Prospectus:

Simple Computer Games: Introduction to Programming in Java

Introduction

Beginning computer science students are overwhelmed by the cognitive overhead of writing their very first line of Java. This is one reason for the general decline in interest in computer science in the United States and the difficulty in recruiting students into the computer science major. A simpler introduction utilizing a spiral approach is necessary to support students in the first course. This prospectus describes a new book, *Simple Computer Games* (*SCG*), that both engages students with examples and projects drawn from a wide range of computer games and holds complexity at bay by abstracting away the first line of Java in a toolkit.

Using a strong, "objects-early" pedagogy, *SCG* teaches the logical underpinnings of sequence, selection, iteration, and delegation with a gradual learning curve suitable for CS1 in a small-liberal arts college. Introducing clear, concise communication with both the human and the machine audience, *SCG* helps beginning students reach a level of maturity with regard to computer programming and thinking about computer games in general.

SCG leverages student interest in computer games and permits them to write interesting games from the very beginning using the SCAN framework. As students mature, they replace more and more of the framework until they are writing entire computer games themselves. The framework supports students as well as instructors by permitting a large number of labs, design assignments, and complete game assignments in the Programming Problem sections in almost every chapter. Instructors are also provided with a large number of reading questions, concept questions, and slightly longer writing assignments in each Chapter Review.

The Problem

Interest in computer science has fallen nearly fifty percent in the last six years. This trend has hit computer science departments in small liberal arts colleges particularly hard. Declining interest leads to declining enrollments in introductory courses and in turn to declining numbers of majors. Several explanations for the problem have been offered including that the current crop of students came of age after the Internet bubble burst, that interest during the Internet bubble were artificially inflated, that current college students are jaded with respect to computers because they use them every single day, and that computer science is fundamentally a difficult discipline.

Similar to learning a new human language, learning to program a computer requires a new, foreign way of thinking. Many students find the required literal, logical thinking difficult because it is different. Their struggle is compounded by the need to "know everything, all at once". This refers to the need to learn an unforgiving machine-readable syntax, a technical vocabulary, and the skills to execute a program, all before the first "Hello, World!" program can be run.

Whether computer science is intrinsically difficult or our pedagogy in CS1 only makes it seem so, any other reason for the decline in interest in computer science is compounded by the steep learning curve in introductory computer programming.

The very success of computerization, the ubiquity of cellphones, digital music, educational software, and hand-held and console-based computer games, has commodified the computer at the same time it

raised the bar on what students expect computers to be able to do. Student outcomes are best when students are *engaged* with the material; traditional introductory textbooks (and, unfortunately, traditional professors) seem frozen in time, starting with "Hello, World!" and sales tax calculations. The current generation of college student is not engaged by this sort of programming.

The other possible explanations of declining student interest focus on student perceptions of computer science. Stories now focus on outsourcing of technology jobs rather than on many-figured signing bonuses offered to computer science graduates. This is both a cause and an effect of the decline in student interest.

SCG addresses all three of these problems. It uses the SCANS framework to abstract away some of the cognitive overhead of learning the Java language, permitting students to focus on learning the new concepts first. It leverages many students' interest in computer games by teaching concepts in the context of building different types of computer games. The consistent context gives students a place to plug in their newly learned concepts, further easing the learning curve. Student perceptions are the hardest to address; *SCG* includes examples and discussions that take the student beyond the computer game context to other parts of computer science. The spiral approach is applied on both a micro and a macro scale so some interesting problems in different parts of the field are presented when students can understand them with guidance that answers will only come in future courses. The approach and benefits of *SCG* are covered in more detail in the **Approach of the Text** section below.

The Competition

The three following books seem closest to *SCG* in approach. A synopsis of each is provided along with an analysis of how similar/different they are from my vision of *Simple Computer Games*. They are presented in chronological order which happens to coincide with increasing commonality in approach.

Biermann, A. and Ramm, D. *Great Ideas in Computer Science with Java.* The MIT Press. Cambridge, MA, USA. 2002. 528pp. (16 Chapters, 510pp; Bibliography, 2pp; Index, 14pp).

Based on earlier Scheme editions of the book, Biermann and Ramm's text is very broad in its coverage, ranging from variables and control structures in the second chapter through simulation, computability, program speed, and finally artificial intelligence in the last chapter. This text is geared toward a well-prepared, motivated audience such as that found at Duke University (where the authors teach).

The breadth of coverage encouraged student engagement. It also provides an overview of the history of algorithms and hardware that have led to the digital society that the current crop of students has grown up in. Starting with HTML, objects, and Java applets in the first two chapters, the learning curve in this text is very steep.

SCG attempts to engage students in a similar fashion, providing an interesting context including both historical and societal connections to computers and computer games. By being more focused and providing a minimalist framework, *SCG* is able to provide beginning students with more support as they learn to program and a gentler learning curve. By focusing on computer games, it is able to address interesting problems including simulation and artificial intelligence covered in Biermann and Ramm's book.

Bruce, K. and Danyluk, A. and Murtagh, T. *Java: An Eventful Approach.* Pearson Prentice Hall. Upper Saddle River, NJ, USA. 2006. 675pp (21 Chapters, 642pp; 4 Appendices, 29pp; Index 4pp).

Bruce, addresses the same problems identified above in a similar manner to that used in *SCG*: the *ObjectDraw* toolkit is provided to permit students to create interactive programs and applets from day one. Early versions of *ObjectDraw* were influential on the design of the SCANS toolkit and Bruce's research on how beginning programmers could understand exceptions (and an exceptions-first

pedagogy) provided some of the impetus for this prospectus.

Bruce, introduce variables, types, and selection in the context of programs derived from the base GUI class in *ObjectDraw*. This use of classes leads naturally to a more formal definition of classes and objects; the spiral approach is used similarly in *SCG*.

SCG differs from Bruce's text by providing a single, unified context, computer games, for the presentation of new material. The SCANS framework is different than *ObjectDraw* in that it was designed from the beginning to be light-weight, to be able to get out of the student's way as she develops as a programmer. One of Bruce's appendices (12pp) focuses on programming without *ObjectDraw*; SCANS is systematically removed throughout the text of *SCG*.

Guzdial, M. and Ericson, B. *Introduction to Computing and Programming with Java: A Multimedia Approach.* Pearson Prentice Hall. Upper Saddle River, NJ, USA. 2007. 558pp (16 Chapters, 542pp; 1 Appendix, 8pp; Bibliography, 2pp; Index, 6pp; CD).

Guzdial and Ericson's book addresses the everything-all-at-once problem through the use of the Dr. Java interpretive Java environment. This permits Guzdial to apply his experience with the Squeak dialect of Smalltalk to teaching Java: students are urged to engage by creating programs that manipulate images, sound, and at the end of the book, simple video.

This book is the closest competitor to *SCG* in that student engagement was designed in from the start and considered the most important part of the book. It differs by focusing on an interactive, objects-first pedagogy and by ignoring the medium of video games; *SCG* is about the medium of computer games and while students are provided with a scaffold in the SCANS framework, it is peeled away, layer by layer, as students master more Java.

Approach of the Text

Declining interest in computer science, exacerbated by a disconnect between traditional introductory course content and the role of the computer in our students' lives, causes declining enrollment in introductory courses. Traditional content's failure to engage students convinces many who do enroll that computer science too hard or too dull for them to continue. *Simple Computer Games* leverages many student's interest in computer games to address these problems. Traditional assignments are replaced with writing computer games and the SCANS (Super-Cool Acronym Name Soon) framework permits students to write simple, interactive games from day one.

Two approaches influenced the development of the SCANS framework: *BlueJ*, and Feleisen’s work in layered programming languages (embodied, for one example, in DrScheme and DrJava*).* SCANS similarly hides the **`main`** declaration and exceptions from the user, holding some of the cognitive overhead at bay. Unlike other educational frameworks, however, every effort has been made to keep SCANS both simple and minimal. This permits the entirety of the framework to be exposed by the end of this textbook. In *Simple Computer Games*, the student goes from depending on SCANS to being able to write their own Java programs by the end of the text

The SCANS framework in this proposed text also permits students to start writing interesting games right away. As many have noted, educating the "Net generation" is hard because they see a seamless integration of technology in their lives. Growing up with graphical user interfaces on everything from the computer in their kindergarten to their set-top cable box to their cellphone has raised their expectations on what they should be able to do *with* technology. The SCANS framework supports their development of games right away, to motivate their learning, and then is moved out of the way, piece by piece, so that the students interact directly with the Java technology. This section provides an overview of several features that support beginning computer scientists and their instructors.

Spiral Approach

Simple Computer Games makes good use of the "spiral approach" to teaching complex topics, presenting them in a reduced form early and then revisiting them multiple times, each time revealing a little more of the complexity [Spicer, *A Spiral Approach to Software Engineering Management Education.* International Conference on Software Engineering, 1984.]. Examples include the `String` and `StringBuffer` classes, used in Chapter 3, expanded in detail in Chapter 6, and then used extensively in the text adventure game of Chapter 10. Event-handlers, too, are presented early (as part of the framework) and their complexity is slowly revealed until Chapter 11 when students learn to write their own.

SCG proposes applying this same approach to many advanced topics, introducing concepts that are built upon in future courses. Examples include computer history (discussed along-side the history of games through-out the book), computer organization (in the discussion of threads, the history of arcade games, and the overview of networked communication), the use of design patterns (Factory in Chapter 10), game design fundamentals (through out the book), and game studies. The inclusion of material from the new field of game studies is one unique aspect of this text, making it suitable as a text for introducing game studies students to programming just as it introduces programming students to game studies. *SCG* takes the best parts of early introduction of topics and stretches out the examples to ease the learning curve for introductory students. *SCG* also benefits from focusing on game-based examples. This context gives students somewhere to put what they learn.

Chapter Designs

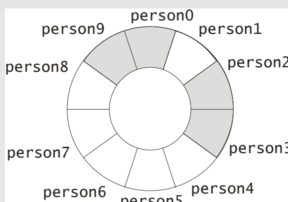
SCG proposes the use of two different chapter designs. One design is focused on presenting new language features in terms of some game. Most chapters begin with the description and then the design of a single game. The game then serves as the motivation for new language features and, in turn, the mental scaffold for holding those new concepts. The following examples, introducing the chapter on arrays, serves as an illustration:

Chapter 8 – Arrays

Pandemic Simulation

Many computer games are simulations, simplified models of the world that can be manipulated for entertainment, education, or research. One reason simulations are useful is that it is possible to simulate things that have not yet happened (or that we hope will never happen). One interesting simulation is of the spread of disease through a population.

Disease simulation may seem deep for a book on computer games but after developing a *very* simple model we will extend it with additional features, features that permit us to "play" with the population as a Public Health Czar and see who should be inoculated against the disease.



Step one is to determine how to model the population. Initially we are interested in the very simplest model that can yield interesting results. If all we model is how many days a person has been sick, then we can represent each person with a single integer variable. Healthy people have been sick zero days; sick people get better after five days.

How do we represent a population of ten people? We will need ten variables. We

could name them **person0, person1, person2, ..., person9**. We can run our simulation in *time steps* which are one day long. Each day we will check the people around any sick person and, if they are not already sick, one of them will become sick. The *neighborhood* of an individual is defined to be the two people with just lower numbers and the two people with just higher numbers; if we consider the group seated around a round table, the following picture shows **person1**'s neighborhood.

Think for a moment how you would write an if statement to pick one of those neighbors based on a random number on the range 0..3. Assuming we had a predicate whether or not a given person was sick, then the following code would permit a sick **person1** to infect one healthy person near them.

```
if (sick(person1)) {
  int whichNeighbor = getRandom(4);
  if ((whichNeighbor == 0) && (!sick(person9)) {
    person9 = 1;
  } else if ((whichNeighbor == 1) && (!sick(person0)) {
    person0 = 1;
  } else if ((whichNeighbor == 2) && (!sick(person2)) {
    person2 = 1;
  } else if ((whichNeighbor == 3) && (!sick(person3)) {
    person3 = 1;
  }
}
```

One if statement like this is required for *each* person. Copying, pasting, and modifying ten versions is daunting. Moving up to fifty, one hundred, or one thousand persons is not possible with this approach. The approach does not *scale* or remain effective as the size of the problem increases. Earlier, when cutting, pasting, and modifying proved ineffective, we learned to think, "Loop!" That is just what we could do if we had a way to use a loop control variable to determine which variable we're working with.

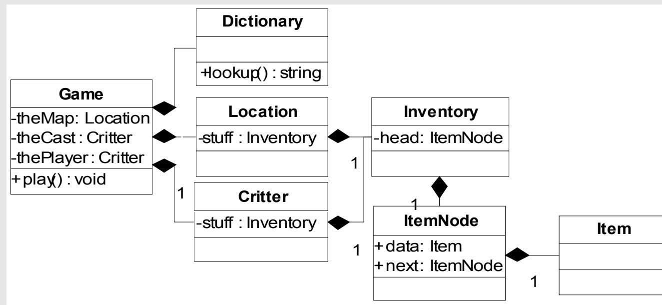
Java (and many other programming languages) has an *array* construct, a container for declaring and holding a collection of elements all of the exact same type. The elements in the array can then be addressed individually using integer indices. The advantages of an array include declaring all the variables we need in one line, being able to pass all the elements at once to member functions, and being able to address each element individually inside a count-controlled loop.

The remainder of Chapter 8 uses this pandemic simulation to develop the idea of an array (and modular arithmetic), starting with the array of integers and moving on to create a Person class that encapsulates the age, health, mobility, and immunization history of a person, permitting the player of the game to determine where to use a limited supply of vaccine and see the results.

The other chapter design focuses on presenting a new game in terms of language features. Each chapter titled **Making Games** begins with an overview of some genre of computer game: text adventure, arcade, side-scrolling platformer, or turn-based multi-player game. The presentation includes a history of the genre, what sorts of computers were available to play it, and some of the methods used to study games in the genre. The architecture of games of the genre is then examined, showing how design patterns apply at the high level. An example game is designed and implemented,

bringing together the language concepts presented in book up to that point. These cumulative chapters appear in the second half of the book. The following section from the chapter on Text Adventure Games illustrates the level of detail in the **Making Games** chapters:

Section 10.3 Text-Adventure Game System Architecture



The text-adventure game system is large and complex. It is composed of a group of interacting class objects. The relationship between the classes is shown in the diagram to the left.

The Game class maintains the current state of the game. It contains the command Dictionary, the map of all Locations in the game world, and the cast of all the Critters in the game world including the player. A Game object processes all input from the user and generates all messages to the user. The main *Game loop* is inside the Game object.

The Game class is special: there should be one and only one object of this type in any text adventure game. Consider our old standby, checkers. The board and pieces, together, record the current state of the game, all captures, kings, occupied, and free squares. There should only be one checkerboard per game of checkers. Similarly, there should be only one Game object. In computer science, a class designed to have only one object ever created is called a *singleton* (there can be only one); the Game class is an example of the singleton design pattern.

The Dictionary class translates the commands the user types into internal versions. It permits synonyms and abbreviations to be handled in a data file rather than by hard-coding cumbersome Boolean expressions in the code. The Dictionary can translate "N" to "NORTH", for example, permitting the user to navigate with less typing.

As designed, the Game class contains only a single Dictionary. Dictionary is not a singleton, however. A Game might make use of different dictionaries if different words within a command came from different vocabularies: commands of the form "verb noun" might translate each word from a different dictionary so that similar abbreviations could be unambiguously expanded. As designed, any number of Dictionary objects could be created.

The Location class encapsulates all of the attributes of one location in the game world. The Game keeps an array of Location. The Location themselves are "threaded" together through their ID numbers; a Location knows the ID of the Location to the north of it. The separation of index (in the array) and ID makes the Location data file easier to maintain. Locations contain an Inventory, a collection of Items. These are Items loose in the Location with which the player can interact.

The Critter class tracks everything there is to know about a "living" thing in the game world: identifier, name, description, amount of "life force" (hit points). One of those living things is the player. Critters keep track of their current Location (by ID) as well as an Inventory of the Items they have.

The Item class encapsulates everything about an Item in the game world: identifier, name, description, attribute modifiers (how does having this item impact the Critter that has it?). An Item is always found in an Inventory, a container belonging to a Critter or a Location.

Sample Chapters and Art

The first four chapters of the book, complete with art and sample code (but without chapter exercises) are in another file included with this one. The sample programs give the feel of SCANS and starting at the beginning of the book gives a feel for how the Java learning curve is supported by abstraction. It also shows how parallels between game design and program design are exploited, giving students multiple views of the same design criteria.

Ancillary Suggestions

This text will need a CD with the SCANS framework. Since the capacity of a CD is much larger than the framework, the CD can also include a copy of the JDK and an editor. Eclipse is a candidate for the editor though a simplified, learning UI might be helpful (I have not yet designed one, nor have I read about one but there might well be one in the works somewhere in academic computer science). The benefit of Eclipse is that it is open source, it is in Java, and it is modifiable.

This text should also be supported by a website. The website will include pointers to updated versions of SCANS, Java, and Eclipse. It will also include additional self-analysis questions (automated) for students to help check their knowledge. An instructor-only section of the website can include additional questions (with keys) for the various chapters; this question bank will be stocked with parameterized questions which are randomly and automatically generated. This will make the exposure of any given key value on the web by an instructor much less of a problem for others using *SCG*.

Slides for lecture, book diagrams, and similar classroom support material could also be provided. I, personally, do not find another person's lecture notes helpful in preparing my own lectures but I would be happy to develop and share slides.

Tentative Contents and Organization

1 Getting Started
  1.1 What is a *game*?
    1.1.1 *Components*: What the Game is Played With
    1.1.2 *Attributes*: Store Component State
    1.1.3 *Rules*: Governing How Components Interact
    1.1.4 *Rules-follower*: How Rules are Enacted
  1.2 What is a *strategy*?
    1.2.1 *Context*: The Game to Which it Applies
    1.2.2 *Rules*: How the Game is to be Played
    1.2.3 *Goal*: Strategies Win Games
  1.3 What is a *computer program*?
    1.3.1 Components: Virtual Objects in the Computer's Memory
    1.3.2 Rules: Instructions for Manipulating Objects
    1.3.3 Rules-follower: One or More
  1.4 What is a *computer game?*
  1.5 Summary
  1.6 Chapter Review Problems
2 Your First Game
  2.1 Programming in Java
    2.1.1 Compiling
    2.1.2 Interpreting
    2.1.3 Running
  2.2 Installing and Testing the Framework
    2.2.1 Copying the Framework
    2.2.2 Copying a Sample Program
    2.2.3 Debugging Setup Errors
  2.3 Why Use a Framework?
    2.3.1 Large + Felxible = Complex
      2.3.1.1 Games Come with Tutorials
      2.3.1.2 Avoid "Everything, all at once"
    2.3.2 Framework Mediates Complexity
  2.4 Drawing a Picture
    2.4.1 Local Variables
    2.4.2 Calling Member Functions
  2.5 Interaction
    2.5.1 Member Variables
    2.5.2 The on* Member Functions
  2.6 First Game: Peg in the Hole
  2.7 Summary
  2.8 Chapter Review Problems
  2.9 Programming Problems
3 Components: Names and Types
  3.1 Components in Computer Programs
    3.1.1 A Component has a *Type*
    3.1.2 A Component Lives in *Memory*
  3.2 Names

3.2.1 Names have Types
3.2.2 Declaring Names
3.2.3 Assigning to Names
3.3 Java Components
3.3.1 Plain-old Data: Primitive Types
3.3.2 Objects
3.4 Naming Conventions
3.5 Using JavaDoc Documentation
3.6 Summary
3.7 Chapter Review Problems
3.8 Programming Problems
4 Attributes: Expressions
4.1 Manipulating Components
4.2 Working with Numbers
4.2.1 Operators
4.2.2 Methods
4.2.3 Special Numeric Methods
4.2.3.1 Random Numbers
4.2.3.2 Trigonometry
4.2.3.3 Time
4.3 Working with Other Types
4.4 Summary
4.5 Chapter Review Problems
4.6 Programming Problems
5 Rules: Manipulating Components
5.1 Selection: Making Choices
5.1.1 The if Statement
5.1.2 The if-else Statement
5.2 Iteration: Repetition
5.2.1 The while Statement
5.2.1.1 Count-controlled loops
5.2.1.2 Sentinel-controlled loops
5.2.2 The for Statement
5.3 Crash: Collision Detection Basics
5.4 How a Java Program Starts
5.4.1 The main method
5.4.2 The arguments
5.5 Summary
5.6 Chapter Review Problems
5.7 Programming Problems
6 Reading and Writing I: String Handling
6.1 Reading User Input
6.2 Breaking up a String
6.3 Word Games: Ig-pay Atin-lay
6.4 Summary
6.5 Chapter Review Problems
6.6 Programming Problems
7 Components Meet Rules: Classes and Objects
(Car Racing (Animated in SCANS))

7.1 Our Own Types
7.1.1 Defining a Class
7.1.2 Instances of a Class
7.1.3 Example: Racing Two Cars
7.2 Summary
7.3 Chapter Review Problems
7.4 Programming Problems
8 Collections: Arrays
(Single Dimension Epidemic Simulator)
8.1 One and Many
8.1.1 Declaring an Array
8.1.2 Filling an Array
8.1.3 Using an Array
8.2 Arrays and Classes
8.2.1 Beyond POD Arrays
8.2.2 Extending Epidemic
8.2.2.1 Immunity
8.2.2.2 Death
8.2.3 Arrays are Objects
8.3 Summary
8.4 Chapter Review Problems
8.5 Programming Problems
9 Reading and Writing II: Text Streams
(Learning 20 Questions Game)
9.1 Reading and Writing Information
9.1.1 Human-readable Data
9.1.2 Exceptions
9.1.3 Opening a File
9.1.4 Closing a File
9.2 Reading a Text File
9.2.1 Strings
9.2.2 Converting Strings
9.2.3 Reading "Serialized" Data from Text Files
9.2.3.1 Multiple Lines per Object
9.2.3.2 Construction
9.2.3.3 Java Serialization
9.3 Writing a Text File
9.3.1 POD and Strings
9.3.2 "Serializing" as Text
9.4 Modding: Separating Code and Data
9.5 Summary
9.6 Chapter Review Problems
9.7 Programming Problems
10 Making Games: Text Adventure Games
10.1 Back to the Future: Interactive Fiction
10.1.1 Brief History of Text Adventure Games
10.1.2 Introduction to Interactive Fiction and the IFA
10.2 The *Game* Loop
Display Game State

Get User Input
Update Game State
10.3 Building a System: Software Architecture
10.3.1 Location
10.3.2 Critter
10.3.3 Item
10.4 Implementing a Text Adventure Game System
10.4.1 An Incremental Approach
10.4.2 Making Objects: Using a Factory
10.4.3 Moving Around: Linking Locations
10.5 Extending the Text Adventure Game System
10.5.1 Keys
10.5.2 Mini-games
10.5.3 Combat
10.5.4 Conversations
10.6 Shining Examples of Text Adventure Games
10.6.1 Adventure
10.6.2 Trinity
10.7 Summary
10.8 Chapter Review Problems
10.9 Programming Problems
11 Reaction: Event-driven Programming
11.1 What do the on* Methods Do?
11.1.1 Handling an Event
11.1.2 An Extra Rules-follower
11.2 Java Event Model
11.3 Summary
11.4 Chapter Review Problems
11.5 Programming Problems
12 Animation: Concurrent Programming
12.1 Rules-followers: Threads
12.2 Java Thread Model
12.3 The Animation Loop
Draw
Pause
Undraw
12.4 Java Drawing Model
12.5 Other Uses for Threads
12.6 Summary
12.7 Chapter Review Problems
12.8 Programming Problems
13 Making Games: Arcade Games
13.1 A Brief History of Arcade Games
13.2 The *Game* Loop Redux
13.3 Frame-rate Independent Timing
13.4 Summary
13.5 Chapter Review Problems
13.6 Programming Problems
14 Interaction: Graphical User Interfaces

- 14.1 Swing: A Widget Toolkit
- 14.2 Simple Interfaces
- 14.3 Layout Managers
- 14.4 Enhanced Animation Stuff
- 14.5 Summary
- 14.6 Chapter Review Problems
- 14.7 Programming Problems

15 Artificial Intelligence: Computer Opponents
- 15.1 The Turing Test in Games
- 15.2 Summary
- 15.3 Chapter Review Problems
- 15.4 Programming Problems

16 Game Boards : Multi-dimensional Arrays
- 16.1 Tic-Tac-Toe
- 16.2 Checkers
  - 16.2.1 Tile-based Graphics
  - 16.2.2 Unrolling the *Game* Loop for Two
- 16.3 Getting Here from There: Mazes
- 16.4 Summary
- 16.5 Chapter Review Problems
- 16.6 Programming Problems

17 Over and Over: Recursion
- 17.1 Circular Definitions: Palindromes
  - 17.1.1 Is a String a Palindrome?
- 17.2 Path Finding
  - 17.2.1 *Solving* a Maze
  - 17.2.2 Paths in Text Adventure Games
- 17.3 State Trees: Analyzing Board Position
  - 17.3.1 Recursion and AI
- 17.4 Summary
- 17.5 Chapter Review Problems
- 17.6 Programming Problems

18 <u>Making Games:</u> Side-scrolling Platform Games
- 18.1 Tile-based Graphics
- 18.2 A Window on the World
- 18.3 Levels, Winning, Mods
- 18.4 Summary
- 18.5 Chapter Review Problems
- 18.6 Programming Problems

19 Over the Shoulder: Isometric Tile Graphics
- 19.1 Image Transformations
- 19.2 "Third-person" Perspective
- 19.3 Summary
- 19.4 Chapter Review Problems
- 19.5 Programming Problems

20 Network Programming
- 20.1 Sockets: The Program View of Other Computers
- 20.2 Taking Turns: Avoiding Gridlock
- 20.3 Sharing: Where is the "Game"?

- 20.4 Tic-Tac-Toe Again
  - 20.4.1 The *Game* Loop
  - 20.4.2 Alternating Control
  - 20.4.3 What if something bad happens?
- 20.5 Summary
- 20.6 Chapter Review Problems
- 20.7 Programming Problems

21 Making Games: Multi-player Turn-based Games
- 21.1 General History of Multi-player Games
  - 21.1.1 MUD/MOO
  - 21.1.2 Twitch Games
  - 21.1.3 Strategy Games
  - 21.1.4 Gambling
- 21.2 Texas Hold-em Poker
- 21.3 Client-Server Architecture
- 21.4 Summary
- 21.5 Chapter Review Problems
- 21.6 Programming Problems

Appendices

- A .Installing and Using Eclipse and the JDK
  - A.1 Downloading the Current Version
  - A.2 Installing the Software
    - A.2.1 Installing the JDK
    - A.2.2 Installing Eclipse
  - A.3 Making Eclipse SCANS-aware
- B .Java Syntax
- C .SCANS Framework Architecture