking process is not completely independent of game play. For instance, multiplayer frenetic-fast games require a stateful networking protocol, and the faster the action, the less overhead the protocol can have. The once-popular game Netrek is a space-themed shoot-em-up with up to sixteen simultaneous players. The original version allowed only TCP/IP connections, which is fast enough for most any networked application, but not Netrek. TCP is a "reliable" protocol; when a client receives a packet, it sends an acknowledgment back to the server. Later versions of Netrek introduced a UDP option that allowed a player to jettison the acknowledgments. He'd occasionally lose a packet or two, and missiles would appear out of nowhere or stop dead in their tracks as packets were lost, but this tradeoff of accuracy for speed was deemed worthwhile by the vast majority of players.

Likewise, the synchronization of the game affects the protocol used to exchange information between players. When all the players must move at once, the game can only be as fast as the slowest connection. In some games, every player action must be broadcast to all the other players; in others, there can be a division between information kept locally, information sent to selected other members (a single player, or a team), and information that everyone has to know about. Some games can be peer-to-peer, and others require a centralized server.

EGGG is able to generate some networked games. In this chapter, we will discuss how this is made possible. First, we will talk about a lot of games with a little networking — the intermittent connectivity made available to *all* EGGG games, and then we'll discuss "serious" networking — multiplayer games that require networking at their core.

The Henhouse, And Stateless Connections

Game designers do not, as a rule, assume that their computer games have an Internet connection available to them. That's unfortunate, because constant connectivity enables new styles of game generation, play, and analysis. If that sounds sweeping, it's because we're not just referring to the potential of networked multiplayer games, which consumer markets have to some extent already realized. Instead, we are talking about features that apply to all games, and to the collaborative design process that can only be made possible with a central repository of game design knowledge.

EGGG maintains a single central repository on a computer at the MIT Media Laboratory. The repository is called the *Henhouse*, and it contains four datasets:

- Game descriptions (`.egg` files) created by game designers.

  This is an important resource for new game designers who want examples of the EGGG language, or for designers who want to create variations on already existing EGGG games.

  Ideally, this will become like the Comprehensive Perl Archive Network, or the Comprehensive TeX Archive Network, where people upload their creations (Perl modules, or TeX utilities) for everyone to use. Such a Comprehensive Game Archive Network would make it possible for users worldwide to share their games with one another.

- Invocations of the EGGG engine.

  Whenever someone runs the EGGG engine to create a game, a message is sent to the Henhouse containing the current time, the EGGG version, and who the game designer is (if the operating system identifies users). This includes bug reports: when EGGG generates a game with a syntax error, or an EGGG game generates an error (either during compilation or run-time), three things are sent to the Henhouse: the error message, the game description triggering the error, and the entire EGGG engine for good measure.

- Move histories for EGGG games.

  EGGG keeps a history of game moves throughout the course of the game. When the game terminates, the sequence of moves is sent to the Henhouse along with the usernames of each player if available. The data structure containing the game moves is of limited size (it holds 8192 moves by default; this prevents games with continuous flurries of key- and mouse-clicking from filling up the Henhouse).

- Computer names and port numbers for networked multiplayer games.

Both the games that EGGG generates and EGGG itself have a `-private` command-line switch. When EGGG or its games are invoked with this option, the

Henhouse notifications are suppressed.

Mail Generation

Network connections can be either *stateless* (also called *connectionless*) or *stateful* (*connection-oriented*). Electronic mail and web pages are stateless mechanisms; `telnet` and FTP are stateful mechanisms. Stateful mechanisms are necessary when the communication requires continuous data exchange. In this section, we'll discuss the only stateless connection that EGGG supports at the moment: electronic mail.

EGGG searches for a mail program — either common Unix mail utilities in expected locations (e.g. `/usr/bin/mail`, `/usr/lib/sendmail`) or the Perl `Mail::` modules that enable automatic mail generation regardless of the operating system. (If it can't find any of these utilities, it doesn't send mail.) All mail is sent to `eggg@media.mit.edu`. The four message types are illustrated below.

```
From: nobody@name.of.machine
To: eggg@media.mit.edu
Subject: Game description

Game description filename (e.g. "mammon.egg")
Game description (mammon.egg contents)


From: nobody@name.of.machine
To: eggg@media.mit.edu
Subject: EGGG engine

Operating system, Perl version, and available modules
The entire eggg program, including the version number



From: nobody@name.of.machine
To: eggg@media.mit.edu
Subject: Bug report

Operating system, Perl version, and available modules
The error message
The entire generated game


From: nobody@name.of.machine
To: eggg@media.mit.edu
Subject: Move history

Game description
Game outcome
Array of player names, rounds, phases, and moves
```

Mail sent to `eggg@media.mit.edu` is filtered through a program named

`egggfilter` that stores the data in an mSQL database and notifies me of any bug reports. People can also send mail requesting instructions for a particular game in the repository; EGGG runs the `egggdescribe` program and mails back the results.

EGGG-generated games read in their own source code before the game begins. This is how the games check themselves for syntax errors: during compilation, they read themselves and invoke a separate instantiation of the Perl interpreter to compile the program. If any errors are found, the program mails itself to the Henhouse before terminating. Likewise, if the program aborts while a game is running, a last-ditch handler traps the error and sends mail.

Global History and Adaptive Learning

In Chapter 2, we talked about the *history* of a game. This refers not to the common usage of history, which suggests the genesis and evolution of the game over years, but to the sequence of player actions during the course of a single session. This is what Condon and other game researchers mean when they say "history".

We call the history of a particular game session the *local history*. Distributed gaming lets us consider the set of every game session ever played by anyone. We call this the global history, and EGGG uses it 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. Pop the first game off `@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 loses more often than it wins.
8. If either move is statistically significant at the 0.05 level, store it in the Henhouse's module for the game after the `__DATA__` token.
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.

Games can update their strategies when they are invoked with the `-update` switch; that establishes a TCP/IP connection to a server that makes the Henhouse results available.

Thus, EGGG games can change every time they're played. Hopefully for the better.

Multiplayer Networked Games, And Stateful Connections

Email is fine for reporting game results and bug reports, but networked multiplayer gaming requires a continuous connection between players' computers.

There are many network topologies for stateful connections; by far, the two most popular are server/client and peer-to-peer. When EGGG games connect to the Henhouse, EGGG is a server, and the games are clients.

When a game designer includes a statement like this in a game description, EGGG creates a game that can be both a server and a client:

```
networked game
```

The game window will have two buttons at the bottom: `Connect` and `Serve`. `Connect` makes the game into a client, first connecting to the Henhouse to find out what servers are currently available. `Serve` makes the game into a server that accepts connections.

The current incarnation of EGGG can only send text back and forth, so typically those statements look like this:

```
networked multiplayer text game
```

(Game designers can stipulate their own protocol, but only by burrowing down into the underlying Perl code.) By default, EGGG creates a TCP/IP connection, including code that creates Berkeley sockets, invoking the `socket`, `connect`, `bind`, `accept`, and `listen` system calls as appropriate. The server is built on the `fork/exec` model: one process listens for new connections; when one is found, a new process is spawned to manage the new connection. Since Windows does not support `fork`, EGGG games on Windows cannot be servers.

A Sample Networked Game: Mammon

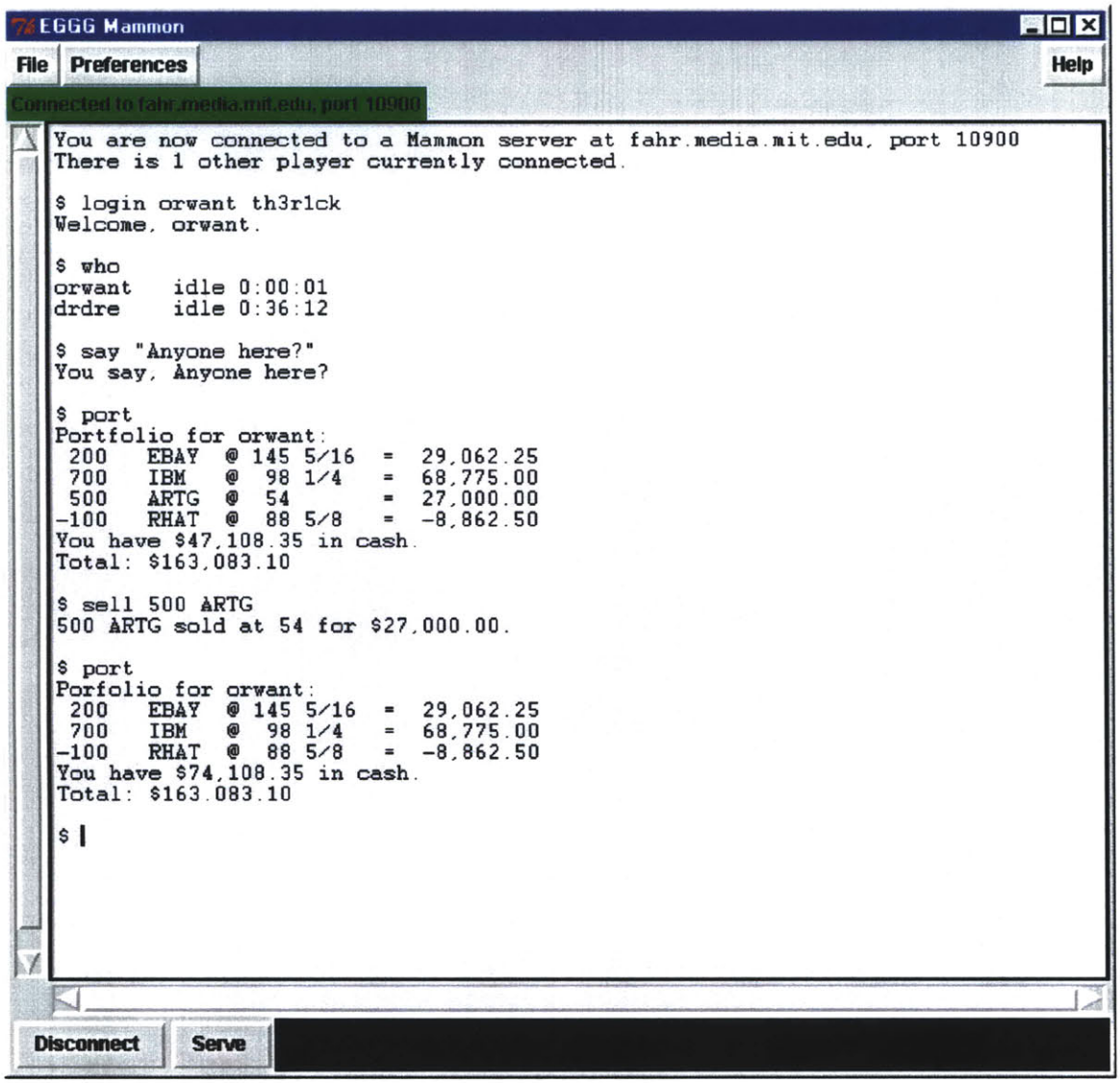
To illustrate a networked multiplayer text game, we'll examine Mammon, a stock picking game where players log in to a dedicated server, buy and sell stocks using play money, and chat with one another. Mammon (in its non-EGGG incarnation) was actually the first Internet stock-picking game, developed by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

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 the Henhouse) and that the server can accept up to twenty simultaneous connections. Why is that written as `0..20` and not `1..20`? Because of the zero, the server will stay up even when no players are present. These three lines alone generate a chat server where people can send messages to one another.

Here is a screenshot of Mammon:



(The `Connect` button changes to a `Disconnect` button once a connection to a server is made.)

Other statements of note from `mammon.egg`:

```
player has %Stocks and $Money
player start is %Stocks = () and $Money = 100000
```

In networked multiplayer games, each player has his own state; the `player has` statement indicates the game-dependent components of that state. (We call attributes like `$state{`*`player`*`}{name}` and `$state{`*`player`*`}{history}` game-independent components because all EGGG games have them.) The two statements shown here create a hash called `$state{`*`player`*`}{Stocks}` to contain the stock portfolio and a scalar called `$state{`*`player`*`}{Money}` to contain the players remaining cash.

```
every minute, { @prices = `download_stocks`;
                for (@prices) { /(.*)\s(.*)/; $prices{$1} = $2} }
```

Everything between the braces is Perl; what is of interest here is the clause before the braces — `every minute`. That instructs the game, when run as a server, to execute the Perl snippet every minute. (The snippet launches an external program to download stock information from a real-time quote feed and store the results into the `%prices` hash.)

```
moves are (Portfolio, Buy, Sell, Limitbuy, Limitsell,
           Stoplimitbuy, Stoplimitsell, Short, Fund, Say,
           Program, Quit, Rank, Who, Login, Register, Price,
           Password)
```

The `moves` are the commands available to each player; in the screenshot, we saw the player execute `login`, `who`, `say`, `portfolio`, and `sell` commands. Players can abbreviate the commands; the shortest unique prefix for each command is determined with Perl's `Text::Abbrev` module; that's why the `portfolio` command was invoked when the player typed `port`.

The rest of the game description contains the definitions of the different commands, which generate the subroutines to be executed (by the server; the client does virtually nothing) when the player types one of the commands. The command is sent to the server as soon as the player hits Return. Here is a sample command definition:

```
Limitbuy(STOCK, AMOUNT, LIMIT) means { Buy(STOCK, AMOUNT)
                                       when $prices{STOCK} <= LIMIT }
```

This illustrates the syntax of commands: the player can type `limitbuy IBM 300 112 3/16` to buy 300 shares of IBM at 112 3/16 or less. The `STOCK` is `IBM`, the `AMOUNT` is `300` shares, and the `LIMIT` price is `112 3/16`. As usual, everything inside the curly braces is Perl (the capitalized words are replaced with the function parameters), but here there is an exception: the `when` token triggers a persistent action in the server. "Persistent" means that the server will remember the condition, check for it periodically, and execute the action when the condition is fulfilled.

Interactive chat servers typically take thousands of lines of code, but most of that code is the same no matter what the server does. EGGG allows game designers to specify what makes their server different; all the rest is generated automatically.

EGGG is able to generate three types of multiplayer games: those with no spatial meaning, like Mammon; games where each player has his 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. However, EGGG is not tolerant of internally inconsistent game

descriptions. For instance, if you take the usual chess description and change `players are white and black` to `players are white and black and red`, EGGG will create a nonsensical game that expects a third player to move—even though he has no pieces to move with.

Chapter 7: Conclusion

Since that's the way we're playing it...let's play it that way...

Hamm, in S. Beckett's play *End Game*

In this dissertation, we have advanced the thesis that the similarities between games are sufficiently great that they can be abstracted into reusable software components. To support this thesis, we have developed a system called EGGG that translates concise game descriptions into fully functional computer games. EGGG exploits these similarities between games, making it possible for game designers to create games with a minimum of programming expertise. Designers have to specify only the rules that make their game different from more generic examples of the genre; EGGG's reusable components supply the rest of the game logic.

In Chapter 1, we talked about how the decoupling of software and hardware enabled video gaming to become popular, and we identified the next logical step: decoupling the soft software (the rules of the game) from the hard software (the implementation); this is what the EGGG system does.

The games that EGGG creates are not polished. They lack the artistry and speed of commercially viable software packages, but this is irrelevant to our thesis. We strive only to show that the similarities between game play are sufficient to create the games; we make no claims that the generated games rival what a dedicated programming team can accomplish.

Nevertheless, we have tried to make EGGG as complete as possible, and this includes adding features that are only tangentially related to game play. For instance, the automatic generation of documentation is not strictly necessary to advance our thesis; it simply enhances the EGGG by-product. Our attention to features like documentation generation underlie the dichotomy pervading this document: first, we have explored the similarities between games and established a taxonomy of game classification. Second, we have developed a real software project in the hope that other people will find it of use.

This dichotomy will pervade this final chapter, which is divided into two parts: What We've Learned, which examines the degree to which our thesis has been proven, and What To Do Next, which suggests future improvements to the EGGG system.

What We've Learned

In this section, we recap the similarities between games, enumerate some tips for game designers, and talk about how games can help turn work into play.

Abstraction Revisited

The similarities between games are best revealed by realizing *just how little* one has to abstract the components. We could have abstracted them far more than we did, extending Condon's probabilistic game automaton to handle more than two players, a model of hand-eye coordination for games requiring frenetic activity, and single-player games involving no competition. We could also have emulated Pell's approach, introducing mathematical operators and parameterized functions to express the play of the game. Or we could have introduced our own formalism and terminology for representing games.

All of those approaches would have been misguided, because they wouldn't have made it easier to create games. Nor would they have effectively communicated the mechanics of game creation to a wide audience. It's also likely that different schemas would be needed to represent game play, strategies, and graphic layout, and this would have precluded one of the contributions of this dissertation: establishing the linkages between these areas.

Similarities Revisited

We can divide the similarities of game play into similarities of structure, similarities of strategy, and similarities of appearance.

Similarities of Structure

- Player actions can be divided into turns, moves, phases, and steps. Some games (typically betting games, or games involving quick play) are repeated many times at a sitting.
- There are a small number of main loops for game programs, and these depend on the synchronization of player moves and the time requirements of the program.
- Games can be divided into Markov games, games of partial history, and games of total history; this determines how much state the game program must keep.
- Any game with spatial meaning has a playing surface, and a few types of playing surfaces suffice to represent the vast majority of games. Games without spatial meaning typically involve the exchange of text.
- Some pieces have state, but most have only a position. Some pieces can be ranked relative to one another; some pieces are part of an ordered set (like

letters) but have no implicit ranking.

- Many game boards are grids — coordinate systems of elements (squares) that typically hold exactly one piece. In a few games, the squares can hold many pieces; in a few games, a piece consists of many squares.

- There are two ways that piece information is concealed from players: from a "hand", where a player physically conceals his pieces from opponents; and from a "bag", where unseen pieces are concealed from everyone.

- Themes don't matter to game programmers; they usually find expression only in the superficial aspects of the game: the artistic aspects of the pieces or board, for example.

- When players need to communicate information to one another, the types of messages fit cleanly into three categories: friendly information (between teammates), unfriendly information (bluffing), and information merely required by the play of the game.

- When the game play is asymmetrical, with one player having a different role from other players, he is likely to be a referee, and can be given unlimited access to the game state.

- Every player in a game has some goal, which is better called an ending since its primary purpose is to determine when the game is over and not who has won. The ending can be binary (win/lose), trinary (win/lose/draw), or comparative (amass the most/least of something).

- Both pieces and sets of pieces can often be sorted and ranked; the sortings and rankings needn't be transitive.

- When the initial state of the game requires randomization, the randomization might involve pieces in the hand, pieces on the board, or pieces in the bag. Sometimes the initial state of the board needs to be concealed from the players.

- Many game situations can be represented as assertions that trigger when particular conditions exist. Buttons are well-suited to assertions; endgames are not if the computer program needs to generate strategies.

- Game play can be divided into variants and invariants, and placed into separate data structures (`%state` and `%game`, in EGGG).

- Game knowledge can be divided into intrinsic and extrinsic characteristics, and placed into separate namespaces (the game description and the game module, in EGGG).

- Piece names sometimes reflect their owner, owner names sometimes reflect their location. All names need to be abbreviated uniquely.

Similarities of Strategy

- A generalized minimax procedure can be applied to many different types of games, not just games of perfect knowledge. If the static evaluator is chosen appropriately, even games of psychology can employ minimax.
- A generalized static evaluator can be applied to many different types of games — even games where it is difficult to estimate the distance between the game state and a player's goal.
- A library of opening moves can be amassed from the aggregate play of a population of players.
- It is easier to generate opponents for asynchronous games than synchronous games — not because the simultaneity of moves poses a problem, but because asynchronous games are easier to analyze.
- When the rank of a piece can't be easily extracted from the rules of a game, it can be estimated by enumerating the number of situations in which the piece is less powerful than others.
- The motility of a piece is not as easy to estimate as it might seem, since the available moves are often determined by interactions with other pieces in situations of unknown frequency.
- The rarity of a piece type can sometimes be inferred from the board configuration, and sometimes from the probability distribution underlying piece generation. When the rarity is known but the piece values aren't, or the piece values are known but the rarities aren't, the unknown quantity can be estimated with Zipf's law.
- When all possible outcomes of the game can be enumerated, the expected value of a piece can be calculated, and then we don't have to use our ad hoc estimate of piece power.
- Most games have discrete moves being made at discrete times, and for these games Hidden Markov Models can be used to identify patterns in a player's behavior.
- Behaving randomly (usually with a weighted distribution) is often a good strategy, even for bluffing. Unfortunately, people aren't very good at behaving randomly; it's often possible to determine when they are trying to behave randomly and even predict what they will do next.
- Even in Markov games (games where the state doesn't rely on the history), histories sometimes provide a clue to how players will move. This is true both for local histories (the history during a particular game session) and for

global histories (the history of all games played).

- One generic strategy is to maximize the ratio of the moves you have available to the moves that your opponents have available.
- When player abilities need to be estimated, either the beta distribution or the normal distribution can be used, depending on whether the game ending is binary/trinary or comparative.
- When there are multiple attractive moves, a weighted distribution can be used to choose one at random to keep play interesting. However, the likelihood of choosing a move must be supralinearly proportional to the attractiveness; otherwise, the premium placed on interesting moves is likely to result in a loss. When no premium is placed on interesting moves, the result is an optimal bluffing strategy for games between rational players.
- People are eager to believe that computers have insights into the game that they might not actually possess, and this can be exploited to the computer's advantage.

Similarities of Appearance

- Non-frenetic games can always be paused, frenetic-fast games can sometimes be paused if you blank out the screen, and frenetic-timed games should never be paused.
- Games need to know not just what information to show a player, but how long the information should stay visible. Put another way, displayed messages should have expiration dates.
- A few attributes of the game state can be made directly manipulable by the player in a Preferences menu; most cannot.
- Players are more likely to perceive their opponent as intelligent when it is depicted with a humanoid face.
- There is no substitute for a geometry manager that provides pixel-by-pixel control over the placement of windows; simple relationships like "this widget should be attached to the top of that widget" aren't sufficient for many games.
- Boards should be oriented so that the flattest side is facing down; the flatter the side, the more likely that it belongs directly in front of a player.
- Large grids need something to help players divide the board into manageable chunks. Checkerings and thickenings help accomplish this.
- Pieces can be on top of squares, attributes, or they can be the squares themselves. However, the appropriate choice is determined more by the type

of display than by the rules of the game.

- The Munsell color space is better than RGB for representing colors in games.

Complexity

Perhaps surprisingly, we have avoided talk of complexity in this dissertation. Complexity can mean a variety of different phenomena. To Condon and Pell, it means computational complexity; to Beasley, it means the difficulty of determining who can win; to a developer, it might mean how hard the game is to implement; to a player, it might mean how hard the game is to learn or play. We now turn briefly to two crude observations about complexity from the player's perspective.

If we define complexity as the number of moves that a player has to consider at a time, we can categorize games by the shape of the complexity over time. Chess complexity peaks in the middle, because beginnings and endings permit fewer moves than the midgame. Nine-men's morris peaks in the middle as well, at the moment you switch from placing pieces to removing them. Scrabble complexity more or less monotonically increases, since the number of possible plays increases as number of words on the board grows. The complexity of crosswords, Black Box, and Stratego monotonically decreases. Go ramps up fast (the board starts simply, and standard openings allow players to move quickly), stays high for a long time, and ramps down fast at the very end.

If we instead define complexity as the amount of detail that a player must consider during the typical move, we can categorize games into those that blind people can easily learn and those that they can't. Poker, tic tac toe, and even chess are in the first category. Monopoly and Diplomacy are in the second category, because they have arbitrary (some would say inelegant) boards that possess a lot of detail. A blind person has to memorize a lot of bits to play Monopoly or Diplomacy. Games in this category are hard to generate with EGGG because there's no concise way to describe them. What's interesting about this classification is that it gives you a hint about when the game was made; most games in the second category were developed after the industrial revolution, since they require custom boards that only modern manufacturing processes can produce inexpensively.

Tips for Game Designers

EGGG helps implement games; it doesn't help design them. Nevertheless, in the course of designing many EGGG games we've made a few observations that might help designers create better games.

Computer Opponents Should Act Smarter Than They Are

In *Conversation With and Through Computers*, Brennan notes that system designers are frequently overzealous in adding human touches to their programs, with the unfortunate result that users ascribe more intelligence to the program than it actually has.

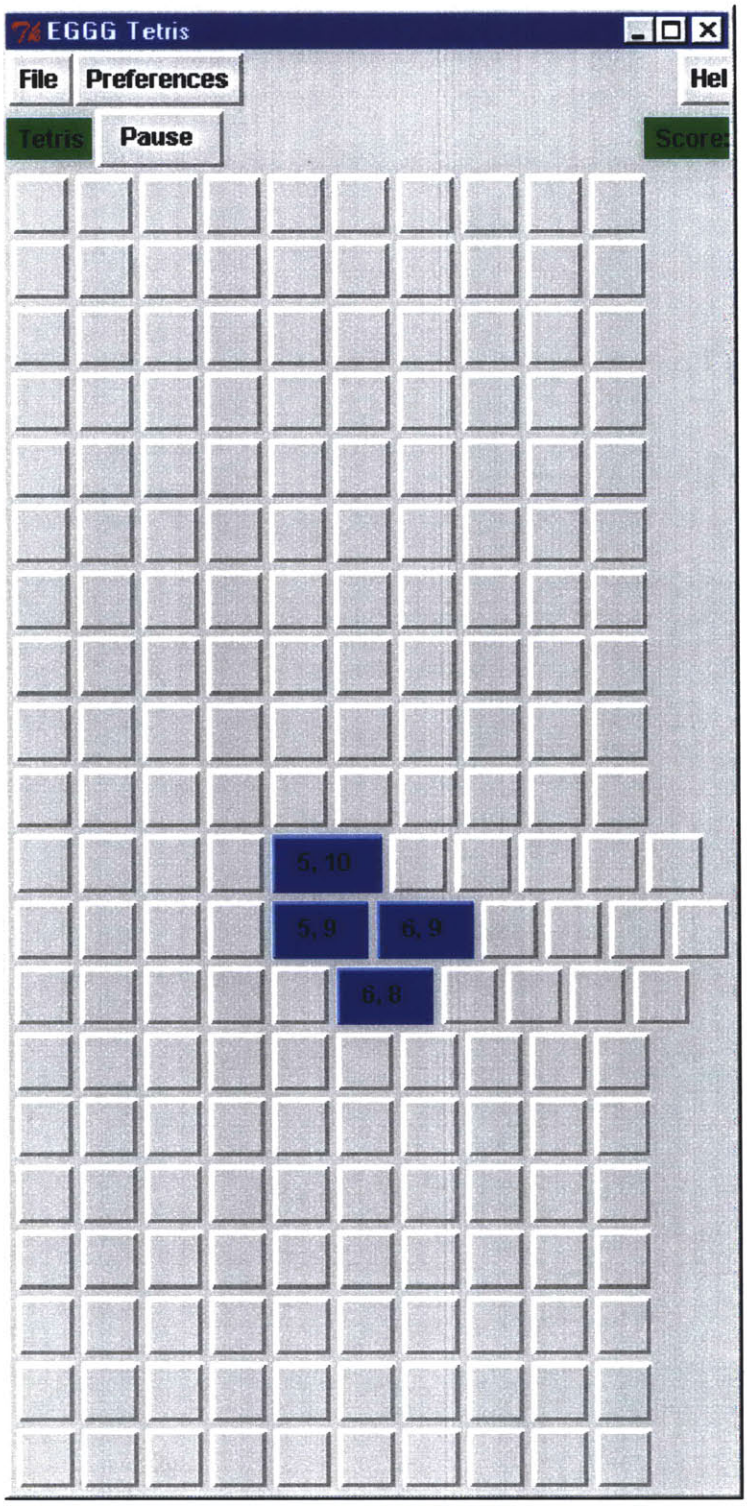
But one man's bug is another's feature: This is an asset to the creation of intelligent-seeming game opponents. In poker, a little bit of posing helps the opponent. Computers don't have much of a chance at seeming stupider than they are, but they're great at pretension.

Make Mistakes

Some of my more interesting variations resulted from bugs as I was designing EGGG. For instance, my first chess game allowed a player to capture his own pieces. I'm a poor chess player, but this seems to me like a perfectly reasonable variation.

One of my first Tetris games never turned squares back to their original colors once occupied; the result was blue piece-tendrils racing down the screen to the bottom. To avoid immediate death, you'd have to rotate each piece so that the narrowest side was facing down, and then hope that the next piece would start from a different horizontal location. These games wouldn't last long, but it was surprisingly fun trying to stay alive for as many pieces as possible.

To fix the problem, I switched EGGG into a debugging mode that displayed the coordinates of each square comprising the Tetris piece. As the Tetris piece moved down the screen, the squares of the piece would display their coordinates — which wouldn't fit entirely inside the square, so the square would grow. After the piece passed by, the square would shrink again. The result was that the piece looked like it was being squeezed through a spongy medium. The game became a little more difficult; since you couldn't align the piece with the bottom of the screen, it was harder to project the piece onto its eventual resting place. A screenshot is shown below.



In one of my poker games, I experimented with a (later abandoned) feature that had EGGG try to figure out what pieces should look like, by searching through the local filesystem and then the web to find appropriate pictures. It found images of cards, but it found images of chess pieces first. The result: aces through jacks looked as you would expect, but the queens and kings were chess pieces instead of cards. This

obviously makes for a lousy poker game, but it does raise the interesting question of what card game variants might result if certain cards were physically different from others, so that opponents had some information about the contents of your hand.

While none of these variants are likely to be more popular than the original versions, they do suggest ideas for other games. When your language is at a high-enough level, some bugs look like brainstorms.

Include Easter EGGGs

Some computer games (typically one-player games) retain their popularity because players are always finding new ways to play, or new avenues to explore. The Easter eggs found in some arcade games are examples: rumors float around that if you perform this action *just so* on level X, the hero grows a tail, and all of a sudden everyone is pumping quarters into the machine to see it happen.

If Microsoft can include a flight simulator in Excel (open a new workbook, press F5, type `X97:L97`, click `Ok`, press tab, hold down Ctrl-Shift and click on the Chart Wizard icon on the toolbar), EGGG chess programs can include Easter eggs. They are most easily implemented with assertions, since assertions provide a way to let game designers specify particular actions to occur when arbitrary conditions are met. Here's a sample:

```
assert after move: { if ($state{board}[8][8] = "white Pawn") {
  use LWP;
  $_=get("http://www.intellicast.com/weather/BOS/content.shtml";);
  s/^.*<\/?BLOCKQUOTE>//gs;s/<[^>]+>//g;s/\n{3,}//g;
  display(wrap(",",$_)); } }
```

If white promotes a pawn to the upper right hand corner of the board, EGGG displays the current weather in Boston.

The only problem is that the `egggdescribe` utility will faithfully document the Easter EGGG. It's not much of a secret when players can discover the egg just by reading the documentation.

Measuring Playability

It's a truism that games evolve to maximize playability: if gamers don't enjoy playing a game, they won't play it, and that lack of popularity will prevent the game from spreading. Games compete for the increasingly scarce resource of leisure time; people can only spend so many hours per month playing games, and so the more playable games extinguish others in the marketplace (or download areas). But some games are tenacious: in the last five hundred years, chess has seen only two rule changes: the rule that allows pawns to move two squares from their initial position,

and the *en passant* rule to counterbalance it.

To what extent could EGGG be used to evaluate the playability of games or variations? This is a difficult question. Not only is any measure of playability subjective, but playability is likely to be confused with enjoyability in the minds of many users. The quality of the graphics in many of today's arcade games disguises insipid play: if the same game were rendered with crude geometric shapes, it would be an utter failure. When an ugly game is nonetheless popular, that's when you know it's playable.

Can EGGG measure the playability of a game just by looking at the game description? It would be nice if EGGG could provide some assistance to would-be game designers trying to create a game from scratch.

One simple metric is the size of the `.egg` file. The larger the file, the more information is necessary to represent the game. The `.egg` files for tic tac toe, chess, Monopoly, and card trading games might well span four orders of magnitude. The notion that the size of the representation is proportional to complexity isn't new: the Kolmogorov complexity of a program is the minimum length of the Turing machine needed to generate its behavior.

Another criterion would be to assume that current popular games are paragons of playability, and the more a variation deviates from the known game, the less playable it is. This is a bit reactionary, assuming as it does that the natural evolution of games cannot be improved upon.

Both of these metrics assume that all rules contribute or detract equally from playability, but obviously not all rules are the same. For instance, would chess be a less playable game if a stalemated player were to lose instead of draw? Perhaps, but only slightly, since the situation occurs so infrequently. So frequently-invoked rules should have larger absolute weights. How often will a particular rule be invoked? We can't know, because there's no way to know beforehand how a game will be played. (It even depends on the quality of the documentation.) When Parker Brothers rejected Monopoly in the 1930's, one of the reasons was that the game took more than 45 minutes to play — that was one of their criteria for playability. The current version of EGGG has no way of knowing how long games take to play. At the end of the day, the only feasible measure of playability is empirical.

Empirical Evidence Of Playability

EGGG gathers empirical evidence of playability: since the games generated by EGGG send messages to the Henhouse identifying which game was played, we can tell how popular each game is. This allows compilation of statistics about which games are being played, and we believe this the best evidence of playability one can hope for. The statistics about the games are automatically compiled and made available on the web.

This feature is keyed to the current year. All of the invocations sent in the years 1999 and 2000 will be included, half the possible messages sent in 2001 will be included, one quarter of the messages in 2002, and so on. Thus, if EGGG becomes popular, this will give the Henhouse a random sampling of the computer gaming universe — at least, that portion of the universe spanned by EGGG.

We borrow our empirical measure of playability from the slogan of Othello: "a minute to learn, a lifetime to master". The measure is simply computer opponent's win-loss ratio divided by the compressed bytecount of the EGGG description. The briefer the description, the easier the game is to learn ("a minute to learn"); the worse the computer generated opponent does, the deeper the strategies get ("a lifetime to master.")

Of course, many people find Othello boring. And the simple-minded EGGG opponents have simple-minded adaptive learning; they improve (slightly) over time, so the playability of a game will decrease over time as the computer's proficiency increases. And EGGG can't generate computer opponents for every game, only for games that can be represented as game trees.

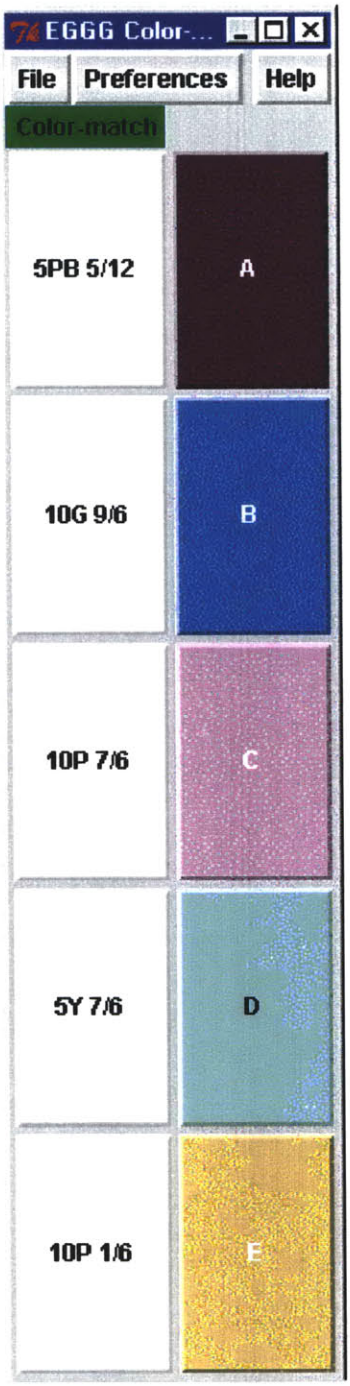
Why There Can Be No Good Measure Of Playability

Without a deeper understanding of games, EGGG has only the game description and player actions available to it. Player actions are a mediocre measure of popularity, and the game descriptions won't always have the information necessary to conclude anything about the game's playability.

Consider this game description:

```
goal is solve
one player
Colors are { open(FILE, "colors.txt"); <FILE> }
shuffle Colors
Randnames are  { ( map (m/BGCOLOR="(.*?)"/, Colors) ) }
Randcolors are { ( map (m!<FONT.*?>(.*?)</FONT>!,  Colors) ) }
board starts [shuffle Randnames, Randcolors]
2x5 grid
squares display numbers
solution displays numbers
solution is { Randcolors => Randnames }
```

This yields a color matching game that teaches players how to match up Munsell names with colors:



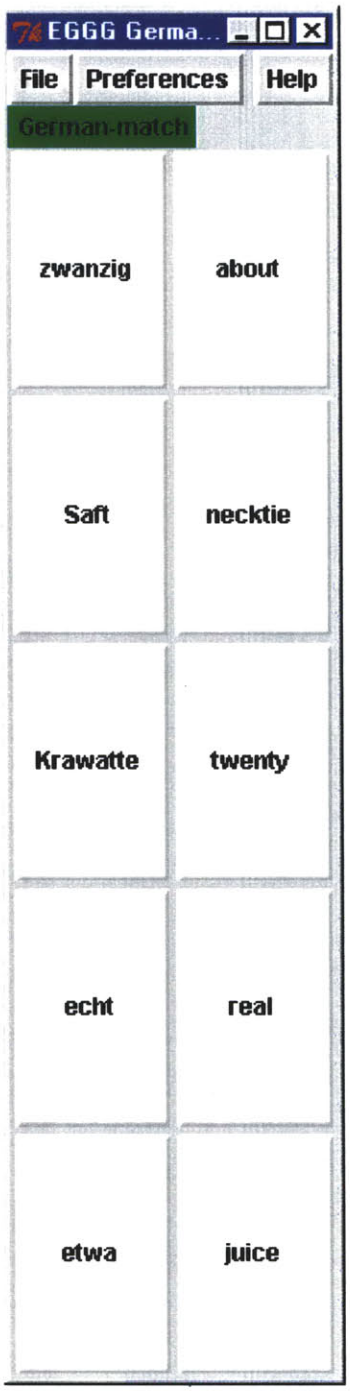
Players type letters in the left-hand column, and the game rearranges the squares in that column. When each name is matched with the appropriate color on the right, the game is won. For instance, the "10G 9/6" color is the cyan color labeled "D", so the player should type "D" on the square labeled "10G 9/6", which makes the second and fourth squares of the left column exchange their labels.

Now let's consider another game:

```
goal is solve
```

```
one player
Dictionary is { open(FILE, "german-words.txt"); <FILE> }
shuffle Dictionary
Randenglish is { ( map (/^(.*?)\t/, Dictionary) ) }
Randgerman  is { ( map (/\t(.*?)$/, Dictionary) ) }
board starts [shuffle Randenglish, Randgerman]
2x5 grid
squares display numbers
solution displays numbers
solution is { Randgerman => Randenglish }
```

This game teaches German vocabulary. Note that the game description is nearly identical to the game description for the color matcher.

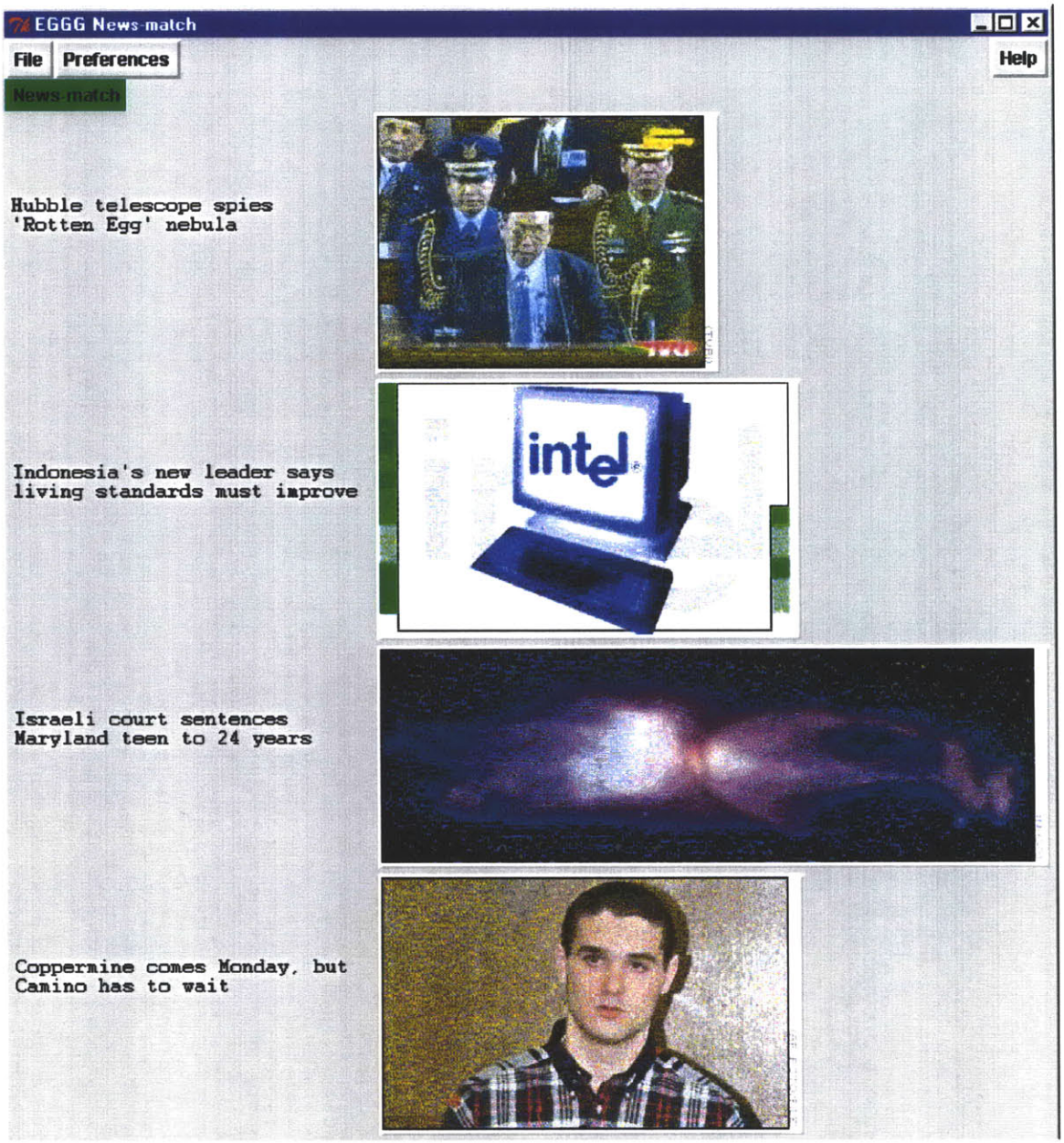


Finally, here is another nearly identical game description.

```
goal is solve
one player
News is { local $/ = ""; open(FILE, "news.txt"); <FILE> }
shuffle News
Randpicture  is { ( map (/^picture\s+(.*?)$/m,  News) ) }
Randheadline is { ( map (/^headline\s+(.*?)$/m, News) ) }
board starts [shuffle Randpicture, Randheadline]
```

```
2x4 grid
squares display numbers
solution displays numbers
solution is { Randheadline => Randpicture }
```

This game has players match headlines of news articles to accompanying photos:



A program was used to gather live data from the Internet (http://www.cnn.com, in particular) and break out the picture, headline, and text of the article into a `news.txt` file. (The text of the article is ignored in this game, but one can envision games in which the text is used for other purposes.)

Each of these three games has essentially identical game descriptions. Some of the

statement names are different, and each uses different regular expressions to extract the two columns of data from the appropriate `.txt` file, but the play of the game is exactly the same in each. Yet while these games might be deemed playable, games with the same play architecture might not be. A game description that matches up, say, the first letter of a web page to the size of the web page would look just like the three games just shown, but wouldn't send Hasbro running for their checkbooks. The game description will never be enough to determine playability.

Making Work Into Play

> *Each player has a string with a total of 80 beads on it. The red, green, blue, and yellow beads are arranged in an irregular sequence. The colors of the beads, as the name "RNA Game" suggests, correspond to the four building blocks of ribonucleic acid...two kinds of linkages — the one joining red beads to green, the other joining blue with yellow — represent the hydrogen bonds that hold the complementary bases together.*
>
> M. Eigen and R. Winkler, *Laws of the Game*, p. 285

Deducto does a pretty good job at fooling people into learning, or more precisely, at making the task of finding signal in noise as enjoyable as possible. It's not hard to imagine a Deducto-like game put to more professional use for domains in which analysis involves pattern recognition. This is the stuff of science fiction, but it raises the larger question of whether EGGG can be used for more than just games — to what extent is it a generalized programming assistant? Unfortunately, the answer is that EGGG makes a lousy programming assistant. The game descriptions are as concise as they are only because the EGGG engine already has a pretty good idea of what it's creating.

Nevertheless, in our opinion there are many ways in which automated game generation, combined with a transformation of the rules of some phenomenon into a game context, could be used to make work into play. Below, we list a sampling of ideas.

- Making language learning into a game.
    - Matchers that embody vocabulary drills like the German word game shown above.
    - Frenetic-fast games that reward players for quick comprehension: translating a sentence, filling in a blank, or following directions in the foreign language before time runs out.
    - Puzzles that involve shapes corresponding to grammatical roles; adjectives can be chained together because the end of one fits into another; verbs cannot.

- Making programming into a game.
    - Simulation games in which players design Turing machines to solve simple problems.
    - Detective games that entice players into debugging programs with an arsenal of tools: magnifying glasses that examine data structures, tripwires that create breakpoints, flashlights that create print statements, probes that provide a multitude of inputs to the program.
    - Black box-like games in which players try to deduce how a program works given input and output.
    - Simulation games in which players assemble programs with visual representations, by combining prebuilt program structures (loops, subroutines, reducers, input and output idioms) in the right order to solve some task.
- Making physics into a game.
    - Arcade games in which players have to quickly solve simple problems involving gravity, friction, rotation, or optics.
    - Multiplayer games in which players create theories to explain fictitious phenomena; the theories are scored according to Occam's razor.
    - Spacewar-like arcade games in which magnetic ships must navigate around (and into) charged geometric objects, as a way of learning about Maxwell's equations.
- Making mathematics into a game.
    - Sequence puzzles in which players have to predict the next number in a sequence.
    - Games in which multiple graphical elements move about following simple constraints, and the player has to predict where they will end up, solving a differential equation.
    - Games in which players combine visual representations of axioms to prove geometric theorems.
- Making news into a game.
    - A race game in which you're the investigative reporter trying to amass stories (gathered from the Internet) for the next edition before time runs out.
    - A betting game in which players predict the news based on real

events, with parimutuel odds based on how many people bet on which side. The Iowa Electronic Markets (http://www.biz.uiowa.edu) and Vadim Gerasimov's News Totalizator (http://vadim.www.media.mit.edu) are two examples.

- A matcher game like the one shown earlier, but with the text of the articles present. Alternately, the texts might all cover the same story, and the player's goal is to match the article to the news source, as a way to illuminate the biases or writing styles of different news sources.

- A game similar to the Qix arcade game in which players must lay out advertisements and articles on the page, balancing revenue with printing costs and subscriber disaffection.

- A game like Balderdash, in which players have to identify a fictitious news story from a collection of real news stories.

Lessons Learned

As stated in the introduction, EGGG is not about children, education, gender, or culture. In this section, we discuss what EGGG *is* good for — not in the obvious sense that it helps people create games, but in the broader context of what systems like EGGG portend for the game-designing and software communities.

- EGGG makes game design easier, and therefore better.

Although EGGG offers no feedback on the games it generates, it indirectly improves game quality 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 problems with their design. If problems are found, they can be fixed, and a new game generated immediately. Game design thus becomes an iterative process of trial and error, an inherently easier way to create and refine programs than to rely on perfect planning and meticulous execution.

EGGG also provides something that hasn't existed before: a centralized repository for games and game designs. The repository shows people the rules governing successful games, making it easier for people to examine what has made other games successful and to modify them, creating variations.

An interesting question is whether EGGG might someday be able to critique the design effort, or even suggest what games to create, instead of merely automating their creation. Earlier in this chapter, we offered evidence that directly estimating the playability of the game from the rules alone isn't possible without a deep understanding of the player's experience. But a data-driven approach is possible: what EGGG might someday be able to do is to calculate the difference between a

newly-created game and already existing games. One way to accomplish this would be to chart the universe of games, treating it as a multidimensional space in which each game is a single point defined by some combination of how it is categorized and the rules that ultimately define the game. The distance between chess and its variations would be small; the distance between chess and poker larger, and the distance between chess and Doom larger still. How many dimensions would the space have, and more importantly, which dimensions are the most important? Would the dimensions correspond to the taxonomy in Chapter 2? For instance, one dimension could be the number of players, and another could be some measure of the complexity of the playing surface.

If such a space were defined, it could be used to tell designers what other games their game resembles. The next step would be to use it to create new games without human intervention. Underrepresented regions of the space could be mined for new game ideas. One could interpolate between games, creating a game halfway between two others; or extrapolate them, making a game less like another game, or less like *all* other games.

- EGGG is an unusual software engineering approach.

Automated programming efforts always embody a compromise between the scope of the automated programs and the degree of automation. The Programmer's Apprentice had a broad scope — it helped programmers create any sort of program, but it was a suite of tools that helped expert programmers program better. In contrast, the 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. Similarly, non-game automated programming systems are also tailored to very narrow domains.

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, modifiability, and so on. EGGG can now be taken in two directions; it can grow upward, with more emphasis placed on formalizing its abstractions and exploring their applicability to domains other than games. Or, it can grow downward, with more attention paid to making the system used by thousands of would-be game programmers. It is a source of satisfaction to us that both options are available, and for all its flaws, we consider EGGG a promising model for automated software design.

The most salient flaw in EGGG is the disconnect between the taxonomy of games described in Chapter 2 and the actual EGGG implementation of the taxonomy. To put it bluntly, the mapping is messy, full of special cases not for particular games (that would be cheating) but for genres of games. The end result works well for

easy-to-pigeonhole games and their variations, but at heart EGGG's categorization of games is too coarse. Many game variations that fall through the cracks, exposing EGGG's inflexibilities. Consider Siamese chess, a game for four players on two chessboards, side by side. When one player captures a piece, he can give it to his partner, who is playing on the other chessboard. This can be represented in EGGG, but only with difficulty. EGGG makes it easy to create a game with any single playing surface; creating two playing surfaces is substantially more difficult, requiring that the designer burrow down into the underlying Perl. The designer could take another approach: define a single 8x16 board, but with an invisible barrier down the middle. Then the board is simple, but the piece movement is now more complex. For instance, a bishop can no longer simply move diagonally up to the board edges; instead, a bishop on the left half-board can move diagonally to the upper, left, and bottom edges, but must obey the invisible barrier to prevent it from crossing over onto the right half-board.

So we have two quite different ways to describe the same game. In one, the board is complex and the piece movement simple; in the other, the board is simple and piece movement complex. This tells us that EGGG's taxonomy can't easily be translated into dimensions in our game space: if our taxonomy were mapped directly onto axes, these two descriptions of Siamese chess would each become different points in the space. However a game space is structured, it must be consistent; games should map uniquely to points in the space regardless of how they are described to EGGG.

- People learn when they design games.

People learn about games when they design EGGG, even though EGGG itself offers no tips about how to make games better. Part of that education comes from users realizing what they have to describe and what they don't — it makes users consider the core of their game play, but little more. In the course of designing games with EGGG, we found that an important criterion is what we'll call *balance*. A game is balanced when players have multiple options available to them for most of the game. Games in which key moves force players into rigidly-defined courses of action (whether as part of a winning strategy or a losing one) tend to be less fun for the player whose decisions have been constrained. EGGG can't determine a game's balance from the game description, but the fact that the games are networked could make it possible for EGGG to analyze this aspect of playability from the history of moves (in game-tree games, where the minimax procedure can enumerate the available moves at each point during the game), or even the time taken for each move.

There is also room for the a deeper analysis of the game description. For instance, EGGG could confirm that the rules governing different players are symmetric. If a chess variant included a rule for white without a corresponding rule for black, that could be identified and fixed during parsing in the same way that EGGG currently queries the designer when a nonexistent definition is referenced in the game description.

To a lesser extent, people learn about programming when they use EGGG. They see

their descriptions translated into a large program complete with documentation and graphics, and they can see how tiny changes in the game description can dramatically affect the structure of the generated program. By seeing what code EGGG generates, and in particular how the code changes from game to game, designers can acquire knowledge about the software components that EGGG uses.

To a still lesser extent, EGGG users can learn a little bit about how complexity arises from simple rules. If the time spent playing them is any indication, games are compelling phenomena, and an understanding of how entertainment emerges from the interaction of a dozen or two rules would be powerful indeed. Perhaps future versions of EGGG will be better able to convey this understanding of not just *how* a game works, but *why*.

What To Do Next

In this final section, we will discuss potential improvements to the EGGG system.

Text Adventures

One genre of games lacking from EGGG's repertoire is the text adventure. There are similarities among text adventure games: most of them have an emphasis on puzzles; most involve a first-person journey through a virtual space of around a hundred locations; most involve the player gathering objects and using them to gain entry to particular locations. EGGG cannot supply the creativity that all good text adventures require, but it should be able to provide a framework to let designers concentrate on the rooms, roles, objects, and puzzles, instead of the programming.

Evidence for the similarities between text adventures is provided by the existence of the Z-machine. Years before Java popularized virtual machines, Infocom (makers of most of the popular text adventures of the 1980's, including their flagship game Zork) invented the Z-machine, a virtual computer that allowed Infocom adventures to run on all popular operating systems of the mid-eighties, from the PDP-10 to the IBM PC. The Z-machine consisted of a series of opcodes and an interpreter to run them; the effect was to allow game designers to separate the platform-independent story file (e.g. `zork1.dat`) from the platform-independent interpreter (`zork1.com`).

Michael Edmonson created `rezrov`, an Open Source Z-code intepreter written in Perl [Edmonson 99], and it may be integrated into a future version of EGGG.

Speed

The early versions of EGGG were quite fast, because they didn't do much. Later

versions got slower and slower as the EGGG language grew in complexity, with EGGG ultimately taking over twenty minutes to render `chess.egg` into a chess game. That prompted me to dispense with EGGG's parser, replacing my recursive descent parser with a series of short, fast passes over the game description. This will curdle a language designer's blood: what EGGG does is to read the game description into one long string; decisions are then made on the basis of regular expression matches.

Here is a typical line from the EGGG engine:

```
$output .= "use MD5; # Message digest algorithms, for security\n"
                                        if $game =~ /^turns\s+synch/m;
```

As the game description is parsed, the generated game is amassed in one long string: `$output`. In this line, the statement `use MD5;` is added to `$output` along with a comment — if the game description contains a statement that begins with turns, one or more spaces, followed by `synch`.

The current version of EGGG is again quite fast; no game takes more than thirty seconds to generate on the two platforms used to develop EGGG: a 233 MHz Pentium II laptop running Windows 98, and a 199 MHz DEC Alphastation running OSF/1 v4.0.

Speed is more of a concern in the play of the generated game. For instance, the minimax procedure is woefully inefficient, because it has none of the shortcuts that a program for a specific game would employ. The extra layer of indirection provided by the generic `%game` and `%state` structures slows the procedure even more. Many optimizations are possible; the first to be implemented will probably be a memoizer for identifying previously evaluated moves.

How About Some *Real* Graphics?

> *You did your best to razzle-dazzle them with what graphics you had, but you had, like, sixteen colors and three blocky things. So a lot of the work just went into the play of the game.*
>
> Eugene Jarvis, creator of Defender and Robotron, in *Joystick Nation.*

The most popular computer games of today are more interactive movies than games. The play of games like Doom, Quake, and Tomb Raider are relatively straightforward; they are little more than lavishly produced visualizations of text adventure games. But their popularity stems from the production values. [Herz 97] says,

> *Those games require people whose sole job it is to do texture*

mapping, or polygon animation, or backgrounding, or puzzle building. "At this point, game worlds have become so immense and complicated that their construction requires crews of postcollegiate code carpenters and graphic design masons working sixteen-hour days for months or years...

EGGG will never be able to generate games like these. EGGG is best at generating algorithmically and geometrically simple games. A parallel can be found in Doug Lenat's seminal AI project, "AM", which generated theorems about geometry. AM was well-suited for geometric theorems, and ill-suited for everything else. Generalizing it to other mathematical disciplines would have required, in effect, creating an entirely new system. In the same way, EGGG works for games that can be expressed with simple algorithms; a system that generated games like Doom would require an entirely different software architecture. That's not to say that similarities between Doom-like games don't exist, only that abstracting them into reusable code components requires an attention to graphical detail that is far removed from the purity of simple game play.

What About Sound?

All of EGGG's games are silent. A reasonable future direction for EGGG would be to have it automatically generate sounds for games.

There are similarities between the sounds that different games use. Some of the similarities stem from what sound effects are available to game designers. Other similarities exist because many of the actions meriting sound are shared among games. When something explodes it should make an explosion sound, whether the thing exploding is a box, bomb, or planet. Sounds with a quick attack and decay are used to indicate that two things have collided and stuck together; sounds with a quick attack and long decay are used for things traveling through space, like missiles from a ship. There are traits common to the sounds of success: chirping trills or short ditties.

Unfortunately, there is no software library that does for sound what Tk does for graphics, so sounds are not planned for EGGG in the near future.

Other Domains

Games are an ideal domain for research into automated programming. They possess just the right amount of diversity; they're popular enough that wide audiences can be obtained; they're never so mission-critical that perfection is required; they can be represented with a concise set of rules; and there will always be a demand for a multitude of different programs — no one game can eliminate the demand for the rest.

What other domains fit this criteria? It's hard to say without comprehensive research into the types of programs that people create. Areas like image processing or financial programming are too broad for generalizations like those explored in this dissertation. In this section, we'll touch on a few other domains.

Certainly there are many similiarities between large numbers of existing web pages. Good web pages embody a few basic design principles, and there are a limited number of ways to make web pages collect information and react to the user dynamically. However, there are many existing tools for automating web page design, and people often want to fine-tune web pages in ways that can't be described much more concisely than whatever HTML or code would be used to implement the fine-tuning.

The domain (if it can be called that) of *downloading* web pages is more promising. Many people would like to be able to download web pages automatically, to collect weather information, comic strips, news articles, or their stock portfolios. Typically, a user wants to download only a portion of the web page, skipping advertisements and navigational information. A variety of mechanisms keep them from doing so: the programming knowledge needed to automate HTTP requests, the effort needed to digest web cookies, links hidden behind links. Furthermore, the ever-changing formats of web pages thwart any permanent solution: whatever effort goes into downloading a web page can be rendered useless when that web page is redesigned, and this is the best argument for an adaptable solution that can examine a web page and generate a program on the fly to download the appropriate information from a web site.

Graph design might constitute a good domain for automatic programming. [Mackinlay 86] develops rules for choosing how to represent and lay out graphs of quantities in two and three dimensions, determining what attributes to use (position, size, color, and so on) based on the relationships between the information to be presented. Many people need to generate graphs, it's often a labor-intensive process, and the absolute best presentation isn't required for the graph to be useful.

Music generation might constitute a rich area for automated programming. There are similarities between genres of music (time signatures, keys, modes), between super-genres (for example, the pentatonic scale common in Eastern music), and between sub-genres (the 2.5-to-five-minute verse-chorus-bridge pattern common in rock songs). Such a system might choose these mid-level "cliches" (as they are called in the Programmer's Apprentice [Rich 90]) at random, which would differ from the completely bottom-up approaches toward music generation (such as fractal music synthesis) or completely top-down approaches that start with a theme, and from there develop it "downward" into movements and phrases and bars. An EGGG-like system can't supply the creativity; all it can do is automate the grunt work. The success of such a system will inevitably be constrained by the extent to which the language hinders or helps the expression of creativity.

A more mundane domain is data translation — converting information from one well-defined format to another. Many formats have ways to tag information, to

delimit beginnings and ends, and ways to escape information or handle binary objects; these are the similarities that could be exploited by an EGGG-like system. There have been many attempts to facilitate conversion between data formats; typically, these operate by defining a more generic format (XDR, ASN.1) capable of representing all of the formats to be converted. Such metaformats typically enjoy very limited success; we suggest that this is because the extra layer of abstraction required makes them harder to learn and easier to forget. But if the mapping between two specific formats can be described with precision, an EGGG-like system could be used to create a utility that converts between them.

Still, none of these areas are completely satisfactory. Most fall short of our last criterion for choosing a domain: the domain should benefit from a plethora of programs. Why create a system that generates programs to download web pages instead of simply creating a generic downloader? Why create a system that generates translators instead of a generic translator? Music synthesis meets the last criterion, because the system's output would be not applications, but artworks — there will always be a demand for new and different music. The similarities are certainly there, and synthesized music won't have to be perfect. But music can't be represented with a concise set of rules. More precisely, only a small portion of the elements comprising a musical performance can be represented with a concise set of rules; an EGGG-like system won't be generating operas, symphonies, or Top 40 hits anytime soon.

Finally, it would be seductively elegant to apply automated programming to itself, creating a system that generates EGGG-like systems, by exploiting the similarities of similarities between domains. Perhaps someday this will be possible, but automated programming has a long way to go before it is.

Scriptability and Glass Boxes

Too many applications these days require physical presence at a computer — click here to make this dialog box go away; move the mouse over here to select that option. This is unfortunate, because it means that you can't operate the applications from another computer. Worse, it means that you can't write programs to operate those programs on your behalf. (Many people insist on calling the controlling programs "scripts", as though there were some qualitative difference between scripts, programs, and applications — even though under the hood they're all just programs.)

All applications should be scriptable; that is, you should be able to control them via some text-based protocol. EGGG programs are not scriptable, but scriptability is part of a planned redesign to make EGGG programs "portable" to ASCII, so that any game can be represented with both the familiar Tk widgets and raw text terminals.

The non-scriptability of EGGG highlights one of its most serious deficiencies. EGGG makes game designers more productive, and it (arguably) helps make them into better game designers by shortening the game development cycle, but it doesn't help them become better game programmers. The convoluted design of EGGG encourages users to treat it as a black box: they provide input (the game description)

and the system turns that, somehow, into output (the game program). The implementation of the abstractions described in chapters 2 and 3 are hidden from the user; exposing these abstractions would let designers use EGGG more intelligently by enabling them to see how EGGG translates game descriptions into programs. Put another way, EGGG doesn't encourage game designers to become power users, and it should.

Open Source Games

Id software, the company that made Doom, had the right idea. They released a demo version of their game for free, and then they released the API for their game engine. This allowed people to write their own Doom levels, and made a lot of people design their first games ever. "Happy" versions of Doom where guns shot flowers, games where the sinister chords of the game were replaced by Muzak-perverted versions of pop songs, and games with the monsters replaced by cartoon chracters. Doom has been used by the military to train soldiers, and by system administrators to find and destroy errant processes on workstations.

Allowing devoted players to embrace and extend a game is, in my opinion, a critical step toward breaking the stagnation of fresh game ideas that has settled over today's arcades. All games are interactive, but this is a new, intellectually promising type of interactivity — drawing people into the game generation process and fostering the creativity, insight, cleverness, and critical thought needed as a result.

Chapter 1 began with this quote from Steve Russell, the author of the first video game:

> *I think the thing I take the most pride in about Spacewar is that it got so many people hooked on computer programming. It caught a lot of eyes and got a lot of interesting people asking, "How do you do that?"*

The game consoles of Sony, Sega, and Nintendo don't encourage players to become designers. The software architectures are proprietary; licenses and development kits cost tens or even hundreds of thousands of dollars. I wish these companies were more like Id software, because when players become designers, both people and games benefit.

> *The game isn't over till it's over.*
>
> Yogi Berra

Bibliography

Abelson, H. and G.J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge MA, 1985.

Aho, A.V., S.C. Johnson, and J.D. Ullman. Deterministic Parsing of Ambiguous Grammars. In *Communications of the ACM* 18:8, pp. 441-452, August 1975.

Aho, A.V., R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

Allis, L.V., H.J. van den Herik, and I.S. Herschberg. Which Games Will Survive, in D.N.L. Levy and D.F. Beal, eds., *Heuristic Programming in Artificial Intelligence 2 — The Second Computer Olympiad.* Ellis Horwood, 1991.

Arnot, M. *Crossword Puzzles For Dummies.* IDG Books, 1998. (Yes, it is a "Dummies" book, but it really does have insightful comments about crossword creation.)

Avedon, E.M. and B. Sutton-Smith. *The Study of Games.* John Wiley & Sons, 1971.

Baumgartner, V. *Graphic Games: From Pattern to Composition.* Prentice Hall, Englewood Cliffs, NJ, 1983.

Beasley, J. *The Mathematics of Games.* Oxford University Press, Oxford, 1989.

Berlekamp, E., and D. Wolfe. *Mathematical Go: Chilling Gets the Last Point.* A.K. Peters, Wellesley MA, 1994.

Bennahum, David S. *Extra Life: Coming of Age in Cyberspace.* Basic Books, New York, 1998.

Bennett, P.G. (ed.) *Analysing Conflict and its Resolution: Some Mathematical Contributions.* Clarendon Press, Oxford, 1987.

Berlekamp, E.R., H.H. Conway, and R.K. Guy. *Winning Ways For Your Mathematical Plays.* Academic Press, 1982.

Blaquiere, A. *Quantitative and Qualitative Games.* Academic Press, New York, 1969.

Blass, A. A Game Semantics for Linear Logic. In *Annals of Pure and Applied Logic* (56) pp. 182-220, 1992.

Blass, A. Degrees of Indeterminacy of Games. In *Fundamenta Mathematicae* (77), pp. 151-166, 1972.

Burns, B. (ed.) *The Encyclopedia of Games.* Brown Packaging Books Ltd., London,

1998.

Brennan, Susan. Conversation With and Through Computers. *Second International Conference on User Modeling*, Honolulu, 1990.

Bruckman, A. *MOOSE Crossing: Construction, Community, and Learning in a Networked Virtual World for Kids.* PhD Thesis, MIT Media Laborary, May 1997.

Caillois, R. *Man, Play, and Games.* Free Press of Glencoe, New York, 1961.

Carse, J.P. *Finite and Infinite Games.* The Free Press, 1986.

Condon, A. *Computational Models of Games.* MIT Press, Cambridge MA, 1989.

Costello, M.J. *The Greatest Games Of All Time.* John Wiley & Sons, New York, 1991.

David, F.N. *Games, Gods, and Gambling.* Hafner Co., New York, 1962.

Deng, X., and S Mahajan. Infinite Games, Randomization, Computability, and Applications to Online Problems. In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pp. 289-298, May 1991.

Dershowitz, N., and J.-P. Jouannaud. Rewrite Systems. In *Handbook of Theoretical Computer Science*, Vol. B, pp. 243-320. J. van Leeuwen (ed.), North-Holland, Amsterdam, 1990.

Diderich, C.G. and M. Gengler. *A Survey on Minimax Trees and Associated Algorithms.* Computer Science Department, Swiss Federal Institute of Technology DI-94/50, May 1994.

Donnelly, R.J. *Active Games and Contests.* Ronald Press Co., New York, 1958.

Duke, R.D. *Game-Generating-Games: A Trilogy of Games for Community and Classroom.* Sage Publications, Beverly Hills CA, 1979.

Edmonson, M. The rezrov Infocom Game Interpreter. In *The Perl Journal*, Spring 1999, pp. 52-60.

Eigen, M. *Laws of the Game: How the principles of nature govern chance.* Harper & Row, New York, 1983.

Elo, A. E. *The Rating of Chessplayers, Past And Present.* Batsford, 1978.

Felleisen, M. On the Expressive Power of Programming Languages. In *Proceedings of the European Symposium on Programming*, pp. 134-151. Springer Press, Berlin, 1990.

Fine, G.A. *Shared Fantasy.* University of Chicago Press, 1983.

Fokkinga, M.M. On the Notion of Strong Typing. In *Algorithmic Languages*,

DeBakker and can Vliet (eds.), pp. 305-320, IFIP North-Holland, Amsterdam, 1981.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1984.

Gamson, W.A. *What's News: A Game Simulation of TV News.* Macmillan Press, New York, 1984.

Gao, X., H. Iida, J.W.H.M. Uiterwijk, and H.J. van den Herik. A Speculative Strategy. In *Computers and Games, Lecture Notes in Computer Science, First International Conference, CG '98*, Tsukuba, Japan, November 1998.

Garey, M.R., and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

Geertz, C. Notes on the Balinese Cockfight. In *The Interpretation of Cultures*, Basic Books, 1973.

Goodfellow, C. *A Collector's Guide to Games and Puzzles.* Chartwell Books, 1991.

Guiasu, S. *Coalition and Connection in Games: Problems of modern theory using methods belonging to systems theory and information theory.* Pergamon Press, New York, 1980.

Gunter, C.A. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, Cambridge MA, 1992.

Gunter, C.A. and D.S. Scott. Semantic Domains. In *Handbook of Theoretical Computer Science*, Volume B., J. van Leeuwen (ed.), pp. 633-674, North-Holland, Amsterdam, 1990.

Halck, O.M. and F.A. Dahl. On Classification of Games and Evaluation of Players — with Some Sweeping Generalizations About the Literature. *Machine Learning in Game Playing Workshop, 16th International Conference on Machine Learning*, Slovenia, June 30, 1999.

Hennessy, M. *The Semantics of Programming Languages: An Elementary Introduction using Structured Operational Semantics.* Wiley, 1990.

Herz, J.C. *Joystick Nation.* Little, Brown & Company, 1997.

Hodges, W. *Building Models by Games.* Cambridge University Press, 1985.

Hofstadter, D. *Fluid Concepts and Creative Analogies.* Basic Books, New York, 1995.

Horowitz, E. *Fundamentals of Programming Languages.* Computer Science Press, 1984.

Jackson, S. *GURPS: Generic Universal Role Playing System.* Steve Jackson Games, 1997.

Jacobson, N., W. Bender, and U. Feldman. Alignment and Amplification as Determinants of Expressive Color. In *Proceedings of the SPIE*, Vol. 1453, February 1991.

Klop, J.W. *Term Rewriting: A Tutorial.* In EATCS Bulletin (32), pp. 143-182, 1987.

Koda, T. *Agents with Faces: A Study on the Effects of Personification of Software Agents.* M.S. Thesis, MIT Media Laboratory, September 1996.

Kotzamani, M.A. *Wittgenstein on Philosophy and Language-Games.* MIT M.S. Thesis, 1987.

Lance, D.F. and B. Allan Tindall, eds. The Anthropological Study of Play: Problems and Prospects. *Proceedings of the first annual meeting of the Association for the Anthropological Study of Play.* Leisure Press, Cornwall N.Y, 1976.

Landin, P.J. The Next 700 Programing Languages. In *Communications of the ACM* (9), pp. 157-166, 1966.

Larrabee, T., K. McCall, C. Mitchell, and B.C. Pierce. *Gambit: A Video Game Programming Language.* Stanford project report CS-242, December 1982.

Larrabee, T. and C. Mitchell. Gambit: A Prototyping Approach to Video Game Design, *IEEE Software*, Vol. 1, No. 4, October 1984.

Lewis, H.R., and C.H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1981.

Liskov, B. and J. Guttag. *Abstraction and Specification in Software Development.* MIT Press, Cambridge MA, 1986.

Loftus, G.R. *Mind At Play: The Psychology of Video Games.* Basic Books, New York, 1983.

London, R.L. Program Verification. In *Research Directions in Software Technology*, P. Wegner (ed.), MIT Press, Cambridge MA, pp. 302-315, 1978.

Luce, R.D., and H. Raiffa. *Games and Decisions: Introduction and Critical Survey.* Dover Publications, New York, 1957.

Mackinlay, J. Automating the Design of Graphical Presentations of Relational Information. In *ACM Transactions on Graphics* 5:2, pp. 110-141, April 1986.

Matsubara, H. *Proceedings of 2nd Game Programming Workshop.* Computer Shogi Association, Tsukuba, Ibraki, Japan, 1995.

McDonald, J.D. *Strategy in Poker, Business, and War.* Norton, New York, 1950.

Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, 1988.

Mitchell, J.C. *Foundations of Programming Languages.* MIT Press, Cambridge MA,

1996.

Mitchell, J.C. On Abstraction and the Expressive Power of Programming Languages. In *Proceedings of the Theoretical Aspects of Computer Software* pp. 290-310, September 1991.

Mohr, Merilyn Simonds. *The New Games Treasury.* Houghton Mifflin, 1997.

Moore, J. *The Complete Book of Sports Betting.* Carol Publishing Group, New York, 1996.

Morehead, A.H., R.L. Frey, and Geoffrey Mott-Smith. *The New Complete Hoyle*, revised ed. Doubleday, 1991.

Morgan, C. *Programming from Specifications.* Prentice-Hall, 1990.

Orwant, Jon. *Doppelgänger Goes To School: Machine Learning for User Modeling.* MIT M.S. Thesis, Media Arts and Sciences, September 1993.

Orwant, Jon. *The Doppelgänger User Modeling System.* MIT B.S. Thesis, Department of Electrical Engineering and Computer Science, June 1991.

Orwant, Jon. For Want Of A Bit The User Was Lost: Cheap User Modeling. In *IBM Systems Journal*, 38:3-4, 1996.

Pell, Barney. METAGAME: A New Challenge for Games and Learning. H.J. van den Herik and L.V. Allis, eds., *Heuristic Programming in Artificial Intelligence 3 - The Third Computer Olympiad.* Ellis Horwood, 1992.

Pell, Barney. *Strategy Generation and Evaluation for Meta-Game Playing.* PhD thesis, University of Cambridge, August 1993.

Rabiner, L.R. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. In *Proceedings of the IEEE*, 77(2):257-285, 1989.

Rapoport, A. and A.M. Chammah. *Prisoner's Dilemma.* University of Michigan Press, 1965.

Reif, J.H. Universal Games of Incomplete Information. In *Conference Record of the Eleventh Annual (ACM) Symposium on Theory of Computing*, pp. 288-308, May 1979.

Rich, C. and R.C. Waters. *The Programmer's Apprentice.* ACM Press, 1990.

Ross, D.T. On Context and Ambiguity in Parsing. In *Communications of the ACM* 7:2, pp. 131-133, February 1964.

Samuel, A.L. Programming Computers to Play Games. In *Advances in Computers* (1), pp. 165-192, 1960.

Scarne, J. *Scarne's Encyclopedia of Games.* Harper & Row, New York, 1973.

Schmittberger, Wayne R. *New Rules for Classic Games.* John Wiley & Sons, 1992.

Schoett, O. *Data Abstraction and the Correctness of Modular Programs.* CST-42-87, University of Edinburgh, 1987.

Sethi, R. *Programming Languages: Concepts and Constructs.* Addison Wesley, 1989.

Singleton, R. *Games and Programs: Mathematics for Modeling.* W.H. Freeman, San Francisco, 1974.

Solomon, E. *Games Programming.* Cambridge University Press, New York, 1984.

Steiner, Peter O. *Thursday Night Poker.* Random House, 1996. (Equal parts probability and psychology, the way every poker book should be.)

Stoyan, H. The Influence of the Designer on the Design — John McCarthy and LISP. In *Artificial Intelligence and the Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz (ed.) pp. 409-426, Academic Press, 1991.

Therrien, C.W. *Decision, Estimation, and Classification.* John Wiley & Sons, New York, 1989.

Weintraub, E.R. (ed). *Toward a History of Game Theory.* Duke University Press, 1992.

Williams, P.W., and D. Woodhead. Computer Assisted Analysis of Cryptic Crosswords. In *The Computer Journal*, 22:1, pp. 67-70, February 1979.

Wittgenstein, L. *Philosophical Investigations.* Macmillan, 1958.

Wittgenstein, L. *The Blue and Brown Books.* Harper, 1958.

Appendix A: EGGG Installation Instructions

0. Retrieve the EGGG distribution from http://orwant.www.media.mit.edu/eggg/eggg.

1. If you are on Windows, go to step 4. Otherwise, unpack the EGGG distribution with

```
gzip -d eggg-0.11.tar.gz
tar xvf eggg-0.11.tar
```

If you don't have gzip, you can FTP the latest version from ftp://prep.ai.mit.edu/pub/gnu.

2. Install Perl 5.004 (or higher) if you don't have it already, from http://www.perl.com/CPAN/src. You can find out what version of Perl you have by typing this at your command prompt, regardless of what directory you're in:

```
perl -v
```

If you see a "command not found" error, try

```
/usr/bin/perl -v
```

or

```
/usr/local/bin/perl -v
```

If you still get "command not found" errors, or if the version is less than 5.004, you'll need to install Perl 5.004. (Perl 5.005, which is the latest verion, is preferable.)

3. Install the following Perl modules. Once you've installed Perl, the CPAN module can install other Perl modules for you. To use the CPAN module, type this at your command prompt, regardless of what directory you're in:

```
perl -MCPAN -e 'shell'
```

That places you into a shell where you'll have to answer all sorts of questions about how you want Perl modules installed. After you've done that, you're ready to install modules by typing at the cpan> prompt:

```
cpan> install Data::Dumper
cpan> install Sys::Hostname
cpan> install MD5
```

```
Those modules should install relatively quickly.  This one
will take a while, however:
```

```
cpan> install Tk
```

Exit the cpan> shell by typing "quit" at the prompt.

Skip to step 6: "Testing your installation".

4. To install EGGG on windows, unzip eggg.zip. If you don't have the WinZip utility, you can download it from http://www.winzip.com.

I like running EGGG from Emacs in shell mode, because Emacs in shell mode is a MUCH better shell than MS-DOS. (You can get Win32 Emacs from http://www.cs.washington.edu/homes/voelker/ntemacs.html.)

5. Install Perl on Win32 by downloading the latest version from http://www.activestate.com/ActivePerl/. This also installs the other modules that EGGG needs, like MD5 and Sys::Hostname and Tk.

6. Testing your installation.

By now, you should have a working Perl with all of the modules that EGGG needs. Move into the directory with the EGGG distribution. The next step is to see if your Perl environment is working properly.

Try running one of the games that I created with EGGG. Anything ending in ".test" will do. The best ones to begin with are: poker.test, cross.test, tictactoe.test, chess.test, tictactoe4.test, rps.test, deducto.test, deducto-rgb.test, deducto-hsv.test.

From a Unix shell prompt, just type the name of the game. Here's how to launch a crossword puzzle:

```
cross.test
```

From a DOS or Win32 Emacs shell, prepend "perl":

```
perl cross.test
```

If any errors occur, please let me know. Perhaps Perl won't be able to find one of the modules; if that happens, the error message will contain the word "@INC" near the beginning. Or perhaps there's a runtime error, in which case the error message will be something like:

```
Warning: something's wrong at cross.test line 432.
```

If you can tell me what the error message is, I'll fix the problem.

7. Running EGGG with a preexisting game description

Assuming step 6 was successful, the next step is to run EGGG itself. (Step 6 was just running a game *generated* by EGGG.) The "eggg" program is invoked like so, on Unix:

```
eggg -o poker poker.egg
```

Or on Win32 (this will work on Unix also):

```
perl eggg -o poker poker.egg
```

This runs the "eggg" program with three arguments: "-o", "poker" and "poker.egg". The first two arguments tell EGGG that it's creating a game named "poker". EGGG will obliterate any currently existing file named poker and replace it with the game.

"poker.egg" is the game description. View it with your favorite editor if you like, so you can see what information EGGG uses to create the game.

Once you've run eggg, you should have a fully functioning game that you can run as shown in Step 6:

```
perl poker
```

8. Running EGGG with a new game description

To create your own game, choose a currently existing game description file that's similar to the game you want to create. (You could start from scratch, but I haven't written up how to do that yet.) Hopefully you'll be able to infer enough from the EGGG language. As a test, I suggest you copy the game description file for poker and edit it to define a new hand.

Unix: cp poker.egg poker-new.egg

```
emacs poker-new.egg
```

or

```
vi poker-new.egg
```

Windows: copy poker.egg poker-new.egg

```
emacs poker-new.egg
```

or

```
edit poker-new.egg
```

Let's say you want to define a hand called BabyStraight, which has four cards of successive ranks, like 2-3-4-5 or 9-10-J-Q. Notice how Straight is defined like so:

```
Straight has (R, s) and (R+1, s) and (R+2, s) and (R+3, s) and (R+4, s)
```

Each ordered pair represents a card; the first term is the rank, and the fact that it's capitalized means that it's important, and should be remembered from term to term. So if R is 4 for the first card, then R+1 is constrained to be 5 for the second card. The s is the suit; since it's not capitalized, it's not important. So a BabyStraight is just a Straight without the last term:

```
BabyStraight has (R, s) and (R+1, s) and (R+2, s) and (R+3, s)
```

You can add this line anywhere in the file you like.

The hands are ranked with this line:

```
hands are [StraightFlush, FourKind, FullHouse, Flush, Straight,
           ThreeKind, TwoPair, Pair, HighCard]
```

You'll need to edit that line and stick BabyStraight in wherever you'd like it to go:

```
hands are [StraightFlush, FourKind, FullHouse, Flush, Straight,
           ThreeKind, BabyStraight, TwoPair, Pair, HighCard]
```

Generate the game with `perl eggg -o poker-new poker-new.egg` and then run it with `perl poker-new`.

Enjoy!

Appendix B: Sample EGGG Games

Rock Paper Scissors

```
move is choose
pieces are Rock and Paper and Scissors
board starts [[Rock, Paper, Scissors]]
turns synchronize
Beat means player(Rock) && opponent(Scissors)
     or player(Scissors) && opponent(Paper)
     or player(Paper) && opponent(Rock)
goal is Beat # success!
score increments
3x1 grid
```

Tic Tac Toe

```
turn is player place piece
3x3 grid
pieces are X and O
turns alternate
players are X and O
goal is &Three_in_a_row
Three_in_a_row means (x-1, y) && (x, y) && (x+1, y)
     or (x, y-1) && (x, y) && (x, y+1)
     or (x-1, y-1) && (x, y) && (x+1, y+1)
     or (x-1, y+1) && (x, y) && (x+1, y-1)
board starts empty
```

Poker

```
turns alternate clockwise

Discard means player removes 0..3 cards or 4 cards if Ace()
2..6 players

game is poker

game is shuffle(deck) and deal(cards, 5) and bet(money) and
     Discard(hand, N) and deal(cards, 5-N) and compare(cards)

StraightFlush has (R, S) and (R+1, S) and (R+2, S) and (R+3, S)
              and (R+4, S)
FourKind has (R, s) and (R, s) and (R, s) and (R, s)
FullHouse has (R, s) and (R, s) and (R, s) and (Q, s)
          and (Q, s)
Flush has (r, S) and (r, S) and (r, S) and (r, S) and (r, S)
```

```
Straight has (R, s) and (R+1, s) and (R+2, s) and (R+3, s)
        and (R+4, s)
ThreeKind has (R, s) and (R, s) and (R, s)
TwoPair has (R, s) and (R, s) and (Q, s) and (Q, s)
Pair has (R, s) and (R, s)
HighCard has (R, s)

hands are [StraightFlush, FourKind, FullHouse, Flush,
           Straight, ThreeKind, TwoPair, Pair, HighCard]

goal is highest(hand)
```

Crossword

```
goal is solve
game is Crossword
text is Clues

squares are black or white
squares have letters and numbers
squares display numbers
solution displays letters
board displays numbers with abbreviations(Clues)
board starts [[1, 1, 1, 0, 0, 1, 1, 1],
              [1, 1, 1, 0, 0, 1, 1, 1],
              [1, 1, 1, 1, 1, 1, 1, 1],
              [0, 0, 0, 1, 1, 1, 0, 0],
              [0, 0, 1, 1, 1, 0, 0, 0],
              [1, 1, 1, 1, 1, 1, 1, 1],
              [1, 1, 1, 0, 0, 1, 1, 1],
              [1, 1, 1, 0, 0, 1, 1, 1]]

solution is [[E, T, A, 0, 0, R, E, F],
             [S, I, R, 0, 0, A, K, A],
             [C, O, M, P, U, T, E, R],
             [0, 0, 0, A, S, E, 0, 0],
             [0, 0, O, L, E, 0, 0, 0],
             [F, L, O, O, R, I, N, G],
             [O, A, R, 0, 0, C, I, A],
             [E, N, T, 0, 0, E, L, M]]

8x8 grid
composite puzzle # as opposed to a puzzle with a single answer
one player
```

Deducto

```
Tester(2) is return grid[3][3]
Tester(1) is Tot++ if grid[x][y]; return Tot > 12
Tester(3) is return 1 if grid[3][y] && (grid[1][y] && grid[2][y] or
                        grid[4][y] && grid[5][y]); return 0

# Tricky:  Tester(4) is true if there's an "on" square in each row.
```

```
Tester(4) is Tot |= (2**y) if grid[x][y]; return (Tot == 62) ? 1 : 0
Tester(5) is Tot++ if grid[x][y]; return Tot % 2

Generate(1) is while (!Tester(1)) { grid[x][y] = flip ? 1 : 0 }
Generate(3) is while (!Tester(3)) { grid[x][y] = flip ? 1 : 0 }
Generate(2) is { grid[x][y] = flip ? 1 : 0 }; grid[3][3] = 1
Generate(4) is while (!Tester(4)) { grid[x][y] = flip ? 1 : 0 }
Generate(5) is while (!Tester(5)) { grid[x][y] = flip ? 1 : 0 }

AntiGenerate(1) is while (Tester(1)) { grid[x][y] = flip ? 1 : 0 }
AntiGenerate(2) is { grid[x][y] = flip ? 1 : 0 }; grid[3][3] = 0
AntiGenerate(3) is while (Tester(3)) { grid[x][y] = flip ? 1 : 0 }
AntiGenerate(4) is while (Tester(4)) { grid[x][y] = flip ? 1 : 0 }
AntiGenerate(5) is while (Tester(5)) { grid[x][y] = flip ? 1 : 0 }

Example means grid becomes Generate(level)
Test means display(Tester(level))
VoteYes means Tester(level) && ++CORRECT or CORRECT = 0
VoteNo  means not Tester(level) && ++CORRECT or CORRECT = 0
assert: after move CORRECT == 5 => level = level+1 => CORRECT = 0
assert: after move CORRECT == 0 => display("you have 0 correct")
assert VoteYes: lastmove("Understand")
assert VoteNo:  lastmove("Understand")

Understand means display("Vote Yes or No.");
assert after VoteYes: flip ? Generate(level) : AntiGenerate(level)
assert after VoteNo: flip ? Generate(level) : AntiGenerate(level)
assert after Understand: flip ? Generate(level) :
 AntiGenerate(level);
       update()

game has Example button
game has Test button
game has Understand button
game has VoteYes button
game has VoteNo button
goal is level(6)
one player
5x5 grid
squares are white and black
click makes square toggle
```

Tetris

```
ClearLines means LineFull(Y) and MoveDown(higher Y) and score
                 increments
LineFull means on(all x, some y)
MoveDown means given(Y) and move(x, Y+1, x, Y)

bottom is sticky
sides are solid

round is merge(ActivePiece, Bottom) and ClearLines() and
         create(random piece P, 5, 20) and set(ActivePiece, P)

goal is no Fill
```

```
Fill is touches(ActivePiece, Top)
piece touches Bottom => round(n+1)

key j moves(x-1, y)
key l moves(x+1, y)
key k transforms(Bar1, Bar2, Bar2, Bar1, Zig1, Zig2, Zig2, Zig1,
                 Zag1, Zag2, Zag2, Zag1, Lell1, Lell2, Lell2, Lell3,
                 Lell3, Lell4, Lell4, Lell1, Rell1, Rell2, Rell2,
                 Rell3, Rell3, Rell4, Rell4, Rell1, Tee1, Tee2, Tee2,
                 Tee3, Tee3, Tee4, Tee4, Tee1)

turn is 0.5 seconds

turn is ActivePiece moves(x, y-1)
10x20 grid
pieces are [Bar1, Bar2, Block, Zig1, Zig2, Zag1, Zag2, Lell1, Lell2,
            Lell3, Lell4, Rell1, Rell2, Rell3, Rell4, Tee1, Tee2,
            Tee3, Tee4]
one player

Bar1  is [[1, 1, 1, 1], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Bar2  is [[0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0]]
Block is [[1, 1, 0, 0], [1, 1, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Zig1  is [[1, 0, 0, 0], [1, 1, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0]]
Zig2  is [[0, 1, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Zag1  is [[0, 1, 0, 0], [1, 1, 0, 0], [1, 0, 0, 0], [0, 0, 0, 0]]
Zag2  is [[1, 1, 0, 0], [0, 1, 1, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Lell1 is [[0, 1, 1, 1], [0, 1, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Lell2 is [[1, 0, 0, 0], [1, 0, 0, 0], [1, 1, 0, 0], [0, 0, 0, 0]]
Lell3 is [[0, 0, 1, 0], [1, 1, 1, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Lell4 is [[1, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0]]
Rell1 is [[1, 1, 1, 0], [0, 0, 1, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Rell2 is [[1, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0]]
Rell3 is [[1, 0, 0, 0], [1, 1, 1, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Rell4 is [[0, 1, 0, 0], [0, 1, 0, 0], [1, 1, 0, 0], [0, 0, 0, 0]]
Tee1  is [[0, 1, 0, 0], [1, 1, 1, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Tee2  is [[0, 1, 0, 0], [1, 1, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0]]
Tee3  is [[1, 1, 1, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Tee4  is [[1, 0, 0, 0], [1, 1, 0, 0], [1, 0, 0, 0], [0, 0, 0, 0]]
```

Chess

```
Stalemate means turn(player P) && no moves(P) => tie

Checkmate means no moves(opponent King) and
                Attacking(player piece P, opponent King) and
                no moves(opponent piece Q remove P)

turn is player move piece
assert: after move not Attacked("King")
goal is &Checkmate
tie is &Stalemate
```

```
turns alternate
pieces are King and Queen and Rook and Bishop and Knight and Pawn
players are white and black

board starts [[black Rook, black Knight, black Bishop, black Queen,
               black King, black Bishop, black Knight, black Rook],
              [black Pawn, black Pawn, black Pawn, black Pawn,
               black Pawn, black Pawn, black Pawn, black Pawn],
              [empty, empty, empty, empty,
               empty, empty, empty, empty],
              [empty, empty, empty, empty,
               empty, empty, empty, empty],
              [empty, empty, empty, empty,
               empty, empty, empty, empty],
              [empty, empty, empty, empty,
               empty, empty, empty, empty],
              [white Pawn, white Pawn, white Pawn, white Pawn,
               white Pawn, white Pawn, white Pawn, white Pawn],
              [white Rook, white Knight, white Bishop, white Queen,
              white King, white Bishop, white Knight, white Rook]]

King moves(7, 1) if white && on(5, 1) && on("white Rook", 8, 1)
           && empty(6, 1) && empty(7, 1) && nocheck("white", 5, 1)
           && nocheck("white", 6, 1) && unmoved("white Rook", 5, 1)
           && unmoved("King") => move(8, 1, 6, 1)
King moves(3, 1) if white && on(5, 1) && on("white Rook", 1, 1)
           && empty(2, 1) && empty(3, 1) && empty(4, 1)
           && nocheck("white", 5, 1) && nocheck("white", 4, 1)
           && nocheck("white", 3, 1) && unmoved("white Rook", 1, 1)
           && unmoved("King") => move(1, 1, 4, 1)
King moves(4, 8) if black && on(5, 8) && on("black Rook", 5, 8)
           && empty(6, 8) && empty(4, 8) && nocheck("black", 5, 8)
           && nocheck("black", 6, 8) && unmoved("black Rook", 5, 8)
           && unmoved("King") => move(5, 8, 6, 8)
King moves(3, 8) if black && on(5, 8) && on("black Rook", 1, 8)
           && empty(2, 8) && empty(3, 8) && empty(4, 8)
           && nocheck("black", 5, 8) && nocheck(4, 8)
           && unmoved("black Rook", 1, 8) && unmoved("King")
           => move(1, 8, 4, 8)

Knight moves(x+2, y+1) if empty(x+2, y+1)
Knight moves(x+2, y-1) if empty(x+2, y-1)
Knight moves(x-2, y+1) if empty(x-2, y+1)
Knight moves(x-2, y-1) if empty(x-2, y-1)
Knight moves(x+1, y+2) if empty(x+1, y+2)
Knight moves(x+1, y-2) if empty(x+1, y-2)
Knight moves(x-1, y+2) if empty(x-1, y+2)
Knight moves(x-1, y-2) if empty(x-1, y-2)
Knight captures as it moves

Pawn moves (x+1, 6) if white and on(x, 5) and
                       lastmove("black Pawn", x+1, 4, x+1, 5)
Pawn moves (x-1, 6) if white and on(x, 5) and
                       lastmove("black Pawn", x-1, 4, x-1, 5)
Pawn moves (x+1, 3) if black and on(x, 4) and
                       lastmove("white Pawn", x+1, 2, x+1, 4)
Pawn moves (x-1, 3) if black and on(x, 4) and
```

```
                        lastmove("white Pawn", x-1, 2, x-1, 4)

Pawn moves (x, 4) if white and on(x, 2) and empty(x, 3) and
                     empty(x, 4)
Pawn moves (x, 5) if black and on(x, 7) and empty(x, 6) and
                     empty(x, 5)
Pawn moves (x, y+1) if white and empty(x, y+1)
Pawn moves (x, y-1) if black and empty(x, y-1)
Pawn captures (x+1, y+1) if white and other(x+1, y+1)
Pawn captures (x-1, y+1) if white and other(x-1, y+1)
Pawn captures (x+1, y-1) if black and other(x+1, y-1)
Pawn captures (x-1, y-1) if black and other(x-1, y-1)

Rook moves (x, y-1..7) if empty(x, y-1..7)
Rook moves (x, y+1..7) if empty(x, y+1..7)
Rook moves (x-1..7, y) if empty(x-1..7, y)
Rook moves (x+1..7, y) if empty(x+1..7, y)
Rook captures as it moves

Bishop moves (x-1..7, y-1..7) if empty(x-1..7, y-1..7)
Bishop moves (x+1..7, y+1..7) if empty(x+1..7, y+1..7)
Bishop moves (x-1..7, y+1..7) if empty(x-1..7, y+1..7)
Bishop moves (x+1..7, y-1..7) if empty(x+1..7, y-1..7)
Bishop captures as it moves

Queen moves (x, y-1..7) if empty(x, y-1..7)
Queen moves (x, y+1..7) if empty(x, y+1..7)
Queen moves (x-1..7, y) if empty(x-1..7, y)
Queen moves (x+1..7, y) if empty(x+1..7, y)
Queen moves (x-1..7, y-1..7) if empty(x-1..7, y-1..7)
Queen moves (x+1..7, y+1..7) if empty(x+1..7, y+1..7)
Queen moves (x-1..7, y+1..7) if empty(x-1..7, y+1..7)
Queen moves (x+1..7, y-1..7) if empty(x+1..7, y-1..7)
Queen captures as it moves

King moves (x-1, y) if empty(x-1, y)
King moves (x, y-1) if empty(x, y-1)
King moves (x-1, y-1) if empty(x-1, y-1)
King moves (x+1, y) if empty(x+1, y)
King moves (x, y+1) if empty(x, y+1)
King moves (x+1, y+1) if empty(x+1, y+1)
King moves (x+1, y-1) if empty(x+1, y-1)
King moves (x-1, y+1) if empty(x-1, y+1)
King captures as it moves

8x8 grid
```

Appendix C: The EGGG Grammar

```
Game : Statement(s) TokenBlock(s?)
Statement: Block | CompoundStatement

Block : BlockWord '{' SimpleStatement(s) '}'
BlockWord : 'forever' | 'while' | 'do'

CompoundStatement: SimpleStatement

SimpleStatement: Assertion Comment(?) "\n"
               | Declaration Comment(?) "\n"
               | Assignment Comment(?) "\n"
               | "\n"

Comment: '#' String(s)

Declaration: Imperative Noun HowVerb Rvalue
           | Action Condition(?) Result(?)
           | Adjective Noun

HowVerb: 'with' | 'by'

Action: Noun StateVerb Rvalue(?)
      | Noun InvertVerb
      | Noun Invocation

InvertVerb: /toggles?/ | /inverts?/ | /flips?/ | /reverses?/

Result: '=>' Invocation Result(?)

Assignment: Noun IsVerb Code
          | Noun IsVerb Action
          | Noun IsVerb Rvalue Result(?)

Rvalue: BooleanRvalue
      | Coordinate (Operator Coordinate)(s?)
      | Result

BooleanRvalue: SimpleRvalue (Junction SimpleRvalue)(s?)

SimpleRvalue: (InvertOp)(?) (Invocation | ArrayRvalue | Time
            | Object | PieceValue | ButtonValue | Direction | Status
            | Adjective) Condition(?)

ArrayRvalue: '[' SimpleRvalue(?) (',' SimpleRvalue)(s?) ']'
ButtonValue: Name 'button'
Direction: 'clockwise' | 'counterclockwise' | 'up' | 'down' | 'left'
         | 'right'
Condition: Predicate Boolean

Boolean: (InvertOp)(?) Invocation (Junction Boolean)(s?)
       | (InvertOp)(?) Adjective (Junction Boolean)(s?)
```

```
Invocation: Function Arguments
Assertion: 'assert:' RelativeTime Phase Boolean Result(?)

RelativeTime: 'before' | 'after'

Phase: 'turn' | 'game' | 'round' | 'move'

Function: /\w+/

Arguments: '(' Argument(?) (',' Argument)(s?) ')'
Argument: GenericObject | Term

Coordinate: '(' Term (/,/ Term)(s?) ')'

Mutator: 'remove'

Term: Code | '"' Word(s?) '"'
    | '"' Adjective Name '"'
    | Factor (Operator Factor)(s?)


Factor: Range | Letter | /\w+/ | Number
Letter: /\w+/ '[' /\w/ ']'

CompactExpression: /\w.\w/

Operator: Junction | InvertOp | '-' | '+' | '*' | '/' | '**'

Junction: 'and' | 'or' | '&&' | '||' | 'with'
InvertOp: 'not' | '!' | 'no' | 'avoid'

Adjective: Color | Size | Shape | Location | Completion | IO
         | Name | Subroutine

IO: 'input' | 'output'
GenericObject: Invocation | Qualifier Thing(?) (Axis | Name
             | Variable | Number)(?) (StateVerb (Variable
                                      | Name))(?)
Object: Qualifier | Thing

Qualifier: 'opponent' | 'self' | 'player' | 'random' | 'all'
         | 'some' | 'higher' | 'lower' | 'greater' | 'smaller'
         | 'more' | 'less' | 'max' | 'min' | 'length' | 'first'
         | 'second' | 'last' | Type
Thing: 'piece' | 'board' | 'card' | 'deck' | 'word' | 'number'
Variable: /[A-Z]/
Axis: /[a-z]/
Type: /strings?/ | /integers?/ | /numbers?/ | /words?/
Status: 'on' | 'off'

Imperative: 'choose' | 'ask' | 'solve'
IsVerb: 'is' | 'am' | 'are' | 'has' | 'have'
      | /\bstart(s|ing)?\b/ | /\bbegin(s|ning)?\b/
      | /\bmean(s|ing)\b/ | /\bmake(s|ing)\b/
      | /\bimpl(y|ies|ying)\b/
StateVerb: /\bplac(e|es|ing)\b/ | /\bbecomes?\b/
```

```
        | /\balternat(e|es|ing)\b/ | /\bsynch(roniz(e|ing|es))?\b/
        | /\b(re)?mov(e|es|ing)\b/ | /\btouch(es|ing)?\b/
        | /\bpresent(s|ing)?\b/ | /\bcompar(e|es|ing)\b/
        | DisplayVerb | CardVerb | BetVerb
DisplayVerb: /\bdisplay(s|ing)?\b/ | /\bshow(s|ing)?\b/
CardVerb: /\bshuffl(e|es|ing)\b/ | /\bfold(s|ing)?\b/
       | /\bdeal(s|ing)?\b/ | /\bdiscard(s|ing)?\b/
BetVerb: /\bbet(s|ting)?\b/ | /\brais(e|es|ing)\b/
      | /\bcall(s|ing)?\b/

Color: 'black' | 'white' | 'red' | 'blue' | 'green' | 'yellow'
     | 'gray' | 'orange'

Size: Dimension | Number Unit | Range Unit(?) | Number
Range: Integer '..' Integer

Dimension: /\d+x\d+(x\d+)?/

Unit: OneDimensionalUnit | TwoDimensionalUnit | ThreeDimensionalUnit
OneDimensionalUnit: /in(ch(es)?)?/ | /(cm|centimeters?)/ | /pixels?/
                  | /char(acters?|s?)/ | /cards?/
TwoDimensionalUnit: /sq(uare)?/ OneDimensionalUnit
                  | /squares?/ | /spots?/ | /places?/
ThreeDimensionalUnit: 'cubic' OneDimensionalUnit | /cubes?/
                    | /spots?/ | /places?/
Shape: 'square' | 'circle' | 'triangle' | 'card' | 'line'

Time: Number (/seconds?/ | /minutes?/)

Location: RelativeLocation | AbsoluteLocation
RelativeLocation: 'top' | 'bottom' | 'left' | 'right' | 'up'
                | 'down'
AbsoluteLocation: '(' Number ',' Number ')' | Number

Number: /\d+(\.\d+)?/ | 'one' | 'two' | 'three' | 'four' | 'five'
      | 'six' | 'seven' | 'eight' | 'nine' | 'ten'

Noun: Invocation | /grids?/ | /pieces?/ | /players?/ | /turns?/
    | /rounds?/ | /cards?/ | /words?/ | 'money' | 'game' | 'goal'
    | 'tie' | 'board' | /hands?/ | /puzzles?/ | /squares?/
    | /numbers?/ | 'origin' | 'solution' | 'button'
    | Key | Click | Name

Key: 'key' /\S/
Click: 'click' | 'left' | 'right' | 'middle'
TokenBlock: '__' ('START' | 'PRELOOP' | 'LOOP' | 'POSTLOOP'
          | 'FINISH' | 'PARSER') '__' String(s) '__END__'

PieceValue: 'piece' | 'card' | /numbers?/ | /letters?/
          | Adjective Name


Completion: 'full' | 'whole' | 'covered' | 'blank' | 'clear'
          | 'empty' | 'done' | 'partial' | 'composite'

Predicate: 'if' | 'unless'
```

```
Code: "'" SnippetQuote(s) "'" | '{' SnippetBrace(s) '}'

Name: /[A-Z]\w*/
Subroutine: /&\w+/
Word: /\w+/
SnippetQuote: /[^']+/
SnippetBrace: /[^}]+/
String: /\S+/
Integer: /\d+/ | /[mnr]/i
```

Colophon

This document was written in pod, which stands for Plain Old Documentation. pod is a very simple markup system, simpler even than HTML. By minimizing the presence of markup characters in a text, pod allows authors to concentrate on the words they are writing instead of the more superficial qualities of appearance and structure.

The document was converted to LaTeX, HTML, and FrameMaker by the pod conversion programs bundled with Perl.

Biographical Note

Jon Orwant received a Bachelor of Science in Computer Science and Engineering, a Bachelor of Science in Cognitive Science, and a Master of Science in Media Arts and Sciences, all from MIT. He worked in the MIT Artificial Intelligence Laboratory from 1987 to 1989, and in the MIT Media Laboratory from 1988 until the present.

He is the author of two books: O'Reilly's Mastering Algorithms with Perl and Macmillan's Perl 5 Interactive Course. A third book, O'Reilly's Manipulating Text with Perl, will be published in 2000.

Mr. Orwant was a founding director of The Perl Institute and served on its board of directors from 1996 to 1999. He is on the advisory board of The Perl Mongers, and is on the technical advisory board of Focalex, Inc.

He has published numerous articles about user modeling, and runs the annual Internet Quiz Show. He was Publisher of The Perl Journal since its inception in 1995, and remains Editor-in-Chief. He has written numerous articles about Perl, and has been on the technical committees of all Perl conferences.