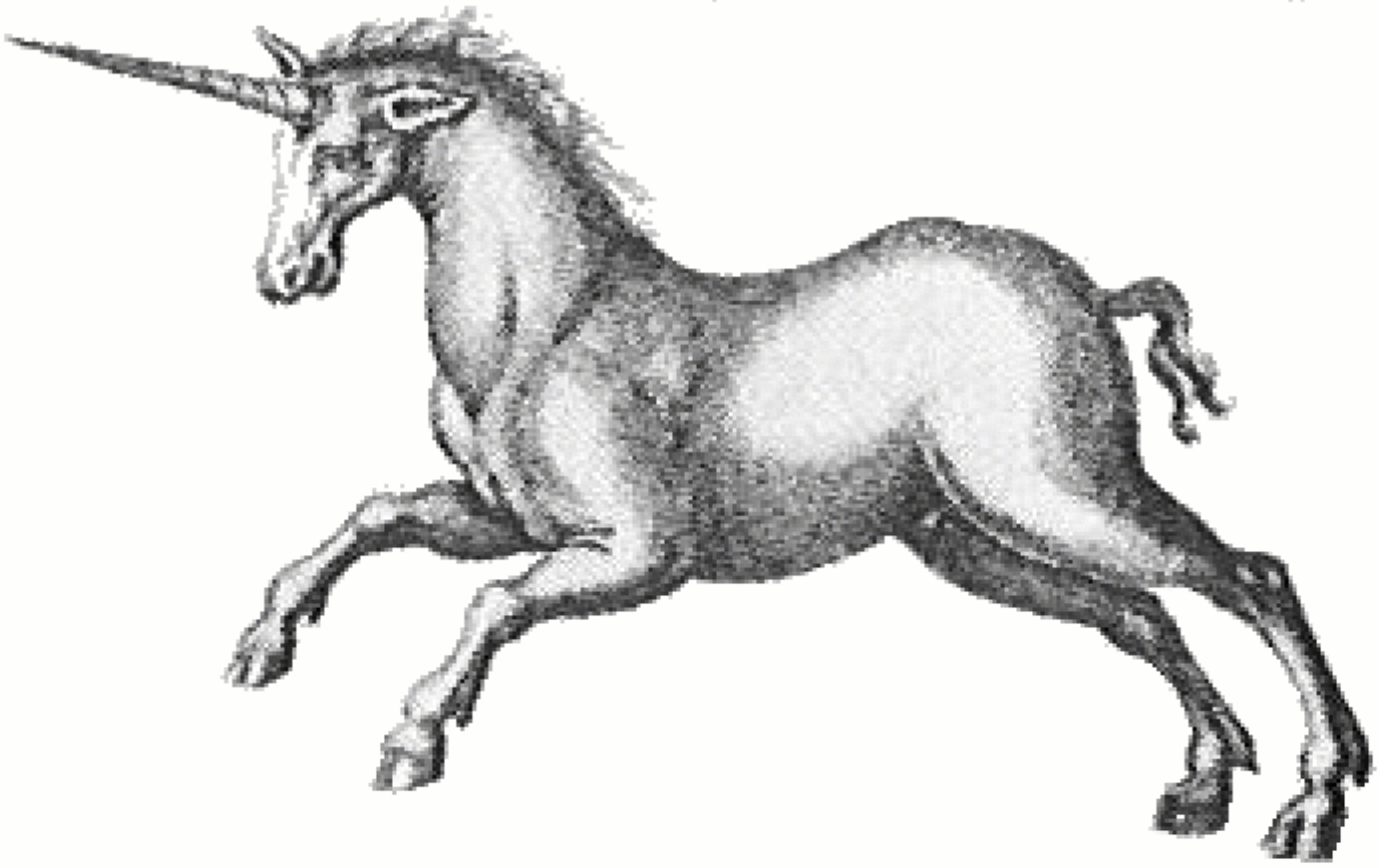

Manual of Puissant Skill at Game Programming

Clinton Jeffery

Manual of Puissant Skill at Game Programming

by Clinton Jeffery

Portions adapted from "Programming with Unicon", http://unicon.org

Dedication

This book is dedicated to Curtis and Cary and future programmers everywhere.

This is a draft manuscript dated 1/6/2015. Send comments and errata to jeffery@cs.uidaho.edu.

This document was prepared using LibreOffice 4.2.

Contents

Preface

This book will teach you game programming using a computer language called Unicon. Writing computer games may be of interest to both computer professionals and hobbyists. Unicon is an excellent language for writing simple games and for rapidly developing prototypes of complex games. Unicon will probably never be the best language for the cutting-edge video games or specialized game hardware, but for many kinds of games it is ideal.

Clinton Jeffery

Introduction

The definition of a *game* used in this book is our own composition, consciously generalizing typical dictionary definitions of the term.

> A competitive or cooperative activity involving skill, chance, or endurance, whose primary goals are amusement, improvement, or both.

There are fun games, and serious games, and every once in awhile you run into a game that is seriously fun.

To use this book, you need to install Unicon from the Internet at http://unicon.org. Do that, and then you can go on to Chapter 1.

Chapter 1: Preliminaries

Every Unicon program starts like this:

```
procedure main()
```

In practice this may be anywhere in a source file, but execution starts with main(). A whole program that does nothing would look like this:

```
procedure main()

end
```

To write something on the screen, you use the write instruction:

```
write( 12 )
write( “Hi folks!”)
```

To run a program you must save the code in a file, and translate it into a machine language that the computer can run. Unicon includes a simple IDE called Ui; more powerful IDE's are available. Fire up Ui (or Wi) by typing “ui” at a command line or launching the menu item labeled "Windows Unicon" and type:

```
procedure main()
   write("Hello, amigo!")
end
```

Your screen should look something like Figure 1-1. The top of the window is where you type your program. The bottom is where the computer tells you what it is doing.

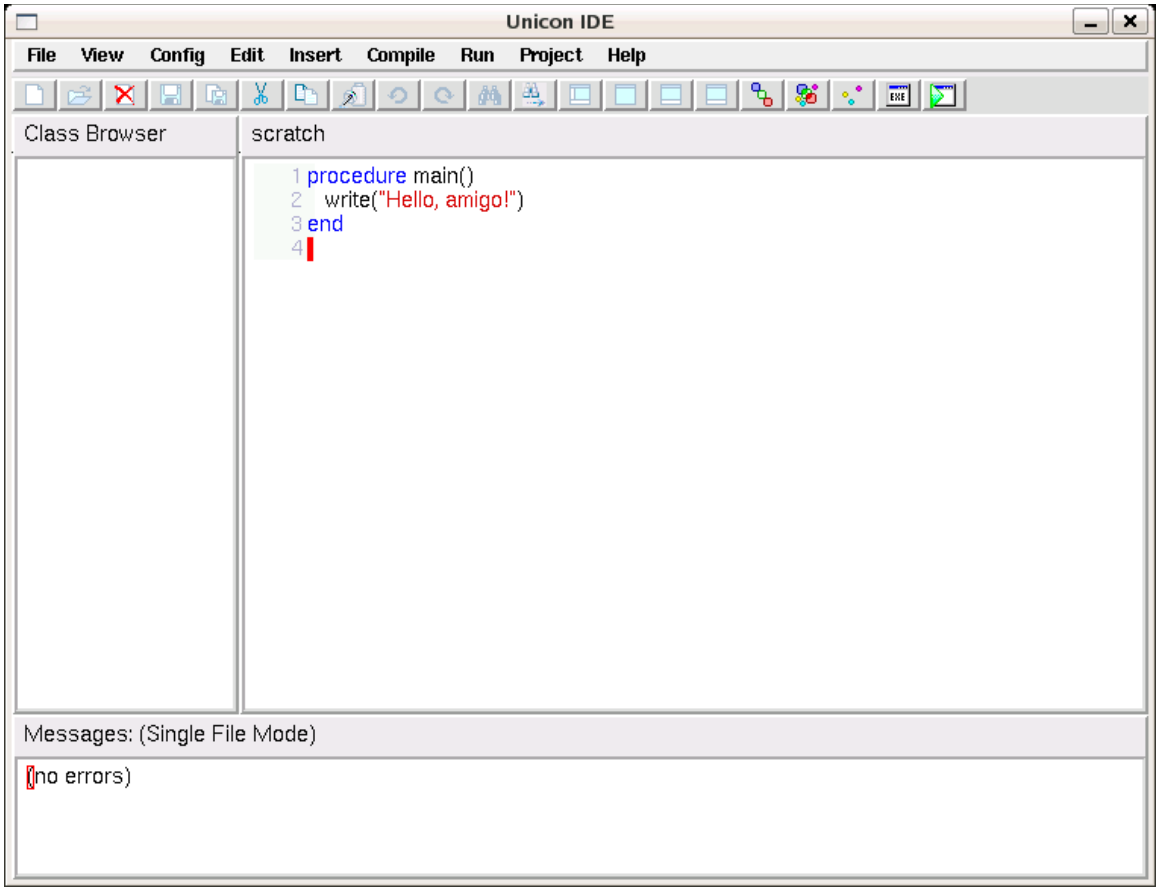

Figure 1-1:
Writing an Unicon program using the Ui program.

If you select **Run->Run**, Ui will do the following things for you.

1. Save the program in a file on disk. Icon and Unicon programs end in .icn.
2. Translate the Unicon program to machine language.
3. Execute the program. This is the main purpose of the **Run** command.

If you type the hello.icn file correctly, the computer should chug and grind its teeth for awhile, and

Hello, amigo!

should appear in a window on your screen.

Variables

Unicon's local variables do not have to be declared. To assign a value to a variable, use a colon (:) followed by equals (=).

```
procedure main()
   answer := 7 + 5
   write(answer)
end
```

Reading from the keyboard

A program can read from the keyboard in Unicon using a function called read(). This function takes what you type on one line and puts it in the program.

```
procedure main()
   write(“What is seven plus five?”)
   answer := read()
   write(“You say it is “, answer, “ and I say it is “, 7+5)
end
```

Run the program to see what it does. The write() instruction is happy to print out more than one value on a line, separated by commas. The write() instruction must have all of its parameters before it can go about its business.

Random Thoughts

To ask the computer to flip a coin or roll a die, just put a question mark in front of a value, and the computer will choose something randomly out of it. For example,

```
   write("die roll: ", ?6)
```

picks a random number between 1 and 6. To flip a coin:

```
   write("coin toss: ", ?["heads", "tails"])
```

The square brackets [ ] here are making a list of two words, and the random operator is choosing one or the other.

Deciding what to do next

There are a number of typical conditional expressions in Unicon, such as if-then-else. For example, you can write

```
if ?6 + ?6 = 2 then write("snake eyes") else write("no dice")
```

Later chapters will show you other ways to decide what instruction to do next, such as case expressions.

Repeating Yourself

Unicon has many ways to repeat an instruction. For example,

```
every 1 to 3 do write("crazy")
```

does the same write() instruction three times. Later chapters will show you other ways to write iterations, such as while loops.

Figure 1-2:
The first rule of random selection is: No peeking.

Chapter 2: Guessing Games

Some of the easiest games are guessing games. Have you guessed which games this chapter will show you? First, a word-unscrambler, then a classic called hangman, and finally, a “serious game”: the thespian's little helper.

Scrambler

In this program, the computer takes a word, scrambles it, and makes the user guess which word it was. Consider the following list of words:

```
words := [“fish”, “beagle”, “minotaur”, “tiger”, “baseball”]
```

The previous chapter showed how to select randomly from a list with the ? operator. The code ?words would pick one of these words. The program can store that choice like this:

```
word := ?words
```

We want to scramble a copy of this word, but we want to remember the original word, so we make a copy to scramble:

```
scramble := word
```

A program can look at or modify individual letters within a word by *subscripting* the word with a position. A subscript is when you pick an element out by following the word with a position in square brackets: [ ]. For example, if the word is “fish”, word[1] is “f”, word[2] is “i”, word[3] is “s” and word[4] is “h”. Inside the computer, the figure below is what the memory for words looks like. The list called words can be subscripted to pick a specific word, and the word can be subscripted to pick a letter. words[2][1] is “b”. You could also write this as words[2,1].

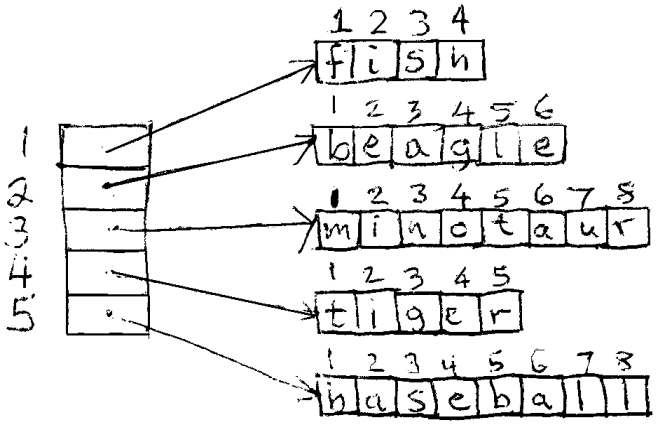

Figure 2-1:
How a list of strings really looks in memory.

The total number of letters in the word is given by *word. The asterisk * is an *operator* with two different meanings. When the asterisk is in between two numbers it multiplies them, but when it has nothing in front of it to multiply, it tells the size of the thing that comes after it.

If ?6 was a random number between 1 and 6, like rolling a dice, then ?*word is a random number between 1 and the number of letters in word, and word[?*word] is a random letter from a word! Now for a crazy jumble of a word you might say

```
every 1 to 99 do
    scramble[?*scramble] :=: scramble[?*scramble]
```

The operator :=: exchanges the thing on its left with the thing on its right, in this case two different random letters. The first line says to do the command on the second line 99 times. The second line "swaps" two characters in the word at random positions. Doing that 99 times will pretty well jumble most ordinary words.

Notice that this "every" command was spread across two lines. We have not done that much so far, but it is usually OK so long as the first line ends with a word that is obviously "unfinished" and needs more after it.

The whole program is

```
procedure main()
   words := ["fish", "beagle", "minotaur", “tiger”, “baseball”]
   word := ?words
   scramble := word
   every 1 to 99 do
      scramble[?*scramble] :=: scramble[?*scramble]
   write("Unscramble: ", scramble)
   answer := read()
   if answer == word then
      write("correct!")
   else write("no, it was ", word)
end
```

The comparison operator == might look strange. In Unicon one equals sign = compares numbers; two equals is similar, == compares words to see if they are the same.

Hangman

Hangman is a classic letter-guessing game. First, the computer picks a word, like in the scramble program.

```
word := ?[“beagle”, “tiger”]
```

You probably want more than just two words here, and probably need to randomize your random numbers as discussed in the previous section. Anyhow, instead of scrambling the letters, hangman makes a word consisting of blanks, the same length as the word chosen.

```
blanks := repl(“-”, *word)
```

The repl(s, number) instruction builds a word consisting of a number of copies of s, one after another.

In hangman each time you miss, more of the person being hanged gets drawn. The program needs to count how many misses the player makes. The program starts with 0 misses and ends if the player misses 5 guesses.

```
misses := 0
```

Here is what the hanged person looks like in text form. The backslash character \ is special and it takes two backslashes in a row to print one out.

```
  hangedperson := [
              "  o  ",
              "\\ | /",
              " \\|/ ",
              " / \ ",
              "/   \\"
  ]
```

In many games, play lasts not some fixed number of times, but until there is a winner or a loser. The repeat instruction takes a list of instructions and does them over and over forever until the computer runs out of electricity...or in this case, until the stop() instruction ends the program!

```
  repeat {
```

To write out part of the hanged person, we are using the every instruction we saw before, except we are using the count of how many times we are repeating ourself, to pick out a different element of hangedperson each time.

```
    every write(hangedperson[1 to misses])
```

The program is finished if the player has missed 5 guesses. If not, for each turn we write out the player's partly filled-in blanks and read a new letter from the player.

```
    if misses = 5 then stop("you lose! The word was ", word)
    write(blanks)
    letter := read()
```

The program looks for that letter in the word, and if it is there, then it fills in the blanks with the correct letter from the word. There is an instruction named find(s1, s2) that looks for s1 inside s2 and returns each place that it finds s1.

```
    if find(letter, word) then
      every i := find(letter, word) do
        blanks[i] := word[i]
    else misses := misses + 1
    if not find("-", blanks) then
      stop("You win! The word was ", word)
    }
end
```

Graphics

It will be more fun if our games use graphics, or pictures, instead of just letters and digits. Drawing pictures on the computer is just as easy as writing words. The following example draws a simple picture of a die. Opening a window is similar to opening a file:

```
w := open(“hangman”, ”g”, “size=400,400”)
```

This gives you a rectangle on the screen within which you might draw a picture of your hanged man. In the hangman program, we would need to open this file sometime after procedure main() and before we start the “repeat” instruction that plays the game.

Dots on computer monitors are called *pixels*, and are numbered using (x,y) coordinates, starting at (0,0) in the upper-left corner. The x coordinate gives the pixel column, and y gives the pixel row. The function

```
DrawCircle(w, 300, 50, 25)
```

draws a circle centered at dot (300,50), with a radius of 25 pixels. This might be a reasonable “head” for a stick-figure person for the hangman game. Drawing the body and arms might look like:

```
DrawLine(w, 300,75, 300,150)
DrawLine(w, 300,75, 250,125)
DrawLine(w, 300,75, 350,125)
DrawLine(w, 300,150, 250,200)
DrawLine(w, 300,150, 350,200)
```

We don't want to draw the whole body all at once like this, we want to draw more each time the “misses” value increases. To do this in the actual hangman program, replace the old way of drawing the body

```
every write(hangedperson[1 to misses])
```

with the following:

```
if misses = 1 then DrawCircle(w, 300, 50, 25)
if misses = 2 then DrawLine(w, 300,75, 300,150)
if misses = 3 then DrawLine(w, 300,75, 250,125)
if misses = 4 then DrawLine(w, 300,75, 350,125)
if misses = 5 then {
```

```
DrawLine(w, 300,150, 250,200)
DrawLine(w, 300,150, 350,200)
}
```

A losing game should end up with a window that looks something like the following:

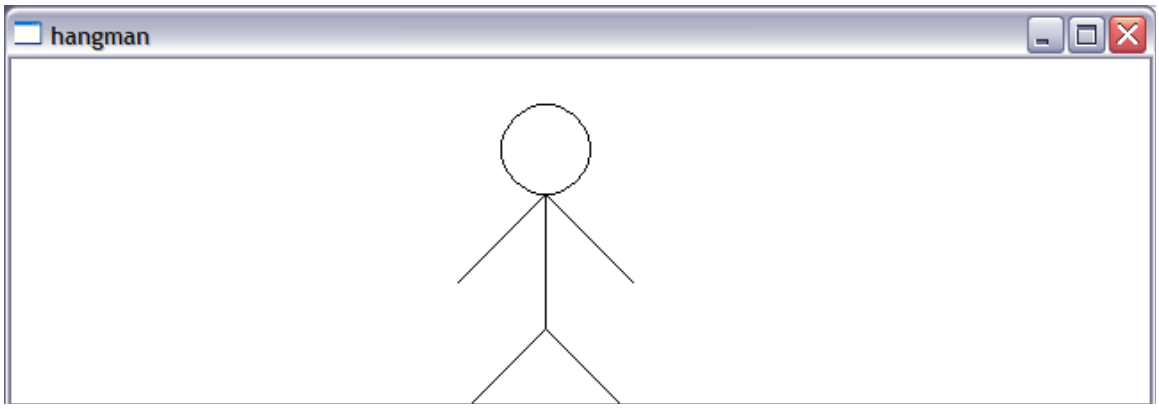

Figure 2-2:
A simple stick figure for a graphical version of the hangman game.

Thespian's Little Helper

Here is a more advanced guessing game of sorts: a “serious game” for folks that are memorizing passages, such as actors. A thespian is an actor, and this program, the *thespian's little helper*, was suggested by a young actress named Rebeca Rond, who wanted a program to help a cast learn their lines. The thespian's little helper, or tlh.icn, is a memory game, and could be used for any recitation based exercise, from plays to scripture memorization to a geography bee. It is a “serious game”, where the hope for this guessing game is that eventually you win by no longer needing to guess.

The program reads the work you are to memorize, which is taken to be a long play, in which the user is practicing one part in one scene. It “reads” the scene to you, slowly printing the text on the screen until it comes to your lines, which you are to recite by typing them.

Play Files and the Gutenberg Repository

The text file format required for the plays is that found in the plays of William Shakespeare in the plain text files available via Project Gutenberg (www.gutenberg.org). A rough description would be: each scene starts with a line of the form ACT n SCENE m, and each actor's lines start with the actor's name in all uppercase.

Reading in a Play

The main procedure of tlh.icn starts with command line argument processing, followed by reading the play into a list of strings, and selecting the scene for that play:

```
procedure main(arg)
   write("Welcome to the Thespian's Little Helper.")
   if *arg ~= 4 then stop("usage: tlh play act scene role")
   if arg[1] == "-help" then
      stop("usage: tlh play act scene role\n",
           "Play abbreviations:\n",
           "   AWEW for All's Well that Ends Well\n",
           "   AC for Antony and Cleopatra, etc.")
   act := arg[2]
   scene := arg[3]
   role := map(arg[4],&lcase,&ucase)
   L := selectscene(readin(arg[1]), act, scene)
```

Giving the User their Queues

The main procedure continues with string processing code that looks for the actor's line(s). This is a loop that reads lines, and by default writes them out to the screen (reading the play to queue the user for their lines). The loop is structured similar to the following:

```
   # for every line in the play
   i := 1
   while i <= *L do {
      s := L[i]
      ... if the line is for the speaker then {
         ... test them on it
         }
      else {
         write(s)
```

```
      delay(500) # half-second, let user read it
      }
   i +:= 1
   }
```

Testing the Responses: How Perfect Must it Be?

The code for testing the speaker on his lines is a bit tricky because it must grab multiple lines (when the actor has a lot to say), and must handle misspellings and punctuation errors which normally would not be considered an error on the part of the actor. First test whether the line is in fact for the speaker. trim(s, ' ',1) skips over any leading whitespace in string s, while match(role, ...) succeeds if the line starts with the character's role.

```
   # if the line is for the speaker
   if match(role, trim(s,' ',1)) then {
```

To grab all the speaker's lines, look for the next line that starts with some all capitals followed by a colon or period – the start of the *next* speaker's lines. This brings up a whole new control structure that is very important in Unicon: string scanning.

A string scanning control structure is an expression *s* ? *expr* in which *expr* performs a scan on string *s*. In this case, the string is tested to see if it starts with upper-case letters followed by a colon or period, indicating a new speaker.

```
   j := i+1
   #
   # grab all the speaker's lines
   #
   while not (trim(L[j], ' \t',1) ? tab(many(&ucase)) &
                    tab(any(':.'))) do {
      L[i] ||:= "\n " || L[j]
      s := L[i]
      delete(L, j)
      }
```

To test the speaker, we must read their input:

```
   #
   # Test the speaker's version
   #
```

```
      write(role,": ")
      line := read()
```

The comparison code is tricky because it should really be an approximate string comparison, and the approximation might well need to use a phone-company-style soundex algorithm for homonyms and similar-sounding words. For now, we just skip the punctuation differences:

```
      if (role || " " || supertrim(line)) == supertrim(s) then {
        i +:= 1
        next
        }
      else {
        write("No, you said --> ",
              image((role || " " || supertrim(line,'\n,.!?'))),
              "\nThe line is ---> ", image(supertrim(s, '\n,.!?')),
              "\n<--")
        }
      }
```

We have now covered the main procedure, but this program has a few vital *helper procedures*. A program can consist of as many procedures as you like, each one starting with the word procedure and ending with the word end, like procedure main() does. Procedure supertrim(s, c) removes characters from c found in s, and removes any leading or trailing spaces or tabs.

```
procedure supertrim(s, c : ',.?!')
   s := trim(s,' \t',0)
   while s[upto(c,s)] := ""
   return s
end
```

Procedure playurl(s) returns an URL for the play named s if one is known. It is initialized with knowledge of a few (15, so far) of Shakespeare's more famous plays, available from Project Gutenberg:

```
procedure playurl(s)
  static t
  initial {
    t := table()
```

```
      t["All's Well that Ends Well"] := t["AWEW"] :=
         "http://www.gutenberg.org/dirs/etext97/1ws3010.txt"
      ...
      }
  return \ (t[s])
end
```

Procedure readin(s) opens an URL if one is known, or a local file, and reads the play into a list of strings.

```
procedure readin(s)
local f, L := []
   if not (f := (open(playurl(s),"m") | open(s))) then
      stop("usage: tlh play act scene role")
   while line := read(f) do put(L, line)
   write("Read ", *L, " lines.")
   return L
end
```

Finally, procedure selectscene(L, act, scene) reduces the play to the desired scene, discarding scenes that come before and after it.

```
procedure selectscene(L, act, scene)
local i := 1, as
   as := "ACT " || act || ". SCENE " || scene || "."
   while L[1] ~== as do pop(L)
   if *L = 0 then stop("didn't find ", as)
   write(pop(L))
   write(pop(L))
   i := 1
   while i < *L do {
      if match("ACT", L[i]) then {
         L := L[1:i]
         break
         }
      i +:= 1
      }
   return L
end
```

Exercises

1. The hangman program needs to tell the human what it is doing and what the human is supposed to do at each step. Add prompts to explain this.
2. Add eyes or other details such as hands or feet to the picture.

Chapter 3: Dice Games

Dice games are easy and fun. This chapter presents a public domain poker dice game called Yatzy that is popular in Scandinavia. It is similar to the trademarked Hasbro game Yahtzee and numerous close relatives. Along the way, it is time to learn some more programming concepts. Suppose you roll 5 dice:

```
dice := [ ?6, ?6, ?6, ?6, ?6 ]
```

What kinds of things should a dice game program do? Roll the dice, and tell the user what the dice say. Let the user pick which dice to keep. Reroll the dice. Let the user decide how to score the turn. For each of these many tasks, the program might want to define a new instruction to perform that task. For now, hold that thought, and just see if the following works:

```
procedure main()
    # roll dice
    dice := [ ?6, ?6, ?6, ?6, ?6 ]
    write("The dice are: ",
        dice[1], dice[2], dice[3], dice[4], dice[5])

    # select keepers
    write("Which dice do you keep?")
    keep := read()

    # OK, ready to reroll the dice?
end
```

I have stopped here, because there is a basic question. When you run this program, what do the input and output look like, and what do they mean? The output looks like:

```
The dice are: 13216
Which dice do you keep?
```

OK, so each digit is one of the dice. How is the player supposed to answer the question: which dice do you keep? Suppose the player wants to keep the 1's and try for more 1's. One possible way to say this is to type an answer of

```
11
```

meaning that the two ones are to be kept. Another possible answer would be to type

14

meaning that the first and fourth dice are to be kept. Neither humans nor computers will know, unless you decide which way the answer should be specified, and explain it clearly. Let us use the first interpretation: the user actually types the dice values they wish to keep. Now we can write computer code to handle the reroll. Continue your main() procedure like so

```
write("Which dice values do you keep?")
keep := read()

# Reroll the dice
every i := 1 to 5 do {
        if not find(dice[i], keep) then dice[i] := ?6
        else keep[find(dice[i], keep)] := ""
        }
write("The dice are: ",
        dice[1], dice[2], dice[3], dice[4], dice[5])
```

This example shows a fundamental way to use an every instruction: to make a name (in this case i) hold each value in a sequence (in this case, 1 2 3 4 5). For each of these values, the instructions in the loop body (the stuff inside the do { ... }) are executed.

For Yatzy, the rules allow the player to select their dice and reroll a third time. Instead of copying the code, you can use every 1 to 2 do { ... } around this whole block of code, all the way from "which dice values do you keep" down to "the dice are...". See if you can figure out how to do that.

Scoring

The turn, in Yatzy ends with whatever dice are the player's after the third roll. First, be lazy and sort your dice so all the numbers that are equal are next to each other.

```
dice := sort(dice)
```

Now, what are the possibilities? In Yatzy the user applies the dice to one of 15 different categories each turn, ending after 15 turns with a score for each category filled in. The scores are kept in a list of 15 elements that are initially null and are filled in with integer point scores as the turns progress.

```
score := list(15)
```

The 15 categories are named after the faces on the dice, and after patterns in the dice that resemble various poker hands.

```
category := [1, 2, 3, 4, 5, 6, "1 pair", "2 pair",
            "3 of a Kind", "4 of a Kind", "Small straight",
            "Large straight", "Full house", "Chance", "Yatzy"]
```

The user can select one of these categories by typing a number from 1 to 15, but they must not pick the number of a category that has already been scored:

```
write("What category do you want to score in (1-15): ")
repeat {
   cat := read() | stop("terminated, end of file")
   if /score[integer(cat)] then break
   write("What category do you want to score in (1-15): ")
   }
```

In the first six categories of a Yatzy board, the points awarded are the sum of those dice with the value of the category number. United States readers should beware that the scoring rules are slightly different than those used by the game Yahtzee. The pair categories award points if at least two of the same dice match; similar points are awarded for attaining three or four dice of the same number. A small straight receives the sum of the dice if they read 1, 2, 3, 4, 5. A large straight receives the sum if they read 2, 3, 4, 5, 6. A user scores a 0 if they select a category for which the requirements are not met.

```
   points := 0
   case category[cat] of {
      !6: every !dice=cat do points +:= cat

      "1 pair": if dice[j := (4|3|2|1)]=dice[j+1] then
                   points := dice[j]*2
      "2 pair": if (dice[j := (4|3)]=dice[j+1]) &
```

```
                (dice[k := (j-2 to 1 by -1)]=dice[k+1]) then
                points := dice[j]*2 + dice[k]*2
        "3 of a Kind": if dice[j := (3|2|1)]=dice[j+2] then
                        points := dice[j]*3
        "4 of a Kind": if dice[j := (2|1)]=dice[j+3] then
                        points := dice[j]*4
        "Small straight": points := if dice[j:=!5]~=j then 0 else 15
        "Large straight": points :=
                            if dice[j:=!5]~=j+1 then 0 else 20
        "Full house": if ((dice[1]=dice[3])&(dice[4]=dice[5])) |
                        ((dice[1]=dice[2])&(dice[3]=dice[5]))
                      then every points +:= !dice
        "Chance":     every points +:= !dice
        "Yatzy":      if dice[1]=dice[5] then points := 50
        }
    score[cat] := points
```

Complete Program

Here is the whole Yatzy program. It is playable, but could sure benefit from a graphical user interface.

```
############################################################
#
#      File:     yatzy.icn
#
#      Subject:  Program to play the dice game yatzy
#
#      Author:   Clinton L. Jeffery
#
#      Date:     January 2, 2015
#
############################################################
#
#   This file is in the public domain.
#
############################################################
#
# From the Wikipedia entry (en.wikipedia.org/wiki/Yatzy):
#
```

```
# Yatzy is a public domain dice game, similar to the Latin
# American game Generala, the English games Poker Dice,
# Yacht, Cheerio, and Yahtzee (trademarked by Hasbro in
# the United States). Yatzy is most popular in the
# Scandinavian countries.
#
############################################################

global scores, categories

procedure main()
   score := list(15)
   category := [1, 2, 3, 4, 5, 6, "1 pair", "2 pair",
                "3 of a Kind", "4 of a Kind", "Small straight",
                "Large straight", "Full house", "Chance", "Yatzy"]

   every turn := 1 to 15 do {
      dice := [?6, ?6, ?6, ?6, ?6]
      write("The dice are: ",
            dice[1], dice[2], dice[3], dice[4], dice[5])

      every 1 to 2 do {
         write("Which dice values do you keep?")
         keep := read()

         every i := 1 to 5 do {
           if not find(dice[i], keep) then dice[i] := ?6
           else keep[find(dice[i], keep)] := ""
           }
         write("The dice are: ",
               dice[1], dice[2], dice[3], dice[4], dice[5])
         }
      dice := sort(dice)
      write("You ended with: ",
            dice[1], dice[2], dice[3], dice[4], dice[5])

      every i := 1 to 15 do {
         writes(i,"\t",left(category[i],15),"\t")
         write(\score[i] | "available")
```

```
      }

   write("What category do you want to score in (1-15): ")
   repeat {
      cat := read() | stop("terminated, end of file")
      if /score[integer(cat)] then break
      write("What category do you want to score in (1-15): ")
      }

   points := 0
   case category[cat] of {
      !6: every !dice=cat do points +:= cat

      "1 pair": if dice[j := (4|3|2|1)]=dice[j+1] then
                   points := dice[j]*2
      "2 pair": if (dice[j := (4|3)]=dice[j+1]) &
                   (dice[k := (j-2 to 1 by -1)]=dice[k+1]) then
                   points := dice[j]*2 + dice[k]*2
      "3 of a Kind": if dice[j := (3|2|1)]=dice[j+2] then
                        points := dice[j]*3
      "4 of a Kind": if dice[j := (2|1)]=dice[j+3] then
                        points := dice[j]*4
      "Small straight": points := if dice[j:=!5]~=j then 0 else 15
      "Large straight": points :=
                           if dice[j:=!5]~=j+1 then 0 else 20
      "Full house": if ((dice[1]=dice[3])&(dice[4]=dice[5])) |
                       ((dice[1]=dice[2])&(dice[3]=dice[5])) then
                       every points +:= !dice
      "Chance": every points +:= !dice
      "Yatzy": if dice[1]=dice[5] then points := 50
      }
   score[cat] := points
   }
write("Final score: ")

   total := 0
   every i := 1 to 15 do {
      total +:= score[i]
      writes(i,"\t",left(category[i],15),"\t")
```

```
      write(\score[i] | "available")
      sum := score[1]+score[2]+score[3]+
                score[4]+score[5]+score[6]
      if (i=6) & (sum > 62) then {
         write("\tbonus\t50")
         total +:= 50
         }
      }
   write("\t",left("Total:",15), total)
end
```

Graphics

The following example draws a simple picture of a die. Opening a window is similar to opening a file:

```
w := open(“yaht”,”g”)
```

To draw a die, we need a square to outline it, and then we need to draw the dots that tell what the die roll is. The function

```
DrawRectangle(w, 10, 10, 180, 180)
```

draws a square starting at dot (10,10), 180 dots wide and 180 dots high. The center of this square will be at (100, 100). Dots on computer monitors are called *pixels*, and are numbered using (x,y) coordinates, starting at (0,0) in the upper-left corner. The x coordinate gives the pixel column, and y gives the pixel row.

```
die := ?6
if die = (1 | 3 | 5) then FillCircle(w, 100, 100, 10)
```

On most dice, a one, a three, and a five have a dot in the middle. This call to FillCircle() draws that dot, with a radius of 10 pixels. You can read the vertical bar (|) as “or”: if the die is one or three or five, draw a dot in the middle. The parentheses are necessary because without them it would be like (die = 1) | 3 | 5: an equals test normally applies to only the nearest thing to it. Here are the rest of the dots on dice:

```
if die = (2 | 3 | 4 | 5 | 6) then {
      FillCircle(w, 30, 30, 10)          # upper left dot
      FillCircle(w, 170, 170, 10)        # lower right dot
      }
if die = (4 | 5 | 6) then {
```

```
        FillCircle(w, 170, 30, 10)        # upper right dot
        FillCircle(w, 30, 170, 10)        # lower left dot
        }
    if die = 6 then {
        FillCircle(w, 30, 100, 10)        # midhigh left
        FillCircle(w, 170, 100, 10)       # midhigh right
        }
    Event(w)                          # waits for user to click
```

Exercises

1. Extend the dice game to handle multiple players and multiple games. Keep score and report each player's single-game and cumulative points after each game.
2. Extend the dice graphics to draw 5 dice instead of 1 die. Tweak the dice graphics to look nicer; for example, maybe the dots are too big or in the wrong places.
3. Extend the dice game to draw the dice graphically on each roll. Call EraseArea(w) in between each roll.
4. Extend the game to display the scorecard graphically, and allow the player to select the category in which to score each turn by clicking on it.

Chapter 4: Tic Tac Toe

Have you ever played Tic Tac Toe? It is easy to play, and it makes for a great computer program. This chapter presents two versions of Tic Tac Toe, one with text, and one with pictures.

The Tic Tac Toe Board

In order to draw the board, the computer has to remember each player's moves. One way to remember all the moves is to store the whole board using names like this.

square[1]	square[2]	square[3]
square[4]	square[5]	square[6]
square[7]	square[8]	square[9]

Choosing a name to remember the contents of the 9 squares is only the beginning. Each of those squares can have three possible states:

- no one has chosen that square yet, it is blank
- the square holds an X
- the square holds an O

The program could use numbers 1, 2, and 3 in each state to mean empty, x, and o, but it is more readable to use “ “, “x”, and “o” to remember these three possibilities. Before the game starts, start the board as empty:

```
square := list(9, “ “)
```

This is the same as saying

```
square := [“ “, “ “, “ “, “ “, “ “, “ “, “ “, “ “, “ “]
```

in both cases the name square holds a *list* of 9 things which are spaces.

Consider the following textual way to display the tic-tac-toe board. The character "-" is used to draw horizontal lines, while "|" draws vertical lines. The next version of this program will draw the boxes using a graphical picture.

```
write(“-------”)
```

```
every i := 1 to 3 do
   write(“|”,square[i*3-2],”|”, square[i*3-1],”|”, square[i*3],”|”)
write(“-------”)
```

Taking Turns

Tic Tac Toe switches back and forth between x and o, so the computer needs to remember which player played last. Player x starts. To remember something like whose turn it is, make up a name and use := to store a value using that name.

```
   turn := "x"
```

Now comes some tricky business. The computer must play a lot of turns (up to 9 of them), and each turn does pretty much the same thing: show the board, wait for the player to make his move, and then draw an x or an o. The following code outline shows how to repeat something 9 times, changing the “turn” each time. The lines that start with a # are telling you what instructions the program needs, but they aren't instructions, they are just comments to any human who happens to read the program. Drawing the board was already presented above. The sections to follow will need to add instructions that read the players' moves and check if they won.

```
   every 1 to 9 do {
      # draw the board
      # read the player's move (x or o)
      # check if game is over
      if turn == “x” then turn := “o” else turn := “x”
      }
```

The game is over when there are three in a row of the same letter, x or o, or if no squares remain blank. You can check for a win using calls to a helper procedure:

```
   if won := check_win(board) then stop(“Player “, won, “wins”)
```

The check_win() procedure is given below. It returns an x or an o if the corresponding player has won. The tests of equality produce their right operand if they succeed; if they do not, the return does not occur. The procedure fails if it falls off the end.

```
procedure check_win(board)
   every i := 1 to 3 do {
      # check horizontally
```

```
      return board[i*3-2]==board[i*3-1]==board[i*3]==(“x”|”o”)
      # check vertically
      return board[i]==board[i+3]==board[i+6]==(“x”|”o”)
      }
   # check diagonals
   return board[1]==board[5]==board[9]==(“x”|”o”)
   return board[3]==board[5]==board[7]==(“x”|”o”)
end
```

Reading the Player's Move

Reading the human user's input sounds simple (just read an x or an o) but it is a bit trickier than that. The program already knows whether the player is drawing an x or an o, it needs to find out what location to draw it in. Locations are identified by a 1-9, but the program has to check and make sure the user does not mark a position that is already played, or type nonsense into the program! It also needs to provide an explanatory prompt to tell the user what it is expecting.

```
      repeat {
        write(“It is player “, turn,
                “'s turn. Pick a square from 1 to 9:”)
        pick := read()
        if square[integer(pick)] == “ “ then break
        }
      square[pick] := turn
```

A Complete Two-Player Tic Tac Toe Program

The complete program is

```
procedure main()
   turn := "x"
   square := list(9, “ “)
   every 1 to 9 do {
      # draw the board
      write(“-------”)
      every i := 3 to 9 by 3 do {
        write(“|”,square[i-2],”|”, square[i-1],”|”, square[i],”|\n-------”)
        }
      # read the player's move (X or O)
      repeat {
        write(“It is player “, turn,
```

```
            "'s turn. Pick a square from 1 to 9:")
         pick := read()
         if square[integer(pick)] == " " then break
         }
      square[pick] := turn
      if won := check_win(board) then stop("Player ", won, "wins")
      # advance to the next turn
      if turn == "x" then turn := "o" else turn := "x"
      }
   write("Cat game!")
end
# Check if the game is over; return the winner if there is one
procedure check_win(board)
   every i := 1 to 3 do {
      # check horizontally
      return board[i*3-2]==board[i*3-1]==board[i*3]==("x"|"o")
      # check vertically
      return board[i]==board[i+3]==board[i+6]==("x"|"o")
      }
   # check diagonals
   return board[1]==board[5]==board[9]==("x"|"o")
   return board[3]==board[5]==board[7]==("x"|"o")
end
```

Graphical TTT

A graphical tic-tac-toe program can look something like this:

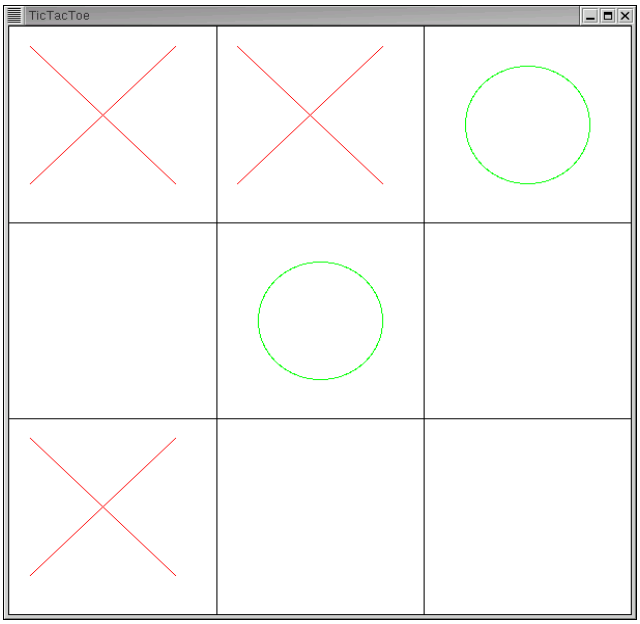

Figure 4-1:
A graphical tic-tac-toe board

Drawing lines and circles in a window is done using using graphics. To draw some graphics you have to first tell the computer to put a window on the screen.

```
procedure main()
   &window := open("TicTacToe", "g",
            "size=600,600", "linewidth=5")
```

The call to open() takes several parameters: first a name, then a mode ("g" stands for graphics), then how big you want the window to be, and lastly how wide the lines should be drawn. In this case we want a window 600 dots wide and 600 dots high. Each dot is called a pixel, and to pick out a particular pixel you give its location as: how far over from the left edge (the "x" coordinate) and how far down from the top (the "y" coordinate).

To draw the Tic Tac Toe board, we want to draw lines from top to bottom at locations 200 and 400, which are one third and two thirds of the way across.

```
   DrawLine(200,0,200,600)
   DrawLine(400,0,400,600)
   DrawLine(0,200,600,200)
   DrawLine(0,400,600,400)
```

There is a function named Event() that waits until the person running your program types a key or clicks the mouse. There are

many different events, but the only one we care about is a left mouse click, called &lpress. So we read one event, but if it isn't a left mouse click, the next instruction will take us back to the repeat so we can ask for another event. "if" and "then" are used to do an instruction only after checking and seeing whether something is true:

```
if Event() ~=== &lpress then next
```

We divide the Tic Tac Toe board into three rows, from top to bottom, numbered 0, 1, and 2. Similarly we divide the boardi into three columns, from left to right, numbered 0, 1, and 2. On a mouse click, the location where the mouse is can be found in &x and &y. These numbers are how many dots from the top left corner of the window. To find the row and column in the Tic Tac Toe board, we divide by 200, and remember the results using a name we can ask for later:

```
y := &y / 200
x := &x / 200
```

Now, draw a red x or a green o, depending on whose turn it is. The name turn is remembering whose turn it is, and after we draw an x or an o we change to the other player's turn. The Fg() instruction tells what foreground color to draw with, and to draw an x we just draw two lines.

```
if turn == "x" then { # draw an X
  Fg("red")
  DrawLine(x*200 + 20, y*200 + 20,
        x*200 + 160, y*200 + 160)
  DrawLine(x*200 + 20, y*200 + 160,
        x*200 + 160, y*200 + 20)
  turn := "o"
  }
```

If you want, the if-then instruction lets you put in an "else" instruction which tells what to do if the condition you checked wasn't true. If it isn't x's turn it is o's turn, so we should draw a circle.

```
else {              # draw on O
  Fg("green")
  DrawCircle(x * 200 + 100, y * 200 + 100, 60)
  turn := "x"
```

```
      }
    }
end
```

This is a pretty interesting program with a lot of ideas in it, but it is only 28 lines of code, and if you ask enough questions you should be able to understand every line. Later on after you learn more, you might come back to this program and teach it how to stop anyone from making an illegal move, how to quit, and how to tell who wins. Here is the complete program so you can type it in and try it:

```
procedure main()
  &window := open("TicTacToe","g", "size=600,600",
                          "linewidth=5")
  DrawLine(200,0,200,600)
  DrawLine(400,0,400,600)
  DrawLine(0,200,600,200)
  DrawLine(0,400,600,400)
  turn := "x"
  repeat {
    if Event() ~=== &lpress then next
    y := &y / 200
    x := &x / 200
    if turn == "x" then {      # draw an X
      Fg("red")
      DrawLine(x * 200+20, y * 200+20,
           x * 200 + 160, y * 200 + 160)
      DrawLine(x * 200+20, y * 200 + 160,
           x * 200 + 160, y * 200 + 20)
      turn := "o"
      }
    else {        # draw on O
      Fg("green")
      DrawCircle(x * 200 + 100,  y * 200 + 100, 60)
      turn := "x"
      }
    }
end
```

Adding a Computer Player

The first step towards providing an “intelligent” computer player for a turn-based game like Tic Tac Toe is to encapsulate the task of

choosing the move for a given side. First, add a command-line argument to specify whether the human is playing “x” or “o”, at the front of the main() procedure:

```
human := (argv[1] == (“x”|”o”)) | “x”
```

The code for choosing the current move then becomes:

```
if human==turn then {
   # read the player's move (X or O)
   pick := humanmove(square, turn)
} else {
   pick := computermove(square, turn)
}
```

The humanmove() code is just a straightforward packaging of the repeat loop given earlier, but the computermove() code is strangely similar. Given the current program state, it asks that a move be selected. In the following example, the move is selected by trying moves at random until a legal move is found:

```
procedure computermove(square, turn)
  write("...")
  repeat {
    pick := ?9
    if square[integer(pick)] == " " then break
    }
  delay(500)
  return pick
end
```

The write(“...”) and the half-second delay are inserted just to give the human time to notice that the computer has made its move.

Making the Computer Smarter

If you are playing solitaire, this “dumb” computer player may be more entertaining than moving the pieces for both sides yourself. A more satisfying computer player will at least rate its available moves in order to select which one it will choose.

```
procedure computermove(square, turn)
  every square[i := 1 to 9] == " " do {
    newboard := copy(square)
    newboard[i] := turn
    pick := evaluate(newboard, turn)
```

```
    if /bestpick | (pick > bestpick) then bestpick := pick
    }
  return bestpick
end
```

The real question, then, is how to evaluate board positions from the point of view of a given player. How to you evaluate a given board position in a strategy game? Usually, there is an absolute best and worst (where the game is won or lost outright), and otherwise it comes down to an opinion based on how the pieces are lined up. The technical term for this is *heuristic* – an evaluation that is just based on (hopefully good) judgment, without a guarantee of correctness or optimality, is a heuristic judgment. When a judgment is formed by a combination of such rules of thumb, we say the evaluation is based on heuristics. Here is a sample set of heuristics:

- if I have a win: award the board position +10000
- if the opponent has a win: award the board position -10000
- for every opponent's unblocked two-in-a-row, award -1000
- for every one of my unblocked two-in-a-row, award +300

Combining these rules, we get the following evaluator:

```
procedure evaluate(board, turn)
  if won := check_win(board) then
    return if won == turn then 10000 else -10000
  points := 0
  every play := two_in_row(board, turn) do
    if play==turn then points +:= 300
    else points -:= 1000
  return points
end
```

In order to finish this version, another helper function is required. Later in the chapter on checkers, we will contemplate how all these helper functions that evaluate board positions would be easier to keep straight if we had a new data type for evaluating and operating on board positions. The way such data types are introduced is called object-oriented programming.

```
# generate all the un-blocked two-in-a-rows
procedure two_in_row(board, turn)
```

```
   every i := 1 to 3 do {
      # check horizontally
      if board[i*3-2]==" " & board[i*3-1]==board[i*3] then
         suspend board[i*3]
      if board[i*3-1]==" " & board[i*3-2]==board[i*3] then
         suspend board[i*3]
      if board[i*3]==" " & board[i*3-2]==board[i*3-1] then
         suspend board[i*3-1]
      # check vertically
      if board[i] == " " & board[i+3]==board[i+6] then
         suspend board[i+6]
      if board[i+3] == " " & board[i]==board[i+6] then
         suspend board[i]
      if board[i+6] == " " & board[i]==board[i+3] then
         suspend board[i]
      }
   # check diagonals
   if board[1]==" " & board[5]==board[9] then suspend board[5]
   if board[5]==" " & board[1]==board[9] then suspend board[1]
   if board[9]==" " & board[1]==board[5] then suspend board[1]

   if board[3]== " " & board[5]==board[7] then suspend board[5]
   if board[5]== " " & board[3]==board[7] then suspend board[3]
   if board[7]== " " & board[3]==board[5] then suspend board[3]
end
```

By way of further study, the Icon Program Library has another version of Tic Tac Toe written by Chris Tenaglia in which the computer plays against the human player reasonably well. You can find it at

http://www.cs.arizona.edu/icon/library/src/progs/ttt.icn

Exercises

1. Merge the various of tic-tac-toe into one version that draws pictures and checks for legal moves and wins properly.
2. Modify Tic Tac Toe's evaluator to look two moves ahead.

Chapter 5: Card Games

Card games are an excellent opportunity to learn more about how computers organize information. A card game is slightly trickier than a dice game from the standpoint that there is a fixed deck and each card you remove from it won't be in the deck the next time you draw, unless it was put back for some reason. In this chapter you will look at a card game called “war” that uses a standard deck of 52 cards (aces through kings in each of hearts, diamonds, spades, and clubs). This version of “war” might not be identical to how you have played it before, since there are many variations.

There are a few basic questions that will be relevant for any card game: how do we store cards, hands, and decks in the computer's memory, and how do we present them legibly to the user? Other issues include: how do we shuffle, and how do we deal out cards?

Representing Cards

It would be easy to store the cards as unique numbers from 1-52, but the user won't know what these numbers mean unless they read the program code. The cards might be stored as a more self-explanatory code such as “ace of spades” or “seven of hearts”, which might be easier for the human but more work for the computer to use. For humans, the ultimate way to show the cards is with a picture, and this chapter will show a way to do that also.

In order to make it easy for both humans and computers to know what a card means, in this chapter we will represent a card in the computer's memory as a list of three items:

```
[ rank, suit, label ]
```

where rank and suit are human-readable and label is a computer code for the card. Rank will be an integer from 1-13 to indicate ace, two, on up to king. Suit will be a string (“hearts”, “clubs”, “diamonds”, or “spades”). The label will be a single letter in the range A-Z or a-z. A-M are ace..king of clubs, N-Z are ace..king of diamonds, a-m are ace..king of hearts, and n-z are ace..king of spades. The reason for the label is that Gregg Townsend wrote instructions to draw pictures of cards using these codes.

The Deck

To create the deck and shuffle it, start your program like this:

```
link random
procedure main()
   randomize()
   deck := [ ]
   suits := [“hearts”, “diamonds”, “spades”, “clubs”]
   every i := 1 to 4 do        # for each suit
      every j := 1 to 13 do   # for each rank
         put(deck, [ j, suits[i], char(ord(“A”) + (i-1)*13 + (j-1)) ] )
```

There is some gross and unexplained code on that last line. The rank is j, and the suits are understandable (“hearts” and so on), but the label depends on a mysterious code called ASCII that the computer uses to store letters like “A” in memory. The call ord(“A”) gives the number that the computer uses to store an “A”, which is our code for “ace of hearts”.

Shuffling

To shuffle the deck, use the random operator (?) like so:

```
every 1 to 100 do ?deck :=: ?deck
```

check whether this is shuffled enough by printing out your deck.

```
        every card := !deck do
              write(card[1], “ of “, card[2])
```

Dealing

In war, the cards are dealt out to two players. In our version, one player will be the human user while the other will be the computer.

```
Player1 := [ ]
Player2 := [ ]
every 1 to 26 do {
   put(Player1, pop(deck))
   put(Player2, pop(deck))
   }
```

Turns in War

Each turn, the top card on each player's hand is turned up, and whoever is higher wins both cards.

```
while *Player1 > 0 & *Player2 > 0 do {
   write(“Player1: “, Player1[1][1], “ of “, Player1[1][2])
   write(“Player2: “, Player2[1][1], “ of “, Player2[1][2])
   delay(1000)
  if Player1[1][1] > Player2[1][1] then {
      write(“Player1 wins”)
      put(Player1, pop(Player2), pop(Player1))
      }
  else if Player1[1][1] < Player2[1][1] then {
      write(“Player2 wins”)
      put(Player2, pop(Player2), pop(Player1))
      }
   else { # tie; should resolve tiebreak better
      write(“Tie”)
      put(Player1, pop(Player1))
      put(Player2, pop(Player2))
      }
   }
if *Player1 = 0 then write(“Player2 wins”)
else write(“Player1 wins”)
```

Graphics

To add graphics, you will want to link in Gregg Townsend's drawcard module. At the top of your program before your procedure main(), add

```
link drawcard
```

To open a window, add the following call on a line at the beginning of procedure main():

```
   &window := open(“War”,”g”)
```

To draw the cards, instead of writing them out, you would write something like the following each turn. The numbers such as 10,50 are x,y coordinates that select the location of the dot at which to start drawing something.

```
   EraseArea()
   DrawString(10, 50, “Player 1:”)
   drawcard   (80, 10, Player1[1][3])
   DrawString(210, 50, “Player 2:”)
```

```
drawcard   (280, 10, Player2[1][3])
```

Exercises

1. Fix the War game's tiebreaker. In the event of a tie, place the cards in a new list called “kitty”, and draw more cards until eventually you have a winner. The winner wins all the cards in the kitty.

2. Write another card game that you would like to play. The hard parts will most likely be scoring and determining who wins.

Chapter 6: Checkers

Checkers is a classic game played on an 8x8 grid. Some of the code might look similar to the tic tac toe program, since that program used a 3x3 grid. One basic difference is that the pieces in checkers start already on the board, and move from location to location. This chapter presents a program for playing checkers with two human players named Red and White. Red will go first.

```
procedure main()
   turn := “red”
```

In order to keep track of the board, the program could number the checkerboard with positions from 1-64 like Tic Tac Toe used positions 1-9, but instead it may be easier to keep things straight using a list of 8 rows, each of which is a list of 8 squares.

```
   square := list(8)
   every !square := list(8, “ “)
```

From now on, you can refer to positions within the checkerboard by saying the name square[x,y] where x and y refer to row and column. The positions will have names that look like this

[1,1]	[1,2]	[1,3]	[1,4]	[1,5]	[1,6]	[1,7]	[1,8]
[2,1]	[2,2]	[2,3]	[2,4]	[2,5]	[2,6]	[2,7]	[2,8]
[3,1]	[3,2]	[3,3]	[3,4]	[3,5]	[3,6]	[3,7]	[3,8]
[4,1]	[4,2]	[4,3]	[4,4]	[4,5]	[4,6]	[4,7]	[4,8]
[5,1]	[5,2]	[5,3]	[5,4]	[5,5]	[5,6]	[5,7]	[5,8]
[6,1]	[6,2]	[6,3]	[6,4]	[6,5]	[6,6]	[6,7]	[6,8]
[7,1]	[7,2]	[7,3]	[7,4]	[7,5]	[7,6]	[7,7]	[7,8]
[8,1]	[8,2]	[8,3]	[8,4]	[8,5]	[8,6]	[8,7]	[8,8]

Checkers features an initial board that looks like the screen below. Half the squares (the white squares) will always be empty. The dark squares have either white or red pieces, or are empty. We will use the values “ “, “white” and “red” to indicate the board contents. Queens will be indicated by “white queen” or “red queen”.

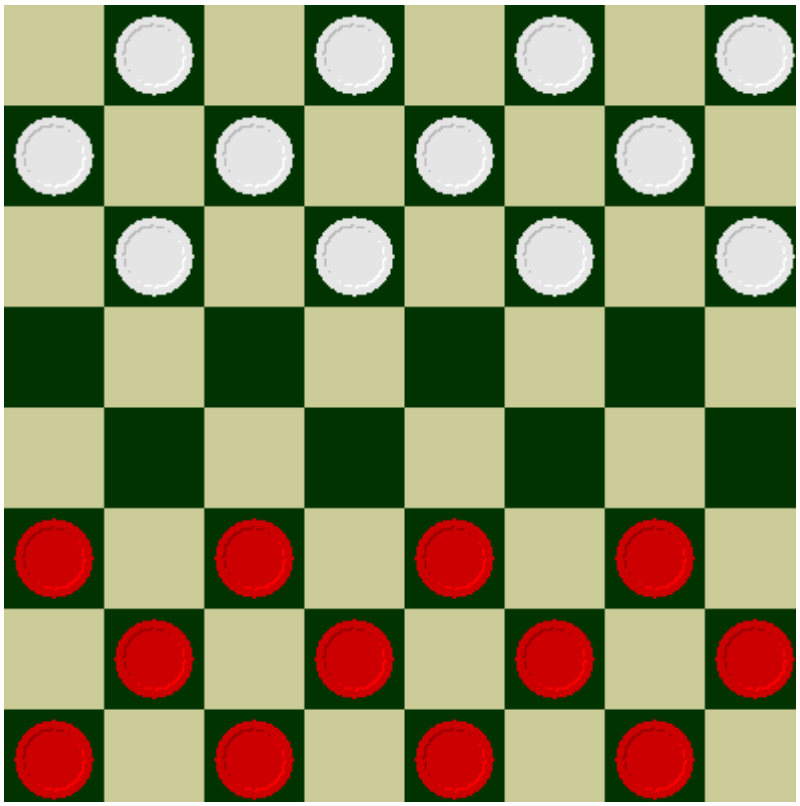

Figure 6-1:
A checkers board

After starting every square out as a space, the initial board contents can be generated as follows. The code uses “red” and “white” as the players' sides, but other player colors can be substituted here and in the section below titled Taking Turns.

```
every row := 1 to 3 do
   every col := 1 + (row % 2) to 8 by 2 do
      square[row,col] := “white”
every row := 6 to 8 do
   every col := 1 + (row % 2) to 8 by 2 do
      square[row,col] := “red”
```

Drawing the Checkers Board Textually

The following displays the checkers board textually. The character “-” is used to draw horizontal lines, while “|” draws vertical lines. If square[i,j] is a word such as “red” or “white”, then square[i,j][1] or square[i,j,1] is the first letter (“r” or “w”). These letters will be written to to textually indicate the positions of pieces on the board. The next version of this program draws the board using a picture.

```
write(“   \\ 1 2 3 4 5 6 7 8 column”)
write(“row ----------------”)
every i := 1 to 8 do {
   writes(“ “, i, “  “)
   every j := 1 to 8 do
      writes(“|”,square[i,j,1])
   write(“|”)
   write(“    ----------------”)
   }
```

The textual version of the board looks like:

Figure 6-2:
A textual representation of checkers.

Taking Turns

The program will repeat until one side or the other wins, similar to the war program in the last chapter. The “while” instruction below checks to see that there are still pieces on each side. Each turn the instructions inside the do { ... } will resemble (at least a little bit) the same steps as in tic-tac-toe.

```
while find(“red”, !!square) & find(“white”, !!square) do {
   # draw the board
   # read the player's move and change the board
```

```
if turn == “red” then turn := “white” else turn := “red”
}
```

Reading the Player's Move

Moving is a bit tricker in checks than it was in tic-tac-toe. First the program most prompt the user and read and interpret their desired move. Then it must tell whether it is legal, and if so, move the pieces accordingly. For this version we will read the player's move using the following format: row,col,...row_n,col_n where the ... might be nothing, or if the move is jumping multiple pieces, the ... would be all the squares that the piece is to move through, separated by commas.

```
write(turn, “'s turn, move in row,col,...row,col format:”)
input := read()
```

Processing this input string is going to be tricky. It is really a list of numbers.

```
L := [ ]
```

To break it up into pieces, look for the commas. This uses a feature of Unicon called string scanning. Strings are just a sequence of letters, which we have been calling “words” up to now. String scanning uses the notation (s ? instructions) in which a string s is examined by the instructions. In checkers, the program puts the numbers it finds into the list.

```
input ? while put(L, integer(tab(many(&digits)))) do =”,”
```

Is this a legal move for the current player? If so, square[L[1],L[2]] holds a piece of his color, and all the other locations are empty. In addition, either the move was to an adjacent diagonal, or the move was a jump. All this must be checked.

```
# read the player's move
repeat {
  write(“It is “, turn, “'s turn, move in x,y,...xn,yn format:”)
  input := read()
  L := [ ]
  input ? while put(L, integer(tab(many(&digits)))) do =”,”
  # if starting square did not hold our piece, re-do
  if not find(turn, square[L[1],L[2]]) then {
    write(“Requested move is illegal, try again.”)
```

```
   next
   }
every i := 3 to *L by 2 do {
   # if target square is not empty, re-do
   if square[L[i], L[i+1]] ~== “ “ then next
   if find(“queen”, square[L[1],L[2]]) then { # queen rules
      if abs(L[3]-L[1]) = abs(L[4]-L[2]) = 1 then {
         square[L[1],L[2]] :=: square[L[3],L[4]]
         break break
         }
      else if abs(L[i]-L[i-2]) = abs(L[i+1]-L[i-1]) = 2 then {
         square[L[i],L[i+1]] :=: square[L[i-2],L[i-1]]
         square[(L[i]+L[i-2])/2, (L[i+1]+L[i-1])/2] := “ “
         }
      else {
         write(“Can't perform requested move.”)
         break next
         }
      }
   else { # regular piece
      direction := (if turn==”red” then -1 else 1)
      if abs(L[2]-L[4]) = 1 & (L[3]-L[1]) = direction then {
         square[L[1],L[2]] :=: square[L[3],L[4]]
         break break
         }
      else if abs(L[i]-L[i-2]) = 2 &
               (L[i+1]-L[i-1]) = direction*2 then {
         square[L[i],L[i+1]] :=: square[L[i-2],L[i-1]]
         square[(L[i]+L[i-2])/2, (L[i+1]+L[i-1])/2] := “ “
         }
      else {
         write(“Can't perform requested move.”)
         break next
         }
      }
   }
break
}
```

Checkers Graphics

The additional graphics functions you need to know in order to draw the checkerboard are Fg(s) to set the foreground color, along with FillRectangle(x,y,width,height) and FillCircle(x,y,radius) to do the drawing. In addition, you will want to know how to open a window of the correct size and shape. Suppose we want each square of the board to be 64 dots wide and 64 dots high. Then the total board should be 64x8 pixels wide and high.

```
&window := open("Checkers","g","size=512,512")
```

The board needs to have alternating light and dark rectangles. The checker color "green" here may be changed to be any dark color.

```
Fg("green")
every i := 1 to 8 do
   every j := 1 + (i % 2) to 8 by 2 do
      FillRectangle( (j-1) * 64, (i-1) * 64, 64, 64)
```

To handle the moves, the program could redraw the board from scratch each time, or it could just erase where the piece has left (a call to FillRectangle() should do the trick) and then draw the piece at the new location (a single call to FillCircle() will work). In both cases the foreground color will need to be set first.

Also it would be nice if the game were played by clicking on the board, instead of having to type coordinates. The function Event() reads mouse clicks and stores their coordinates in &x and &y. If you take those pixel coordinates and divide by 64 (and add 1) you will get the row and column.

Moving Pieces Around

The main challenge in moving pieces around a checkerboard is to calculate the (x,y) pixel coordinates that identify the position of the dot at which to draw. Once you have those coordinates, erasing where a checker piece used to be looks like:

```
Fg("green")
FillRectangle(x, y, 64, 64)
```

Drawing a checker piece in its new location looks like:

```
Fg(turn)
FillCircle(x+32, y+32, 28)
```

The +32 parts are because FillCircle() works from its center point, not its upper-left corner the way FillRectangle() works.

Animation

In a really cool checkers game, the pieces would *slide* or *jump* to their new positions. A slide can be done by many calls to FillRectangle() and FillCircle(), with x and y just changing by 1 (or a small number) each time going from the old values to the new values. The stuff to be erased is going to be some mixture of up to four different squares, so maybe four different squares have to be redrawn at each step.

Suppose you are moving from position (64,64) down to position (0,128). You might try the following code:

```
x := y := 64
every i := 0 to 63 do {
   Fg("green")
   FillRectangle( x-i, y+i, 64, 64)
   Fg(turn)
   FillCircle( x-i + 32, y+i + 32, 28)
   delay(50)
   }
```

Try numbers other than 50 in the delay() and see what number looks best. The delay is a number of milliseconds (the units are 1/1000th of a second), so 50/1000 is 1/20th of a second.

Making a jump "look cool" remains an exercise for the reader, and you might have to bring in some trigonometry or physics equations to do it properly. You would be amazed, however, at how much you can do without any advanced math, if you are clever.

A Computer Checkers Player

Checkers is an almost ideal platform for introducing a more serious computer player, and this game was one of the first to be conquered by artificial intelligence researchers. For turn-based games such as checkers and chess (it would have worked for tic-tac-toe, but was overkill), the most popular brute force technique for computer play looks ahead several moves, considering every possibility. The algorithm is called minimax.

The Minimax Algorithm

Given a board position P, the computer player's goal is to calculate the best possible move. Position P is the root of a tree of board positions in which P's children are the board positions of each of the possible moves. Each of P's children either denote a terminal position such as a win for one side or the other, or they have one or more children that are possible moves.

In order to evaluate all possible moves and select one from the current position, the minimax algorithm selects the child whose board position is best (maximizing the computer's position). Evaluation of board positions can consider further moves (looking further ahead) to the extent that the CPU allows within some real-time constraint. The algorithm pessimistically assumes that the adversary will make their best possible move (minimizing the computer player's position). The computer's best move, short of forcing a win, will be whatever gives the opponent the least opportunity to win.

This algorithm can be generically applied to a broad range of turn-based games. It requires a game-specific board position representation (board positions are nodes in the game tree), and several helper functions that operate on a (possible, future) board position. These requirements can be specified using the following abstract class. A game such as checkers will implement a subclass that provides a board position in one or more member variables, and use that data structure to implement at least these five methods.

```
class GameState(player,        # list of players
                turn)          # whose turn it is
  abstract method evaluate()
  abstract method finished()
  abstract method generate_moves()
  abstract method copy()
  abstract method draw_board()
```

The methods finished() and evaluate() return whether a board position is terminal, and a heuristic assessment of the strength of the board position, respectively. The method generate_moves() generates all the possible board positions that can immediately follow a given position. The method copy() creates a physical copy

of the game state, with which the minimax algorithm applies prospective moves and evaluates them recursively. The method draw_board() is used to render a current or prospective game state visually for the human user. The abstract class GameState does implement two methods, the first of which is the minimax() algorithm itself.

```
method minimax(depth, player)
local alpha
  /player := turn
 if (depth = 0) | finished() then {
    ev := evaluate()
    if ev === &null then {
       stop("evaluate() returned null for preceding board")
       }
    return ev
    }
 else if player == turn then {
    every childmove := generate_moves(node) do {
        child := copy()
        child.apply_move(childmove)
        kidval := child.minimax(depth-1, player)
        if not (/alpha := kidval) then
            alpha <:= kidval
        }
    }
 else { # minimizing player
    every childmove := generate_moves(node) do {
        child := copy()
        child.apply_move(childmove)
        kidval := -child.minimax(depth-1,player)
        if not (/alpha := kidval) then
            alpha >:= kidval
        }
    }
 return \alpha | 0
end
```

The GameState class also provides a method advance_turn() that sets the turn member variable to the next player in the list of players, wrapping around to player[1] when it reaches the end.

```
method advance_turn()
   if turn == player[i := 1 to *player] then {
      i +:= 1
      if i > *player then i := 1
      }
   else stop("no player named ", image(turn))
   turn := player[i]
end
```

Given the abstract class, the checkers game logic is captured in a subclass. The representation of the board is a list of lists of strings with names like “red queen” or “white”.

```
class CheckersGameState : GameState(square, mydepth)
```

The representation of a move is a list of row1,col1,row2,col2... that moves a piece from row1,col1 to rowN,colN through a series of 0 or more intermediate locations.

```
method apply_move(L)
   i := 0
   while i+4 <= *L do {
      srcrow := L[i+1]; srccol := L[i+2];
      destrow := L[i+3]; destcol:=L[i+4]
      square[srcrow,srccol] :=: square[destrow, destcol]
      if abs(srcrow-destrow)=2 then
         square[(srcrow+destrow)/2,(srccol+destcol)/2] := " "
      i +:= 2
      }
end
```

The heuristic for evaluating a board position in checkers is vital to the skill level exhibited by the computer player. A checkers expert could undoubtedly provide a much stronger evaluation function here. The basic tenets of the following relatively brute-force evaluation are: add 1000 points for each friendly piece, where rank indicates how far that piece has advanced towards the end. Add 10,000 points for each friendly queen. Subtract points for distance

from promotion, for non-queen pieces. Assess enemy pieces and positions with the symmetric opposite point scores.

```
method evaluate()
   local dir # direction this player is moving
   if turn=="white" then dir := 1 else dir := -1
   points := 0
   every row := 1 to 8 do
      every col := 1 to 8 do {
         if square[row,col] == " " then next
         else if find(turn,square[row,col]) then
            sgn := 1
         else sgn := -1
         points +:= 1000*sgn
         if find("queen", square[row,col]) then {
            points +:= 10000*sgn
            }
         else {
            points -:= sgn * 2 ^ (if dir=-1 then row else (8-row))
            }
         }
   return points
end
```

The code to generate all possible moves from a given board configuration must first find every piece owned by the current player and then consider all moves in all directions that are possible.

```
method generate_moves()
   # for every current-player's piece on the current board...
   every row := 1 to 8 & col := 1 to 8 &
            match(turn,square[row,col]) do {
```

An important issue is: how are moves represented? In this game, a move is represented as a list of positions, where each position is denoted by two elements, a row followed by a column. Regular moves are easy, they move the piece forward one row and into an adjacent column that must be empty.

```
         if turn=="white" then dir := 1 else dir := -1
         every square[row+dir,(c := ((col-1)|col+1))]== " " do {
```

```
        suspend [row,col,row+dir,c]
        }
```

Jumps are somewhat more involved. The conditions include an enemy piece in an adjacent diagonal, with an empty square behind it. In addition, such a jump allows the possibility of multiple jumps if the same conditions are present at the destination.

```
    # jumps
    every (csgn := (1|-1)) &
        (square[row+dir*2, col+csgn*2]== " ") &
        enemy_in(row+dir,col+csgn) do {
      L:= [row,col,row+dir*2, col+2*csgn]
      suspend L
      # multijumps logic here
      suspend generate_jumps(L)
      }
```

Queen moves are similar, but less constrained since they can move both forward and backward.

```
    every (dir := (1|-1)) & (csgn := (1|-1)) do {
          if square[row+dir,col+csgn] == " " then
            suspend [row,col,row+dir,col+csgn]
          if square[row+dir*2,col+csgn*2] == " " &
            enemy_in(row+dir, col+csgn) then {
            L := [row,col,row+dir,col+csgn]
            suspend L
            # multijumps logic here
            suspend generate_jumps(L)
            }
          }
```

The generate_moves() code depended on a couple helper functions. A simple predicate function enemy_in() succeeds if an enemy piece is located at a given row and column.

```
  method enemy_in(row,column)
    if (row|column)=0 then fail
    s := square[row,column]
    return not (s == (" " | turn))
  end
```

A more significant helper function generate_jumps() looks at a given jump move and generates additional moves possible due to jumping a second or subsequent pieces. The function is recursive.

```
method generate_jumps(L)
   local dir # direction this player is moving
   if turn=="white" then dir := 1 else dir := -1
   row := L[-2]
   col := L[-1]

   every (rsgn:=(dir|
                  (if find("queen",square[row,col]) then -dir))) &
      (csgn := (1|-1)) &
      (square[row+rsgn*2, col+2*csgn]==" ") &
      enemy_in(row+rsgn, col+csgn) do {
        L2 := L ||| [row+rsgn*2, col+2*csgn]
        suspend L2
        suspend generate_jumps(L2)
        }
end
```

Detection of a terminal board position is achieved by method finished(), which returns the winning player if there is one. The version below is incomplete in that conditions need to be added to detect when the game is over because a player is unable to move.

```
method finished()
   if not find(player[1], !!square) then return player[2]
   if not find(player[2], !!square) then return player[1]
end
```

The following ASCII art rendition of the checkers board might be replaced with a graphical version.

```
method draw_board()
local i, j
write("(draw board ",image(square),")")
   write("   \\ 1 2 3 4 5 6 7 8  column")
   write("row ----------------")
   every i := 1 to 8 do {
      writes(" ",i,"  ")
      every j := 1 to 8 do {
```

```
      if find("queen", square[i,j]) then
        writes("|",map(square[i,j,1],&lcase,&ucase))
      else
        writes("|",square[i,j,1])
      }
    write("|\n    ----------------")
    }
end
```

The actual game A/I is provided in a method computermove() that rates each possible move and selects the best one using minimax. The game tree nodes are constructed by copy+modify of the current game state. The moves are flat lists of alternating row,col coordinates; returning the computer move like this, instead of the winning "game state" allows the computer player to look just like a human player who input their move as a list of row,col coordinates.

```
method computermove()
  list_of_possible := []
  t1 := &time
  every possible := generate_moves() do {
    put(list_of_possible, possible)
    mv := copy()
    mv.apply_move(possible)
    thepoints := mv.minimax(mydepth)
    mv.draw_board()
    if /bestpoints | thepoints>bestpoints then {
      bestpoints := thepoints
      bestmove := possible
      list_of_possible := [possible]
      }
    else if thepoints=bestpoints then {
      put(list_of_possible, possible)
      }
    }
  if *list_of_possible = 0 then stop("no possible moves")
  possible := ?list_of_possible
    t2 := &time
    if t2-t1 < 1000 then {
      delay(1000-(t2-t1))
```

```
            mydepth +:= 1
            }
         else {
            mydepth -:= 1
            }
      return possible
   end
```

The initialization for the CheckersGameState class is given in an initially section. Minimax depth starts at 1; players are initialized to be named “red” and “white”, and the board is initialized to a list of 8 lists of 8 strings as seen at the beginning of this chapter.

```
initially(p)
   mydepth := 1
   player := ["red", "white"]
   human := p
   turn := "red"
   # … code for initializing square[ ], as shown earlier
end
```

The main() procedure starts by initializing a CheckersGameState object. The human player's color is read from the first command line argument, defaulting to “red”, which is the color that moves first. The skeleton of the main game loop is given below.

```
procedure main(argv)
   game := CheckersGame((argv[1]==("red"|"white")) | "red")
   while not game.finished() do {
      game.draw_board()
      if not (L := game.get_move()) then stop("goodbye")
      if game.apply_move(L) then
         game.advance_turn()
      }
   write((game.finished() || " wins") |
         "game over, result a draw?")
end
```

Exercises

1. Test and correct the checkers program. There are many missing rules that aren't being checked.
2. Modify the graphical version of the checkers program so that the user specifies their move by left-clicking the mouse on the piece to move and the (sequence of) destination square(s). Right-click on the final destination.
3. Modify the checkers program so the computer player is smarter. This may involve improving the evaluation function, performance tuning, or allowing the computer more time for deeper lookahead.

Chapter 7: Text Adventures

Text adventures are one of the early forms of computer game, originating in the mid-1970's on timesharing computers' terminals and early PC's where little or no graphics were supported. Text adventures were (perhaps) directly inspired by pencil-and-paper roleplaying games such as Dungeons and Dragons, which took a popular genre of fiction and allowed players to imagine being in the story and interacting with it. A text adventure typically does so using a computer, but this form of *interactive fiction* became so popular that it in turn spawned a particularly odd form of books in which the reader makes choices which determine the outcome of the story – books imitating the computer games which imitate the paper games which imitate the great stories in books.

Besides spawning “interactive books”, text adventures went on to spawn other genres of computer games (graphical adventures such as King's Quest, and multiple-user dungeons (MUDs) which were the precursors to the modern Massively Multi-player On-line Role Playing Games that have become a multi-billion dollar industry. Text-adventure-style prose descriptions and location-based puzzles still form a core of the gameplay: they handle the exposition of quests and give meaning or purpose to the virtual lives of the players' characters.

Design

A text adventure needs to store a lot of text, which could be in an external file or could be embedded directly in the code. Far more importantly though, a text adventure needs to store a *model* of the imaginary world and the player's progress through the game. This includes: the locations and how to get from place to place, the objects the player needs, and a record of what the player has done.

The virtual world in a text adventure is typically a graph of nodes, where each node is a room or other discrete location. If each node is labeled with an integer, the player's location can be very easily stored and updated as a simple integer. It might be just as easy to store it as a string name of the location, such as “kitchen” or “backyard”. Text adventures typically allow the player to go from

place to place by commands such as “go north”, the programmer noting for each direction whether it has a door (or opening, or trail, or whatever) that constitutes an edge in the graph that takes you to a different location.

In each location, there is a text description of what you see when you get there, and usually one or more objects you can look at more closely, or possibly take with you. Besides recording about the player what room they are in, the program maintains an *inventory* of virtual objects that they are carrying.

CIA

CIA is an example text adventure inspired by an old game written in BASIC by Susan Lipscomb and Margaret Zuanich. It illustrates typical text adventure structure and features. The program features the following state variables.

```
# cia.icn – a CIA text adventure, inspired by Lipscomb/Zuanich

global verbs,    # set of verbs allowed in actions
       directions, # four directions in which one can move
       ca,         # list of (integer codes of) what the player is carrying
       vs,         # the current verb/action the player is performing
       rooms,      # array of rooms (adjacency list representation of graph)
       ob,         # array of obj (objects)
       li,         # list of (integer codes of) evidence toward conviction
       tt,         # game time elapsed
       maxtime,    # maximum game time allowed for victory
       g,          # 0 = Griminski not home, 1 = dead, 2 = he's attacking
       r,          # current room (integer code)
       rs,         # "read string" (or "response") == what the user said to do
       n, ns       # the noun (integer code, string) the player is acting on
```

Two complex structure types are used, to represent information about nouns (objects) and about rooms. These are both really record types, although the room type is declared as a class in order to take advantage of flexible constructor parameter handling.

```
record obj(as, ds, m, l, v, t)

class room(ds, e)
initially(descrip, h, e1,e2,e3,e4)
  ds := descrip
  hs := h
  e := [e1, e2, e3, e4]
```

```
end
```

CIA recognizes two kinds of input: single-word commands and verb-noun actions. The main() routine determines whether to call action() or command() based on whether a space character is found.

```
procedure main()
   init()
   r := 1
   write("\n", rooms[r].ds, "\n")
   repeat {
      if input() then {
           if parsing() then
              action()
         }
      else command()
   }
end
```

The input handler checks if time has run out, writes a prompt, and reads the user's answer. If it had a space in it, the procedure succeeds to indicate a verb-noun action; if there was no space, it fails, indicating a command.

```
procedure input()
   rs := ""
   tt +:= 3
   if tt > maxtime then stop("sorry... you ran out of time")
   writes("\nNow what? ")
   rs := read()
   write()
   rs ? if vs := tab(find(" ")) then { =" "; ns := tab(0); return }
end
```

“Parsing” is an overstatement, but verb-noun actions are validated by checking whether the verb is recognized, and checking whether the noun can be used with that verb. A primary side effect here is to compute the integer code of the noun of interest.

```
procedure parsing()
   if member(verbs, vs) then {
      every n := 1 to *ob do {
           if ns == ob[n].as & (ob[n].m = (r | 100)) then return
           }
      write("it won't help")
      }
   else
      write("i don't know how to ", vs)
```

```
   vs := ns := &null
end
```

Movement (a phrase such as “go west”) in this world is handled by checking the direction requested to see if the current room has an adjacency (edge) to a new room.

```
procedure go()
   every j := 1 to 4 do
      if directions[j] == ns then {
           if rooms[r].e[j] = 0 then { write("I can't go that direction"); return }
           r := rooms[r].e[j]
           write("\n", rooms[r].ds, "\n")
           return
         }
end
```

Initialization sets up several large static lists, particularly of objects and of rooms. Each object has a short name, a detailed description, and four codes indicating the object's location (field m, value 100 means the user possesses it), a link (in the sense of a linked list) to related/contained objects, an evidence value (field v), and a “take code” (field t) which determines which verbs work on that object.

```
procedure init()
   directions := [ "north", "east", "south", "west" ]
   ca := [ ]
   li := [ ]
   tt := g := 0
   et := 1000

   ob := [
      obj("north","it doesn't help",100,0,0,4),
      obj("east","it doesn't help",100,0,0,4),
      obj("south","it doesn't help",100,0,0,4),
      obj("west","it doen't help",100,0,0,4),

      obj("shelves","Shelves for weapons and tools line the wall next to your_
           desk.\nThere are numerous items which may help you on your _
           assignment.", 1,6,0,3),

      obj("screwdriver", "an all-purpose screwdriver with collapsible handle.",
             1,7,0,1),

      obj("bomb", "a mark mx high-intensity smoke bomb", 1,8,0,1),
      obj("pistol", "an automatic ppk-3 pistol", 1,9,0,1),
      obj("key", "a skeleton key", 1,10,0,1),
      obj("drug","a small can of insta-knockout drug", 1,11,0,1),
```

```
obj("gun", "a mark 3k harpoon gun with grapple and line", 1,0,0,1),
obj("door","The heavy door is painted black. A brass keyhole and _
    doorknob are here. You can see the circular holes on either side _
    of the door which must mean an electronic alarm beam.", 2,13,0,5),

obj("alarm","The alarm is screwed into place.",2,0,0,5),

obj("dog","The savage doberman leaps at you with bared fangs.\n _
    He will not let you past him.", 3,0,0,4),

obj("table","The venetian front hall table has a tortoise shell letter\n_
    tray on it for business cards and mail. There is a letter on the _
    tray.", 3,0,0,1),

obj("letter","This is apparently a phone bill that has been paid and\n_
    is being sent to the telephone company.", 3,0,10,1),

obj("umbrella","There is a black businessman's umbrella with a _
    pointed end.", 4,18,0,1),
obj("briefcase","There is a black leather briefcase with a _
    combination lock.", 4,0,0,1),

obj("desk", "The large oak desk has a blotter and pen set on it. _
    A phone is here. a blank notepad is by the phone. _
    The desk has a pigeonhole and one drawer on it.", 5,0,0,1),

obj("notepad", "Although the notepad is blank, you can see the _
    indentation of writing on it.", 5,0,0,1),

obj("drawer", "This is a standard pull desk drawer.", 5,0,0,4),
obj("pigeonhole", "The pigeonhole has a paid bill in it.",5,0,0,4),
obj("bill","The bill is from the telephone company.",5,0,0,1),
obj("phone","This is a beige pushbutton desk phone.",5,25,0,4),
obj("number","The telephone number is printed on the base",5,0,0,4),
obj("panel","The panels are tongue-in-groove. One of the panels _
     seems more worn than the others", 5,0,0,4),
obj("shelves","There are software programs, manuals, and blank _
    disks on the shelves.",6,0,0,4),
obj("program","One program is for communicating with the U.S. _
    defense department's mainframe computer.", 6,0,10,5),
obj("phone","This is a standard desk-type dial telephone. _
    The receiver is set into a modem.",6,30,0,4),
obj("number","The telephone number is printed on the base",6,0,0,1),
obj("computer","this is a standard office computer with a keyboard.\n_
    A cd is inserted into one of the drives. The power switch is off.",
    6,0,0,5),
obj("monitor","This is a high resolution LCD monitor. _
```

```
      The power switch is off.",6,0,0,5),
obj("modem","The modem is one that can use an automatic dialing\n_
      communications program. The power switch is off.", 6,0,0,5),
obj("tray","the silver tray holds a decanter partially filled with _
      claret.", 7,0,0,1),
obj("decanter","the decanter is of etched crystal. it probably holds _
      some claret.", 7,0,0,1),
obj("claret", "An amber liquid", 7,0,0,1),
obj("cabinet", "this is a fairly standard kitchen cabinet.",8,0,0,4),
obj("bottle","a bottle of capsules are here.",8,39,0,2),
obj("capsule","the capsules are elongated and have a slight aroma _
      of burnt almonds.",8,0,0,1),
obj("table","the bedside table has a phone on it. _
      a piece of paper and a lamp are here.", 9,0,0,3),
obj("phone","there is a number printed on the phone.", 9,0,0,4),
obj("paper","a piece of monogrammed writing paper",9,43,0,1),
obj("combination","there is a combination written on it",9,0,0,4),
obj("safe","this is a standard combination safe.",9,0,0,4),
obj("gum","a pack of stick-type peppermint gum. _
      each stick is wrapped in paper.", 9,0,0,2),
obj("microfilm","the microfilm has been developed but you can't _
      see it without special equipment. Nevertheless it's pretty _
      certain what you have found.",9,0,10,2),
obj("shelves","a very sophisticated camera is on one of the shelves.",
      10,0,0,4),
obj("camera","This camera is used to transfer documents to microfilm.",
      10,0,10,1),
obj("cabinet","this is a large mirrored bathroom cabinet.", 10,0,0,4),
obj("bureau","a wall safe is set into the wall above the low _
            mahogony carved bureau.", 9,0,0,3),
obj("bottles","bottles of fixer and photoflo are on the shelves.",
      10,52,0,2),
obj("tank","there is a film developing tank and a film apron _
            and tank cover here too.", 10,0,0,2),
obj("headquarters","headquarters",100,0,0,4),
obj("capsules","the capsules are elongated and have a slight aroma _
            of burnt almonds.",8,0,0,1),
obj("sideboard","a large ornate sideboard with a beveled glass mirror _
            dominates the east wall.", 7,34,0,4),
obj("number","there is a number printed on the phone.", 9,0,0,1),
obj("paper","the numbers 2-4-8 are written on a piece of paper _
            on the top of the drawer.", 5,0,0,2),
obj("griminski","the white-haired man is dressed in evening clothes.",
      6,0,0,4),
obj("corner","you are looking at the corner of the closet.", 4,17,0,4)
    ]
```

```
rooms := [

  room("You are in your office at the CIA.\n_
        On the shelves are tools you've used in past missions.\n_
        Ambassador Griminski's apartment is North.",
         "You'll need some tools to get into the apartment.", 2,0,0,0),

  room("You are at 14 Parkside Avenue. The entrance to ambassador\n_
       Griminski's small but elegant bachelor apartment. You see a \n_
       heavy wooden door with a notice on it warning of an alarm system.",
         "Maybe your tools will help you.", 0,0,1,0),

  room("This is the marbled foyer of the ambassador's apartment. \n_
      A table is in the corner. The master bedroom is east, the drawing \n_
      room is north, and a closet west. A fierce dog charges to attack.",
         "Something from your office could be helpful now.", 0,0,2,0),

  room("You are in the front hall cedar closet. Heavy overcoats and a \n_
    trenchcoat are hanging up. Boots are on the floor and other items \n_
    are in the corner.", "First impressions can be deceiving.", 0,3,0,0),

  room("You are in the drawing room. A desk is here. A sofa and a\n_
   coffee table are in front of the fireplace set into the paneled \n_
   east wall. The dining room is north.",
        "There is more here than meets the eye.", 7,0,3,0),

  room("You can see a microcomputer, monitor, and a cable modem \n_
   on a table against the east wall of this over-sized closet. A phone is by_
   the computer. A chair and shelves are here.",
        "Running a program is always interesting.", 0,0,0,5),

  room("You are standing in a small formal dining room. The table \n_
   seats six guests. A sideboard with a tray on it is against the east\n_
    wall. The kitchen is to the north.", "I can't help you here", 8,0,5,0),

  room("You are in the apartment kitchen which shimmers with\n_
   polished chrome appliances and butcherblock counters. A long\n_
    cabinet above the stainless steel sinks is closed.",
        "Be suspicious of items in small bottles.", 0,0,7,0),

  room("This is ambassador Griminsky's bedroom. A bed and bedside\n_
     table are here. A safe is in the wall above the bureau. The\n_
        bathroom and dressing area are to the north.",
         "Things are often not what they seem.", 10,0,0,3),

  room("You are in a combined bathroom / dressing area. The\n_
   ambassador's clothes are hanging neatly on rods and open shelves\n_
```

```
      hold towels and sweaters. The medicine cabinet is closed.",
          "Don't overlook the obvious.", 0,0,9,0) ]

   verbs := set([ "look","get","take","go","crawl","walk","open","read",
                  "drop","call","unscrew","spray","push","load","run","drink",
                  "eat","chew","unwrap","talk","shoot","unlock","on","off"])
   return
end
```

There are seven commands, each of which has a corresponding procedure.

```
procedure command()
   case rs of {
      "help": help()
      "quit": quit()
      "inventory": inventory()
      "look": look()
      "time": printtime()
      "score": printscore()
      "restart": restart()
      default: write("I can't understand ", rs)
      }
end
```

The help command writes out a string that is determined by what room the player is in (“context sensitive help”).

```
procedure help()
   write(rooms[r].hs)
end

procedure quit()
    write("are you sure you want to quit? (yes/no)")
    rs := read()
    if rs == "no" then return
    printtime()
    printscore()
    stop()
end
```

The inventory is maintained in a list of items carried (ca) which are subscripts into the global object list.

```
procedure inventory()
   if *ca = 0 then { write("you aren't carrying anything"); return }
   write("you have")
   every write(ob[!ca].as)
end
```

The look command simply prints out the detailed description of the current room. The current (game) time is kept in a simple counter (tt), while the score is tracked by counting up the values of the objects the player is carrying (state secrets count more than screwdrivers or pistols).

```
procedure look()
  write(rooms[r].ds)
end

procedure printtime()
  write("elapsed time is ", tt, " minutes.")
end

procedure printscore()
   s := 0
   every s +:= ob[!ca].v
   write("you have ", s, " points for evidence.")
end

procedure restart()
 writes("are you sure you want to restart? ")
 if read() == "yes" then {
    main()
    stop()
 }
 write("Since you don't want to restart...")
end
```

Like the commands, the actions are handled by helper procedures. It is easy to extend this game with new verbs.

```
# verb handlers
procedure action()
  if ob[n].t ~= 2 then {
    case vs of {
        "look": verblook()
        "take"|"get": takeget()
        "go"|"crawl"|"walk": go()
        "open": verbopen()
        "read": verbread()
        "drop": drop()
        "call": call()
        "unscrew": unscrew()
        "spray": spray()
        "push": verbpush()
        "load": verbload()
        "run": verbrun()
```

```
        "drink": drink()
        "eat"|"chew": cheweat()
        "unwrap": unwrap()
        "talk": talk()
        "shoot": shoot()
        "unlock": unlock()
        "on": onoff("on")
        "off": onoff("off")
        default: write("invalid verb ", v)
      }
    }
  else write("You can't ", vs, " ", ns, " yet.")
end
```

To look at an object, you write its detailed string. Objects can in fact point at a linked list of contents or subobjects.

```
procedure verblook()
  write(ob[n].ds)
  repeat {
     if ob[n].l = 0 then return
     n := ob[n].l
     if ob[n].m = r then
          write(ob[n].ds)
  }
end
```

By default, objects can be “taken” and placed in one's inventory. Exceptions are marked in the object's t field.

```
procedure takeget()
  k := ob[n].t
  case k of {
    1: {
        if *ca < 6 then takeit()
        else write("you can't carry anything else")
        }
    2: { write("you can't take ", n, " yet"); return }
    3: { write("silly, that's too heavy to carry"); return }
    4: { write("that's ridiculous!"); return }
    5: { write("you can't take ", n, " yet"); return }
    default: {
        write("invalid take code for object ", ob[n].as,ob[n].t)
        return
      }
    }
end

procedure takeit()
```

```
  if ob[n].m = 100 then { write("you already have it"); return }
  write("taken.")
  ob[n].m := 100
  put(ca, n)
end
```

Opening an object is one of the most complex actions in the game, since several different types of objects can be opened with different effects.

```
procedure verbopen()
  case ns of {
    "door": {
        if ob[12].t = 4 & ob[13].t = 4 then {
          write("opened")
          rooms[2].e[1] := 3
          }
        else if ob[12].t = 5 then write("the door is locked.")
        else if ob[13].t = 5 then
          stop("You didn't disconnect the alarm. It goes off and the\n",
                "police come and arrest you. Game over.")
        else write("can't get through door yet")
        }
    "briefcase": {
        write("combination ")
        cs := read()
        if cs == "2-4-8" then {
          write("opened")
          ob[18].ds ||:="parts of an rr-13 rifle are inside the padded case."
          }
        else write("sorry you don't have the right combination")
        }
    "safe": {
        write("combination ")
        cs := read()
        if cs == "20-15-9" then {
          write("opened")
          ob[44].l := 45
          ob[45].t := 1
          ob[44].ds ||:= "   inside is"
          }
        else write("sorry you don't have the right combination")
        }
    "cabinet": {
        write("opened")
        if n = 49 then {ob[51].t := 1; ob[49].l := 51; rooms[10].ds ||:= " open"}
        else  {ob[38].t := 1; ob[37].l := 38; rooms[8].ds ||:= " open"}
        }
```

```
    "umbrella": {
        stop("you stab yourself with the tip, which is a poisoned dart.\n",
            "you are rushed to the hospital, but it is no use.\n",
            "Game over.")
        }
    "drawer": {
        write("opened")
        ob[21].l := 57
        ob[57].t := 1
        }
    default: {
        write("A ", ns, " can't be opened.")
        }
    }
end
```

Reading secret messages is an important part of the clue finding in CIA.

```
procedure verbread()
  case n of {
  (r = 3) & 16: {
    write("The telephone bill is made out to 322-9678 -V.Grim, P.O. X\n",
          "Grand Central Station, NYC\n", “The amount is $247.36 _
          for long distance charges to Washington DC")
    return
    }
  20: {
    write("You can just make out this message: HEL-ZXT.93.ZARF.1")
    return
    }
  23: {
    write("The bill is made out to 322-8721, Dr. Vladimir Griminski",
          "14 Parkside Avenue - NYC.\n",
          "The bill is for $68.34 for mostly local calls.")
    }
  25: write("322-8721")
  30: write("322-9678")
  42: write("20-15-9")
  56: write("322-8721")
  default: write("There is nothing to read.")
  }
end
```

To drop an object, you tell it what room it is now in (set its .m field) and delete it from the carry list.

```
procedure drop()
  every i := 1 to *ca do
```

```
      if n = ca[i] then {
            ob[ca[i]].m := r
            delete(ca, i)
            write("dropped")
            return
      }
   write("You aren't carrying a ", ns)
end
```

Phoning home in this game allows you to check whether you've solved the puzzle yet or not. This game dates to before cellphones!

```
procedure call()
   if n = 53 & (r = (5 |6 | 9)) then {
      write("Ring...ring")
      write("Hello, agent.  This is your control speaking.")
      write("List your tangible evidence.")
      ll := 0
      li := [ ]
      if get_evidence() >= 40 then {
          write("Fantastic job!!")
          write("We'll be over in a flash to arrest the suspect!")
          tt +:= 6
          if tt > maxtime then stop("sorry... you ran out of time")
          write(" -----------------")
          write("Ambassador Griminski arrives home at 10:30 to find\n",
                "operatives waiting to arrest him.")
          write(" -----------------")
          write("You are handsomely rewarded for your clever sleuthing.")
          write("You solved the mystery in ", tt, " minutes")
          exit()
          }
      }
   else if n ~= 53 then write("it's no use to call ", ns)
   else write("You are not near a phone")
end
```

Listing tangible evidence requires that the user remember what they are carrying.

```
procedure get_evidence()
   local ev := 0
   repeat {
      rs := read()
      if rs == "" then return ev
      every i := 1 to *ca do {
          if rs == ob[ca[i]].as then {
             if !li = ca[i] then {
                write("you already said ", rs)
```

```
            break next
            }
          ev +:=ob[ca[i]].v
          put(li, ca[i])
          break next
          }
        }
    write("You're not carrying a ", rs)
    }
end
```

Several of the remaining verbs solve unique puzzles that are part of the game's challenge. Reading the source code gives spoilers.

```
procedure unscrew()
  if n = 13 then {
    if ob[!ca].as=="screwdriver" then {
        write("The alarm system is off.")
        ob[13].t := 4
        ob[13].ds := "The alarm system is disabled."
        return
        }
    write("you have nothing to unscrew with")
    }
  else write("you can't unscrew a ", ns)
end

procedure spray()
  if n = (13|10) then {
    if !ca = 10 then {
        write("The dog is drugged and falls harmlessly at your feet.")
        rooms[3].e[1] := 5
        rooms[3].e[2] := 9
        rooms[3].e[4] := 4
        rooms[3].ds[-31:0] := " The drugged dog is on the floor."
        ob[14].ds := "The fierce doberman is drugged on the floor."
        delete(ca, i)
        write("The drug is used up and is no longer in your inventory.")
        return
        }
    write("you have nothing to spray with")
    }
 else {
    write("you can't spray a ", ns)
 }
end

procedure verbpush()
```

```
  if n = 26 then {
     write("The panel pops open to reveal the presence of a \n",
           "previously hidden room.")
     rooms[5].e[2] := 6
     ob[26].ds ||:= "A hidden room can be seen behind one panel."
    }
  else
     write("It doesn't do any good to push a ", ns)
end

procedure verbload()
   if n = 28 then {
      if ob[28].m = 6 then {
           write("The program is already loaded.")
      }
      else write("That won't help you.")
     }
   else write("can't load a ", ns)
end

procedure verbrun()
   if n = 28 then {
     if ob[31 | 32 | 33].t = 5 then {
         write("The computer can't run the program yet.")
         return
       }
     ob[28].t := 1
     write("The program dials a Washington D.C. number.\n",
           "A message appears on the monitor.\n")
     writes("PLEASE LOG IN: ")
     cs := read()
     if cs == "HEL-ZXT.93.ZARF.1" then {
         write("The following message appears on the monitor.\n")
         write("  WELCOME TO THE U. S. DEPARTMENT OF DEFENSE")
         write("     RADAR RESISTANT AIRCRAFT PROGRAM.")
         write("ALL INFORMATION ON THIS SYSTEM”)
         write(“IS CLASSIFIED TOP SECRET.")
       }
     else if g = 0 then {
         g := 2
         write("\n\nINVALID LOGON CODE\n\n")
         write("The screen goes blank. You hear footsteps.\n",
               "Griminski looms in the doorway with an 8mm Luger in hand.")
         write("You'd better have the PPK-3 pistol or you're doomed.")
         input()
         if b~= 0 then {
           parsing()
```

```
            if vs == "shoot" & (n=(8 | 58)) then {
               if shoot(1) === "return" then
                    return
               }
             }
          write("It's hopeless! Griminski fires....")
          stop("You crumple to the floor.  End of game.")
        }
     else write("INVALID LOGON CODE")
     }
  else write("you can't run a ", ns)
end
```

Beware, the espionage business is dangerous!

```
procedure drink()
  if n = 36 then {
     write("You are poisoned.")
     stop("You stagger to the phone and call the ambulance. Game over.")
     }
  write("You can't drink ", ns)
end

procedure cheweat()
  if n = (39 | 54) then {
     write("You fool! These are cyanide capsules.")
     stop("You fall to the floor and die in agony.  Game over.")
     }
  if n = 45 then {
     write("You idiot! The gum is a plastic explosive.")
     stop("You have just blown yourself to smithereens. Game over.")
     }
  write("You can't ", vs, " ", ns)
end

procedure unwrap()
  if n = 45 then {
     write("The wrapper conceals a tiny strip of microfilm.")
     ob[46].t := 1
     }
  else write("It doesn't help to unwrap ", ns)
end

procedure talk()
  if n = 14 then write("He doesn't speak English.")
  else write("That won't help you.")
end
```

Shooting the gun is really a last resort, intended to be used when the comes home while you are still in the house.

```
procedure shoot(x)
   if \x | (n=(8 | 14 | 58)) then {
      every i := 1 to *ca do {
          if ca[i] = 8 then {
             if r = 3 & n = (8 | 14) then {
                write("The dog bites your hand.")
                return
                }
             if r ~= 6 then {
                write("That just makes a big mess.")
                return
                }
             if g ~= 2 then {
                write("That won't help.")
                return
                }
             write("Your shot grazes his forehead. He crashes to the floor")
             write("You have time to gather more evidence to apprehend him.")
             g := 1
             rooms[6].ds ||:=" Griminski is lying unconscious on the floor."
             }
          }
      if r=6 & g = 2 then {
          write("You don't have the pistol. Anything else is too slow.")
          fail
          }
      write("You have nothing to shoot with.")
      fail
      }
   else write("That won't help")
end

procedure unlock()
   if n = 12 then {
      if ob[!ca].as == "key" then {
         write("Unlocked.")
         ob[12].t := 4
         return
           }
      write("You have nothing to use to unlock.")
      }
   else
      write("You can't ", vs, " a ", ns)
end
```

```
procedure onoff(o)
   case n of {
        31: m := 137
        32: m := 57
        33: m := 111
        default: { write("You can't turn ", o, " a ", ns); fail }
   }
   if ob[n].ds[-3:0] == ("off"|" on") then {
        while ob[n].ds[-1] ~== "o" do ob[n].ds[-1] := ""
        ob[n].ds[-1] := ""
   }
   ob[n].ds ||:= " " || o
   write(o, ".")
   if o=="on" then ob[n].t := 3 else ob[n].t := 5
end
```

The Adventure Shell

Many people like text adventures more than their command-line interface shells, enough so that it was suggested as an alternative interface. Certainly it has the potential to be more user friendly. For example, why should I want to “cd ~” when I could “go home” instead? Why should I “ls -la” or “dir /w” when I could “look”?

Chapter 8: Resource Simulation

Resource simulation games are another ancient genre that spun off of the early use of the computer for running simulations.

Hamurabi

Hamurabi is one of the oldest and simplest games ever written. It simulates the king of Babylon giving orders regarding the management of the city's agriculture in order to keep everyone fed and the empire growing. Much of the charm of this game is derived from the sardonically obsequious tone of its narrator, your nameless prime minister. The discrete simulation is coarse-grained, with one turn per year.

Our version of Hamurabi is translated from BASIC into a simplistic object-oriented program with one singleton class. A more advanced simulation game like Civilization might start by creating multiple instances to simulate different cities. Hamurabi executes by creating an instance of class Babylon() and invoking a play() method.

```
procedure main()
   write("Try your hand at governing ancient Sumeria\n_
        successfully for a 10 year term of office.")
  Babylon().play()
end
```

The header comments are preserved for historic pedigree purposes.

```
# Hamurabi
# Converted from the original FOCAL program and modified
# for EDUSYSTEM 70 by David Ahl, Digital.
# Modified for 8K Microsoft BASIC by Peter Turnbull.
# Then translated to C# by Bill Lange on codeproject.com.
# Translated to Unicon by Clint Jeffery
```

The class in this program simulates a city very very crudely, with a small number of integer parameters including population, size (in acres), and how much grain is in the granary.

```
class Babylon (
      Z, # Year
      D, # People died this turn
```

```
        D1, # People died, cumulative
        I, # Immigrants
        P, # Population
        P1, # Starvation percentage
        A, # Acres
        Y, # Yield
        E, # Eaten
        S, # Store
        C, # Random yearly yield modifier
        Q, # Input
        L, # Acres per person
        H
       )
```

If you look at the original BASIC source, you will never guess that the spaghetti code executes a one-turn = one-year simulation consisting of reporting the state of the country, taking buy (for acres) and sell orders, taking Hamurabi's instructions on how much grain to plant, and calculating the amount of grain harvested.

```
method play()
   repeat {
      report()
      buy()
      sell()
      feed()
      plant()
      harvest()
   }
end
```

The annual report is fairly refreshing. In reports on births and deaths, the current population and grain reserves. The game is over in year 11.

```
method report()
  write("\n\nHamurabi:  I beg to report to you,")
  Z +:= 1
  write("in year ", Z, ", ", D, " people starved; ", I,
        " came to the city.")
```

```
   P +:= I
   if Q <= 0 then {
      P /:= 2
      write("A horrible plague struck!  Half the people died.")
      }

   write("Population is now " , P, ".  The city now owns " ,
         A , " acres.")
   write("You harvested ", Y, " bushels per acre. Rats ate ",
         E, " bushels.")
   write("You now have " , S , " bushels in store.\n")
   if Z == 11 then evaluate()
end
```

The simulation gets interesting when it starts asking for your orders. The price of land is made to vary randomly. Generally, all input is taken using a loop that repeats until the input is legal.

```
method buy()
   C := ?11-1
   Y := C + 17
   write("Land is trading at " , Y , " bushels per acre.")
   repeat {
      writes("How many acres do you wish to buy? ")
      Q := integer(read())
      if Q < 0 then finish("quit")
      else if Y * Q <= S then return
      else {
         write("Hamurabi:  think again. You have only ", S)
         write(" bushels of grain.  Now then,")
         }
      }
end
```

The user (Hammurabi) is only asked if they want to sell acres if they did not buy acres.

```
method sell()
   if Q == 0 then {
      repeat {
```

```
      writes("How many acres do you wish to sell? ")
      Q := integer(read())
      if Q < 0 then finish("quit")
      else if Q < A then {
        A -:= Q; S +:= Y * Q; C := 0
        write()
        break
        }
      else {
        write("Hamurabi:  think again. You own only " , A ,
              " acres.  Now then,")
        }
      }
   else {
     A +:= Q; S -:= Y * Q; C := 0
     write()
     }
end
```

If you don't feed the people enough, they will starve.

```
method feed()
   repeat {
     writes("How many bushels shall we feed your people? ")
     Q := integer(read())
     if Q < 0 then finish("quit")
     else {
       # Trying to use more grain than in the silos?
       if Q <= S then break
       else {
         write("Hamurabi:  think again. You have only")
         write(S , " bushels of grain.  Now then,")
         }
       }
     S -:= Q
     C := 1
     write()
     }
```

```
end
```

If you don't plant enough grain for next year, you will starve then. It takes workers and seed grain to plant acres.

```
method plant()
  repeat {
    writes("How many acres do you wish to plant? ")
    D := integer(read())
    if D == 0 then return
    else if D < 0 then finish("quit")
    # Trying to plant more acres than you own?
    else if D <= A then {
      # Enough grain for seed?
      if D / 2 < S then {
        # Enough people to tend the crops?
        if D < 10 * P then {
          S -:= D / 2
          return
          }
        else {
          write("but you have only " , P ,
                " people to tend the fields. Now then,")
          next
          }
        }
      else {
        write("Hamurabi:  think again. You have only ", S,
              " bushels of grain.  Now then,")
        next
        }
      break
      }
    else {
      write("Hamurabi:  think again. You own only " , A ,
              " acres.  Now then,")
      }
    }
end
```

The harvest method does a lot more than just selecting randomly how much grain per acre is harvested, simulating the highly variable weather conditions. Randomly, the rats are especially bad some years. Population changes are calculated and if things look bad enough, the game ends prematurely.

```
method harvest()
  C := integer(?0 * 5 + 1)
  # A bountiful harvest!!
  Y := C; H := D * Y; E := 0
  C := integer(?0 * 5 + 1)
  if C % 2 = 0 then {
    # THE RATS ARE RUNNING WILD!!
    E := S / C
    }

  S := S - E + H
  C := ?0 * 5 + 1
  # Lets have some babies
  I := integer((C * (20 * A + S) / P / 100 + 1))
  # How many people had full tummies?
  C := integer(Q / 20)
  # Horrors, a 15% chance of plague
  Q := integer(10 * (2 * ?0 - .3))

  if P < C then { D:= 0; return }
  else {
    # Starve enough for impeachment?
    D := P - C
    if D > .45 * P then {
      write("\nYou starved " , D , " people in one year!!!")
      finish("fink")
      }
    else {
      P1 := ((Z - 1) * P1 + D * 100 / P) / Z
      P := C
      D1 +:= D
      return
      }
```

```
    }
end
```

The normal end of the game includes an evaluation of how you did in your ten years as emperor.

```
method evaluate()
  write("In your 10-year term of office, " , P1 ,
        " percent of the population starved per year on _
        average, i.e., a total of", D1 , " people died!!")
  L := A / P

  write("You started with 10 acres/person and ended with ",
      L , " acres/person.\n")

  if P1 > 33 |  L < 7 then finish("fink")
  else if P1 > 10 | L < 9 then finish("nero")
  else if P1 > 3 | L < 10 then finish("notbad")
  else finish("fantastic")
end
```

The exit message you receive depends on your performance.

```
method finish(as)
  case as of {
    "quit": {
      write("Hamurabi:  I cannot do what you wish.")
      write("Get yourself another steward!!!!!")
      }
    "fantastic": {
      write("A fantastic performance!!!  Charlemagne, _
            Disraeli, and Jefferson combined could not _
            have done better!")
      }
    "fink": {
      write("Due to this extreme mismanagement you have _
            not only been impeached and thrown out of _
            office but you have also been declared 'national
            fink' !!")
      }
```

```
      "nero": {
        write("Your heavy-handed performance smacks of _
              Nero and Ivan IV. The people (remaining) find _
              you an unpleasant ruler, and, frankly, hate _
              your guts!")
        }
      "not bad": {
        write("Your performance could have been somewhat _
              better, but really wasn't too bad at all. ",
              integer(P * .8 * ?0) , " people would dearly like _
              to see you assassinated but we all have _
              our trivial problems.")
        }
    }
   stop("So long for now.")
end
```

The class initialization sets parameters up the same each time.

```
initially
    Z := D := I := P := A := Y := E := S := C := Q := D1 := P1 :=
      0
    P := 95
    S := 2800
    H := 3000
    E := H - S
    Y := 3
    A := H / Y
    I := 5
    Q := 1
end
```

Taipan

Coming soon.

Chapter 9: Turn-based Role-Playing

The text adventures described in Chapter 7 model the world as a series of rooms, depicting the spaces with prose text at a coarse granularity (or resolution, if you prefer) of one move = one room. Their imaginary worlds were followed by a more detailed level of play, where some rooms are larger than others and each move might be 5 feet. Turn-based role-playing games such as rogue and nethack created vastly different, tactical fun than text adventure games which were typically mysteries or puzzles. These games used a bird's-eye view from the top to show the scene, often using ASCII art. Later turn-based role-playing games replaced each text character with a sprite, resulting in 2D games that lead to arcade games once they switch from turn-based to real-time.

Pirate Duel

This example turn-based role-playing game pits the player against a series of progressively more ferocious pirates. The version in this book is incomplete, showing a single level against a single pirate foe. Figure 9-1 shows an example screen shot.

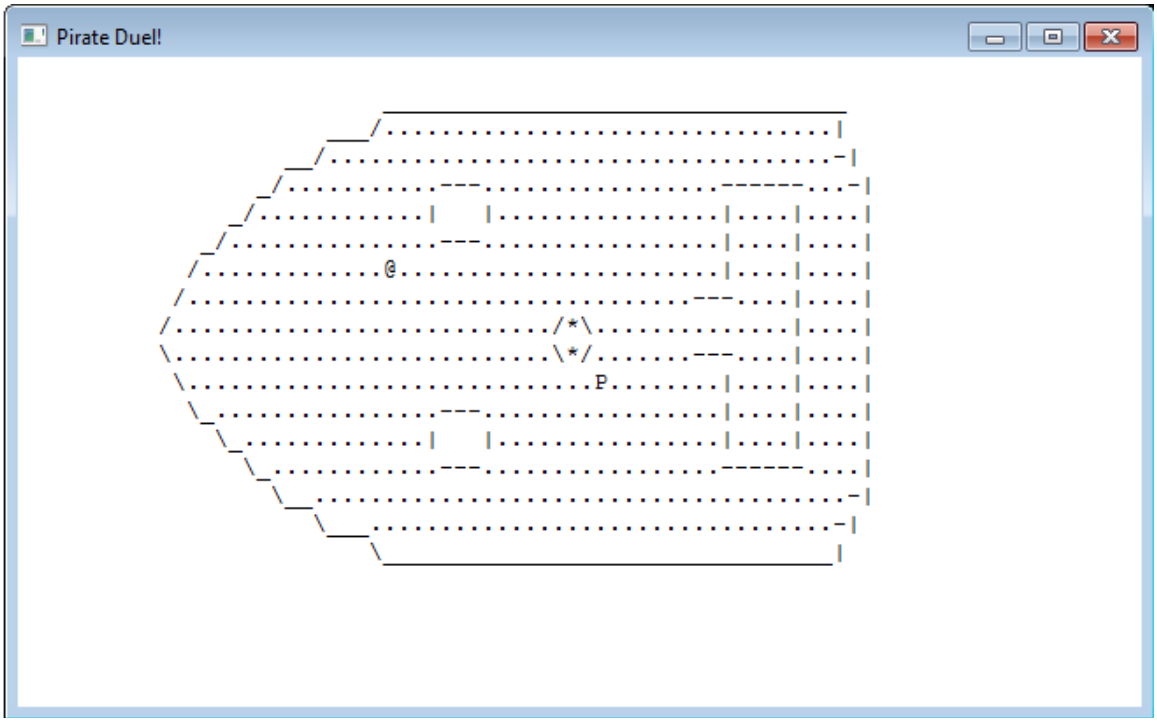

Figure 9-1:
The Pirate Duel

The code for Pirate Duel uses a graphics mode to do its ASCII art at present, which is ironic but portable. Classic text games were written for terminals connected using serial ports, and used non-portable escape sequences to position the cursor, clear the screen and so on. This style of programming has almost disappeared from the modern world of computing. Some printers are still controlled in a similar fashion.

```
#
# piraduel.icn - a simple rogue-like pirate duel game
#
link graphics
global player, world

# open a window, create the world, create the player, draw
procedure main()
   &window := open("Pirate Duel!","g",
                   "rows=25","columns=80")
   world := World()
   player := character()
   world.add_npc(?"PO") # create an enemy pirate or orc
   world.draw()

   repeat {
      player.turn()
      if (!world.npcs).turn() then break
      world.draw()
      }
   world.draw()
   while Event() ~== "q"
   stop()
end
```

Patterned after the dungeon structure from classic Dungeons and Dragons, a World object has a list of levels, within which all action transpires on the current level. World methods delegate their responsibilities to the current level object. The Pirate Duel world is preinitialized to two levels, one read from a text file named ship.dat, and the second one generated randomly.

```
class World(levels, cur)
   method npcs()
      return levels[cur].npcs
   end
   method grid()
      return levels[cur].grid
   end
   method add_npc(s)
      levels[cur].add_npc(s)
   end
   method draw()
      levels[cur].draw()
   end
   method level()
      return levels[cur]
   end
initially
   levels := [ AALevel("ship.dat"), PGLevel() ]
   cur := 1
end
```

Level is a parent class with core functionality consisting of management of a map represented as a list of strings named grid, and a list of non-player (computer-controlled) characters.

```
class Level(grid, npcs)
   method add_npc(s)
      put(npcs, npc(s))
   end
   method draw()
      every r := 1 to 24 do {
         GotoRC(r,1)
         WWrites(left(grid[r], 80, " "))
         }
      grid[1] := ""
   end
initially
```

```
   npcs := []
end
```

There are three subclasses of Level. The shape and contents of an AALevel is read from a file. An instance of OneBigLevel consists of a single gigantic room. A PGLevel is procedurally generated, making them ideal to add content and replayability with gameplay in between signature levels that advance the story line.

```
class AALevel : Level(filename)
initially
   self.Level.initially()
   grid := [ ]
   f := open(filename) | stop("can't open ", image(filename))
   while line := read(f) do {
      put(grid, line)
   }
   close(f)
end

class OneBigLevel : Level()
initially
   self.Level.initially()
   grid := list(24)
   every !grid := repl(" ", 80)
   grid[2][2 : -1] := repl("_",78)
   every i := 3 to 22 do grid[i][2 : -1] := "|" || repl(".",76) || "|"
   grid[23][2 : -1] := repl("_",78)
   grid[?21+1][?76+1] := "<"
end

record room(row,col,width,height)

class PGLevel : Level()

   method plot(x, y)
      saved := grid[y,x]
      grid[y,x] := "*"
```

```
      draw()
      Event()
      grid[y,x] := saved
      if any('.<>', grid[y,x]) then fail
      grid[y,x] := "."
   end

   method DrillFrom(x1,y1,x2,y2)
      while x1 < x2 do { plot(x1, y1); x1 +:= 1 }
      while x1 > x2 do { plot(x1, y1); x1 -:= 1 }
      while y1 < y2 do { plot(x1, y1); y1 +:= 1 }
      while y1 > y2 do { plot(x1, y1); y1 -:= 1 }
   end

initially
   roomlist := []
   self.Level.initially()
   grid := list(24)
   every !grid := repl(" ", 80)
   rooms := ?4+2
   r := 1
   while r < rooms do {
      # generate a random location and size for the new room
      rm := room(?19, ?72, ?30+3, ?4+3)
      rm.width >:= 80-rm.col-1
      rm.height >:= 24-rm.row-1

      # check against past rooms
      every oldr := !roomlist do {
         if oldr.row-2 <= rm.row+(0|rm.height) <=
oldr.row+oldr.height+2 &
            oldr.col-2 <= rm.col+(0|rm.width) <=
oldr.col+oldr.width+2 then {
               # a corner is in old room
               break next
            }
         }
      grid[rm.row][rm.col +: rm.width] := repl("_", rm.width)
      every i := rm.row+1 to rm.row+rm.height-1 do
```

```
        grid[i][rm.col +: rm.width] := "|" || repl(".",rm.width-2) ||
"|"
      grid[rm.row+rm.height][rm.col +: rm.width] := repl("_",
rm.width)

      if r>1 then {
         # pick a random previous room, drill a cooridoor to it.
         or := ?*roomlist
         oldr := roomlist[or]
         centerrow := rm.row + rm.height/2
         centercol := rm.col + rm.width/2
         destrow := oldr.row + oldr.height/2
         destcol := oldr.col + oldr.width/2
         DrillFrom(centercol,centerrow,destcol,destrow)
         }

      put(roomlist, rm)
      r +:= 1
      }
   # pick a random location in room 1 to have a stairs up
   uprow := roomlist[1].row + ?(roomlist[1].height-1)
   upcol := roomlist[1].col + ?(roomlist[1].width-1)
   grid[uprow, upcol] := "<"
   # pick a random location in room n to have a stairs down
   dnrow := roomlist[-1].row + ?(roomlist[-1].height-1)
   dncol := roomlist[-1].col + ?(roomlist[-1].width-1)
   grid[dnrow, dncol] := ">"
end


# the player character is modeled with a location, name, etc.
class character(x, y, name, hitpoints, armorclass, damage)
   method hittest(enemy)
      die := ?20
      if die > 10 + (9 - enemy.armorclass) then {
         enemy.hitpoints -:= ?damage
         return
         }
   end
```

```
method attack(enemy)
  if hittest(enemy) then {
    world.grid[1]||:= "You have hit the " || enemy.name || ". "
    enemy.awake := 1
    if enemy.hitpoints < 1 then {
      world.grid()[1] ||:= "You have defeated the " ||
                           enemy.name
      enemy.die()
      }
    }
  else {
    world.grid()[1] ||:= "You missed the "||enemy.name || ". "
    if / (enemy.awake) := (?1=1) then
      world.grid()[1] ||:= "The enemy awakes. "
    }
end
method move(newx,newy, letter : "@")
  world.grid[()y,x] := "."
  x := newx; y := newy
  world.grid()[y,x] := letter
end
method trymove(newx, newy)
  case world.grid()[newy, newx] of {
    "." : move(newx, newy)
    ">" : {
      if world.cur < *world.levels then {
        let := world.grid()[y,x]
        world.grid()[y,x] := "."
        world.cur +:= 1
        place(let)
        }
    }
  default: {
    if (m := !world.npcs()) & m.y = newy & m.x = newx then
      attack(m)
    }
  }
end
method turn()
```

```
      e := Event()
      case e of {
         "q": { stop("goodbye") }
         "h": { trymove(x-1,  y) }
         "l": { trymove(x+1,  y) }
         "k": { trymove( x , y-1) }
         "j": { trymove( x , y+1) }
         "y": { trymove(x-1, y-1) }
         "u": { trymove(x+1, y-1) }
         "b": { trymove(x-1, y+1) }
         "n": { trymove(x+1, y+1) }
        }
   end

   # keep trying random placement until you get a legal spot
   method place(c, lvl)
   local grid := (\lvl).grid | world.grid()
      repeat {
         x := ?80
         y := ?24
         if grid[y,x] == "." then {
            grid[y,x] := c
            break
            }
         }
   end

initially
   place("@")
   hitpoints := 12
   damage := 6
   armorclass := 6
end

class npc : character(awake, letter)
   method die()
      world.grid()[y,x] := "."
      every 1 to *world.npcs() do {
          put(world.npcs(), self ~=== pop(world.npcs()))
```

```
      }
   end
   method distance(enemy, x, y)
      return abs(y-enemy.y)^2 + abs(x-enemy.x)^2
   end
   method attack(enemy)
      if hittest(enemy) then {
         world.grid()[1] ||:= "The " || name || " has hit you. "
         if enemy.hitpoints < 1 then {
            world.grid()[1] ||:= "You were defeated by the "|| name
            return
            }
         }
      else
         world.grid()[1] ||:= "The " || name || " missed you. "
   end
   method turn()
   local dist, minx, miny
      if /awake then fail
      every row := y-1 to y+1 do
        every col := x-1 to x+1 do {
           if world.grid()[row,col] == "@" then {
              return attack(player)
              }
          }
      dist := distance(player, x, y)
      every row := y-1 to y+1 do
        every col := x-1 to x+1 do {
           if world.grid()[row,col] == "." &
              dist >:= distance(player,col,row) then {
              minx := col
              miny := row
              }
          }
      if \minx then move(minx, miny, letter)
   end
initially(c)
static nametable
   /nametable := table("O","Orc","P","Pirate")
```

```
   letter := \c | ?"PO"
   name := nametable[letter]
   if awake := (?3 = 1) then
      world.grid()[1] ||:=  "The " || name || " is awake. "
   hitpoints := ?8
   armorclass := 6
   damage := 8
   place(letter)
end
```

Exercises

1. Change procedural generation so that it performs the vertical drilling before the horizontal about 25% of the time.

2. Add more kinds of monsters.

3. Add weapons, armor, or other magical or non-magical items.

Chapter 10: Paddle Games

The simplest and earliest arcade video games involve hitting bouncing balls with paddles (or if you prefer, rackets).

Ping

Ping is a remake of the original classic VideoGame, Atari's **Pong**. It is stunning to think that the program for this game is around 80 lines of code in Unicon, short enough that you can type it in in just a few minutes, but Pong made many millions of dollars in quarters, arcade tokens, and game cartridge sales. Ping starts by opening a window (640 pixels wide, 480 high) with a black background color and a white foreground color. The foreground color is the color that will be used in drawing operations.

A special name called &window will be used to remember the window the game is playing in. The reason it is special is because when this name is used, we do not have to refer to it in graphics calls, it is the *default window*.

```
procedure main()
   &window := open("Ping","g",
                  "size=640,480", "bg=black", "fg=white")
```

Ping draws rectangles for the players' paddles, the ball, and a center stripe. The rectangles are filled (solid white) using FillRectangle(). The function DrawRectangle() would instead draw an outline of the shape.

```
   FillRectangle(10,220,10,40)  # left player
   FillRectangle(318,0,4,480)   # center stripe
   FillRectangle(620,220,10,40) # right player
   FillRectangle(340,240,10,10) # ball
```

Ping has to remember where the ball is, and what direction it is heading. dx and dy tell how to change the ball's position on each turn.

```
   x := 340
   y := 240
   dx := 1
   dy := 0
```

Ping also has to remember the players' paddle positions.

```
p1x := 10
p1y := 220
p2x := 620
p2y := 220
```

Lastly, Ping has to keep score of who wins how many points.

```
p1score := 0
p2score := 0
```

After all of these preliminaries, the Ping program does the same thing over and over...

```
repeat {
```

First it erases the ball at the old position. It redraws the center stripe just in case part of it got erased.

```
    EraseArea(x,y,10,10)
    FillRectangle(318,0,4,480)
```

It calculates the ball's new position by adding dx to x and dy to y.

```
    # calculate new position
    x +:= dx
    y +:= dy
```

If the ball hits player 1's paddle, it bounces off. Bouncing off reverses the delta x so it is heading at the other player. The angle of the the return shot (determined by dy) is proportional to the distance from the center of the ball (y+5) to the center of the paddle (p1y+20). The "divides by 15" part was determined by guessing and replaying until it "felt OK".

To tell if the ball "hit", you check whether x and y are inside the paddle rectangle. Actually, to be picky you check whether either left corner of the ball hits the paddle. The coordinates are (x,y) and (x,y+10) but you have to check x and y separately.

```
    if p1x <= x <= p1x + 10 &
       p1y <= (y | y+10) <= p1y + 40 then {
         dx := dx * -1
         dy +:= ((y+5) - (p1y+20))/15
       }
```

You do the same thing for player 2's paddle, except check the right corners of the paddle.

```
if p2x <= x+10 <= p2x + 10 &
   p2y <= (y | y+10) <= p2y + 40 then {
     dx := dx * -1
     dy +:= ((y+5) - (p2y+20))/15
    }
```

If the ball hits the top of the screen it bounces down. If it hits the bottom of the screen it bounces up.

```
if y <= 0 then dy *:= -1
if y+10 >= 480 then dy *:= -1
```

After drawing the ball, on Linux at least, you must call WSync() to make sure the X Window System receives your command immediately instead of waiting until later.

```
# draw ball at new position
FillRectangle(x,y,10,10)
WSync()
```

The last thing in each step is to check if the ball reaches either the left or right screen edge, and if it does, score a point for the appropriate player.

```
if x + 10 > 640 then {
   dx := dx * -1
   p1score +:= 1
   dy := 0
   GotoXY(10,10)
   writes(&window, p1score)
   }
if x <= 0 then {
   dx := dx * -1
   p2score +:= 1
   dy := 0
   GotoXY(600,10)
   writes(&window, p2score)
   }
```

A small delay helps so that players can hope to keep up with the action. The units here are 1/1000 of a second.

```
    delay(8)
```

At each step, the program needs to read the players' paddle commands. Player 1 uses "q" and "a" keys to go up and down. Player 2 uses "p" and "l" to go up and down. Pending() tells whether any input is waiting; it returns a list of events. The * operator checks the size of the list, and if the size is more than 0 there is a user event to read.

```
    if (* Pending() > 0) then {
      e := Event()
      if e == ("q"|"a") then EraseArea(p1x,p1y,10,40)
      if e == "q" then p1y -:= 5
      if e == "a" then p1y +:= 5
      if e == ("q"|"a") then FillRectangle(p1x,p1y,10,40)
      if e == ("p"|"l") then EraseArea(p2x,p2y,10,40)
      if e == "p" then p2y -:= 5
      if e == "l" then p2y +:= 5
      if e == ("p"|"l") then FillRectangle(p2x,p2y,10,40)
    }
```

The program finishes its "repeat" instruction with a } and then is at its end. But, this means it really goes on forever; what happens when the players get tired?

```
    }
end
```

Brickout

Brickout is a remake of the classic Atari game, Breakout. This program is directly based on the Ping program, with a few exceptions. The paddle moves horizontally across the bottom of the screen, and the ball bounces up and down. The most important difference, however, are a large number of bricks in the upper portion of the screen. The object of the game is to knock the ball into each of these bricks in order to destroy them and clear the screen. The following figure is an example of the screen from Brickout.

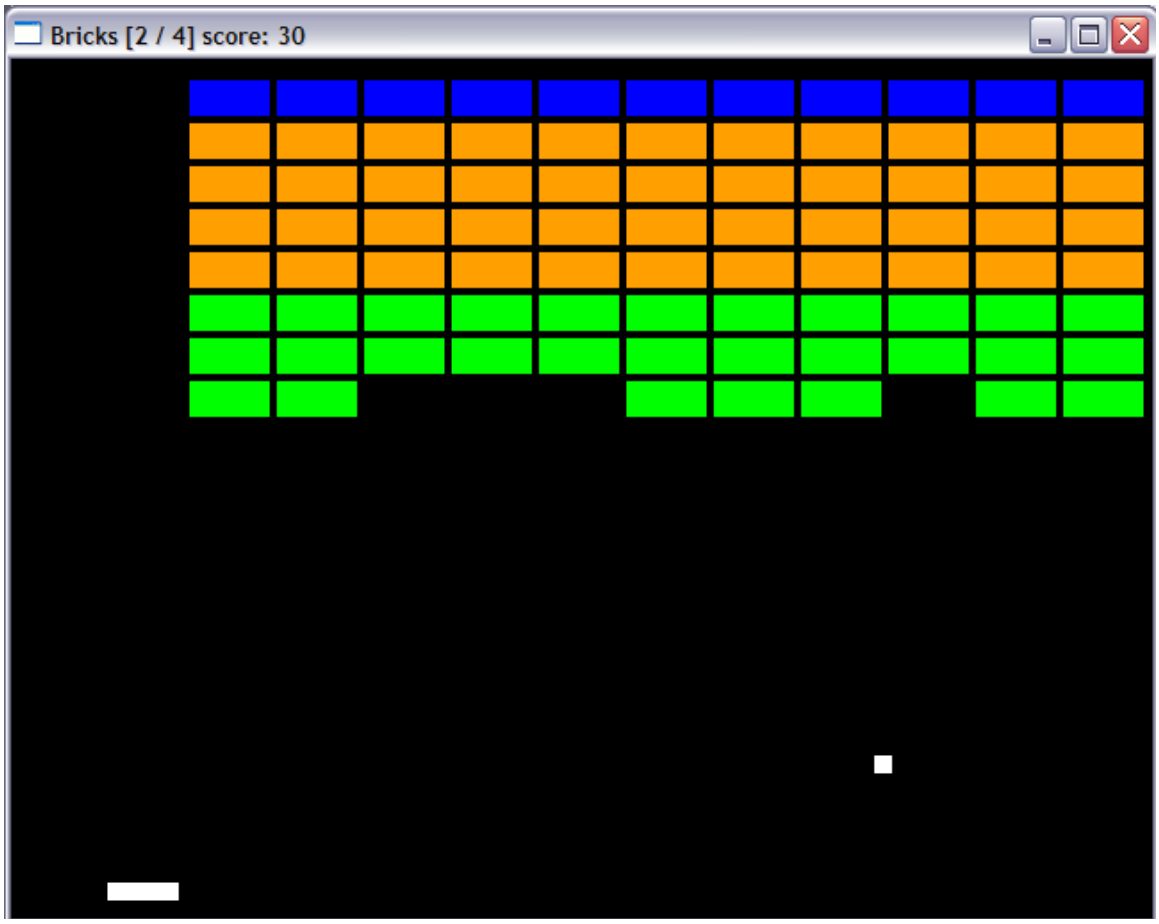

Figure 10-1:
The Brickout program.

The Brickout program starts by opening a window almost identically to the Ping program.

```
procedure main()
   &window := open("Brickout", "g", "size=640,480",
                                "fg=white","bg=black")
```

The bricks are organized into rows and columns, similar to the checkerboard from Chapter 6. However, the contents of each brick position will be a Brick object instead of a word. Brick objects don't behave like numbers or words, they behave like bricks. Actually, each brick will keep track of its (x,y) position and color, and know how to do exactly four things: draw itself, erase itself, report whether or not a ball hits it, and report how many points it is worth. The details of the Brick type will be given soon, but for now, creating a brick is done by calling Brick(x, y, color).

```
   bricks := [ ]
```

```
every row := 0 to 7 do {
   if row = 0 then color := "blue"
   else if 1 <= row <= 4 then color := "yellow orange"
   else if row > 4 then color := "green"
   put(bricks, [ ] )
   every column := 0 to 12 do {
      b := Brick( column * 49 + 2, row * 24 + 12, color)
      put(bricks[ -1 ], b)
      }
}
```

A lot of the initialization of Brickout looks just like Ping. The paddle is drawn short and wide at the bottom of the screen instead of tall and thin. The ball motion starts going down instead of right.

```
Fg("white")
FillRectangle(320,460,40,10) # paddle
FillRectangle(320,260,10,10) # ball
x := 320
y := 260
dx := 0
dy := 1
px := 320
py := 460
score := 0
lives := 0
```

A big difference between Brickout and Ping is seen in the drawing of the bricks on the screen. This could be done simply enough using FillRectangle(), but Unicon has a notation for this: if b is a Brick, b.draw() asks that Brick to draw itself. If bricks is a list of lists of bricks, !bricks will produce all the lists of bricks, and !!bricks will produce all the bricks in all those lists, so to draw all of the bricks one may write:

```
every (!!bricks).draw()
```

Having drawn the paddle, ball, and bricks, the program enters its main loop, again based closely on that of the Ping program.

```
repeat {
   # erase ball at old position
   EraseArea(x,y,10,10)
```

```
# calculate new position
x +:= dx
y +:= dy
# draw ball at new position
FillRectangle(x,y,10,10)
WSync()
# bounce off paddle
if px <= x <= px + 40 &
    py <= y+10 <= py + 10 then {
      dy := dy * -1
      dx +:= ((x+5) - (px+20))/10
    }

# bounce off left, right, or top wall
if x <= 0 then dx := dx * -1
if x+10 >= 640 then dx := dx * -1
if y <= 0 then dy := dy * -1

# take a life when the ball hits the bottom edge
if y + 10 > 480 then {
   dy := dy * -1
   lives +:= 1
   dx := 0
   if lives = 4 then {
      WAttrib("label=Bricks [game over, q to quit] score: " ||
                score)
       while *Pending() > 0 do Event()
       while Event() ~== "q"
       exit(0)
       }
   else
      WAttrib("label=Bricks [" ||  lives || " / " || "4] score: " ||
                score)
   }
```

The part where Brickout really diverges from Ping is in the handling of the bricks. Given the ball's (x,y) position, every brick is checked to see whether it was hit by the ball, and if so, the score is updated and the brick is erased. With the Brick type, this is

accomplished by using three aptly named messages. b.hittest(x,y) succeeds if the ball at (x,y) will hit Brick b. b.score() produces the points awarded if b is hit. b.erase() removes the brick from the field.

```
every b := !!bricks do {
   if b.hittest(x,y) then {
      score +:= b.score()
      b.erase()
      WAttrib("label=Bricks [" ||  lives ||
                  " / "||"4] score: "||score)
      dy := dy * -1
   }
}
```

An early version of this program used a call to delay() to slow the program down enough for the human to keep up. This might work well on some platforms, but if the operating system timer is too coarse, delay() may wait longer than desired and make the game too slow. In that case, one can get a smaller delay by simply telling the program to do something stupid for a little while, such as this instruction to count to 75000. This isn't a very good solution, since the delay will get smaller and smaller over time as CPU's and compilers get faster and more efficient.

```
# use a delay smaller than delay(1); may need to adjust
# for CPU speed and implementation efficiency
every 1 to 75000
```

The easiest way to give the player smooth control over the position of the paddle is to have it follow the mouse. Keypress events and even mouse clicks and drags may not provide this information as fast as the program can explicitly request it. QueryPointer() generates two results (the x and y locations of the mouse pointer) of which this program only needs the first value.

```
# move paddle left or right in response to user input
EraseArea(px,py,40,10)
px := QueryPointer()
FillRectangle(px,py,40,10)
}
Event()
```

```
end
```

The program is finished except for one small problem: there is no Brick data type in Unicon. In order to create one, a programmer declares a *class* to model the behavior of bricks in the game. A class is a user-defined type of value that can be stored in variables and used in many ways similar to built-in types such as numbers and words. Instead of using arithmetic operators such as + - * or /, a class defines a set of named operations called methods to manipulate values of that class. The methods are basically a set of procedures that work on Brick objects. For each class there can be many distinct occurrences, or *instances*.

The class Brick() used in this program is given below. Each instance will have its own, separate x, y, and color, but all instances will use the same code to determine their behavior.

```
class Brick(x, y, color)
  method score()
    if color == "blue" then return 10
    else if color == "orange" then return 5
    if color == "green" then return 1
  end
  method draw()
    Fg(color)
    FillRectangle(x,y,45,20)
  end
  method erase()
    color := "black"
    EraseArea(x,y,45,20)
  end
  method hittest(ballx, bally)
    if color == "black" then fail
    if x-10 < ballx < x + 45 &
      y-10 < bally < y + 20 then return
    fail
  end
end
```

Exercises

1. Fix the paddle programs to check and prevent a player from moving their paddle above the top or below the bottom of the screen.

Chapter 11: Sesrit

To fully understand some of the graphics facilities used in this chapter you may wish to consult the book Graphics Programming in Icon. This chapter shows you:

- How to animate objects within a graphics screen.
- How to represent on-screen objects with internal data structures.
- Steps to take in an object-oriented design for a complex game.
- An example of a moderately sophisticated custom user interface design.

The Gameplay of Falling Blocks

Sesrit is a game written by David Rice back when he was a high school student, based on the classic game of Tetris. A free software program called *xtetris* inspires Sesrit's look and feel. Sesrit is written in about five hundred lines of Unicon.

In Sesrit, like Tetris, pieces of varying shape fall from the top of a rectangular play area. Sesrit, however, uses randomly generated shapes, making it more difficult than tetris, which uses a small, fixed set of shapes for its pieces. The object of the game is to keep the play area clear so that there is room for more objects to fall. The play area is a grid of cells, ten cells wide and thirty cells high. Sesrit pieces fall at a rate that begins slowly, and increases in speed as play progresses.

A piece stops falling when it reaches the bottom row, or when one of its cells has another piece directly beneath it, preventing its fall. When a piece stops falling, a new randomly selected piece is created and starts to fall from the top of the play area. If the cells in a given row completely fill, they dissolve, and all cells above that row drop down.

The user moves the falling piece left or right by pressing the left or right arrow keys. The user may rotate the piece clockwise or

counterclockwise by pressing the up arrow or down arrow, respectively. The spacebar causes the falling object to immediately drop as far as it can. Figure 9-1 shows a sample screen from *Sesrit*.

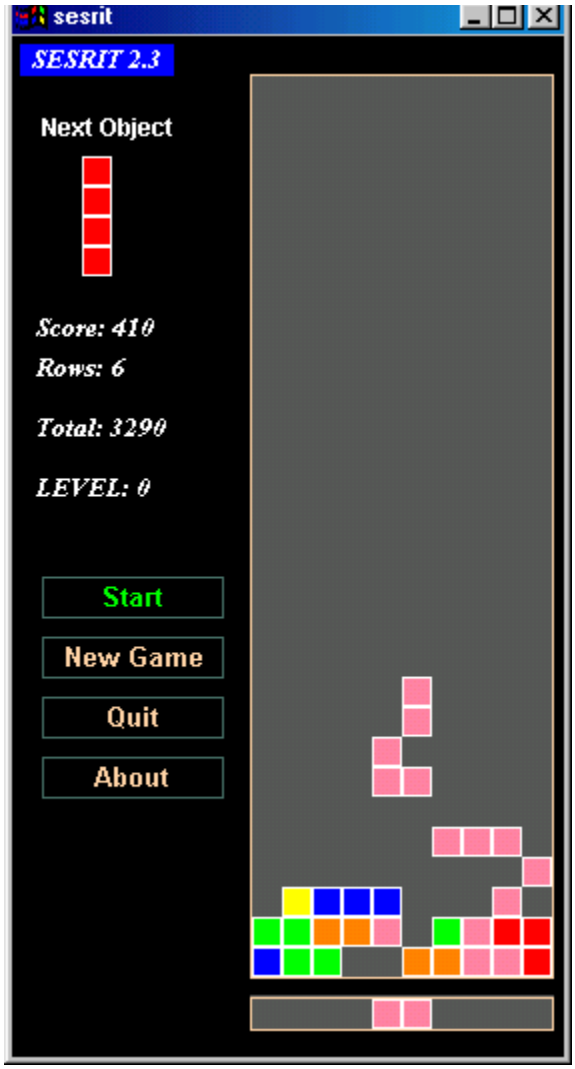

Figure 11-1:
An example screen image from the Sesrit game.

This section presents the key elements of *Sesrit*'s design and implementation. A few of the more mundane procedures are not described. The complete *Sesrit* source code is on the book's web site. Like most games with a graphical user interface, *Sesrit* starts with link declarations that give it access to graphics library procedures and to a function that randomizes program behavior on each run. *Sesrit* also links in the Internet high score client procedure from Chapter 15 of the book “Programming with Unicon”. Including the file "keysyms.icn" introduces defined

symbols for special keyboard keys such as the arrow keys used in *Sesrit*.

```
link graphics
link random
link highscor
$include "keysyms.icn"
```

Sesrit has a number of global variables that are used to maintain its internal representation of the game state and screen contents. The global variable L represents the actual playing area contents as a list of lists of cells, where each cell is represented by the string name for the color of that cell. For example, L[12,7] could have the value "red".

Several other global variables are important. A list of two-element lists containing (x,y) coordinates represents the current piece (variable activecells) and the next piece (variable nextpiece). Other global variables maintain details such as the rate at which the pieces fall, and the current score. The global table named colors holds a mapping from colors' string names to window bindings whose foreground colors are set to that color.

```
global activecells, activecellcolor, nextpiece, nextcolor, L,
      colors, score, numrows, level, delaytime, pieceposition,
      button_status, game_status, tot_score
```

Sesrit's procedure drawcell(x,y,color) fills a rectangle of the appropriate color and then (for nonempty cells) draws a white border around it. You can draw a cell such as L[12,7] with a call to drawcell(x, y, L[12, 7]).

```
procedure drawcell(x,y,color)
   FillRectangle(colors[color],x,y,15,15)
   if color ~=== "black" then
      DrawRectangle(colors["white"], x, y, 14, 14)
end
```

The main() procedure of *Sesrit* initializes the graphics and state variables and then executes a loop that allows the user to play one game on each iteration. After each game the user has the option of playing again, or quitting. For each game, the main action is accomplished by a call to procedure game_loop().

```
procedure main()
   init()
   repeat
      if buttons(15, 105, 270, 290, ["Start", "green", 45, 285],
                ["Pause", "red", 40, 285]) == "done" then break
   repeat {
      game_loop()
      init()
      }
end
```

Sesrit performs initialization with a procedure named init(). The first time init() is called, it must do a bit more work than in subsequent calls, so it has an initial section, which starts by opening a window and creating the color table. Initialization would be where you would set the random number seed, if you cared about whether the random sequence was repeatable. Like its predecessor the Icon language, Unicon's random number generator used to use the same sequence each time it executes, which is very helpful for debugging, but not a good feature in games that are supposed to be unpredictable. The Unicon language default behavior was modified for the sake of games to set the random number seed based on the current time, so every game is different.

```
procedure init()
   initial {
      &window := Wopen("label=sesrit","size=276,510",
                                  "posx=20")
      colors := table(&window)
      every c := ("blue"|"yellow"|"cyan"|"green"|"red"|"white"|
                  "red-yellow" | "purple-magenta") do
         colors[c] := Clone("fg=" || c)
      colors["black"] := Clone("fg=dark vivid gray")
      }
```

The rest of the init() procedure consists of drawing the window's starting contents and initializing global variables. Most of that is not worth presenting here, but it is worth showing how *Sesrit*'s "playing field" (global variable L) and first two objects are initialized. L is a list of 30 lists of 10 elements that should all start as "black". The list(n,x) call creates a list of n elements, all with the

initial value x. You cannot just initialize the variable L to list(30, list(10, "black")) because that would create a list of 30 references to a single list of 10 cells. The inner call to list() would only get called once. Instead, each of L's thirty rows is initialized with a different list of 10 cells.

```
   ...
   L := list(30)
   every !L := list(10, "black")
   newobject()
   activecells := copy(nextpiece)
   activecellcolor := copy(nextcolor)
   every point := !activecells do
      L[point[1], point[2]] := activecellcolor
   newobject()
end
```

With the window and variables initialized, the main task of the game is to repeat a sequence of steps in which the current piece falls one row each step, until the game is over. Like many other games, this infinite loop starts with a check for user input, and since several events could be waiting, the check for user input is itself a loop that terminates when the window's list of pending events (returned by Pending()) is empty. A case expression performs the appropriate response to each type of user input. Notice how ordinary keyboard characters are returned as simple one-letter strings, while special keys such as the arrows have defined symbols. Mouse event codes have keyword constants such as &lpress to represent their value. The &lpress constant indicates a left mouse key press. When Event() returns it assigns keywords &x and &y to the mouse location, so it is common to see code that checks these keywords' values while processing an input event. Several other keywords (&control, &meta, &shift) are also set to indicate the state of special keys (Control, Alt, and Shift on most keyboards). These keywords fail if the special key was not pressed at the time of the event.

```
procedure game_loop()
   game_status := 1
   repeat {
      while *Pending() > 0 do {
```

```
    case Event() of {
      Key_Left : move_piece(-1, 0)
      Key_Right: move_piece(1, 0)
      Key_Down : rotate_piece(1, -1)
      Key_Up   : rotate_piece(-1, 1)
      " "      : while move_piece(0, 1) # drop to bottom
      "q"      : if &meta then exit()
      "a"      : if &meta then about_itetris()
      "p"      : if (&meta & game_status = 1) then pause()
      "n"      : if &meta then return
      &lpress  : {
        if 15 <= &x <= 105 then {
          if 270 <= &y <= 290 then pause()
          else if 300 <= &y <= 320 then return
          else if 360 <= &y <= 380 then about_sesrit()
          }
        }
      &lrelease :
        if ((15 <= &x <= 105) & (330 <= &y <= 350)) then
          exit()
      } # end case
    } # end while user input is pending
```

Once user input has been handled, the piece falls one row, if it can. If the object could not fall, it is time to either bring the next object into play, or the game is over because the object is still at the top of the screen. The game_over() procedure is not shown in detail; it marks the occasion by drawing random colors over the entire screen from bottom to top and top to bottom and then asks whether the user wishes to play again.

```
  if not move_piece(0, 1) then {
    if (!activecells)[1] < 2 then { # top of screen
      game_over()
      return
      }
```

In the more common occurrence that the object could not fall but the game was not over, procedure scanrows() is called to check whether any of the rows are filled and can be destroyed. The next piece replaces the active cell variables, and procedure newobject()

generates a new next piece. The score is updated for each new object, and the current and next pieces are drawn on the display.

```
      while get(Pending())
      scanrows()
      Fg("black")
      drawstat(score, , , tot_score)
      score +:= 5
      tot_score +:= 5
      Fg("white")
      drawstat(score, , , tot_score)
      activecells := copy(nextpiece)
      activecellcolor := copy(nextcolor)
      every point := !activecells do
        L[point[1], point[2]] := activecellcolor
      newobject()
      EraseArea(120,481,150,15)
      Bg("black")
      every cell := !activecells do {
        EraseArea(-40 + (cell[2]-1)*15,
                  60 + (cell[1]-1)*15, 15, 15)
        drawcell(120 + (cell[2]-1)*15, 481,
                 activecellcolor)
        }
      every cell := !nextpiece do
        drawcell(-40 + (cell[2]-1)*15,
            60 + (cell[1]-1)*15, nextcolor)
      }
```

Each step is completed by a call to WSync() to flush graphics output, followed by a delay period to allow the user to react. The WSync() procedure is only needed on window systems that buffer output for performance reasons, such as the X Window System. The delay time becomes smaller and smaller as the game progresses, making it harder and harder for the user to move the falling pieces into position.

```
    WSync()
    delay(delaytime)
    }
end
```

Procedure newobject() generates a new object, which will be displayed beside the game area until the current object stops falling. The object is stored in global variable nextpiece, and its screen color is given in variable nextcolor. Objects are represented as a list of (row,column) pairs where the pairs are two-element lists.

One quarter of the objects (?4 = 1) are of a random shape; the remainder are taken from the set of four-cell shapes found in *xtetris*. Random shapes are obtained by starting with a single cell, and adding cells in a loop. Each time through the loop there is a 25% chance that the loop will terminate and the object is complete. The other 75% of the time (?4 < 4), a cell is added to the object, adjacent in a random direction from the last cell. The expression ?3-2 gives a random value of 1, 0, or -1. This expression is added to each of the x and y coordinates of the last cell to pick the next cell.

```
procedure newobject()   pieceposition := 1
  if ? 4 = 1 then {
    nextcolor := "pink"
    nextpiece := [[2,6]]
    while ?4 < 4 do {
      x := copy(nextpiece[-1])
      x[1] +:= ?3 - 2
      x[2] +:= ?3 - 2
      if x[1] = ((y := !nextpiece)[1]) & x[2] = y[2] then
        next
      put(nextpiece, x)
      }
```

There is a bunch of sanity checking that is needed for random objects, given below. If the object is so big that it won't fit in the next piece area beside the playing field, it is filtered out. In addition, we need to center the object horizontally. The subtlest task is to move the "center" cell of the random object (if it has one) to the front of the list, so the object rotates nicely. To do all this, the code first computes the boundaries (min and max) of the random object's row and column values. The solution for finding a center cell, checking each cell to see if it is at row (miny + maxy)/2, column (minx + maxx)/2 is pretty suboptimal

since it only succeeds if an exact center is found, instead of looking for the cell closest to the center. How would you fix it?

```
      miny := maxy := nextpiece[1][1]
      minx := maxx := nextpiece[1][2]
      every miny >:= (!nextpiece)[1]
      every minx >:= (!nextpiece)[2]
      every maxy <:= (!nextpiece)[1]
      every maxx <:= (!nextpiece)[2]
      every i := 2 to *nextpiece do
        if nextpiece[i][1] == (miny + maxy) / 2 &
           nextpiece[i][2] == (minx + maxx) / 2 then
            nextpiece[1] :=: nextpiece[i] # swap!
      if miny < 1 then every (!nextpiece)[1] +:= -miny + 1
      every minx to 3 do every (!nextpiece)[2] +:= 1
      if (!nextpiece)[1] > 5 then return newobject()
      if (!nextpiece)[2] > 8 then return newobject()
      }
```

In contrast to the random shapes, the standard shapes use hardwired coordinates and colors.

```
   else
      case nextcolor := ?["red-yellow","red","yellow","green",
                          "cyan", "blue", "purple-magenta"] of {
   "red-yellow":     nextpiece := [ [1,5], [1,6], [2,5], [2,6] ]
   "yellow":         nextpiece := [ [2,6], [1,6], [2,5], [2,7] ]
   "blue":           nextpiece := [ [2,6], [1,5], [2,5], [2,7] ]
   "purple-magenta": nextpiece := [ [2,6], [1,7], [2,5], [2,7] ]
   "red":            nextpiece := [ [3,6], [1,6], [2,6], [4,6] ]
   "green":          nextpiece := [ [2,6], [1,5], [1,6], [2,7] ]
   "cyan":           nextpiece := [ [2,6], [1,6], [1,7], [2,5] ]
   }
end
```

Procedure move_piece(x,y) moves the active cells for the current object by an offset (x,y) that handles movement left and right, as well as down. The new desired location of all the cells is calculated, and then procedure place_piece() is called to try to put the piece at the new location.

```
procedure move_piece(x, y)
```

```
   newactivecells := []
   every cell := !activecells do
      put(newactivecells, [cell[1]+y, cell[2]+x])
   return place_piece(newactivecells, x)
end
```

The place_piece() procedure checks whether the object can go into the location proposed, and if it can, it draws the piece in the new location and updates the activecells variable. Testing whether parameter horiz = 0 allows the code to skip redrawing the object's "footprint" below the play area in the common case when the object just drops one row. The backslash in \horiz causes the expression to fail if horiz is null, so the second parameter may be omitted.

```
procedure place_piece(newactivecells, horiz)
   if collision(newactivecells) then fail
   if not (\horiz = 0) then
      EraseArea(120,481,150,15)
   every cell := !activecells do {
      FillRectangle(colors["black"],
                120 + (cell[2]-1)*15,
                20 + (cell[1]-1)*15, 15, 15)
      L[cell[1], cell[2]] := "black"
      }
   every cell := !newactivecells do {
      L[cell[1], cell[2]] := activecellcolor
      drawcell(120 + (cell[2]-1)*15, 20 + (cell[1]-1)*15,
            activecellcolor)
      if not (\horiz = 0) then
         drawcell(120 + (cell[2]-1)*15, 481, activecellcolor)
      }
   WSync()
   activecells := newactivecells
   return
end
```

Procedure collision() checks each cell that the object is moving into to see if it is occupied by something other than part of the currently active piece. Check out the cool break next expression when a black cell is found. It exits the inner loop and skips to the next

iteration of the outer loop, preventing some "false positive" collision results.

```
procedure collision(cells)
   every c := !cells do {
      if not ((1 <= c[1] <= 30) & (1 <= c[2] <= 10)) then return
      if L[c[1], c[2]] === "black" then next
      every a := !activecells do {
         if (c[1] = a[1]) & (c[2] = a[2]) then
            break next
         }
      if L[c[1], c[2]] ~=== "black" then return
      }
   fail
end
```

Rotating a piece is similar to moving it, in that it involves calculating a new location for each cell. The first active cell in the list is considered the "center" around which the rest of the cells rotate. To rotate a cell by ninety degrees around the center, compute its x- and y-coordinate offsets from the center cell. The cell's rotated position uses these same offsets, but swaps the x-offset with the y-offset, and reverses the signs of one offset or the other, depending on which quadrant the rotation is coming from and going into. Figure 9-2 shows how the signs are reversed depending on the quadrant.

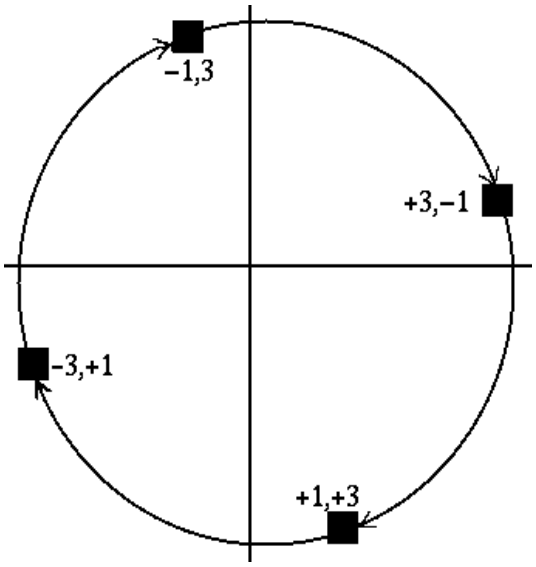

Figure 11-2:
Rotating a cell swaps its x and y offsets, with sign changes

Four standard cell shapes, identified by color, are handled specially during rotation. Squares (red-yellow) have no rotation. The other three standard cell types (red, green, and cyan) are symmetric shapes whose rotation appears smoother if it alternates clockwise and counterclockwise. This alternation is handled by global variable pieceposition.

```
procedure rotate_piece(mult1, mult2)
   if activecellcolor === "red-yellow" then fail
   newactivecells := list()
   centerpoint := copy(activecells[1])
   differencelist := list()
   every point := ! activecells do {
      temp := [centerpoint[1]-point[1], centerpoint[2]-point[2]]
      put(differencelist, temp)
      next
      }
   every cell := !activecells do
      put(newactivecells, copy(cell))
   if activecellcolor === ("red" | "green" | "cyan") then {
      if pieceposition = 2 then {
         mult2 :=: mult1
         pieceposition := 1
         }
      else pieceposition := 2
      }
   every foo := 1 to *newactivecells do
      newactivecells[foo] := [
         centerpoint[1] + differencelist[foo,2] * mult1,
         centerpoint[2] + differencelist[foo,1] * mult2
         ]
   return place_piece(newactivecells)
end
```

Each time a piece stops falling, procedure scanrows() checks to see whether any rows in the playing area are filled and can be removed. If no black is found on a row, that row is put on a list called rows_to_delete.

The player scores $50 * 2^{k-1}$ points for k deleted rows. To maximize your score you should try to always delete several rows at once!

```
procedure scanrows()
   scanned_rows := table()
   rows_to_delete := []
   every point := !activecells do {
      if \scanned_rows[point[1]] then next
      scanned_rows[point[1]] := 1
      every x := 1 to 10 do {
         if L[point[1], x] === "black" then
            break next
         }
      put(rows_to_delete, point[1])
      }
   if *rows_to_delete > 0 then {
      Fg("black")
      drawstat(score, numrows, level, tot_score)
      numrows +:= *rows_to_delete
      level := integer(numrows / 10)
      score +:= 50 * (2 ^ (*rows_to_delete - 1))
      tot_score +:= 50 * (2 ^ (*rows_to_delete - 1))
      delaytime := 200 - (10 * level)
      Fg("white")
      drawstat(score, numrows, level, tot_score)
      deleterows(rows_to_delete)
      }
end
```

The code to delete rows takes the list of rows to delete, sorts it, and builds a corresponding set. It then moves the bottom rows of L, up to the first row to be deleted, into a temporary list. For each row in the temporary list, if it is to be deleted, a new row of black cells is inserted at the top of L, otherwise the row is re-appended to the end of L. When the play area has been reassembled it is redrawn.

```
procedure deleterows(rows_to_delete)
   temp := []
   rows_to_delete := sort(rows_to_delete)
   row_set := set()
   every insert(row_set, !rows_to_delete)
```

```
   current_row := 30
   while current_row >= rows_to_delete[1] do {
      push(temp, pull(L))
      current_row -:= 1
      }
   current_row := 1
   basesize := *L
   while *temp>0 do {
      if member(row_set, basesize + current_row) then {
         push(L, list(10, "black"))
         pop(temp)
         }
      else
         put(L, pop(temp))
      current_row +:= 1
      }
   refresh_screen()
   WSync()
end
```

Sesrit provides several buttons to the left of the play area that allow the user to pause, quit, start a new game, or see author information. The procedure buttons() is called to handle input events whenever the game is not actually running, such as before it starts or when it is paused. Its code is analogous to the user input handling in game_loop() and is not shown here. The procedure about_sesrit() implements the about box, a simple dialog that shows author information until the user dismisses it. It illustrates several graphics library procedures related to drawing of text. CenterString() is a useful library procedure that draws a string centered about an (x,y) location. Incidentally, the GUI facilities described in Chapter 17 of the Unicon book include an interface builder for constructing dialogs such as this, but for Sesrit the use of such a tool is overkill.

```
procedure about_sesrit()
   about := WOpen("label=About Sesrit",
                  "size=330,200", "fg=white",
                  "bg=black", "posx=10", "posy=155") | fail
   Bg("black")
   every cell := !nextpiece do
```

```
      EraseArea(-40 + (cell[2]-1)*15, 60 + (cell[1]-1)*15,15,15)
   FillRectangle(colors["black"],
                    120,20,150,450,120,481,150,15)
   CenterString(about, 165, 25, "Written By: David Rice")
   CenterString(about, 165, 50,
              "Communications Arts HS, San Antonio")
   CenterString(about, 165, 90, "and")
   CenterString(about, 165, 115, "Clinton Jeffery")
   CenterString(about, 165, 180, "Spring 1999")
   Event(about)
   while get(Pending())
   WClose(about)
   if game_status = 1 then refresh_screen()
end
```

The last procedure from *Sesrit* that we present is the one that redraws the play area, refresh_screen(). It draws the next piece cells to the left of the play area, it draws the footprint outline of the active cell below the play area, and then it draws the entire play area with a big loop that draws filled rectangles. Cell L[x,y]'s colors are looked up in the color table to determine the color of each rectangle that is drawn.

```
procedure refresh_screen()
   every cell := !nextpiece do
      drawcell(-40 + (cell[2]-1)*15, 60 + (cell[1]-1)*15,
            nextcolor)
   every cell := !activecells do
      drawcell(120 + (cell[2]-1)*15, 481, activecellcolor)
   every (x := 1 to 30, y := 1 to 10) do
      drawcell(120 + (y-1)*15, 20 + (x-1)*15, L[x, y])
end
```

Chapter 12: Blasteroids

The classic Atari game asteroids is a more involved example of 2D animation and simple physics. The program Blasteroids presented in this chapter was written originally by Jared Kuhn while he was an undergraduate student. It is 1100+ lines long and is organized into four different .icn files which are combined together to form a complete program. These files are separately compiled and then linked together, along with many graphics and user interface library modules. Blasteroids uses the following makefile which links a main module (blaster), a game component (game), an options dialog (optionsdialog), and an about box (aboutbox):

```
# makefile for blaster
blaster: blaster.u game.u optionsdialog.u aboutbox.u
	unicon blaster.u game.u optionsdialog.u aboutbox.u
blaster.u: blaster.icn
	unicon -c blaster
game.u: game.icn
	unicon -c game
aboutbox.u: aboutbox.icn
	unicon -c aboutbox
optionsdialog.u: optionsdialog.icn
	unicon -c optionsdialog
```

Most of the code in Blasteroids really lives in the game module, but our description starts in blaster.icn because that is where the main() procedure is located. The main procedure goes like this:

```
procedure main()
  local d
  d := blaster()
  d.show_modal()
end
```

This is not very long, but it bears some explaining. Blasteroids uses a graphical interface built with Unicon's official GUI package, a remarkable code library written mainly by Robert Parlett, an open source volunteer from the United Kingdom. Most of the code in blaster.icn, optionsdialog.icn, and aboutbox.icn is automatically generated from an interface drawing tool called *ivib*, Unicon's

“improved visual interface builder”. This main procedure just creates a “blaster” object and tells it to show itself. A blaster object (an instance of class blaster) is a type of D*ialog;* a Dialog is an object which controls a window by attaching a set of user interface components to it and handles the window's input. From the point the blaster dialog is asked to show itself, the GUI library takes over and opens the window, draws the interface, and runs the game.

Creating Graphical User Interfaces with Ivib

The key aspects in writing programs with graphical user interfaces are: (a) selecting and placing user interface elements on a dialog, and (b) specifying how the program should respond or handle user input, which is delivered in the form of “event” objects. The first step is normally performed with Ivib. A screenshot of Ivib editing the blaster dialog for this game looks like this:

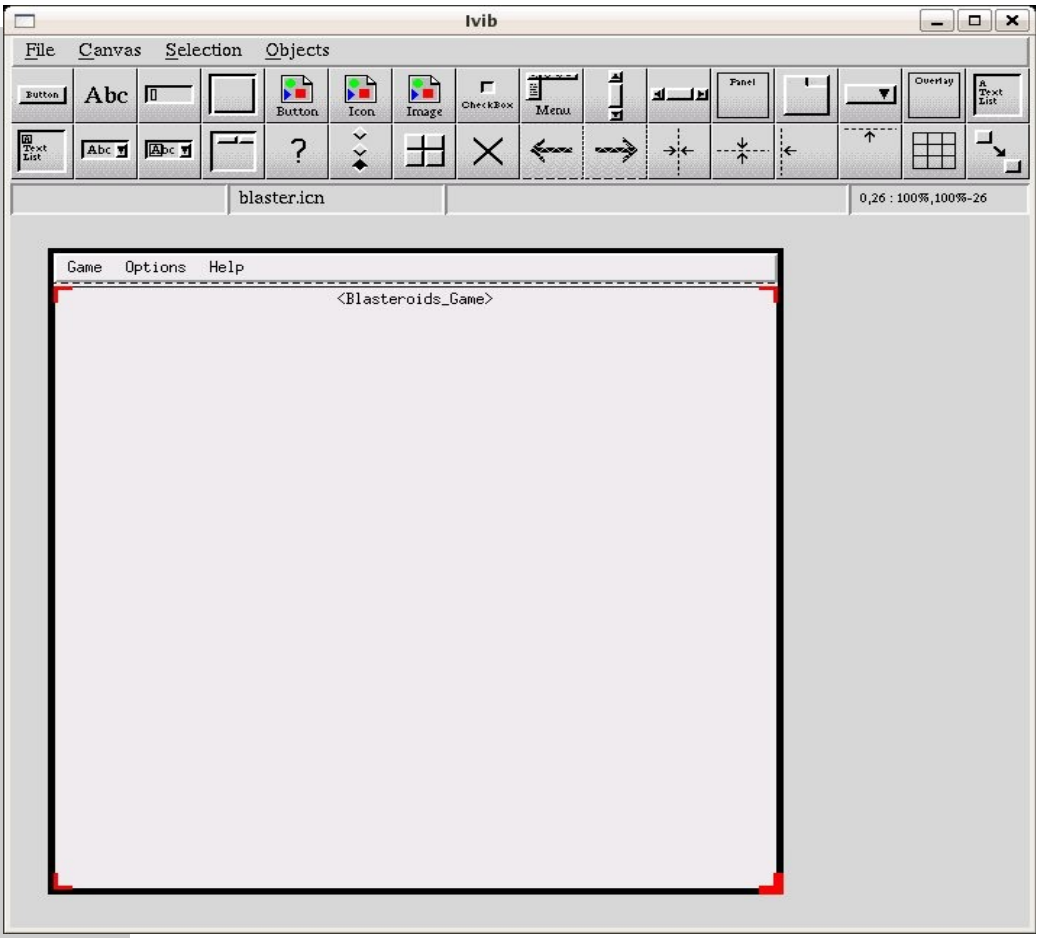

Figure 12-1:
The Ivib graphical interface builder.

A full description of how to use Ivib is beyond the scope of this book, but there is a fine tutorial: Unicon Technical Report #6 available at unicon.org. For the purposes of this book it is enough to say that the Ivib toolbar at the top allows you to easily create most routine user interface elements such as buttons or scrollbars. The blaster interface consists of a simple menu and a custom component (class Blasteroids_Game) written in game.icn to implement the main game visual elements (ships, asteroids, etc.). This custom component was inserted into the Ivib drawing using the “custom” button (the button with the “?” in the second row of the Ivib toolbar.

To tell what code should execute when the user interacts with a user interface element, you right click the element within Ivib to bring up a component Setup dialog that looks something like this:

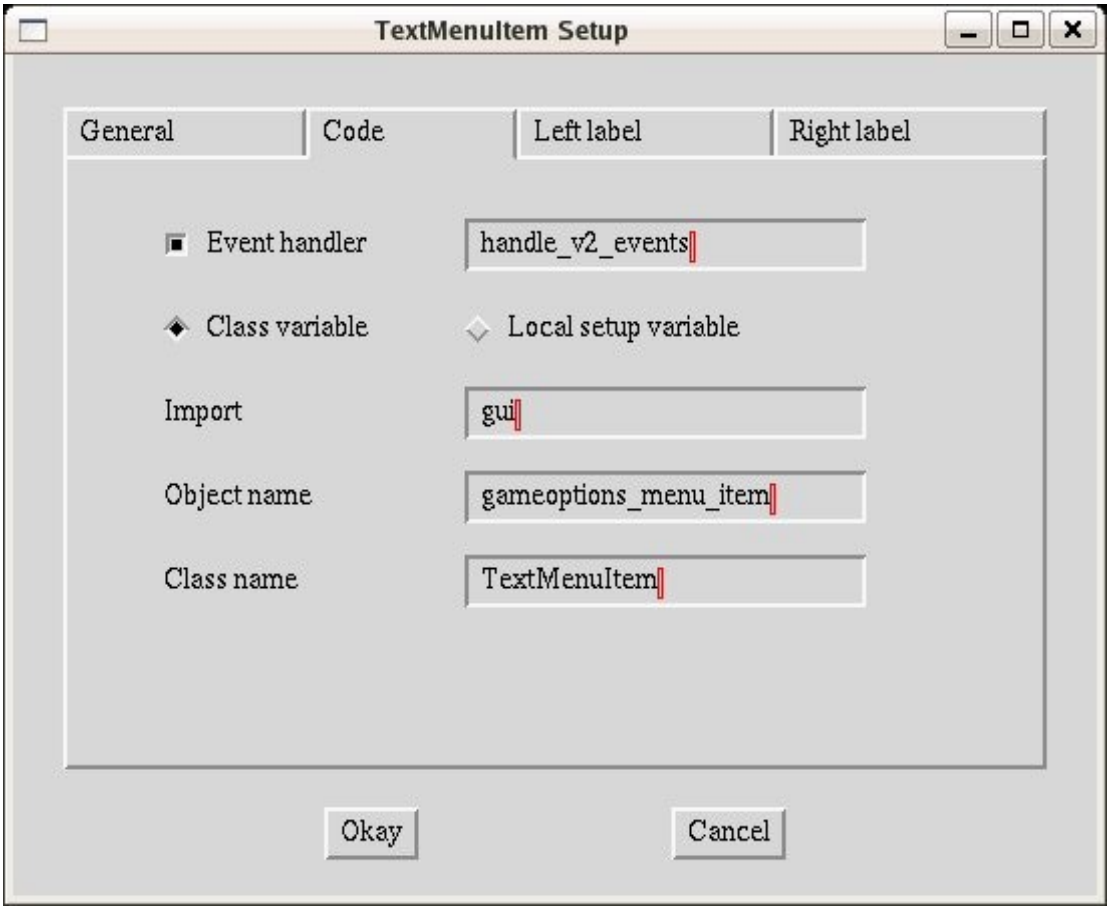

Figure 12-2:
GUI components attributes are modified by means of a Setup dialog.

Poking around in the Setup dialog, you can find places where you can set the name of the variable, what class it is an instance of, and

what method to call when the component is clicked on. There are lots of options; see UTR6 and “Programming with Unicon” for more details.

A final, special part of the blaster dialog class is a tweak for videogame-style real-time input handling. Inside a method init_dialog() which is called automatically after the window has been created and the dialog is about to start, the blaster turns on some extra input events: key release events. While mouse presses and releases are separate events by default, keyboard events are normally triggered on the press, with no event to tell you when the key is released. This is bad for videogames, which need (for example) to keep the spaceship turning as long as the arrow key is held down. To get Unicon to report key release events, the initi_dialog() looks like:

```
method init_dialog()
   WAttrib(win, "inputmask=k")
end
```

The Blasteroid Game Class

Almost all of the “real” code in Blasteroids lives in the custom component found in game.icn. It is a pristine example of creating a complex graphical element that in turn fits comfortably into a standard user interface dialog. The file game.icn starts by importing the GUI package (“import gui” will appear at the top of all modern Unicon GUI programs) and including a file keysyms.icn which contains symbolic names for the various integer codes used for special keys such as the left arrow (Key_Left), the function keys, and so forth. It then proceeds with a lot of $define'd symbols such as

```
$define SPEEDINC 0.1          #acceleration constant
$define MAXSPEED 5.0          #maximum speed
$define NEGMAXSPEED -5.0 #negative of max speed
```

These make it relatively easy to change game behavior. For example, after a few years the specified delay between each frame was too small and the game was unplayably fast. It was increased from 10ms (100fps) to 33ms (30fps) in order to slow the asteroids

down enough for a human to shoot them. On a PDA or other slow platform, a smaller delay might be better.

```
$define DELAY_TIME 33        #length of a timeslice (ms)
```

The game module introduces several user-defined data types – records – to improve the game organization and readability. A record is just a class without any methods. If you have an array or table full of data, organized into fields and want to refer to them by name, records are the type to use.

```
record point(x,y)       #a point on the screen
record rtrack(          #asteroid tracking record
        x, y,           #center coordinates
        angle,          #angle of rotation (not used)
        points,         # polygon point coords (relative to center)
        size, speed,    #size / speed of asteroid
        offset,     #precomputed multiplier, saves computations
        offset2x,   #same
        active)         #boolean, active or not

record btrack(          #blast tracking record
        x, y,           #center coordinates
        angle,          #angle of rotation of blast
        flag,           #active / inactive
        traveled,       #how far the blast has gone
        deactnext)      #flag: to be deactivated next time slice
```

The main Blasteroids_Game is a class, not a record. It is in fact a subclass of the generic GUI package Component, from which it inherits the fields and methods needed to reside and function comfortably within a GUI dialog. The downside of this is that you have to go read library documentation or source code in order to fully understand a Blasteroids_Game object. There is a learning curve for the GUI classes in all major languages. In Unicon that learning curve is modest, but some will learn it more quickly than others. Anyhow, here is the class header:

```
class Blasteroids_Game : Component(starsdone,
                                   tapped, gamestart,
                                   shipx, shipy, c,
                                   newwin, backwin,
```

```
      lookups, lookupc,
      currblast, blasts, blastwin,
      roids, currroids, roidwin,
      explodewins, shiphit,
      speedx, speedy, angle, currspeed,
      oldshipx, oldshipy,
      masterspeed, oldangle, negx, negy,
      redrawship,
      x1, y1, firsttime,
      livesleft, livesdone, score, oldscore,
      num_ships, num_range,
      chk_explode, chk_sound,
      num_torpedos, num_speed,
      num_rotation, num_level,
      left_pressed, right_pressed,
      up_pressed, down_pressed)
```

These class fields (instance variables) should really be commented. For a class as large as this one, the programmer needs all the help they can get. At least the variable names are good for the most part.

Class instances (objects) get initialized in an *initially section*, which is just a special method that is automatically called when the instance is created. The game class initially section is long, and interesting, but we present the highlights (... indicates additional lines of code which are omitted here).

```
initially
   self.Component.initially() ...
   blasts := list(MAXBLASTS)
   roids := list(MAXROIDS) ...
   every !roids := rtrack(-500, -500, 0, list(8), 0, 0, 0, 0, 0)
   every (!roids).points[1 to 8] := point(0,0)
   every !blasts := btrack(-500, -500, 0, 0, 0)
```

The preallocation of all the data keeps things short and sweet in-game. The initially also creates an invisible, off-screen window (attribute “canvas=hidden”) which is used for redrawing the scene as dynamic objects whiz through space.

How do you handle real-time behavior in a graphical user interface? The question is complicated because the user interface

owns the control flow and wants it to stick to a nice, tight “event processing loop” which revolves around handling user input, but in a real-time game the other objects (rocks in this case, but in other games they might be AI-controlled beings with intelligent behavior) want to move or act continuously whether the user is idle or not.

In this game, at least, the “idle time” is utilized in a method called main_game_loop() which is called after each user input event. As long as there is nothing for the GUI to handle (if the Pending() queue is empty) the code executes time steps in which the asteroids and the ships blasts are updated.

```
method main_game_loop()
...
while *Pending() = 0 do {
   ...
        # if the ship has moved or turned, redraw it
        if (shipx~= x1 | shipy ~= y1 | oldangle ~= angle) &
           (shiphit = 0)then{
           redrawship := 0
           oldangle := angle
           #finally, draw the ship
           eraseblasts()
           drawship(x1, y1, angle-1)
           drawblasts()
           WDelay(DELAY_TIME)
           #already did delay, don't do it again
           delayed := 1
           }
       # ... animate asteroids, check if ship is hit, etc.
}
```

To move an object you first erase it (by copying from the off-screen background window) and then draw it:

```
   method eraseship()
      # copy background over area where ship is (done before
      # animating it)
      CopyArea(backwin, newwin, shipx-24, shipy-24,
                     48,48, shipx-24, shipy-24)
```

```
  end
  method drawship(x, y, rot)
    local cs, si, si16, si10, cs16, cs10, count
    #make sure rot (angle of rotation) is between 0 and 359
    rot := rot % 360
    if rot < 0 then rot := rot + 360
    #erase
    CopyArea(backwin, newwin, shipx-24, shipy-24,
                  48, 48, shipx-24, shipy-24)
    #use a lookup table for speed
    cs := lookupc[rot+1]
    si := lookups[rot+1]

    si16 := 16*si
    cs16 := 16*cs
    si10 := 10*si
    cs10 := 10*cs
    #do rotation (using x, y as center), and draw the 'ship'
    #might be a faster way (w/out so many multiplies...
    DrawLine(x - cs16, y-si16, x+(4*cs), y+(4*si))
    DrawLine(x-cs10-si16, y + cs16-si10, x, y,
          x-cs10+si16, y-cs16-si10)
    DrawLine(x+si16-cs16, y-cs16-si16, x+si16,
          y-cs16)
    DrawLine(x-cs16-si16, y+cs16-si16, x-si16, y+cs16)
    DrawPoint(x+cs10, y+si10)
    DrawCircle(x+cs10, y+si10, 6)
    #reset globals defining ship location
    shipx := x
    shipy := y
  end
```

Although the game class has many more details which deserve study, the last aspect that we will present is the user input handling. The input handling code is “legacy” code, so it is not entirely representative of all GUI applications. But, in the Ivib program Setup dialogs, you can specify what type of events a component responds to and what method to call, which results in Ivib-generated code of the form

your_item.connect(self,"*your_method*", *event_type*)

Most components that respond to mouse clicks will specify an event_type of ACTION_EVENT, but there are many other specific kinds of events that you can request. In addition, your components can write a handle_event(e) method that receives events in their "raw" form (strings for regular keystrokes, and small integer codes for mouse events and special keys). Inside the game class handle_event(e) method, there is code that looks like:

```
        case e of {
          " ": {
            #space bar, fire guns
             if shiphit = 0 then {
                redrawship := 5
                eraseblasts()
                eraseship()
                initblast(x1, y1, angle - 1)
                eraseblasts()
                drawship(x1, y1, angle)
                }
             }
    ...
          "q" | "Q": {
            exit()
            }
          Key_Left:      {
            angle -:= 5
            if(angle <= 0) then angle +:= 360
            left_pressed := 1
            }
          -(Key_Left)-128: {
            left_pressed := &null
            }
```

Between such events, the method main_game_loop() described earlier is busy updating the asteroids' positions.

Exercises

1. Modify Blasteroids so that it gets more difficult as time progresses.

2. Fix the high score code to present results attractively in a GUI dialog.
3. Add alien ships which attempt to shoot the player's ship.
4. Add limited-time shields so the player can protect their ship.

Chapter 13: Network Games and Servers

An Internet Scorecard Server

Many games with numeric scoring systems feature a list of high scores. This feature is interesting on an individual machine, but it is ten times as interesting on a machine connected to the Internet! The following simple server program allows games to report their high scores from around the world. This allows players to compete globally. The scorecard server is called scored. By convention, servers are often given names ending in "d" to indicate that they are daemon programs that run in the background.

The Scorecard Client Procedure

Before you see the server program, take a look at the client procedure that a game calls to communicate with the scored server. To use this client procedure in your programs, add the following declaration to your program.

```
link highscor
```

The procedure highscore() opens a network connection, writes four lines consisting of the protocol name "HSP", the name of the game, the user's identification (which could be a nickname, a number, an e-mail address, or anything else), and that game's numeric score. Procedure highscore() then reads the complete list of high scores from the server, and returns the list. Most games write the list of high scores to a window for the user to ponder.

```
procedure highscore(game, userid, score, server)
   if not find(":", server) then server ||:= ":4578"
   f := open(server, "n") | fail

   # Send in this game's score
   write(f, "HSP\n", game, "\n", userid, "\n", score) |
      stop("Couldn't write: ", &errortext)

   # Get the high score list
   L := ["High Scores"]
```

```
   while line := read(f) do
      put(L, line)
   close(f)
   return L
end
```

The Scorecard Server Program

The scorecard server program, scored.icn illustrates issues inherent in all Internet servers. It must sit at a port, accepting connection requests endlessly. For each connection, a call to score_result() handles the request. The main() procedure given below allows the user to specify a port, or uses a default port if none is supplied. If another server is using a given port, it won't be available to this server, and the client and server have to agree on which port the server is using.

```
procedure main(av)
   port := 4578 # a random user-level port
   if av[i := 1 to *av] == "-port" then
      port := integer(av[i+1])

   write("Internet Scorecard version 1.0")
   while net := open(":" || port, "na")  do {
      score_result(net)
      close(net)
      }
   (&errno = 0) | stop("scored net accept failed: ",
      &errortext)
end
```

The procedure score_result() does all the real work of the server, and its implementation is of architectural significance. If any delay is possible in handling a request, the server will be unable to handle other simultaneous client requests. For this reason, many server programs immediately spawn a separate process to handle each request. You could do that with the system() function, but for scored this is overkill. The server will handle each request almost instantaneously itself.

Some small concessions to security are in order, even in a trivial example such as this. If a bogus Internet client connects by accident, it will fail to identify our protocol and be rejected. More subtly, if a rogue client opens a connection and writes nothing, we do not want to block waiting for input or the client will deny service to others. A call to select() is used to guarantee the server receives data within the first 1000 milliseconds (1 second). A last security concern is to ensure that the "game" filename supplied is valid; it must be an existing file in the current directory, not something like /etc/passwd for example.

The score_result() procedure maintains a static table of all scores of all games that it knows about. The keys of the table are the names of different games, and the values in the table are lists of alternating user names and scores. The procedure starts by reading the game, user, and score from the network connection, and loading the game's score list from a local file, if it isn't in the table already. Both the score lists maintained in memory, and the high scores files on the server, are sequences of pairs of text lines containing a userid followed by a numeric score. The high score files have to be created and initialized manually with some N available (userid,score) pairs of lines, prior to their use by the server.

```
procedure score_result(net)
   static t
   initial t := table()
   gamenamechars := &letters++&digits++'-_'

   select(net, 1000) | { write(net, "timeout"); fail }
   read(net) == "HSP" | { write(net, "wrong protocol"); fail }
   game := read(net) | { write(net,"no game?"); fail }
   game++gamenamechars === gamenamechars | { fail }
   owner := read(net) | { write(net,"no owner?"); fail }
   score := numeric(read(net)) | { write("no score?"); fail }

   if t[game] === &null then {
      if not (f := open(game)) then {
         write(net, "No high scores here for ", game)
         fail
```

```
      }
   t[game] := L := []
   while put(L, read(f))
   close(f)
   }
else
   L := t[game]
```

The central question is whether the new score makes an entry into the high scores list or not. The new score is checked against the last entry in the high score list, and if it is larger, it replaces that entry. It is then "bubbled" up to the correct place in the high score list by repeatedly comparing it with the next higher score, and swapping entries if it is higher. If the new score made the high score list, the list is written to its file on disk.

```
if score > L[-1] then {
   L[-2] := owner
   L[-1] := score
   i := -1
   while L[i] > L[i-2] do {
      L[i] :=: L[i-2]
      L[i-1] :=: L[i-3]
      i -:= 2
      }
   f := open(game,"w")
   every write(f, !L)
   close(f)
   }
```

Note

List *L* and *t[game]* refer to the same list, so the change to L here is seen by the next client that looks at *t[game]*.

Lastly, whether the new score made the high score list or not, the high score list is written out on the network connection so that the game can display it.

```
   every write(net, !L)
end
```

Is this high score application useful and fun? Yes! Is it secure and reliable? No! It records any scores it is given for any game that has

a high score file on the server. It is utterly easy to supply false scores. This is an honor system.

Chapter 14: Galactic Network Upgrade War

The next example is an illustration of object-oriented design for a complex strategy game of a depth competitive with commercial offerings. The scope of this example is enormous, and this chapter only presents highlights from the design of this 10,000+ line program. The focus is on the challenge of implementing an object-oriented design in Unicon. Like many large programs, this game is a "living thing" that is slowly evolving and never really finished.

The game presented in this section is a parody of many sacred cows, from the software industry to the political system to the great science fiction and fantasy epics of the twentieth century. It owes inspirational credit to a charming old war game called *Freedom in the Galaxy*, designed by Howard Barasch and John Butterfield, and published originally by Simulations Publications, Inc. and later by the Avalon Hill Game Company. Freedom in the Galaxy is in turn a homage (or parody) to George Lucas' *Star Wars* films. Our game is farther over-the-top than either of these fine works.

The Play's the Thing

What better medium for setting the mood, than epic poetry?

> ***The Lord of the Nets***
>
> *Green nets to snare compiler wizards from ivory towers in deep space,*
> *Gold nets to lure database gurus, their galactic servers in place,*
> *Silver nets for IT media pundits and their damned lies,*
> *A black net for the Dark Lord, to mask his dark face.*
>
> *One Net to take them all,*
> *One Net to buy them,*
> *One Net to tax them with*
> *And in the license bind them*
> *In the land of Redmond, where Executives lie.*

Galactic Network Upgrade War (GW) is an epic space-fantasy strategy game in which an evil empire holds an entire galaxy in its iron grip by means of monopolistic software practices. A small band of rag-tag independent software developers fight a hopeless crusade against the ruthless tyrant, the Microsaur Corporation and its Dark Lord, Microsauron.

To play the game, you take on the role of a young hacker who must work your way up through the ranks, on the side of good or evil, until you earn the respect and influence that enable you to lead your side to victory, crush your opponents, and gain control over the galactic Internet.

Background

In the year 9,999, as the entire galaxy struggles for its very survival in the throes of the Y10K bug, one company holds all the cards. Governments have long since defaulted on their loans and been replaced by the rule of the multi-stellar corporations. After a time of cataclysmic battles, all of the surviving corporations swore fealty to one ruler, becoming subsidiaries with varying nano-degrees of freedom. That ruler was the Microsaur Corporation.

Some say Microsaur took over through superior innovation, blending the best aspects of biological and technological species into a super-race, a hybrid of voracious reptile and implanted artificial intelligence. Others say that the Microsaur Collective, a predator unchecked, simply ate all organized resisters on the galactic network. Perhaps both accounts are true. In any case, Microsaur Corporation took over by controlling the amalgamated IT resources of the entire galaxy, on every desk in every home and office.

But although organized corporate resistance was futile, lonely bands of mysterious rebels persisted in conveying an oral tradition of a time when machines ran open source software. Unable to maintain any presence on digital media, these groups, such as the Gnights who say Gnu, memorized entire programs such as Gnu Emacs in their heads and passed them on to their children for generations. This went on until one day Tux the Killer Penguin inexplicably appeared in a puff of logic on an unguarded workstation at the edge of the galaxy in the office of one Doctor Van Helsingfors. The Microsaur collective's presence on that machine flickered out too instantaneously and too faintly to even be noticed in the corporate campus at the center of the galactic Internet. Finding him playful, cuddly, and more useful than the Microsaur that Tux had slain, Doctor Van Helsingfors shared Tux

with a few thousand of his closest friends, little realizing his communiqué would change the galaxy forever.

Tux the Killer Penguin was friendly to Sapes (from Homo sapien) and other bio life forms, but had a rare property: an insatiable hunger for Microsaurs. Tux was a network aware, open system artificial intelligence (an "Aint," as they are called). With the assistance of Sape rebels, Tux spread like a virus through the galactic network, paving the way for other open system software on small machines across the galaxy. Eventually, the Microsaur collective detected that it no longer held its 100 percent market share, and once it identified Tux, it began the most radical binary purge in galactic history, expecting little or no resistance. Thus begins the epic war depicted in GW.

The Map

Each player gets a different view of the galaxy, reflecting both his or her own focus of attention and the limited information available to them. Their map is drawn on a window, and contains glyphs (visual objects, or symbols) corresponding to a subset of the objects in the galaxy. In general, several players may have different glyph objects (with different location, size, or other attributes) that refer to the same underlying galaxy object.

The User Interface

GW's user interface centers around a map of the galaxy that depicts everything the player knows about the movements and activities of friendly and hostile forces. With limited screen space, it is tempting to show only one piece of information at a time to present it in detail, say for example, one habitat (or other type of space object, with its contents) at a time. The problem with this is that strategic decisions about a group's activities requires more information than that provided in a single habitat; strategy requires a global view.

Another way to state the problem with this type of "modal" interface is that the player will be constantly switching modes and reorienting themselves when the view changes. Instead of this, GW's map uses a technique called a *graphical fisheye view* to show multiple focus star systems in detail, while preserving a global

view of the system. Figure 12-1 shows a view of a galaxy. A light (yellow) circle, orbited by one or more planets depicts each solar system. Solar systems of interest are drawn in vastly more detail than others, controlled by clicking the left (enlarge) and right (shrink) mouse buttons on the different stars. The names of stars or planets are drawn in varying font sizes, as long as sufficient space is available. Planet details such as habitats, defenses, and loyalties are depicted. The simple colors and textures used for stars and planets are legible and easily learned, but these prototype-quality graphics should be upgraded with more artistic or photo-like images of astral bodies.

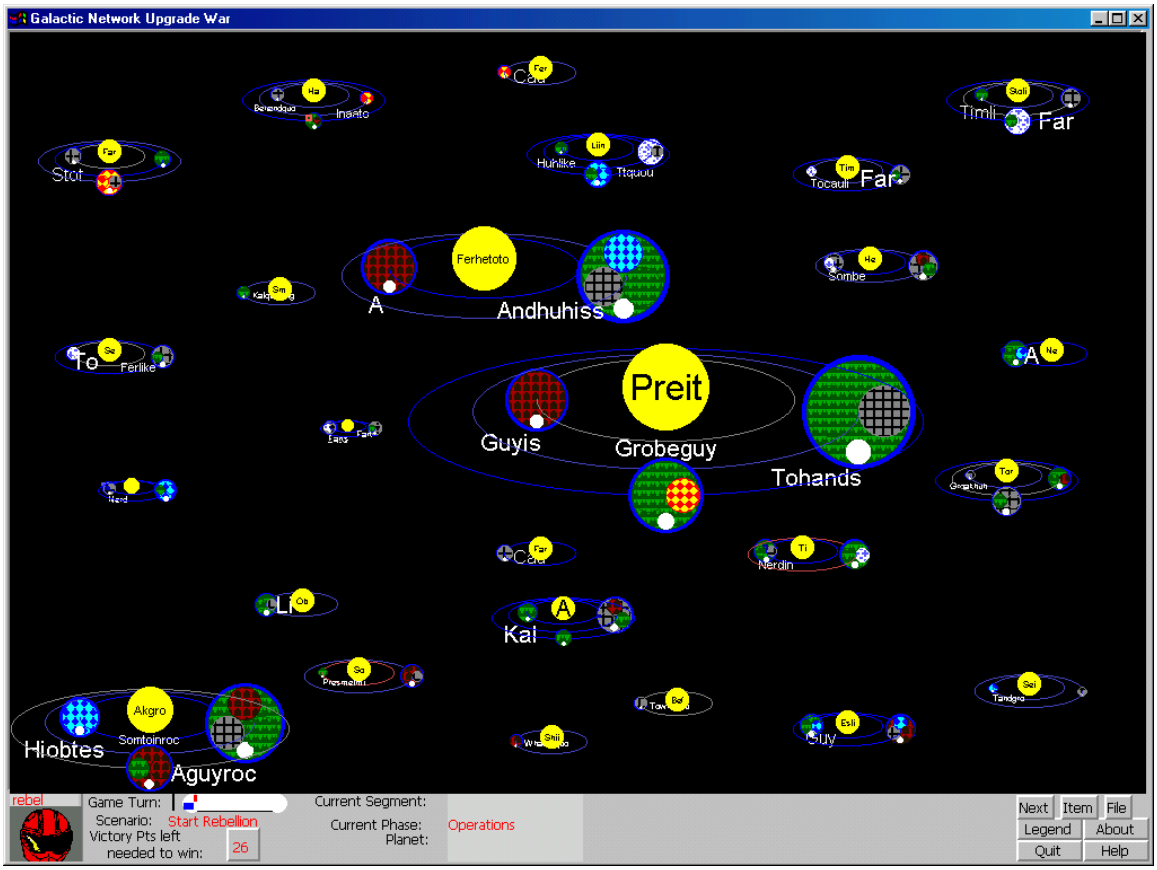

Figure 14-1:
The Graphical Fisheye View in GW

GW's other activities, such as giving orders, can generally be performed using keyboard shortcuts, or using various dialogs that present detail information. Figure 12-2 shows a dialog used for assigning orders within a habitat. The units are depicted with humorous mock-up images and various attribute details in the upper area of the window. The bottom center contains a list of

orders these characters are eligible for in this habitat. The currently selected order is described in detail in the lower right area of the dialog. As play commences, players gain or lose characters and military units, and separately, they gain or lose installed base and "mind share" for their software's cause on each planet in the galaxy. The player can define winning in terms of survival, domination, or eradication of all opposition.

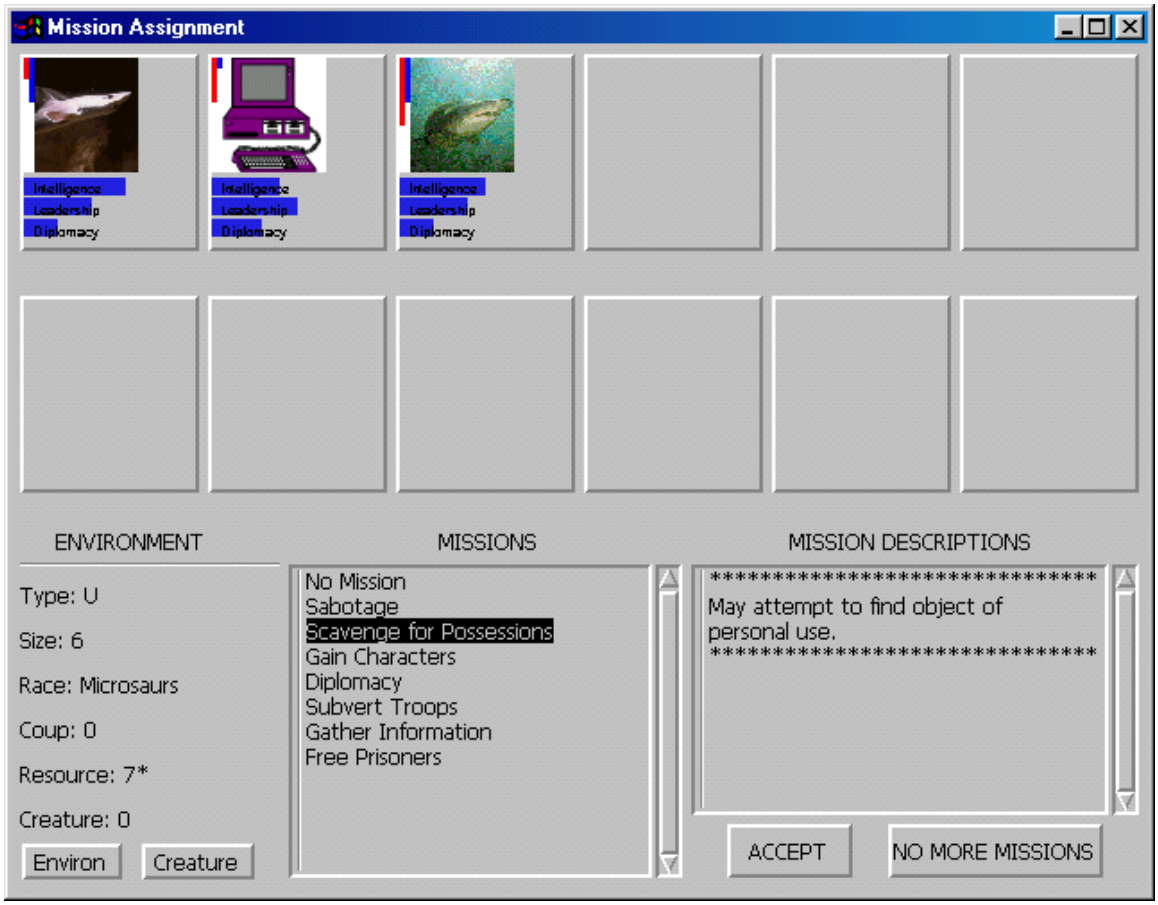

Figure 14-2:
Assigning Orders Dialog

Summary

Unicon is a fun language to use for writing many different kinds of games. Games usually make use of graphics facilities and increasingly, Internet facilities as well. Thanks to the high score server from Chapter 11, any game that includes a point-scoring system can easily provide global competition for players with Internet access.

For complex games, designing the game may be as hard a job as programming it. It helps to start from a complete prose description of all game activities, to develop use cases, and then to describe the kinds of objects and relationships between objects that will be necessary for those game activities to occur. This chapter is able to suggest that process, but to show it in detail would be the subject of another entire book.

Index