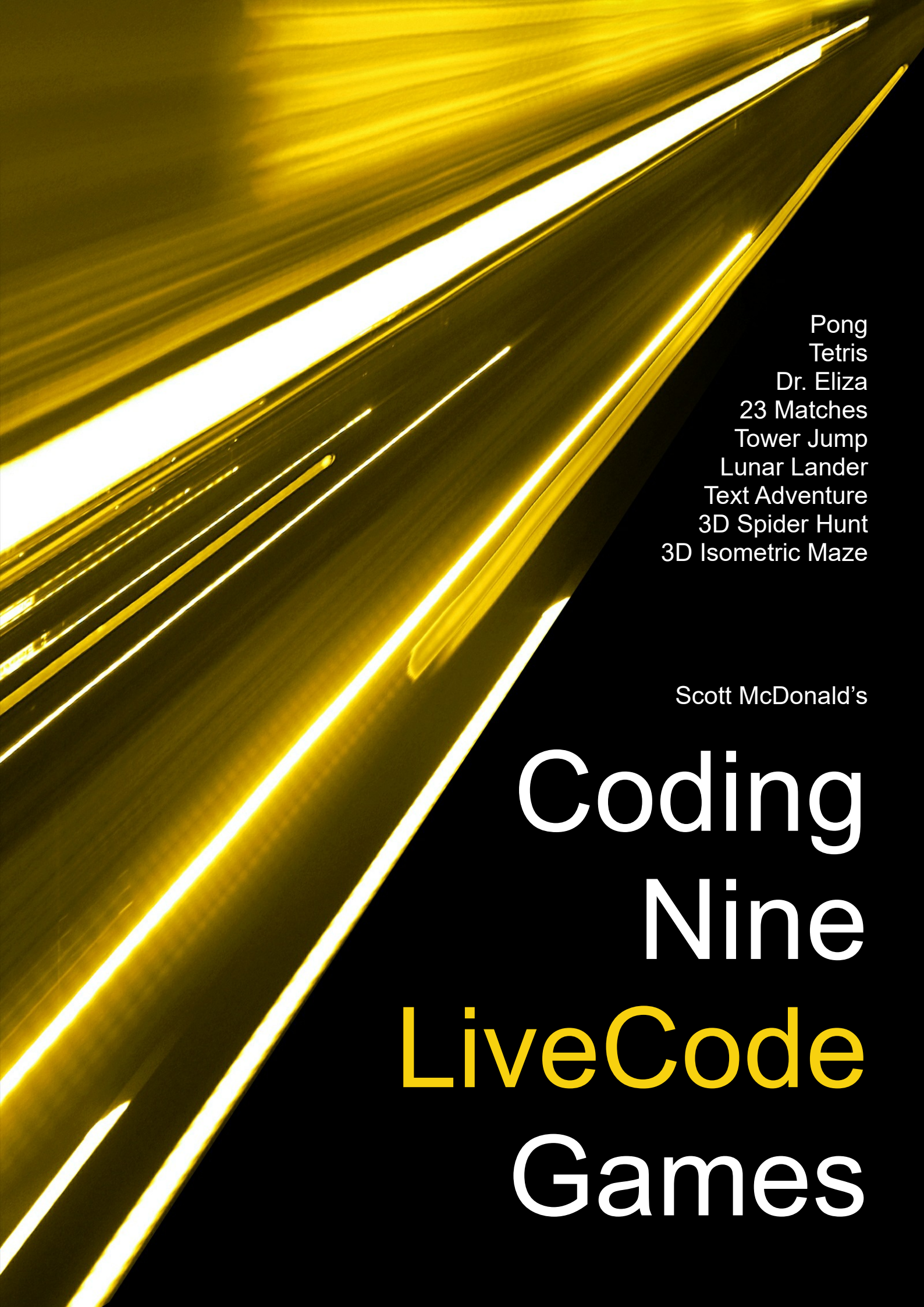


Pong
Tetris
Dr. Eliza
23 Matches
Tower Jump
Lunar Lander
Text Adventure
3D Spider Hunt
3D Isometric Maze

Scott McDonald's

Coding Nine LiveCode Games

Note about the code listings

Long lines of code that are longer than the page width, show the rest of the line as right justified text on the following line. For example in this code fragment

```
repeat for each line loopImage in sImageList
  set the loc of image ID (item 3 of loopImage) to item 1 of loopImage +
                                sScreenOffsetX,item 2 of loopImage + sScreenOffsetY
end repeat
```

The set statement is a single line code when viewed in the source file. Long strings enclosed in quotations may not follow this convention when there is little ambiguity.

Contents

Introduction

LiveCode Games

Coding Nine LiveCode Games has content from my old *LiveCode Game Developer* blog plus 3 new LiveCode games:

- Tower Jump
- Doctor Eliza
- Lunar Lander

This eBook PDF was originally published and sold in 2014, but is now released for free under the Creative Commons license: Attribution-NonCommercial 4.0 International.

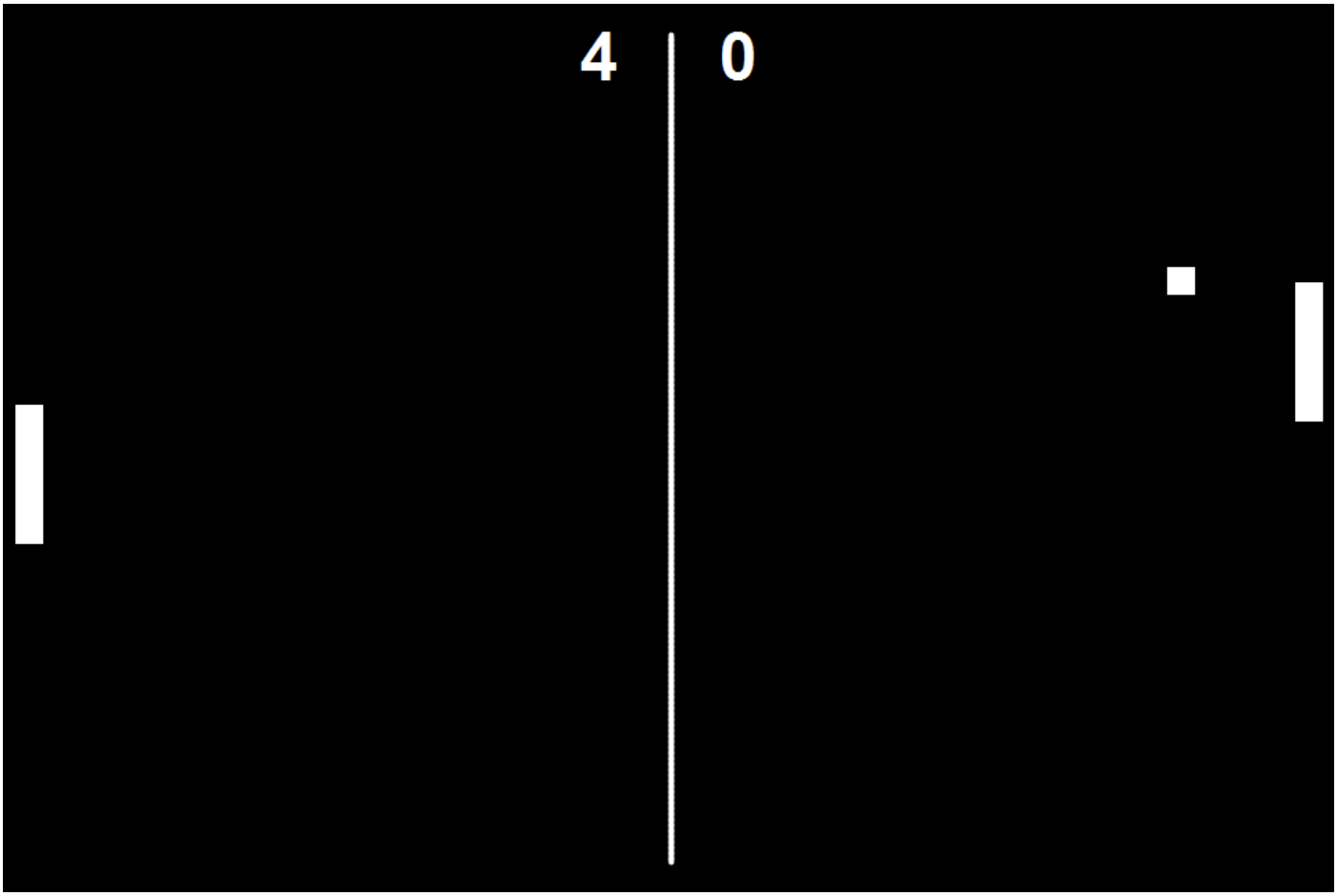
With LiveCode you can make a game with a fraction of the effort of other languages.

With the exception of Spider Hunt, all the games in this book are less than 500 lines long, and can be coded in a weekend

By playing the games and examining the techniques used, you will discover ways of creating interesting games with a minimum of code, that are fun to play. Each chapter includes suggestions for enhancing the existing code to make each game even better. How will you improve each program, and make it your own unique version of a classic game?

Happy LiveCoding!

Scott McDonald.



I think the computer cheats, that’s me at the right on zero.

Game 1

Pong

Pong is good place to start making games. The name is an abbreviation of Ping Pong (table tennis). A "ball" (actually a small square of white light) bounces around the screen, while two players control a paddle (a skinny white rectangle) at the left and right ends of the screen. The ball bounces off the top and bottom edges of the screen, but if your paddle is not in position to hit it back when it gets to your end of the screen, then off the edge it goes and your opponent gets a point.

Practically every language has code for the original game of Pong. So this makes a good starting point and allows you to compare LiveCode with other languages. It is also short to code in LiveCode and so a good game to start with.

In this game you will see:

- A game loop in action
- One way to make a ball bounce
- LiveCode graphics used instead of images
- "Artificial Intelligence" code

Before examining the game, a few words about a couple of coding conventions use in this book. Strict Compilation Mode is on in the LiveCode IDE Preferences so variables are always declared. It may seem time-saving to keep Strict Compilation Mode turned off and not declare any variables, but don't do it. Sure in a small game all goes well but once your games get larger, wasting an hour trying to find out why your program doesn't work is not fun. You then find a typo that Strict Compilation Mode would have highlighted immediately you did a save.

With variables, 3 conventions are used.

- Variable names at the card or stack level begin with a lowercase s followed by an uppercase letter.
- Variable names in a handler begin with a lowercase letter.
- Variables use CamelCase, where a capital letter is used if the variable name is a compound word.

These conventions are a personal preference and make code more readable to me. Name your variables any way you want, but trust me about Strict Compilation Mode.

With that out of the way, let's look at some code.

Further below is the StartGame handler. (It may be easier to download the stack and have the Main card script open in LiveCode, while reading this page.) This is called to initialize all the variables declared on the card and to call GameLoop for the first time.

The first 3 lines remove any messages that are in the message queue. This is normally a good idea when starting a game, because if there are already messages pending that will call your GameLoop (which can easily occur during development and debugging) if you don't first remove them, weird things could happen and you will be left scratching your head wondering what has gone wrong.

Many of the variables don’t change once they are initialized. So why waste time declaring variables at the top of the card, and then filling them with values that are not going to change during the running of the game? Three reasons:

- Putting the width of the graphic for the ball into the variable sBallWidth saves time later, typing sBallWidth is much easier than the width of graphic "ball".
- Putting, for example, a value into sPlayerPaddleX based on the card size, means that if the size of the card is changed code still works.
- When your game runs it takes LiveCode much longer to process the width of graphic "ball" than getting the value stored in sBallWidth.

For turn based games, or Pong where not much processing is required, the extra time taken in reason (3) is unimportant, but reasons (1) and (2) still apply. A little investment at the start of your coding means less typing and more flexibility later on. When your games get more complex and require smooth animation, the faster processing from (3) will be an asset. Why not start with good habits that will repay you ten-fold in the future?

```
command StartGame
  repeat for each line loopLine in the pendingmessages
    cancel item 1 of loopLine
  end repeat

  put the width of graphic "ball" into sBallWidth
  put the height of graphic "ball" into sBallHeight

  put the width of this card into sCardWidth
  put the height of this card into sCardHeight

  put the width of graphic "playerPaddle" into sPaddleWidth
  put the height of graphic "playerPaddle" into sPaddleHeight

  put sCardWidth-sPaddleWidth into sPlayerPaddleX
  put sPaddleWidth into sCPUPaddleX
  put sCardHeight div 2 into sCPUPaddleY
  set the loc of graphic "cpuPaddle" to sCPUPaddleX,sCPUPaddleY

  put kSpeedX into sSpeedX
  put kSpeedY into sSpeedY
  put sCardWidth div 2 into sBallX
  put sCardHeight div 2 into sBallY
  set the loc of graphic "ball" to sBallX,sBallY

  put 0 into field "cpuScore"
  put 0 into field "playerScore"
  hide field "GameOver"

  put 1 into sCPUState
```

```
  put kCPUSpeed into sCPUSpeed

  put true into sBallInPlay
  put false into sGameOver
  -- start motion
  GameLoop
end StartGame
```

After setting up the variables, at the end of StartGame call GameLoop. Games with animation (that happen even when the player is doing nothing), nearly always have a "game loop". These actions are continually repeated to check for events and update the display. Below is the game loop for Pong. It does a set of actions and unless the game is over, it calls itself in 2 ticks. Since there are 60 ticks each second, sending the GameLoop message in 2 ticks results in a game that runs at 30 frames per second.

So what does GameLoop do? It moves the ball, does the AI (artificial intelligence, the overblown term for the code that moves the CPU paddle), checks for collisions with the card edges and paddles, and finally checks whether the ball has left the card.

```
command GameLoop
  MoveBall
  ProcessAI
  CheckCollisions
  CheckMissedBall
  if sGameOver then
    GameOver
  else
    -- run game at 30 fps
    send "GameLoop" to me in 2 ticks
  end if
end GameLoop
```

MoveBall adds the speed of the ball in horizontal and vertical directions to the current position. Then the loc (location) of the ball is set to the new position.

In LiveCode a graphic is a visual object that LiveCode draws. This is different from a bitmap image that you load into your game after creating it in a separate image editor. For a simple game like Pong you can draw directly on the card the required graphics in the LiveCode IDE before you start coding. This can save time in the early stages of development, and in the case of Pong is more than adequate for the finished game.

```
command MoveBall add sSpeedX to sBallX
  add sSpeedY to sBallY
  set the loc of graphic "ball" to sBallX,sBallY
end MoveBall
```

After moving the ball the paddle on the left is moved according to the AI. If you are thinking this isn't artificial intelligence, fair enough, but ProcessAI is a useful label for the handler that controls the CPU player.

This handler uses what is called a state machine. A state machine is good for algorithms that have a number of different "states" or conditions that each do a different action, and normally depend on the previous state. A state machine can be represented as a variable at the card level to keep track of the state, and a switch statement.

Here the CPU player has four possible states:

1. Getting ready to move randomly
2. Moving randomly while waiting for you to hit the ball
3. Getting ready to move towards the incoming ball
4. Moving towards the ball.

If you look carefully at this code you may wonder how the state changes from 2 to 3 and from 4 back to 1. The answer is that sCPUState is also set outside of this handler. In CheckCollisions, examined later, sCPUState is changed after the ball hits a paddle.

Note, state machines are not limited to AI applications and games. If you have code with lots of if then statements, where this depends on that, which affects this, and you find yourself having trouble understanding the logic, then consider converting your code to a state machine.

The random numbers put into the sCPUCounter, sCPUWobble1 and sCPUWobble2 variables give the CPU paddle some "personality." Even though you (the human player) know you are playing against a stupid computer, it adds to the fun if the computer acts more human. The wobbling in the paddle movement helps you pretend (even if only subconsciously) that you are playing against a machine who, just maybe, is thinking.

```
command ProcessAI
  local paddleRect
  switch sCPUState
    case 1 -- prepare to move randomly
      put random(60) into sCPUCounter
      put -sCPUSpeed div 2 into sCPUSpeed
      put 2 into sCPUState
      break
    case 2 -- move randomly
      subtract 1 from sCPUCounter
      if sCPUCounter > 0 then
        add sCPUSpeed to sCPUPaddleY
        set the loc of graphic "cpuPaddle" to sCPUPaddleX,sCPUPaddleY
        if sCPUPaddleY < sPaddleHeight then put abs(sCPUSpeed) into sCPUSpeed
        if sCPUPaddleY > sCardHeight - sPaddleHeight then
          put -abs(sCPUSpeed) into sCPUSpeed
        end if
      else
        put -sCPUSpeed into sCPUSpeed
        put random(60) into sCPUCounter
      end if
      break
    case 3 -- prepare to follow ball
      put 4 into sCPUState
      put random(sPaddleHeight div 2) into sCPUWobble1
      put random(sPaddleHeight div 2) into sCPUWobble2
      break
    case 4 -- following ball
      put the rectangle of graphic "cpuPaddle" into paddleRect
      -- only move the paddle if not inline with ball
      if (sBallY < item 2 of paddleRect + sCPUWobble1) or (sBallY > item 4 of
                                                          paddleRect - sCPUWobble2) then
        if sBallY < item 2 of paddleRect + sCPUWobble1 then put -kCPUSpeed into
                                                                sCPUSpeed
        if sBallY > item 4 of paddleRect - sCPUWobble2 then put kCPUSpeed into
                                                                sCPUSpeed
        add sCPUSpeed to sCPUPaddleY
        set the loc of graphic "cpuPaddle" to sCPUPaddleX,sCPUPaddleY
      end if
      break
  end switch
end ProcessAI
```

Next in the GameLoop is a call to CheckCollisions to check when the ball hits the top or bottom of the card or a paddle. This is the longest handler in Pong. It could be shorter, but the original Pong makes the gameplay more interesting by changing the angle of bounce depending on where the paddle is hit.

Many of the versions of Pong you find on the internet use a simple bounce algorithm where the ball bounces off the paddle at the supplementary angle of the impact. This is easy to code, but results in a ball that bounces around the screen in a less interesting way, often at angles that are multiples of 45 degrees. Boring.

Instead, here the the paddle is divided into 9 regions using this line:

```
put round(9 * ((sBallY - item 2 of paddleRect) / sPaddleHeight)) into paddleRegion
```

Then the vertical speed of the ball is varied depending on the value of paddleRegion. Since the horizontal speed of the ball never changes, increasing or decreasing the vertical speed has the effect of changing the angle of bounce. This has a side affect that at the steeper angles the ball moves faster in the direction it is heading. This adds to the illusion that you have just pulled off a tricky shot and given the ball an extra hard hit.

At the bottom of CheckCollisions a simple check is done to make the ball bounce off the top and bottom edges of the card. In CheckCollisions the code used for the two paddle collisions is almost identical. To reduce the amount of lines in Pong, this common code could be put into a separate handler and called twice. This time laziness got the better, and it worked. Even though it could have been made more elegant.

```
command CheckCollisions
  local buffer,paddleY,paddleRect,paddleRegion
  -- hit player paddle?
  if (sBallX > (sPlayerPaddleX - (sBallWidth + sPaddleWidth) div 2)) and sBallInPlay
                                                                                then
    put the rectangle of graphic "playerPaddle" into paddleRect
    if (sBallY > item 2 of paddleRect - sBallHeight div 2) and (sBallY < item 4 of
                                                  paddleRect + sBallHeight div 2) then
      put 3 into sCPUState
      put -sSpeedX into sSpeedX
      -- needed in case ball is "inside" the the paddle during collision
      put sPlayerPaddleX - ((sBallWidth + sPaddleWidth) div 2) into sBallX
      -- divide paddle into 9 regions from 0 to 8
      put round(9 * ((sBallY - item 2 of paddleRect) / sPaddleHeight)) into
                                                                      paddleRegion
      if paddleRegion < 4 then
        put -kSpeedY - 0.2 * (3 - paddleRegion) into sSpeedY
      else
        if paddleRegion > 4 then
          put kSpeedY + 0.2 * paddleRegion into sSpeedY
        else
          put 0 into sSpeedY
        end if
      end if
    else
      put false into sBallInPlay
    end if
  end if
  -- hit CPU paddle?
  if (sBallX < (sCPUPaddleX + (sBallWidth + sPaddleWidth) div 2)) and sBallInPlay then
    put the rectangle of graphic "cpuPaddle" into paddleRect
    if (sBallY > item 2 of paddleRect - sBallHeight div 2) and (sBallY < item 4 of
                                                  paddleRect + sBallHeight div 2) then
      put 1 into sCPUState
      put -sSpeedX into sSpeedX
      -- needed in case ball is "inside" the the paddle during collision
      put sCPUPaddleX + ((sBallWidth + sPaddleWidth) div 2) into sBallX
      -- divide paddle into 9 regions from 0 to 8
      put round(9 * ((sBallY - item 2 of paddleRect) / sPaddleHeight)) into
                                                                      paddleRegion
      if paddleRegion < 4 then
        put -kSpeedY - 0.2 * (3 - paddleRegion) into sSpeedY
      else
        if paddleRegion > 4 then
          put kSpeedY + 0.2 * paddleRegion into sSpeedY
        else
          put 0 into sSpeedY
        end if
      end if
    else
      put false into sBallInPlay
```

```
    end if
  end if
  -- hit top or bottom wall
  if (sBallY < sBallHeight div 2) or (sBallY > sCardHeight - sBallHeight div 2) then
                                                                    put -sSpeedY into sSpeedY
end CheckCollisions
```

So the ball has moved, the CPU has moved the left paddle, and we have checked whether the ball has hit anything. What if a player misses the ball? Then CheckMissedBall is next in the game loop.

Here a couple of checks are made. If the ball goes off the left or right edge of the card, update the appropriate score. If a player reaches 15, the game is over.

If the game is not over, a ball is "served" towards the player who lost the point. Here an interesting line is:

```
put any item of "-1,1" * random(kSpeedY) into sSpeedY
```

The "any item" phrase is one of those powerful LiveCode features that can substitute for a few lines of code in other languages. The any keyword means randomly return one of the items in the list. Similar sort of code can be used for randomly picking a line or character too. There are other ways to set the vertical speed, but this code makes it clear that the direction will be either positive or negative.

```
command CheckMissedBall
  local ballMissed
  if sBallX < -sBallWidth then
    put true into ballMissed
    add 1 to field "playerScore"
    if field "playerScore" = 15 then put true into sGameOver
    -- serve ball towards CPU
    put -kSpeedX into sSpeedX
    put 3 into sCPUState
  end if
  if sBallX > sCardWidth + sBallWidth then
    put true into ballMissed
    add 1 to field "cpuScore"
    if field "cpuScore" = 15 then put true into sGameOver
    -- serve ball towards player
    put kSpeedX into sSpeedX
    put 1 into sCPUState
  end if
  if ballMissed and not sGameOver then
    wait 2 seconds
    put true into sBallInPlay
    put any item of "-1,1" * random(kSpeedY) into sSpeedY
    put sCardWidth div 2 into sBallX
    put sCardHeight div 2 into sBallY
    set the loc of graphic "ball" to sBallX,sBallY
  end if
end CheckMissedBall
```

If it is game over the GameOver card is shown after a short pause. Because the scores are stored in the fields that show the information, there is a little trick used to access the scores. Note the line:

```
if field "playerScore" of me > field "cpuScore" of me then
```

includes the of me phrase. This is needed because after:

```
go to card "GameOver"
```

the score fields are no longer visible and cannot be accessed without some extra information. The *of me* means, "refer to the object that is on the object that contains this script." Since all this code is the script for the card that contains the fields, this:

```
field "playerScore" of me
```

works in the same way as:

```
field "playerScore" of card "Main"
```

but is simpler and if you change the name of the card, you do not need to update your code. How convenient is that?

```
command GameOver
  show field "GameOver"
  wait 2 seconds
  go to card "GameOver"
  if field "playerScore" of me > field "cpuScore" of me then
    put "You Win. Do you want to play again to let me have another go?" into field
                                                                        "GameOverMessage"
  else
    put "I Win. Do you want to play again to try and beat me?" into field
                                                                        "GameOverMessage"
  end if
end GameOver
```

Near the end now. There are two more handlers. To remind me that these handlers are for messages that LiveCode sends they begin with a lowercase letter. The moveMouse handler updates the position of the player paddle. The mouseMove message is sent regularly by LiveCode as you move the mouse, so if you were wondering why GameLoop has no code for moving the player paddle, this is the reason. It happens in this handler asynchronously from the game loop.

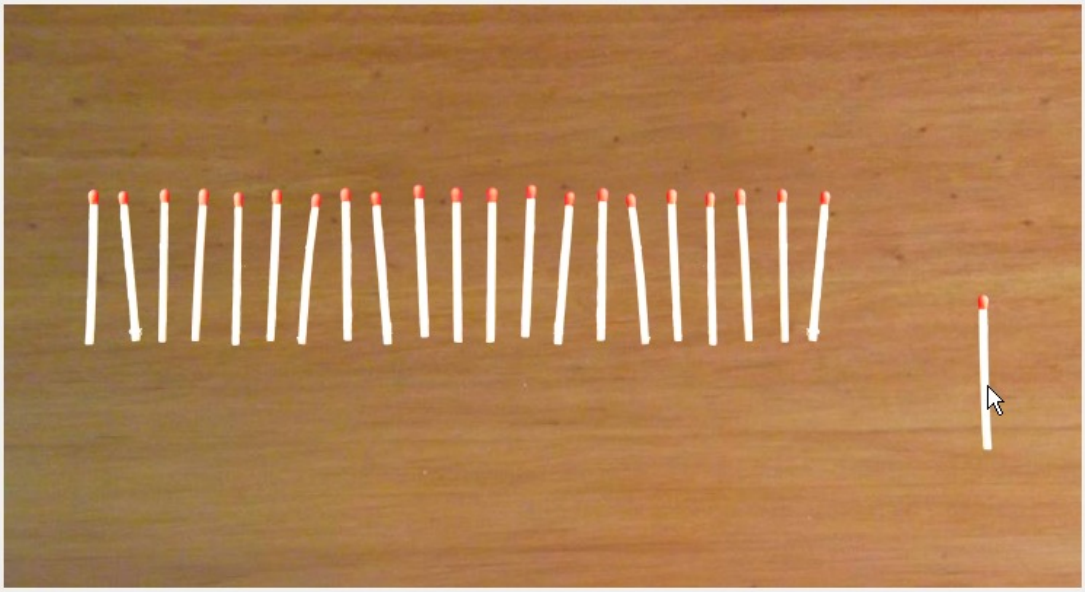
```
command openCard
  StartGame
end openCard

on mouseMove pMouseH,pMouseV
  set the loc of graphic "playerPaddle" to sPlayerPaddleX,pMouseV
end mouseMove
```

That's about it. So without working hard to keep the code small, a fully working Pong re-creation in under 200 (not counting comments) lines of LiveCode.

Here are some suggestions for improving the game.

- The original Pong would not let the paddles go all the way to the top and bottom edges of the screen. This was to stop games going forever when the player is good and never misses. The gap made some balls impossible to hit. Make it so the CPU player behaves this way.
- Make a two player game where a human controls the paddle on the left.
- Increase the speed of the ball when there has been no has misses for a while.

Each turn you can take away 1, 2 or 3 matches. To win, force the CPU to take the last match.



22 matches remain.

Play Again

Finished Turn

Can you think up a winning strategy and out smart the computer?

Game 2

23 Matches

Another simple game. 23 Matches is a completely unoriginal classic game. There is a row of matches. You and another player, here you play against the CPU, take turns at removing 1, 2 or 3 matches from the end at the right. Your goal is to get to a position where at the end of your turn only one match remains. You do this because the player who is forced to take the last match is the loser. So if there is only 1 match left at the start of the CPU's turn you win! You will need to play smart because the CPU plays to win, some of the time.

In this game you will see:

- A turn based game without a game loop
- Object names used to simplify coding
- How to autonomously move an image with a single line code
- Images you can click and drag around the screen

Further below is the complete StartGame handler. This handler is called from the openCard handler and when the Play Again button is clicked. There are only 5 local variables on the card so this handler is quite short.

The sSmartCPU variable keeps track of whether the CPU player is going to play to win, or in a random way. When sSmartCPU is true, the algorithm used for the CPU player is optimal where (unless you are careful) it will win. When sSmartCPU is false, the CPU player takes a random number of matches each turn, and it can only win by luck.

The line:

```
put not sSmartCPU into sSmartCPU
```

means the CPU alternates between smart and random each time you play. This gives you a fighting chance of winning every second game. We don't want you to get too discouraged by making the CPU too smart. Who wants to play against a computer who wins every time? Boring.

Next a few variables are initialised so the human player goes first, 23 matches are visible, and none taken away.

Then a loop lines up the matches in neat row. Each match is a PNG image that was added to the card with the Import As Control, Image File command in the File menu. Actually, this command wasn't used for all the matches. After using it a few times, realising there must be a better way, the Import As Control, All Images in Folder command was used instead. After putting the remaining image files into a folder of their own, this command made the process too easy.

The images were created by taking a photo of actual matches on a black background and then using a bitmap image editor to remove the background. The surrounds were left transparent and each match was cut and copied into a separate PNG image file 25 by 100 pixels in size. Each image is larger than the actual match to make it easier to click and drag later in the game. The PNG image format allows transparency in the bitmap. You can also use the GIF format if you like. The images were named Match1.png, Match2.png, Match3.png and so on.

Using a row of real match images gives the game some class, and not repeating the same image 23 times helps make an otherwise simple game a little more impressive. After creating the 23 image controls, the .png extension was removed from the name of each control in the LiveCode Property Inspector.

This work and preparation with the images makes the loop to lay out the matches in a line a simple piece of code. Each image is made visible because after a game all the matches are invisible after being dragged off the "table". The table is another photo that has been suitably sized in a bitmap editor and imported as a control and then put behind the matches with the Send to back command in the Object menu.

Lastly, the two buttons are disabled and UpdateView is called to display the status below the table.

```
command StartGame
  local x
  put not sSmartCPU into sSmartCPU
  put empty into sMatchToDrag
  put true into sYourTurn
  put 23 into sNumberOfVisibleMatches
  put 0 into sRemoveCount
  put 100 into x
  repeat with loopMatchNumber = 1 to sNumberOfVisibleMatches
    set the loc of image ("Match" & loopMatchNumber) to x,280
    add 25 to x
    show image ("Match" & loopMatchNumber)
  end repeat
  disable button "butnPlayAgain"
  disable button "butnFinishTurn"
  UpdateView
end StartGame
```

UpdateView has a few conditional statements to update the status indicating whose turn it is, and the number of matches that are left. An interesting part of this handler is the enabling and disabling of the Play Again and Finished Turn buttons.

In the first version of LiveCode 23 Matches, the buttons were never disabled. This meant that if a button was clicked at the wrong time, for example, a click on Finished Turn before taking any matches, required a message saying in effect, "You can't do that." That was fine, but depending on the game, it is better to prevent a player from being able to do an invalid action. Allowing an action to be selected and then saying don't do that, is annoying. Also, not allowing invalid actions can make for easier programming because you no longer need to consider and handle events that shouldn't happen anyway. Hence the enabling and disabling of the buttons.

```
command UpdateView
  if sYourTurn then
    if sNumberOfVisibleMatches = 1 then
      enable button "butnPlayAgain"
      put "Game Over. I WIN!" into field "labeTurn"
    else
      put "Your turn. " & RemainingMatches(sNumberOfVisibleMatches) into
                                                                    field "labeTurn"
    end if
  else
    if sNumberOfVisibleMatches < 1 then
      enable button "butnPlayAgain"
      put "Game Over. You WIN. Do you want to play again to give me a chance
                                          to show how smart I am?" into field "labeTurn"
    else
      put "My turn." into field "labeTurn"
    end if
    disable button "butnFinishTurn"
    put 0 into sRemoveCount
  end if
end UpdateView
```

The FinishedTurn handler is called by the button "butnFinishTurn" object. It ends your turn and then calls ProcessAI for the CPU to have a go.

```
command FinishTurn
  put false into sYourTurn
  UpdateView
  ProcessAI
end FinishTurn
```

When the CPU is playing a smart game, a simple mathematical formula with the mod operator sets the best number of matches to take. If you want to understand the details of how this works, a careful search on Google may reveal the answer.

If the CPU is not playing smart, then a random number of matches are taken. Always taking at most one less than the number of visible matches if there are less than 4. You don't want the CPU to lose by taking the last match during its own turn!

A loop then repeatedly waits a random interval, moves the right most match off the table, and hides the image. The move command is one of those neat LiveCode features that can achieve what would take many lines in a less high-level language.

```
move image ("Match" & sNumberOfVisibleMatches) to (880,150+random(300)) in 60 ticks
```

This command moves a control, here the match at the right end of the line, to a location in a set amount of time. This is where the name of the image control for each match simplifies the code. The name of the visible match at the right end is always "Match" followed by the number of visible matches.

The match is moved to a location off the right end of the table, with a random value used for the y position to make the movement more interesting. Lastly, the move takes 60 ticks which is the same as one second. As the number of matches gets less and each match needs to move a further distance, the speed of movement increases so the distance is still covered in the one second. Perhaps it would be more "realistic" if the number of ticks had a random variation, but the variation in the direction of the match movement is sufficient to make it seem more human. As with Pong, in a simple game like this, little effects that make the CPU player seem more human make for more enjoyable gameplay.

After the loop to move the matches if there are any left, then it is your turn and the status is updated.

```
on ProcessAI
  local takeCount
  put 0 into takeCount
  if sSmartCPU then put (sNumberOfVisibleMatches - 1) mod 4 into takeCount
  if takeCount = 0 then put random(max(1,min(3,sNumberOfVisibleMatches - 1))) into
                                                                          takeCount
  repeat takeCount times
    wait 1 + random(60) ticks
    move image ("Match" & sNumberOfVisibleMatches) to (880,150+random(300)) in
                                                                           60 ticks
    hide image ("Match" & sNumberOfVisibleMatches)
    subtract 1 from sNumberOfVisibleMatches
  end repeat
  if sNumberOfVisibleMatches > 0 then put true into sYourTurn
  UpdateView
end ProcessAI
```

RemainingMatches is a small handler that is called when displaying the number of remaining matches. Computers are stupid, but it is annoying when software uses incorrect grammar like 1 matches remain. Sure, you could get around the problem by using text like 1 match(es) remain(s) but that is ugly and emphasises that you are playing against an unthinking computer. Given only a few lines are needed, why not use a handler like RemainingMatches to give the game some polish?

```
function RemainingMatches pNumber
  if pNumber = 1 then
    return pNumber & " match remains."
  else
    return pNumber & " matches remain."
  end if
end RemainingMatches
```

The next three handlers let the player click and drag to remove matches from the table. In mouseDown a check is made that it is actually your turn and then there is an interesting line.

```
put the short name of the target into targetName
```

The target is the object that first receives a message, which in this case is the mouseDown messge sent to the image control you clicked on. Using the short name phrase is a way of getting just the name, i.e. what you see in the Name field of the Property Inspector for the control and throwing away details like the type of control it is. Since we know only the match image controls have a short name starting with "Match" the short name is adequate to identify a click on a match.

When a match has the mouseDown message, a check is made that it is the one at the right end and the name is stored in the sMatchToDrag variable on the card. This variable is on the card (not in the handler) because it must maintain its value after the mouseDown handler ends.

```
on mouseDown
  local targetName
  if sYourTurn then
    put the short name of the target into targetName
    if targetName = "Match" & sNumberOfVisibleMatches then
      put targetName into sMatchToDrag
    end if
  end if
end mouseDown
```

While the mouse button is kept pressed down, the mouseMove handler moves the match object until mouseUp is called when the button is released. Then a check is made to see if the match has been dragged off the table. If it is, the match image is hidden and the number of visible matches is reduced by 1. If this is the third match that has been removed the turn automatically ends, otherwise the status is updated and the Finish Turn button is enabled.

The button is enabled here because at the start of your turn it is disabled. This ensures that you take at least one match before ending your turn.

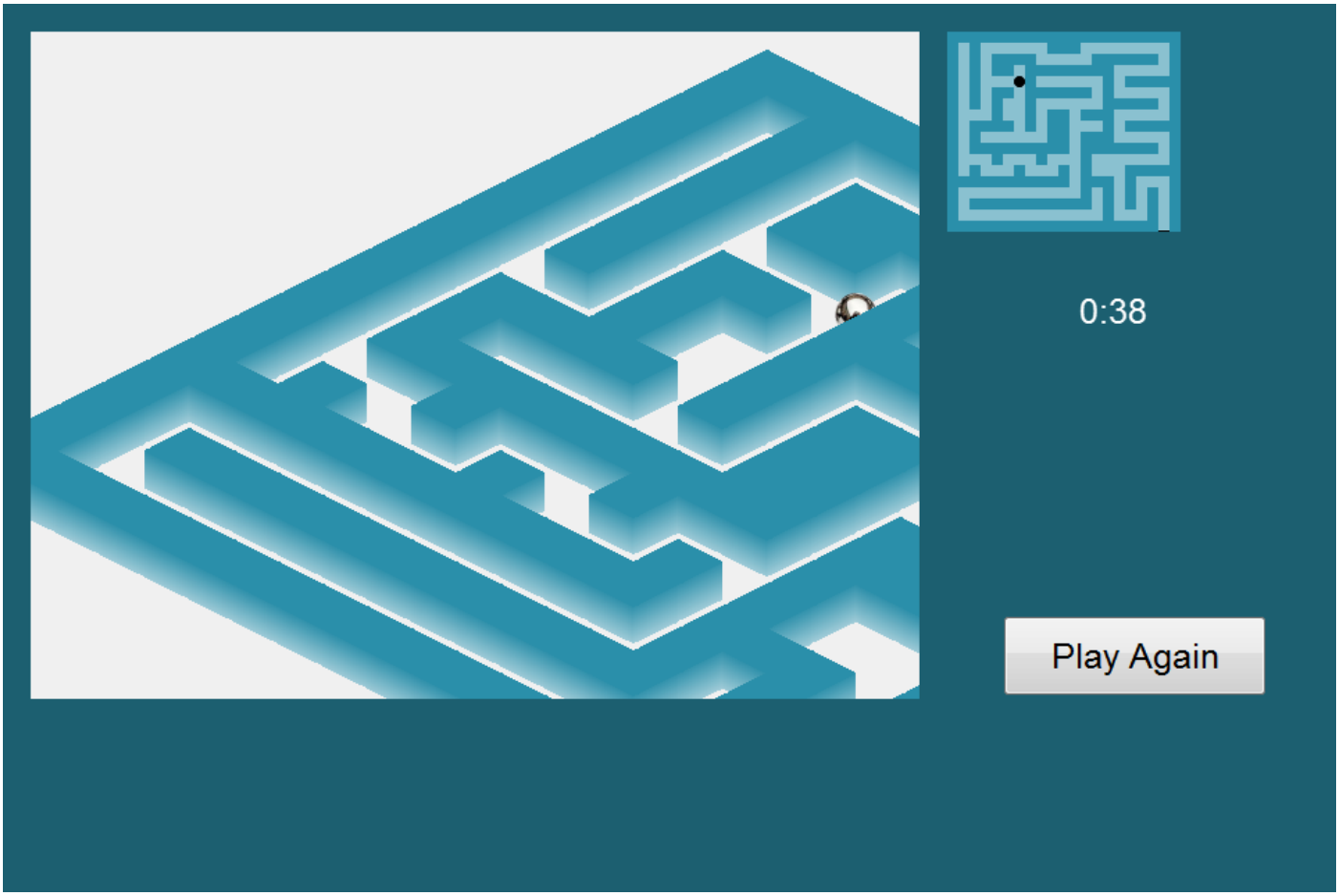
```
on mouseMove
  if sMatchToDrag is not empty then
    set the loc of image sMatchToDrag to the mouseLoc
  end if
end mouseMove

on mouseUp
  if sMatchToDrag is not empty then
    if the mouseLoc is not within the rectangle of image "Table.png" then
      hide image sMatchToDrag
      subtract 1 from sNumberOfVisibleMatches
      add 1 to sRemoveCount
      if sRemoveCount = 3 then
        FinishTurn
      else
        put RemainingMatches(sNumberOfVisibleMatches) into field "labeTurn"
      end if
      enable button "butnFinishTurn"
    end if
    put empty into sMatchToDrag
  end if
end mouseUp
```

This ends the code for the game. Nothing interesting in the openCard handler to comment on. As you can see a game loop is not used. Instead nothing happens until a button is clicked, or an image control (of a match) has a click and drag with the mouse. The 23 Matches code does nothing until the LiveCode engine (which is working all the time behind the scenes) sends a message about an event started by a mouse click.

That's about it. Preparing the image controls seemed like almost as much work as the actual coding. The ability of LiveCode to work with images, including motion, in few lines makes the required coding quite easy. That's one reason to use LiveCode for making games and other software.

Here are some suggestions for improving the game.

- Make the game alternate between who goes first.
- Make the CPU play to lose when the human player is not winning any games.
- Allow matches to be taken from anywhere in the row.



A 3D maze and mini-map in a few hundred lines of code? Thank you LiveCode.

Game 3

3D Isometric Maze

Here is a puzzle game. It is a maze that uses an isometric projection to achieve a 3D effect. The maze is easy to solve, but it can easily be expanded and have more complicated paths to increase the challenge. To make it a little more interesting a timer is included that counts the seconds until you finish. It is you against the clock.

In this game you will see:

- One way to store room data to allow easy changes to the maze
- How 2D images and layers give the illusion of depth
- A technique to scroll a maze that is larger than the viewport
- A minimap to show the entire maze

Before examining this game, a warning about versions of LiveCode from 6.0 and newer. Close the Project Browser before testing games like this that frequently change the layer property of controls. If the Project Browser is open when layers are changing, your program will not run smoothly and the resulting jerky animation may mislead you to think that the LiveCode Engine is not up to the task.

StartGame is the longest handler in this game, and so it is split up into smaller pieces of code below.

The first 3 lines remove any messages that are in the message queue. This is a good idea when starting a game, because if there are already messages pending that call your GameLoop (which can happen during development) if you don't first remove them the game animation will be unpredictable.

```
repeat for each line loopLine in the pendingMessages
  cancel item 1 of loopLine
end repeat
```

During the rest of StartGame there is a lot of creating and deleting of images and graphics so the screen is locked to speed up the execution. Here all the images and graphics from the previous run of the game are deleted. In a finished game this would not be necessary, but since this a demo and the maze may be changed those controls are deleted. Both loops use the way that every image and graphic that is created from code at

runtime has a name that begins with the image or graphic respectively. This means that controls that I want to keep between each run have names that do not begin with either of these are are kept.

```
set the cursor to busy
lock screen
repeat with loopIndex = the number of images down to 1
  if the short name of image loopIndex begins with "image" then delete image loopIndex
end repeat
repeat with loopIndex = the number of graphics down to 1
  if the short name of graphic loopIndex begins with "graphic" then delete graphic
loopIndex
end repeat
```

The roomMap list represents the maze which is made up of equal sized squares which are referred to as cells. Each character of each line in roomMap represents a cell where W is a wall, B is the initial position of the ball and E the exit. The size of the maze is then put into sRoomMaxX and sRoomMaxY. Variables that include Room in the name refer to the cells of the maze which in this case is 11 by 11 in size. Other variables you later see that refer to X and Y and do not include Room are screen coordinates.

```
put empty into roomMap
put "WWWWWWWWWWWWWWWWWWWWW" & cr after roomMap
put "WBW       WWW         W" & cr after roomMap
put "W W WWWW     WWWWWW W" & cr after roomMap
put "W W WW WWWWWWWW      W" & cr after roomMap
put "W W    W       W WWWWW" & cr after roomMap
put "W W WW WWWWWW W      W" & cr after roomMap
put "W W W  W       WWWWW W" & cr after roomMap
put "W   W  W WW WWW      W" & cr after roomMap
put "W WWWW W WW   W WWWWW" & cr after roomMap
put "W W       WW WWW     W" & cr after roomMap
put "W WWWWWWWWW WWWWWWW W" & cr after roomMap
put "W   W   W   W        W" & cr after roomMap
put "W W   W   W W   WWWWW" & cr after roomMap
put "WWWWWWWWWWW WWW W   W" & cr after roomMap
put "W            W W W W W" & cr after roomMap
put "W WWWWWWWWWW W W W W" & cr after roomMap
put "W              W   W W" & cr after roomMap
put "WWWWWWWWWWWWWWWWWWWEW" & cr after roomMap
put the number of chars in line 1 of roomMap into sRoomMaxX
put the number of lines in roomMap into sRoomMaxY
```

A single 64 by 64 bitmap is used for the walls and since they move when scrolling, the layer mode is set to dynamic. These two properties are set in the templateImage, so they are inherited by every image for the walls of the maze.

```
set the rectangle of the templateImage to 1,1,64,64
set the layerMode of the templateImage to "dynamic"
set the acceleratedRendering of this stack to true
```

The image for the walls was made in Paint.NET, and surprisingly a single image is all that is required for the entire maze. The shading makes it look interesting, but was tricky to get right in the bitmap editor so it flows smoothly when the images are repeated to make a long wall.

Next some variables are set. There will be more detail about the viewport later. The sFloorList stores information for translation from the screen coordinates to each cell in the room.

```
put 0 into sMaxX
put 0 into sMaxY
put 0 into sScreenOffsetX
put 0 into sScreenOffsetY
put kViewPortX1 + 200,kViewPortY1 + 100,kViewPortX2 - 200,kViewPortY2 - 100 into
sViewPortMiddle
put kCornerX into cornerX
put kCornerY into cornerY
put empty into sImageList
```

```
put empty into sFloorList
```
The next loop fills the sRoomArray, sFloorList and sImageList. With this isometric projection the floor of each cell is represented by a rectangle that is 64 pixels wide and 32 pixels high. Part of the 3D illusion is because your brain assumes the room is a grid of squares, but since they look like diamonds you think they are being viewed at an angle. Each image overlaps with those adjacent, by 32 pixels along the horizontal axis and 16 pixels along the vertical axis. Next when the layer of image is set the 3D illusion is complete.

```
put 1 into roomY
repeat for each line loopRoomRow in roomMap
  put cornerX into x
  put cornerY into y
  put 1 into roomX
  repeat for each char loopRoomItem in loopRoomRow
    put loopRoomItem into sRoomArray[roomX, roomY]
    put y,y+32,x,y+16,roomX,roomY & cr after sFloorList
    switch loopRoomItem
      case "B"
        put roomX into sBallRoomX
        put roomY into sBallRoomY
        break
      case "E"
        put roomX into sExitRoomX
        put roomY into sExitRoomY
        break
      case "W"
        create image
        put the short ID of the last image into imageID
        put image "tileWall" into image ID imageID
        set the loc of image ID imageID to x,y
        put x,y,imageID & cr after sImageList
        break
    end switch
    put max(x,sMaxX) into sMaxX
    put max(y,sMaxY) into sMaxY
    add 32 to x
    add 16 to y
    add 1 to roomX
  end repeat
  subtract 32 from cornerX
  add 16 to cornerY
  add 1 to roomY
end repeat
put (sMaxX div kScrollDelta) * kScrollDelta into sMaxX
put (sMaxY div kScrollDelta) * kScrollDelta into sMaxY
```

After creating the images and filling the image list with each image ID and the screen coordinates, the list is sorted so the images with a smaller y coordinate (i.e. those at the back) are first and each image with the same y coordinate is sorted so those with a smaller x coordinate (i.e. those on the left) come before those on the right. Then the layer of each image is set based on this sorted order. This is all about getting the wall images to overlap in a way that those closer to you are always in front of those further away.

But what about the ball? Later when the ball is rolling in the maze, its layer needs to change as it moves closer and further away. To speed up the later calculations for every possible y screen coordinate, the necessary layer for the ball is stored in sLayerArray

```
sort sImageList ascending numeric by item 1 of each
sort sImageList ascending numeric by item 2 of each
put 1 into layerIndex
put 1 into prevY
repeat for each line loopLine in sImageList
  set the layer of image ID (item 3 of loopLine) to layerIndex
  put item 2 of loopLine into y
  if y <> prevY then
    repeat with loopY = prevY to y
      put layerIndex + 1 into sLayerArray[loopY]
```

```
    end repeat
    put y + 1 into prevY
  end if
  add 1 to layerIndex
end repeat
```

To increase the challenge, and to allow mazes that are larger than the stack window, only part of the maze is visible at any time. If the entire card displays the maze, then the parts of the maze that don't fit are clipped at the edge of the window of the stack by LiveCode. But here there is a border around the maze to display the minimap, the time and the Play Again button.

So 4 graphics are created to cover the areas of the card where the maze is hidden. A single image with a transparent rectangle could have been used, but instead separate graphics allow the size and position of the viewport to be more easily changed during development. In a polished game an image could allow for a more interesting border.

```
set the style of the templateGraphic to "rectangle"
set the backgroundColor of the templateGraphic to 28,95,112
set the penColor of the templateGraphic to 28,95,112
set the showFill of the templateGraphic to true
set the rectangle of the templateGraphic to 0,0,960,kViewPortY1
create graphic
set the layer of the last image to top
set the rectangle of the templateGraphic to 0,kViewPortY2,960,640
create graphic
set the layer of the last image to top
set the rectangle of the templateGraphic to 0,kViewPortY1,kViewPortX1,kViewPortY2
create graphic
set the layer of the last image to top
set the rectangle of the templateGraphic to kViewPortX2,kViewPortY1,960,kViewPortY2
create graphic
set the layer of the last image to top
set the layer of button "butnPlayAgain" to top
set the layer of field "Time" to top
set the layer of image "tileWall" to bottom
```

The properties of the ball are next set, converting the room coordinates to the screen and setting the move to values of the current position, so the ball is stationary at the start of the game. The layer is set based on the screen y coordinate so the ball is in front of and behind the appropriate wall images.

```
ConvertRoomToScreenXY sBallRoomX,sBallRoomY,sBallX,sBallY
put sBallX into sMoveToX
put sBallY into sMoveToY
put sLayerArray[sBallY - sScreenOffsetY] into sBallLayer
set the loc of image "Ball" to sBallX,sBallY
set the layer of image "Ball" to sBallLayer
set the layerMode of the image "Ball" to "dynamic"
```

Lastly the timer and minimap are initialised and drawn.

```
put 0 into sTime
put 0 into sGameCounter
UpdateTime
DrawMiniMap
GameLoop
set the cursor to arrow
unlock screen
```

The DrawMiniMap handler is only called once from StartGame, but the StartGame handler is already way too long, so it is a handler of its own. It uses the information now in sRoomArray and creates the graphic controls to make a top down representation of the maze. Once it is drawn, UpdateMiniMap is called to show the position of the ball in the minimap.

```
command DrawMiniMap
  local x,y
  reset templateGraphic
  set the style of the templateGraphic to "rectangle"
  set the rect of the templateGraphic to kMiniMapCornerX,kMiniMapCornerY,
                                         kMiniMapCornerX + kMiniMapUnit * sRoomMaxX,
                                          kMiniMapCornerY + kMiniMapUnit * sRoomMaxY
  set the brushColor of the templateGraphic to 138,193,208
  set the penColor of the templateGraphic to "black"
  set the filled of the templateGraphic to true
  set the layerMode of the templateGraphic to "static"
  create graphic

  set the brushColor of the templateGraphic to 42,143,170
  set the penColor of the templateGraphic to 42,143,170
  put kMiniMapCornerY into y
  repeat with loopY = 1 to sRoomMaxY
    put kMiniMapCornerX into x
    repeat with loopX = 1 to sRoomMaxX
      if sRoomArray[loopX, loopY] = "W" then
        create graphic
        set the rect of the last graphic to x,y,x + kMiniMapUnit,y + kMiniMapUnit
      end if
      add kMiniMapUnit to x
    end repeat
    add kMiniMapUnit to y
  end repeat
  set the style of the templateGraphic to "oval"
  set the brushColor of the templateGraphic to "black"
  set the penColor of the templateGraphic to "black"
  create graphic
  set the width of the last graphic to kMiniMapUnit
  set the height of the last graphic to kMiniMapUnit
  set the loc of the last graphic to kMiniMapCornerX,kMiniMapCornerY
  put the short ID of the last graphic into sMiniMapBallID

  UpdateMiniMap
end DrawMiniMap
```

Next are two handlers that translate between the room and screen coordinates. The first calculates the screen position of the centre of the floor for each cell. There is a little complication because the cells go diagonally down the screen and there is an offset to allow the maze to scroll. The screen coordinates are returned as reference parameters so the x and y values do not need to be separated after returning.

ConvertScreenToRoomXY does the reverse, from a screen coordinate calculate what cell (if any) is at that position. It is a function because a click outside the maze returns false as there is no cell at the point. A 2 line formula like ConvertRoomToScreenXY would be better, but this was tricky to get working and instead a brute force algorithm was chosen. Not elegant, but it works and was quick to code. The sFloorList variable that was filled back in GameStart stores the screen coordinates of the rectangle around the floor of each cell. By running through this list (which is fast because LiveCode is efficient with this type of "for each" loop) and keeping track of which cell is closest to pScreenX and pScreenY the room coordinate is calculated.

```
command ConvertRoomToScreenXY pRoomX,pRoomY,@qScreenX,@qScreenY
  put kCornerX + sScreenOffsetX + 32 * (pRoomX - pRoomY) into qScreenX
  put kCornerY + sScreenOffsetY + 16 * (pRoomX + pRoomY - 2) into qScreenY
end ConvertRoomToScreenXY
function ConvertScreenToRoomXY pScreenX,pScreenY,@qRoomX,@qRoomY
  local mazeXY,delta,bestDelta
  subtract sScreenOffsetX from pScreenX
  subtract sScreenOffsetY from pScreenY
  put empty into qRoomX
  put empty into qRoomY
  put 640 into bestDelta
  repeat for each line loopLine in sFloorList
    if (pScreenY >= item 1 of loopLine) and (pScreenY <= item 2 of loopLine) then
```

```
      put abs(pScreenX - item 3 of loopLine) + abs(pScreenY - item 4 of loopLine) into
                                                                                  delta
      if delta <= bestDelta then
        put item 5 of loopLine into qRoomX
        put item 6 of loopLine into qRoomY
        put delta into bestDelta
      end if
    end if
  end repeat
  return bestDelta < 48
end ConvertScreenToRoomXY
```

The GameLoop does the following. Calls MoveBall which updates the position of the ball if the current position is not at the MoveTo point. If the ball is at the coordinates of sMoveToX,sMoveToY the view is scrolled if the ball is at the edge of the viewport. Then the minimap is updated. Unless the game is over (you reached the exit) GameLoop is called again in 2 ticks so the game runs at 30 fps.

```
on GameLoop
  local roomX,roomY
  MoveBall
  if (sBallX = sMoveToX) and (sBallY = sMoveToY) then
    ScrollViewPort
    UpdateMiniMap
    if ConvertScreenToRoomXY(sBallX,sBallY,roomX,roomY) then
      put roomX = sExitRoomX and roomY = sExitRoomY into sGameOver
    end if
  end if
  add 1 to sGameCounter
  if sGameCounter = 30 then
    UpdateTime
    add 1 to sTime
    put 0 into sGameCounter
  end if
  if sGameOver then
    answer information "You reached the exit."
  else
    send "GameLoop" to me in 2 ticks
  end if
end GameLoop
```

MoveBall moves the ball if the current coordinates are not where the ball should be. The sDeltaX and sDeltaY variables are set when you click on the screen and move the ball a small amount in the right direction. Since MoveBall is called 30 times a second, the delta is kept small to keep the motion smooth. A comparison is made to check whether the ball is at the move to point. The motion of the ball may mean it does not directly go over the move to point, so provided it is close enough it is stopped and snaps to the middle of the cell.

Then the layer is set so the ball is in front of the walls behind it, while behind the walls in front of it.

```
command MoveBall
  local y,layerIndex
  if (sBallX <> sMoveToX) or (sBallY <> sMoveToY) then
    lock screen
    add sDeltaX to sBallX
    add sDeltaY to sBallY
    if (abs(sMoveToX - sBallX) <= kSpeedX) or (abs(sMoveToY - sBallY) <= kSpeedY) then
      put sMoveToX into sBallX
      put sMoveToY into sBallY
      put sMoveToRoomX into sBallRoomX
      put sMoveToRoomY into sBallRoomY
    end if
    put round(sBallY) into y
    set the loc of image "Ball" to round(sBallX),y
    put sLayerArray[y - sScreenOffsetY] into layerIndex
```

```
    if layerIndex <> sBallLayer then
      put layerIndex into sBallLayer
      set the layer of image "Ball" to sBallLayer
    end if
    unlock screen
  end if
end MoveBall
```

In the GameLoop after MoveBall, ScrollViewPort is called. It checks whether the ball is near the edge of the viewport and if necessary enters a repeat loop that updates these variables: sBallX, sMoveToX, sBallY, sMoveToY, screenOffsetX, screenOffsetY to move the ball closer to the middle of the screen. You may be wondering why go to all this trouble? Why not use a LiveCode group?

While a LiveCode group is powerful and requires little work to implement scrolling, it has one major problem for this game. When controls are in a group, the layer property of each control cannot be uniquely set. For an isometric 3D game this rules out using a group for the images of the room.

```
command ScrollViewPort
  local deltaX,screenOffsetX,deltaY,screenOffsetY
  put sScreenOffsetX into screenOffsetX
  put sScreenOffsetY into screenOffsetY
  if kViewPortX2 - sBallX < 32 then
    put -kScrollDelta into deltaX
    put kViewPortX2 - sMaxX - 32 into screenOffsetX
  end if
  if sBallX - kViewPortX1 < 32 then
    put kScrollDelta into deltaX
    put 0 into screenOffsetX
  end if
  if kViewPortY2 - sBallY < 32 then
    put -kScrollDelta into deltaY
    put kViewPortY2 - sMaxY - 32 into screenOffsetY
  end if
  if sBallY - kViewPortY1 < 32 then
    put kScrollDelta into deltaY
    put 0 into screenOffsetY
  end if
  repeat while (sScreenOffsetX <> screenOffsetX) or (sScreenOffsetY <> screenOffsetY)
    add deltaX to sScreenOffsetX
    add deltaY to sScreenOffsetY
    lock screen
    repeat for each line loopImage in sImageList
      set the loc of image ID (item 3 of loopImage) to item 1 of loopImage +
                                    sScreenOffsetX,item 2 of loopImage + sScreenOffsetY
    end repeat
    add deltaX to sBallX
    add deltaX to sMoveToX
    add deltaY to sBallY
    add deltaY to sMoveToY
    set the loc of image "Ball" to sBallX,sBallY
    unlock screen
    if sScreenOffsetX = screenOffsetX then put 0 into deltaX
    if sScreenOffsetY = screenOffsetY then put 0 into deltaY
    if sBallX,sBallY is within sViewPortMiddle then
      put sScreenOffsetX into screenOffsetX
      put sScreenOffsetY into screenOffsetY
    end if
  end repeat
end ScrollViewPort
```

These two commands update the position of the ball in the minimap and the elapsed time.

```
on UpdateMiniMap
  set the loc of the graphic ID sMiniMapBallID to kMiniMapCornerX + sBallRoomX *
                        kMiniMapUnit - kMiniMapUnit div 2,kMiniMapCornerY + sBallRoomY *
                                                    kMiniMapUnit - kMiniMapUnit div 2
end UpdateMiniMap

command UpdateTime
  put (sTime div 60) & ":" & sTime mod 60 into field "Time"
end UpdateTime
```

The code up to here does all the calculations and drawing to make a convincing 3D maze using isometric techniques. But there is one last element missing: responding to the user input. The mouseDown handler first checks if the click is inside the maze and what cell in the room it is. Most of the code that follows is about what values are needed to move the ball to the mouse click, and whether there is a wall in the way of where you have clicked. Since the ball only travels in straight lines, a loop starts from the current position of the ball and stops and changes the move to value if a wall is reached.

Once valid values for sMoveToRoomX and sMoveToRoomY are found they are converted to screen coordinates. The difference between the current position and the move to position is used to set appropriate values for sDeltaX and sDeltaY that are used in MoveBall.

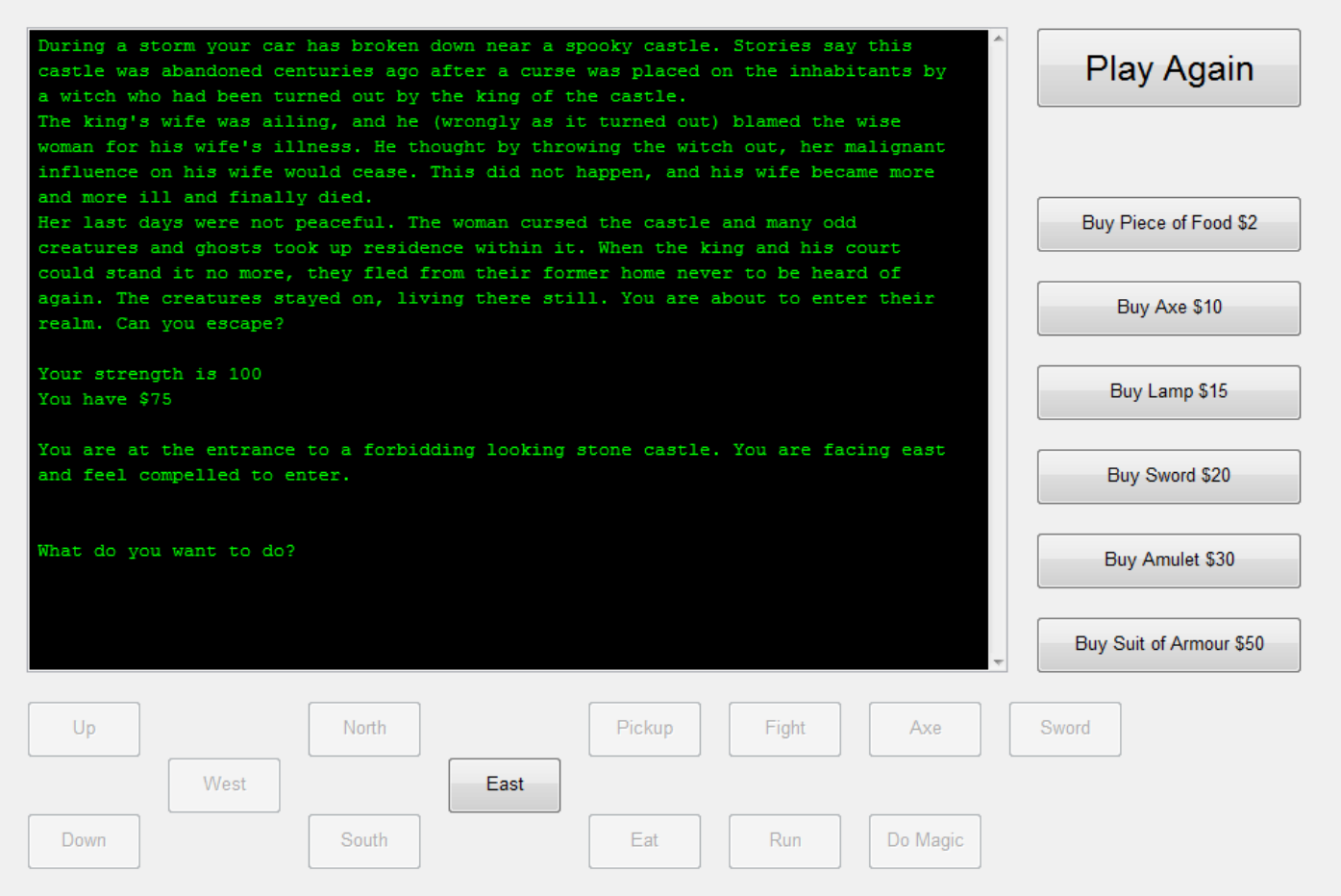
```
on mouseDown pButton
  local dX,dY
  if ConvertScreenToRoomXY(the mouseH,the mouseV,sMoveToRoomX,sMoveToRoomY) then
    -- can only go is straight line
    if not (sMoveToRoomX <> sBallRoomX and sMoveToRoomY <> sBallRoomY) then
      if sMoveToRoomX <> sBallRoomX then
        if sMoveToRoomX > sBallRoomX then
          put 1 into dX
        else
          put -1 into dX
        end if
        repeat with loopX = sBallRoomX + dX to sMoveToRoomX step dX
          if sRoomArray[loopX,sBallRoomY] = "W" then
            put loopX - dX into sMoveToRoomX
            exit repeat
          end if
        end repeat
      end if
      if sMoveToRoomY <> sBallRoomY then
        if sMoveToRoomY > sBallRoomY then
          put 1 into dY
        else
          put -1 into dY
        end if
        repeat with loopY = sBallRoomY + dY to sMoveToRoomY step dY
          if sRoomArray[sBallRoomX,loopY] = "W" then
            put loopY - dY into sMoveToRoomY
            exit repeat
          end if
        end repeat
      end if

      if sMoveToRoomX <> sBallRoomX or sMoveToRoomY <> sBallRoomY then
        ConvertRoomToScreenXY sMoveToRoomX,sMoveToRoomY,sMoveToX,sMoveToY
        if sMoveToX > sBallX then
          put kSpeedX into sDeltaX
        else
          put -kSpeedX into sDeltaX
        end if
        if sMoveToY > sBallY then
          put kSpeedY into sDeltaY
```

```
        else
          put -kSpeedY into sDeltaY
        end if
      end if
    end if
  end if
end mouseDown
```

That's about it. Sure, the maze itself is rather pathetic, but you can modify the roomMap in StartGame to make it bigger or more complex. Note, StartGame does not check that the number of rooms in each row are the same. If they are not, you will get unexpected results. When editing the map, using a fixed width font and setting your editor to overwrite mode can make it easier.

Here are some suggestions for improving the game.

- Make the maze harder to solve, or make it bigger.
- Make the minimap update itself while the ball is rolling.
- Add keyboard control for the ball.



Some free advice, you *will* need the Lamp.

Game 4

Text Adventure

Here is a simple text adventure with no graphics. Calling it an adventure is a little generous because there isn't an in-game story, and the only puzzle is finding your way through the rooms to escape. But as a hack and slash text adventure it still fits in the broad category.

The original version of this game is from 1985, before mouse driven interfaces were standard, and so commands were entered by pressing a single key to indicate a command. For example, press N to go North and C to eat. Pressing C to eat shows one problem of such a simple interface, because the letters N, S, E, W were used for the four basic directions, E could not be used for eating. Instead you Consume food, obvious, right?

In this version, this problem is avoided by using a mouse driven interface, with a heap of dedicated buttons. Perhaps this is no more elegant than the original interface, but it is easy to code and allows the entire game to be played with only clicks, and hence, potentially on a touch driven device without a virtual keyboard.

In this game, you are in a castle full of treasure and dangerous creatures. Your goal is to leave the castle, alive. The original game is called "Werewolves and Wanderer" and has a back story about a mysterious faded letter found in a trunk telling a history of royalty, thunderstorms, a witch and a curse. The tired old story doesn't matter here, because this book is about game programming.

In this game you will see:

- A two dimensional array used as a room, treasure and monster map
- A variable used for inventory with boolean operations
- A single mouseUp handler for all the button presses
- An interface that does not allow invalid actions
- The random function used to add an element of unpredictability

Below is the StartGame handler. This handler is called from the openCard handler and when the Play Again button is clicked and calls commands to initialise the rooms, treasure, monsters and the hero (you!). Then ShowStatus displays the description of the room and other important information.

```
command StartGame
  InitializeRooms
  InitializeTreasure
  InitializeMonsters
  InitializeHero
  put false into sGameOver
  ShowIntroduction
  ShowStatus true
end StartGame
```

InitializeRooms puts information about the castle rooms into a list and then scans the list to set up a two dimensional array that is the map of the game. This two step approach is used because the list makes it easy to edit the rooms during development and planning, while the array results in simpler code to control the game at runtime.

Each line of the roomData has 7 comma delimited items. Each room is represented by a number. The first line is for room number 1, and the first 4 items are numbers for the rooms that are connected in the directions of North, South, East or West. If the item is 0 that means you cannot go in that direction. For example, in room 1 (the first line of roomData) there is an exit to the south that goes to room 2.

In the castle of this game there is more than one level, so the fifth and sixth items of each line is for the directions Up and Down. In room 1 these items are both 0 because there are no stairs in that room. The last item of the line is the contents of the room. Every room has this value equal to 0 because the rooms are initially empty before they are randomly filled by InitializeTreasure and InitializeMonsters.

Now that you understand the structure of the roomData variable, a pair of repeat loops read the list and convert it to a 2 dimensional array. The first index of the array is the room number, and the second index is a property of the room, i.e. the direction or the contents.

```
command InitializeRooms
  local roomData,roomNumber,roomProperty
  -- N,S,E,W,U,D
  put 0,2,0,0,0,0,0 & cr & \ -- room 1
  1,3,3,0,0,0,0 & cr & \ -- room 2
  2,0,5,2,0,0,0 & cr & \ -- room 3
  0,5,0,0,0,0,0 & cr & \ -- room 4
  4,0,0,3,15,13,0 & cr & \ -- room 5
  0,0,1,0,0,0,0 & cr & \ -- room 6
  0,8,0,0,0,0,0 & cr & \ -- room 7
  7,10,0,0,0,0,0 & cr & \ -- room 8
  0,19,0,8,0,8,0 & cr & \ -- room 9
  8,0,11,0,19,0,0 & cr & \ -- room 10
  0,0,0,0,0,0,0 & cr & \ -- room 11
  0,0,0,13,0,0,0 & cr & \ -- room 12
  0,0,12,0,5,0,0 & cr & \ -- room 13
  0,15,17,0,0,0,0 & cr & \ -- room 14
  14,0,0,0,0,5,0 & cr & \ -- room 15
  17,0,19,0,0,0,0 & cr & \ -- room 16
  18,16,0,14,0,0,0 & cr & \ --room 17
  0,17,0,0,0,0,0 & cr & \ -- room 18
  0,0,0,16,0,10,0 & cr into roomData --19
  put 1 into roomNumber
  repeat for each line loopRoom in roomData
    put kRoomNorth into roomProperty
    repeat for each item loopData in loopRoom
      put loopData into sRoomArray[roomNumber][roomProperty]
      add 1 to roomProperty
    end repeat
    add 1 to roomNumber
  end repeat
end InitializeRooms
```

RandomEmptyRoom, InitializeTreasure and InitializeMonsters are listed together below. RandomEmptyRoom returns a number between 1 and 19 (because there are 19 rooms) and has a loop so if

the random number is for a room with something already in it, sRoomArray[roomNumber][kRoomContents] is not zero, the loop repeats and picks another number. The loop also continues if the number is the start or exit room which are kept empty.

RandomEmptyRoom is called by InitializeTreasure to put a random number between 11 and 110 into seven rooms. When the contents of a room is a number larger than 10 it is treasure of that amount. Then InitializeMonsters puts one of each of the six different monsters into a random room. Each monster is represented by a number from 1 to 6, so they cannot be confused with the treasure.

```
function RandomEmptyRoom
  local roomNumber
  put random(19) into roomNumber
  repeat while sRoomArray[roomNumber][kRoomContents] <> 0 or roomNumber = kRoomStart
                                                          or roomNumber = kRoomExit
    put random(19) into roomNumber
  end repeat
  return roomNumber
end RandomEmptyRoom

command InitializeTreasure
  repeat 7 times
    put random(100) + kMinimumTreasure into
sRoomArray[RandomEmptyRoom()][kRoomContents]
  end repeat
end InitializeTreasure

command InitializeMonsters
  repeat with loopMonster = kRat to kWumpus
    put loopMonster into sRoomArray[RandomEmptyRoom()][kRoomContents]
  end repeat
end InitializeMonsters
```

InitializeHero fills some variables and puts the player in the starting room.

```
command InitializeHero
  put 100 into sStrength
  put 75 into sWealth
  put 0 into sFood
  put 0 into sInventory
  put 0 into sTally
  put 0 into sMonsterKilled
  put kRoomStart into sCurrentRoom
end InitializeHero
```

ShowText and ShowTextDelay append text to the text field and make sure the most recent text is visible with the select line. This is a simple way to scroll to the bottom of a field.

ShowScore does a calculation to work out the score based on the number of monster kills and so on, and is called when the game ends, either because you have escaped the castle, or have died from zero strength.

```
command ShowText pText
  put pText & cr after field "memoOutput"
  select after last char of field "memoOutput"
end ShowText

command ShowTextDelay pText
  put pText & cr after field "memoOutput"
  select after last char of field "memoOutput"
  wait 1 second
end ShowTextDelay
```

```
command ShowScore
  local score
  put 30 * sMonsterKilled + 5 * sStrength + 3 * sTally + 2 * sWealth + sFood into
                                                                                score
  ShowText "Your final score is " & score
end ShowScore
```

ShowRoomDescription is a switch statement that calls ShowText with the description for the room. In a more polished game, I would put the descriptions in a separate plain text file and then load the descriptions into an array. But for a demo game like this, using a handler with the switch statement is more than adequate, even if it is less elegant.

```
command ShowRoomDescription pRoom
  local description
  switch pRoom
    case 1
      put "You are in the hallway." & cr & cr & "There is a door to the south. The
entrance door to the west is locked. Through windows to the north you can see a secret
herb garden." into description
      break
    case 2
      put "This is the audience chamber." & cr & cr & "There is a window to the west,
by looking to the right through it you can see the entrance to the castle. Doors leave
this room to the north, east and south." into description
      break
    case 3
      put "You are in the great hall." & cr & cr & "An L-shaped room, there are doors
to the east and to the north. In the alcove is a door to the west." into description
      break
    case 4
      put "This is the monarch's private meeting room. There is a single exit to the
south." into description
      break
    case 5
      put "This inner hallway contains a door to the north, and one to the west, and a
circular stairwell passes through the room." & cr & cr & "You can see an ornamental
lake through the windows to the south." into description
      break
    case 6
      put "You are at the entrance to a forbidding looking stone castle. You are facing
east and feel compelled to enter." into description
      break
    case 7
      put "This is the castle's kitchen." & cr & cr & "Through windows in the north
wall you can see a secret herb garden. A door leaves the kitchen to the south." into
description
      break
    case 8
      put "You are in the store room, amidst spices, vegetables, and vast stacks of
flour and other provisions." & cr & cr & "There is a door to the north and one to the
south." into description
      break
    case 9
      -- not used
      break
    case 10
      put "You are in the rear vestibule." & cr & cr & "There are windows to the south
from which you can see the ornamental lake. There is an exit to the east and north.
Dark stairs head updwards." into description
      break
    case 11
      -- exit, no description
      break
    case 12
      put "You are in the dank, dark dungeon." & cr & cr & "There is a single exit, a
small hole in the wall towards the West." into description
      break
```

```
    case 13
      put "You are in the prison guard room, in the basement of the castle." & cr & cr
& "The stairwell ends in this room. There is one other exit, a small hole in the east
wall." into description
      break
    case 14
      put "You are in the master bedroom on the upper level of the castle..." & cr & cr
& "Looking down from the window to the west can see the entrance to the castle, while
the secret herb garden is visible below the north window. There are doors to the east
and to the south." into description
      break
    case 15
      put "This is the L-shaped upper hallway." & cr & cr & "To the north is a door,
and there is a stairwell in the hall that goes up and down. You can see the lake
through the south windows." into description
      break
    case 16
      put "This room was used as the castle treasury in bygone years..." & cr & cr &
"There are no windows, just exits to the north and to the east." into description
      break
    case 17
      put "Ooooh... you are in the chambermaid bedroom." & cr & cr & "There is an exit
to the west and a doors to the north and south..." into description
      break
    case 18
      put "This tiny room on the upper level is the dressing chamber." & cr & cr &
"There is a window to the north, with a view of the herb garden down below. A door
leads to the south." into description
      break
    case 19
      put "This is the small room near dark stairs that head down." & cr & cr & "To the
west is a door. You can see the lake through the southern windows." into description
      break
  end switch
  ShowText cr & description
end ShowRoomDescription
```

ShowMonsterDescription is similar to ShowRoomDescription, but also sets the sDanger variable according to how difficult the monster is to defeat. ShowTextDelay is used to add a little suspence when showing the text.

```
command ShowMonsterDescription pMonster
  put empty into sCreature
  put 0 into sDanger
  switch pMonster
    case kRat
      put "hungry rat" into sCreature
      put 5 into sDanger
      break
    case kBat
      put "vampire bat" into sCreature
      put 10 into sDanger
      break
    case kWerewolf
      put "ferocious werewolf" into sCreature
      put 15 into sDanger
      break
    case kZombie
      put "brain eating zombie" into sCreature
      put 20 into sDanger
      break
    case kSpider
      put "giant spider" into sCreature
      put 25 into sDanger
      break
    case kWumpus
      put "weird Wumpus" into sCreature
```

```
      put 30 into sDanger
      break
  end switch
  if sCreature is not empty then
    ShowTextDelay "Danger there is a creature here..."
    ShowTextDelay "It is a " & sCreature & ", the danger level is " & sDanger
  end if
end ShowMonsterDescription
```

ShowIntroduction tells a story to set the scene.

```
command ShowIntroduction
  put empty into field "memoOutput"
  ShowText "During a storm your car has broken down near a spooky castle. Stories say
this castle was abandoned centuries ago after a curse was placed on the inhabitants by
a witch who had been turned out by the king of the castle. "
  ShowText "The king's wife was ailing, and he (wrongly as it turned out) blamed the
wise woman for his wife's illness. He thought by throwing the witch out, her malignant
influence on his wife would cease. This did not happen, and his wife became more and
more ill and finally died."
  ShowText "Her last days were not peaceful. The woman cursed the castle and many odd
creatures and ghosts took up residence within it. When the king and his court could
stand it no more, they fled from their former home never to be heard of again. The
creatures stayed on, living there still. You are about to enter their realm. Can you
escape?"
  ShowText empty
end ShowIntroduction
```

ShowStatus does a few comparisons to see if you are dead or weak, and then shows your inventory followed by the room and monster descriptions. From a programming point of view the most interesting part is the boolean operators used to check what items you have in sInventory and then construct a grammatically acceptable sentence about those items.

```
command ShowStatus pKeep
  local buffer
  if pKeep <> true then put empty into field "memoOutput"
  if sCurrentRoom = kRoomExit then
    ShowTextDelay "You have found the exit from the castle."
    ShowTextDelay "You succeeded and managed to get out of the castle!"
    ShowScore
    put true into sGameOver
  else
    if sStrength < 1 then
      ShowText "You have died of hunger or weakness..."
      ShowScore
      put true into sGameOver
    else
      if sStrength < 10 then ShowText "Your strength is running low!"
      ShowText "Your strength is " & sStrength
      ShowText "You have $" & sWealth
      if sFood > 0 then
        if sFood = 1 then
          ShowText "Your provisions sack holds " & sFood & " piece of food"
        else
          ShowText "Your provisions sack holds " & sFood & " pieces of food"
        end if
      end if
      if (sInventory bitAND kArmour) <> 0 then ShowText "You are wearing armour"
      put empty into buffer
      if (sInventory bitAND (kAxe + kSword + kAmulet)) <> 0 then
        if (sInventory bitAND kAxe) <> 0 then put "an axe" after buffer
        if (sInventory bitAND kSword) <> 0 then
          if buffer is not empty then put " and " after buffer
          put "a sword" after buffer
        end if
        if (sInventory bitAND kAmulet) <> 0 then
```

```
          if buffer is not empty then put " and " after buffer
          put "the magic amulet" after buffer
        end if
        ShowText "You are carrying " & buffer
      end if
      if (sInventory bitAND kLamp) <> 0 or sCurrentRoom = kRoomStart then
        ShowRoomDescription sCurrentRoom
      else
        ShowText "It is too dark to see clearly!"
      end if
      put sRoomArray[sCurrentRoom][kRoomContents] into sRoomContents
      if sRoomContents > kMinimumTreasure then
        ShowText empty
        ShowText "There is treasure here worth $" & sRoomContents
      else
        ShowText empty
        ShowMonsterDescription sRoomContents
      end if
      ShowText empty
      ShowText "What do you want to do?"
      UpdateButtons
    end if
  end if
end ShowStatus
```

Then you are asked "What do you want to do?" and UpdateButtons is called. UpdateButtons disables the buttons that do not make sense, or cannot be used at this point in the game. Games and other programs that allow you to attempt an action, but then complain that you cannot do that are annoying! So while UpdateButtons may look complicated and verbose, it actually makes the rest of the game a lot simpler to code because there is no need to do any error checking, anywhere. The pRun parameter is used by the TryToRun handler mentioned later.

```
command UpdateButtons pRun
  local monsterInRoom,runaway,canSee
  put pRun = true into runaway
  put (sInventory bitAND kLamp) <> 0 or sCurrentRoom = kRoomStart into canSee
  put sRoomContents > 0 and sRoomContents < kMinimumTreasure into monsterInRoom

  set the enabled of button "butnUp" to (sRoomArray[sCurrentRoom][kRoomUp] <> 0) and
                              (runaway or not monsterInRoom) and canSee and not sGameOver
  set the enabled of button "butnDown" to (sRoomArray[sCurrentRoom][kRoomDown] <> 0)
                          and (runaway or not monsterInRoom) and canSee and not sGameOver
  set the enabled of button "butnNorth" to (sRoomArray[sCurrentRoom][kRoomNorth] <> 0)
                          and (runaway or not monsterInRoom) and canSee and not sGameOver
  set the enabled of button "butnSouth" to (sRoomArray[sCurrentRoom][kRoomSouth] <> 0)
                          and (runaway or not monsterInRoom) and canSee and not sGameOver
  set the enabled of button "butnEast" to (sRoomArray[sCurrentRoom][kRoomEast] <> 0)
                          and (runaway or not monsterInRoom) and canSee and not sGameOver
  set the enabled of button "butnWest" to (sRoomArray[sCurrentRoom][kRoomWest] <> 0)
                          and (runaway or not monsterInRoom) and canSee and not sGameOver

  set the enabled of button "butnFight" to monsterInRoom and not runaway and
                                                                    not sGameOver
  set the enabled of button "butnRun" to monsterInRoom and not runaway and
                                                                    not sGameOver

  set the enabled of button "butnAxe" to false
  set the enabled of button "butnSword" to false

  set the enabled of button "butnPickup" to (sRoomContents >= kMinimumTreasure) and
                                                                    not sGameOver

  set the enabled of button "butnMagic" to ((sInventory bitAND kAmulet) <> 0) and
                                                    not runaway and not sGameOver
  set the enabled of button "butnEat" to (sFood > 0) and not monsterInRoom and
                                                    not runaway and not sGameOver
```

```
  set the enabled of button "butnBuyFood" to (sWealth >= 2) and not monsterInRoom and
                                                    not runaway and not sGameOver
  set the enabled of button "butnBuyAxe" to (sWealth >= 10) and
                               ((sInventory bitAND kAxe) = 0) and not monsterInRoom and
                                                    not runaway and not sGameOver
  set the enabled of button "butnBuyLamp" to (sWealth >= 15) and
                              ((sInventory bitAND kLamp) = 0) and not monsterInRoom and
                                                    not runaway and not sGameOver
  set the enabled of button "butnBuySword" to (sWealth >= 20)
                         and ((sInventory bitAND kSword) = 0) and not monsterInRoom and
                                                    not runaway and not sGameOver
  set the enabled of button "butnBuyAmulet" to (sWealth >= 30)
                        and ((sInventory bitAND kAmulet) = 0) and not monsterInRoom and
                                                    not runaway and not sGameOver
  set the enabled of button "butnBuyArmour" to (sWealth >= 50) and
                             ((sInventory bitAND kArmour) = 0) and not monsterInRoom and
                                                    not runaway and not sGameOver
end UpdateButtons
```

With the buttons now ready for clicking on, where should the code be put that responds to each click? One place is in the mouseUp handler of each button. The LiveCode IDE makes this an obvious place. The contextual menu for each button has the Edit Script command that opens the Script Editor and enters a skeleton mouseUp handler. Since the button is the first control to receive that message, it should contain the code, right?

Not necessarily. When developing a game like this one, I prefer to have all the code in the one place. One pane in the Script Editor with less than a 1000 lines is easy to navigate, and not putting handlers in the buttons means I can change and delete the buttons during development without the worrying about accidentally deleting code. Putting all code in the Card Script is easier and quicker. (There are also reasons related to the benefit of separating logic and user interface of a program, but that discussion is not for here.)

Each button receives a mouseUp message when it is clicked, and if there isn't a mouseUp handler in the script of the button, the message continues along the message path where is comes to the mouseUp handler below. The short name of the target is the name of the clicked button and the switch statement executes the required code for each action. Except for the logic required for fighting a monster, the code for each action has not been put in a separate handler. The professional coder in me knows this is not good practice, but with only a few lines in a case statement there is little harm in leaving it like this.

```
on mouseUp
  local needToUpdateStatus
  put true into needToUpdateStatus
  switch the short name of the target
    case "butnUp"
      put sRoomArray[sCurrentRoom][kRoomUp] into sCurrentRoom
      subtract 5 from sStrength
      break
    case "butnDown"
      put sRoomArray[sCurrentRoom][kRoomDown] into sCurrentRoom
      subtract 5 from sStrength
      break
    case "butnNorth"
      put sRoomArray[sCurrentRoom][kRoomNorth] into sCurrentRoom
      subtract 5 from sStrength
      break
    case "butnSouth"
      put sRoomArray[sCurrentRoom][kRoomSouth] into sCurrentRoom
      subtract 5 from sStrength
      break
    case "butnEast"
      put sRoomArray[sCurrentRoom][kRoomEast] into sCurrentRoom
      subtract 5 from sStrength
      break
    case "butnWest"
```

```
      put sRoomArray[sCurrentRoom][kRoomWest] into sCurrentRoom
      subtract 5 from sStrength
      break
    case "butnFight"
      PrepareFight
      put false into needToUpdateStatus
      break
    case "butnAxe"
      put round(sDanger * 0.8) into sDanger
      set the enabled of button "butnAxe" to false
      set the enabled of button "butnSword" to false
      ShowFight
      put false into needToUpdateStatus
      break
    case "butnSword"
      put round(sDanger * 0.75) into sDanger
      set the enabled of button "butnAxe" to false
      set the enabled of button "butnSword" to false
      ShowFight
      put false into needToUpdateStatus
      break
    case "butnRun"
      TryToRun
      put false into needToUpdateStatus
      break
    case "butnMagic"
      ShowTextDelay "You use the magic amulet to transport you to a random room in the
castle..."
      wait 5 seconds
      put random(19) into sCurrentRoom
      repeat while sCurrentRoom = kRoomStart or sCurrentRoom = kRoomExit
        put random(19) into sCurrentRoom
      end repeat
      break
    case "butnPickup"
      add sRoomContents to sWealth
      put 0 into sRoomArray[sCurrentRoom][kRoomContents]
      break
    case "butnEat"
      ShowTextDelay "You eat a piece of food."
      subtract 1 from sFood
      add 5 to sStrength
      break
    case "butnBuyFood"
      subtract 2 from sWealth
      add 1 to sFood
      break
    case "butnBuyAxe"
      subtract 10 from sWealth
      put sInventory bitOR kAxe into sInventory
      break
    case "butnBuyLamp"
      subtract 15 from sWealth
      put sInventory bitOR kLamp into sInventory
      break
    case "butnBuySword"
      subtract 20 from sWealth
      put sInventory bitOR kSword into sInventory
      break
    case "butnBuyAmulet"
      subtract 30 from sWealth
      put sInventory bitOR kAmulet into sInventory
      break
    case "butnBuyArmour"
      subtract 50 from sWealth
      put sInventory bitOR kArmour into sInventory
      break
```

```
    default
      put false into needToUpdateStatus
      break
  end switch
  if needToUpdateStatus then ShowStatus
end mouseUp
```

When a monster is in a room you have a choice of fighting or running away. To make it interesting, you cannot run away every time. I want to allow the player to run away 70% of the time. To do this the random function generates a number between 1 and 100 and then checks if the number is less than or equal to 70. If thinking in percentages suits you, this sort of statement is easy to code and understand. On average 70 times out of 100 the comparison will be true, and the other 30 times it is false. Since it is random, you cannot be sure what will happen each time, but in general 30% of the time you will be forced to fight.

The true parameter of UpdateButtons means only the relevant direction buttons are enabled. This stops you from deciding to add an item to your inventory, such as a sword, just when you need it. When there is a monster in the room you can fight or flee, but you don't have time to eat and do a bit of shopping.

```
command TryToRun
  if random(100) <= 70 then
    put 0 into sRoomContents
    ShowTextDelay "Which way do you want to flee?"
    UpdateButtons true
  else
    ShowTextDelay "No, you must stand and fight."
    PrepareFight
  end if
end TryToRun
```

If you choose, or are forced, to fight PrepareFight is called. This adjusts the level of danger depending on what is in the inventory. Multiplying the danger by a number less than zero for each useful item improves your chances of beating the monster. If you have an axe and sword the Axe and Sword buttons are enabled to allow you choose the weapon to fight with.

```
on PrepareFight
  if (sInventory bitAND kArmour) <> 0 then
    ShowTextDelay "Your armour increases your chance of success."
    put round(sDanger * 0.75) into sDanger
  end if
  if (sInventory bitAND (kAxe + kSword)) = 0 then
    ShowTextDelay "You have no weapons you must fight with your bare hands."
    put round(sDanger * 1.2) into sDanger
    ShowFight
  else
    if (sInventory bitAND (kAxe + kSword)) = kAxe + kSword then
      ShowTextDelay "Which weapon do you want to fight with?"
      set the enabled of button "butnFight" to false
      set the enabled of button "butnRun" to false
      set the enabled of button "butnMagic" to false
      set the enabled of button "butnAxe" to true
      set the enabled of button "butnSword" to true
    else
      if (sInventory bitAND kAxe) <> 0 then
        ShowTextDelay "You have only an axe to fight with."
        put round(sDanger * 0.8) into sDanger
      end if
      if (sInventory bitAND kSword) <> 0 then
        ShowTextDelay "You have only a sword to fight with."
        put round(sDanger * 0.75) into sDanger
      end if
      ShowFight
    end if
  end if
end PrepareFight
```

After sDanger has been calculated the fight begins with ShowFight. If you haven't played this style of game before it may be surprising, but once the fight begins you have no control. Other than choosing your weapon, if you have one, the fight plays out without intervention from the player. Instead a loop is entered and your initial strength, how dangerous the monster is, and random numbers determine your fate.

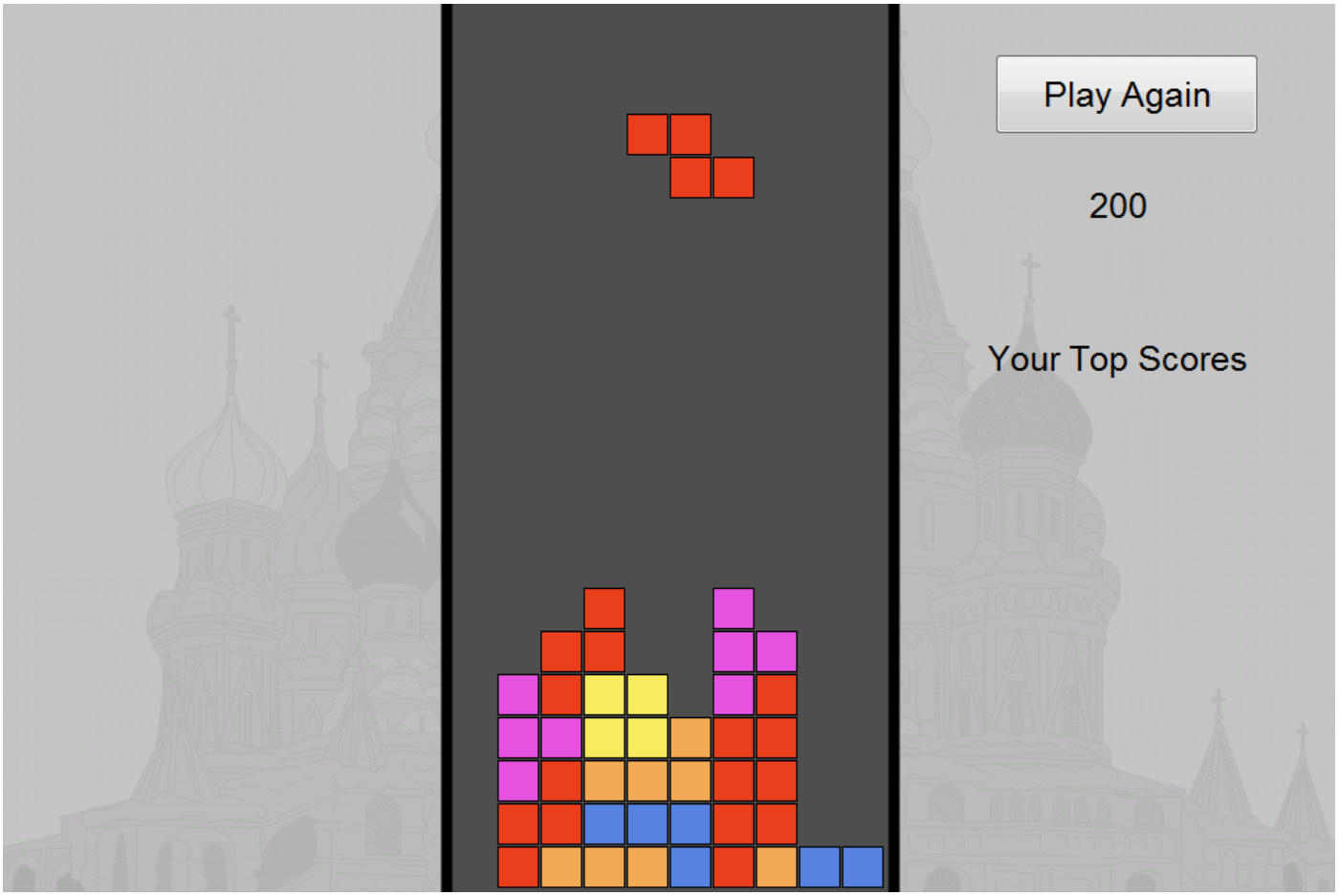
Each time through the loop, there is a 50% chance for these events: the monster wounds you, or you wound it. There is then a 65% chance that the fight continues. When the fight ends there is another random comparison based on how much damage you inflicted on the monster to decide if it is dead. Even if it is, you may still lose if your strength is too low. If the monster defeats you, your strength is halved, and provided it is not too low, you live to continue your adventure.

```
command ShowFight
  local fighting
  put true into fighting
  repeat while fighting
    if random(100) > 50 then
      ShowTextDelay "The " & sCreature & " attacks."
    else
      ShowTextDelay "You attack."
    end if
    if random(100) > 50 then
      ShowTextDelay "You manage to wound it."
      put round(sDanger * 0.83) into sDanger
    end if
    if random(100) > 50 then
      ShowTextDelay "The monster wounds you."
      subtract 5 from sStrength
    end if
    put random(100) <= 65 into fighting
  end repeat
  if random(16) > sDanger then
    ShowTextDelay "You managed to kill the " & sCreature & "!"
    add 1 to sMonsterKilled
    put 0 into sRoomArray[sCurrentRoom][kRoomContents]
    if sStrength < 6 then
      ShowTextDelay "But..."
    end if
  else
    ShowTextDelay "The " & sCreature & " defeated you and disappeared."
    put 0 into sRoomArray[sCurrentRoom][kRoomContents]
    put sStrength div 2 into sStrength
  end if
  ShowStatus true
end ShowFight
```

That's it, a small, but complete hack and slash text adventure in just under 500 lines of code. Can you escape the castle, or will a werewolf get the better of you?

Here are some suggestions for improving the game.

- Make the room descriptions more detailed.
- Make monsters reappear in another room if you do not kill them.
- Add more rooms or items for the inventory.



There’s never a long straight piece when you really need one.

Game 5

Tetris

Tetris. The original falling block, fill a row game. The game that helped ensure the success of the GameBoy. This game is a true classic and still has fans. (Addicts?) Some of the other games in this book (23 Matches and 3D Isometric Maze) are more demonstrations of technique than compelling games. After making and testing for a few minutes, it was time to move on to something more interesting. But Tetris is different. This is the first game in this book that has high replay value, and it required less than 350 lines of LiveCode code.

In this game you will see:

- Collision detection based on a tile grid
- A compact way of representing and rotating the blocks
- A game world larger than it needs to be to simplify coding
- Storing of high scores to add incentive
- Addictive gameplay

While LiveCode can't take credit for the last point, the other points are in the code you can examine below.

A little disclaimer about the second point: A compact way of representing and rotating the falling blocks. While the code that rotates the tetrominoes (each pattern of falling blocks) is small, about 5 lines, setting up the data to achieve this was a real pain. Too much time was spent with sheets of grid paper and pencil figuring out the numbers for each transformation. If this game was coded again from scratch, a more general purpose approach with matrices to do rotations is probably a better way.

StartGame begins by removing any messages that are in the message queue. This is a good idea when starting a game, because if there are already messages pending that call your GameLoop (which can happen during development) if you don't first remove them the game animation will be unpredictable. Then the graphics from the previous run of the game are deleted.

Each group of 4 falling blocks is defined by calling CreateBlockDefinition and CreateBlockTransform. How these work and the format of the data will be discussed later when these handlers are listed in full.

The sPlayArea array represents the visible area where the blocks are on the screen. This area is 10 blocks wide and 20 blocks high. Each element of the array is set to empty. An empty element means there is no block at that point in the array. Then there are two extra loops that fill sPlayArea with a 0 for the columns and row outside of the visible area. These extra elements in the array simplify the later code.

In Tetris the falling blocks stop moving when they come into contact with the stationary blocks below. So there needs to be a handler that detects contact and stops the motion of the blocks that collide. But what about stopping the blocks from falling through the bottom of the screen, or from moving sideways out of the play area? Another handler could be used to detect motion outside of the play area. Code that handles the special cases when the column is less than 1 or greater than 10, or the row is greater than 20, would work but means more complexity.

Instead, by extending the sPlayArea by 1 along the left, bottom and right edges and putting 0 into those elements of the array, no extra code is required to check when the blocks try to leave the visible area or hit the bottom.

Then some variables are initialized, including the high score list, and the first set of 4 blocks about to fall are created before calling GameLoop.

```
on StartGame
  local pendMsg,blockID
  put the pendingmessages into pendMsg
  repeat for each line loopLine in pendMsg
    cancel item 1 of loopLine
  end repeat

  lock screen
  repeat with loopIndex = the number of graphics down to 1
    if the pvBlock of graphic loopIndex then delete graphic loopIndex
  end repeat

  -- block
  CreateBlockDefinition 1,"0,0, 1,0, 0,1, -1,0"
  CreateBlockTransform 1,"0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0"
  -- line
  CreateBlockDefinition 2,"0,0, 1,0, 1,0, 1,0"
  CreateBlockTransform 2,"1,-1,0,0,-1,1,-2,2, 2,1,1,0,0,-1,-1,-2, -2,2,-1,1,0,0,1,-1,
-1,-2,0,-1,1,0,2,1"
  -- s
  CreateBlockDefinition 3,"0,0, 1,0, 0,-1, 1,0"
  CreateBlockTransform 3,"1,-1,0,0,1,1,0,2, 1,1,0,0,-1,1,-2,0, -1,1,0,0,-1,-1,0,-2,
-1,-1,0,0,1,-1,2,0"
  -- z
  CreateBlockDefinition 4,"0,0, 1,0, 0,1, 1,0"
  CreateBlockTransform 4,"2,-1,1,0,0,-1,-1,0, 0,2,-1,1,0,0,-1,-1, -2,0,-1,-1,0,0,1,-1,
0,-1,1,0,0,1,1,2"
  -- t
  CreateBlockDefinition 5,"0,0, 1,0, 1,0, -1,1"
  CreateBlockTransform 5,"1,-1,0,0,-1,1,-1,-1, 1,1,0,0,-1,-1,1,-1, -1,1,0,0,1,-1,1,1,
-1,-1,0,0,1,1,-1,1"
  -- l
  CreateBlockDefinition 6,"0,0, 1,0, 1,0, -2,1"
  CreateBlockTransform 6,"1,-1,0,0,-1,1,0,-2, 1,1,0,0,-1,-1,2,0, -1,1,0,0,1,-1,0,2,
-1,-1,0,0,1,1,-2,0"
  -- j
  CreateBlockDefinition 7,"0,0, 1,0, 1,0, 0,1"
  CreateBlockTransform 7,"1,-1,0,0,-1,1,-2,0, 1,1,0,0,-1,-1,0,-2, -1,1,0,0,1,-1,2,0,
-1,-1,0,0,1,1,0,2"

  repeat with loopRow = 1 to 20
    repeat with loopCol = 1 to 10
      put empty into sPlayArea[loopCol,loopRow]
    end repeat
  end repeat
```

```
  repeat with loopCol = 1 to 10
    put 0 into sPlayArea[loopCol,21]
  end repeat
  repeat with loopRow = 1 to 20
    put 0 into sPlayArea[0,loopRow]
    put 0 into sPlayArea[11,loopRow]
  end repeat
  put 1 into sGameLoopCounter
  put kNormalAnimation into sFallingDelay
  put 0 into sScore
  put empty into sKeyPress
  put false into sGameOver
  hide field "GameOver"
  put URL("file:" & TopScorePath()) into sScoreList
  put line 1 to 10 of sScoreList into field "TopScoreList"

  CreateNewTetromino
  unlock screen
  GameLoop
end StartGame
```

CreateBlockDefinition accepts a list of 4 pairs of numbers. For easier typing, these are entered as a list on a single line. All CreateBlockDefinition does is break this list into number pairs, each pair on a new line. Each pair indicates the position of the next block relative to the current block. So the top left corner of the tetromino is at 0,0. Then using the example of the square tetromino, 1,0 means draw the next block one column to the right, 0,1 means one row down and finally -1,0 is one column to the left.

```
command CreateBlockDefinition pType,pData
  local index,list
  replace space with empty in pData
  put 1 into index
  repeat for each item loopItem in pData
    put loopItem after list
    if index mod 2 = 0 then
      put cr after list
    else
      put comma after list
    end if
    add 1 to index
  end repeat
  put list into sBlockDefinition[pType]
end CreateBlockDefinition
```

CreateBlockTransform sets up sBlockTransform using a similar idea. Each block of the tetromino is moved relative to its current position. Each rotation is 90 degrees, so four transformations are needed to get it back to the original position. As with CreateBlockDefinition most of the work is about converting a single line of numbers into the correct pairs for each transformation. As hinted in the disclaimer near the beginning, while not much code is required, figuring out the numbers for each transformation was time consuming.

```
command CreateBlockTransform pType,pData
  local index,list,transformCount,transformIndex
  replace space with empty in pData
  put 2 * (the number of lines in sBlockDefinition[pType]) into transformCount
  put 1 into index
  put 1 into transformIndex
  repeat for each item loopItem in pData
    put loopItem after list
    if index mod 2 = 0 then
      put cr after list
    else
      put comma after list
    end if
    if index mod transformCount = 0 then
      put list into sBlockTransform[pType,transformIndex]
      put empty into list
```

```
      add 1 to transformIndex
    end if
    add 1 to index
  end repeat
end CreateBlockTransform
```

The sPlayArea array represents the position of the stationary blocks with column and row indices from 1 to 10 and 1 to 20. To translate each position into a coordinate on the screen, ConvertColRowToScreenXY does some maths. TopScorePath returns a path for the top scores file. In this case it is the same folder as the stack. In a standalone program, and especially on a mobile device this would need to be changed.

```
function ConvertColRowToScreenXY pCol,pRow
  return 340 + kBlockSize * (pCol - 1),32 + kBlockSize * (pRow - 1)
end ConvertColRowToScreenXY

function TopScorePath
  local buffer
  put the effective filename of this stack into buffer
  set itemDelimiter to slash
  delete the last item of buffer
  put slash & "livecode-tetris-scores.txt" after buffer
  return buffer
end TopScorePath
```

CreateNewTetromino creates the four LiveCode graphic objects, positioning each one according to the numbers in sBlockDefinition. Each tetromino has a different colour and the specific type of tetromino is randomly selected. The short id of the graphic and its column and row are put into a list in the sFallingBlocks variable. Storing the information for the falling blocks in a variable separate from sPlayArea (which only stores the blocks that have stopped falling) makes MoveBlocks easier to code.

A property named pvBlock is added to each graphic. This is done to allow the loop back in StartGame to only delete the graphics of the blocks, and nothing else (for example, the border of the play area which is also LiveCode graphic objects) on the card.

If the "s" tetromino is created it is positioned down the screen by one row. This is done because its definition draws two of the blocks higher up the screen, which results in a negative y value for the screen coordinates. When creating a graphic object from code with the templateGraphic, LiveCode adjusts the loc property to remove negative values, even if you want the control to be created off screen. You can always set the loc property after the call to create graphic, but here it is just moved down the screen by one row before it is created.

```
command CreateNewTetromino
  local blockID,blockCol,blockRow
  put random(kMaxBlockType) into sBlockType
  put 1 into sBlockRotation
  put 5 into blockCol
  put 1 into blockRow
  if sBlockType = 3 then add 1 to blockRow
  put empty into sFallingBlocks
  reset templateGraphic
  set the style of the templateGraphic to "rectangle"
  set the rect of the templateGraphic to 1,1,kBlockSize,kBlockSize
  set the penColor of the templateGraphic to "black"
  set the filled of the templateGraphic to true
  repeat for each line loopLine in sBlockDefinition[sBlockType]
    add item 1 of loopLine to blockCol
    add item 2 of loopLine to blockRow
    set the loc of the templateGraphic to ConvertColRowToScreenXY(blockCol,blockRow)
    set the brushColor of the templateGraphic to item sBlockType of
                              "#F8EB5F,#53B9ED,#60CC3A,#E93F1C,#E652E0,#F2AB54,#5882DF"
    create graphic
    put the short ID of the last graphic into blockID
    set the pvBlock of graphic ID blockID to true
    put blockID,blockCol,blockRow & cr after sFallingBlocks
```

```
  end repeat
end CreateNewTetromino
```

That is all the preliminary handlers out of the way. Almost half the code. Now time to start the game with GameLoop. The game loop runs at 30 fps and HandleKeyboard is called each time. While GameLoop is called 30 times per second, there is no need to update the position of the falling blocks that often. The sGameLoopCounter mod sFallingDelay expression means the blocks fall one row every second and then collisions checked for. If necessary, any filled rows are deleted and the score updated.

```
on GameLoop
  HandleKeyboard
  if (sGameLoopCounter mod sFallingDelay) = 0 then
    if StopFallingOnCollision() then
      RemoveFullRows
      CreateNewTetromino
    else
      UpdateFallingBlocks
    end if
  end if
  if sGameOver then
    GameOver
  else
    -- run game at 30 fps
    send "GameLoop" to me in 2 ticks
    add 1 to sGameLoopCounter
  end if
end GameLoop
```

HandleKeyboard acts on the sKeyPress variable and if necessary moves or rotates the falling blocks. When the spacebar is pressed, sFallingDelay is set to 2. This speeds up the falling blocks, while the game loop is still at 30 fps, UpdateFallingBlocks is called every second time around the loop to make the tetromino fall faster.

```
command HandleKeyboard
  if sKeyPress is not empty then
    switch sKeyPress
      case "left"
        MoveBlocks -1
        break
      case "right"
        MoveBlocks 1
        break
      case "up"
        RotateBlocks
        break
      case "space"
        put kFastAnimation into sFallingDelay
        break
    end switch
    put empty into sKeyPress
  end if
end HandleKeyboard
```

MoveBlocks moves the blocks one column, either to the left or the right. A loop makes a local copy of sFallingBlocks, but with the new positions and this is then checked to see if there are any collisions. If all positions are free, DrawBlocksInNewPosition is called to update sFallingBlocks and the loc of each block.

```
command MoveBlocks pDelta
  local blockInfo,fallingBlocks
  repeat for each line loopBlockInfo in sFallingBlocks
    put loopBlockInfo into blockInfo
    add pDelta to item 2 of blockInfo
    put blockInfo & cr after fallingBlocks
  end repeat
  if CollisionFree(fallingBlocks) then
```

```
    DrawBlocksInNewPosition fallingBlocks
  end if
end MoveBlocks
```

RotateBlocks is similar to MoveBlocks but the position of each block is not adjusted by the same amount but instead by the contents of sBlockTransform which contains the necessary delta values for each block at each point of the rotation. Again, if all positions are free, DrawBlocksInNewPosition is called to update sFallingBlocks and the loc of each block.

```
command RotateBlocks
  local transIndex,fallingBlocks,delta,transformation
  put sBlockTransform[sBlockType,sBlockRotation] into transformation
  put 1 into transIndex
  repeat for each line loopBlockInfo in sFallingBlocks
    put line transIndex of transformation into delta
    put (item 1 of loopBlockInfo),(item 2 of loopBlockInfo + item 1 of delta),
                  (item 3 of loopBlockInfo + item 2 of delta) & cr after fallingBlocks
    add 1 to transIndex
  end repeat
  if CollisionFree(fallingBlocks) then
    DrawBlocksInNewPosition fallingBlocks
    add 1 to sBlockRotation
    if sBlockRotation > 4 then put 1 into sBlockRotation
  end if
end RotateBlocks
```

DrawBlocksInNewPosition does what is says and updates the loc of the graphic objects.

```
command DrawBlocksInNewPosition pFallingBlocks
  lock screen
  put pFallingBlocks into sFallingBlocks
  repeat for each line loopBlockInfo in sFallingBlocks
    set the loc of the graphic ID (item 1 of loopBlockInfo) to
              ConvertColRowToScreenXY(item 2 of loopBlockInfo,item 3 of loopBlockInfo)
  end repeat
  unlock screen
end DrawBlocksInNewPosition
```

Checking for a collision in CollisionFree is simplified because only the stationary blocks are in the sPlayArea array. If sPlayArea also stored the falling blocks there would be extra checks to do.

```
function CollisionFree pFallingBlocks
  local continue
  put true into continue
  repeat for each line loopBlockInfo in pFallingBlocks
    if sPlayArea[item 2 of loopBlockInfo, item 3 of loopBlockInfo] is not empty then
      put false into continue
      exit repeat
    end if
  end repeat
  return continue
end CollisionFree
```

StopFallingOnCollision has its own loop to check for any overlap (collision) when the blocks are moved down one row. If there would be an overlap, sPlayArea is filled at the positions of the now stationary blocks and sFallingBlocks is emptied, ready for the next tetromino. sCollisionRow is set to reduce the number of rows that RemoveFullRows needs to check next. If there is a collision at the top row the game is over.

```
function StopFallingOnCollision
  local collision,collisionRow,topRow
  repeat for each line loopBlockInfo in sFallingBlocks
    if sPlayArea[item 2 of loopBlockInfo, item 3 of loopBlockInfo + 1] is not empty
                                                                              then
      put true into collision
      exit repeat
```

```
    end if
  end repeat
  if collision then
    put 0 into sCollisionRow
    put 10 into topRow
    repeat for each line loopBlockInfo in sFallingBlocks
      put item 3 of loopBlockInfo into collisionRow
      put min(collisionRow,topRow) into topRow
      put max(collisionRow,sCollisionRow) into sCollisionRow
      put item 1 of loopBlockInfo into sPlayArea[item 2 of loopBlockInfo, collisionRow]
    end repeat
    put topRow = 1 into sGameOver
    put empty into sFallingBlocks
    put kNormalAnimation into sFallingDelay
  end if
  return collision
end StopFallingOnCollision
```

After a tetromino has stopped moving, RemoveFullRows is called. This is a long handler. The sCollisionRow variable set in StopFallingOnCollision means only 4 rows (the maximum height of a tetromino is 4 blocks) in sPlayArea need checking. Sure, sPlayArea is a small array and the performance (speed) of the game would be the same if the entire array was checked, but if for the cost of one line of code and a variable the loop is 5 times faster, why not make it faster? Thinking about these issues, and processing only the data that needs to be, can be the difference between a slick game and a sluggish one.

When rows are removed the score is updated.

```
command RemoveFullRows
  local checkRow,rowFull,rowEmpty,blockID,removeRowCount,emptyPlayArea
  put sCollisionRow into checkRow
  put 0 into removeRowCount
  put false into emptyPlayArea
  repeat until checkRow < (sCollisionRow - 3)
    put true into rowFull
    repeat with loopCol = 1 to 10
      if sPlayArea[loopCol, checkRow] is empty then
        put false into rowFull
        exit repeat
      end if
    end repeat
    if rowFull then
      put true into emptyPlayArea
      repeat with loopCol = 1 to 10
        delete graphic ID sPlayArea[loopCol, checkRow]
      end repeat
      wait 200 milliseconds with messages
      repeat with loopRow = checkRow down to 1 step -1
        put true into rowEmpty
        repeat with loopCol = 1 to 10
          put sPlayArea[loopCol, loopRow-1] into blockID
          put blockID into sPlayArea[loopCol, loopRow]
          if blockID is not empty then
            set the loc of the graphic ID blockID to
                                              ConvertColRowToScreenXY(loopCol,loopRow)
            put false into rowEmpty
            put false into emptyPlayArea
          end if
        end repeat
        if rowEmpty then exit repeat
      end repeat
      add 1 to removeRowCount
      wait 200 milliseconds with messages
    else
      subtract 1 from checkRow
    end if
  end repeat
```

```
  if removeRowCount > 0 then
    add item removeRowCount of "50,150,350,1000" to sScore
    if emptyPlayArea then add 2000 to sScore
    put sScore into field "Score"
  end if
end RemoveFullRows
```

If there isn't a collision in StopFallingOnCollision the falling blocks of the tetromino are moved down one row by the call to UpdateFallingBlocks.

```
command UpdateFallingBlocks
  local blockInfo,fallingBlocks
  repeat for each line loopBlockInfo in sFallingBlocks
    put loopBlockInfo into blockInfo
    add 1 to item 3 of blockInfo
    put blockInfo & cr after fallingBlocks
    end repeat
  DrawBlocksInNewPosition fallingBlocks
end UpdateFallingBlocks
```

The next two handlers receive the LiveCode messages related to the keys and puts them into the sKeyPress variable for use by HandleKeyboard.

```
on arrowKey pKey
  put pKey into sKeyPress
  pass arrowKey
end arrowKey

on keyDown pKey
  if pKey = space then put "space" into sKeyPress
  pass keyDown
end keyDown
```

Lastly, GameOver is called from the game loop if sGameOver is true. This updates the high score list and uses the LiveCode sort command and some handy string processing before storing the score.

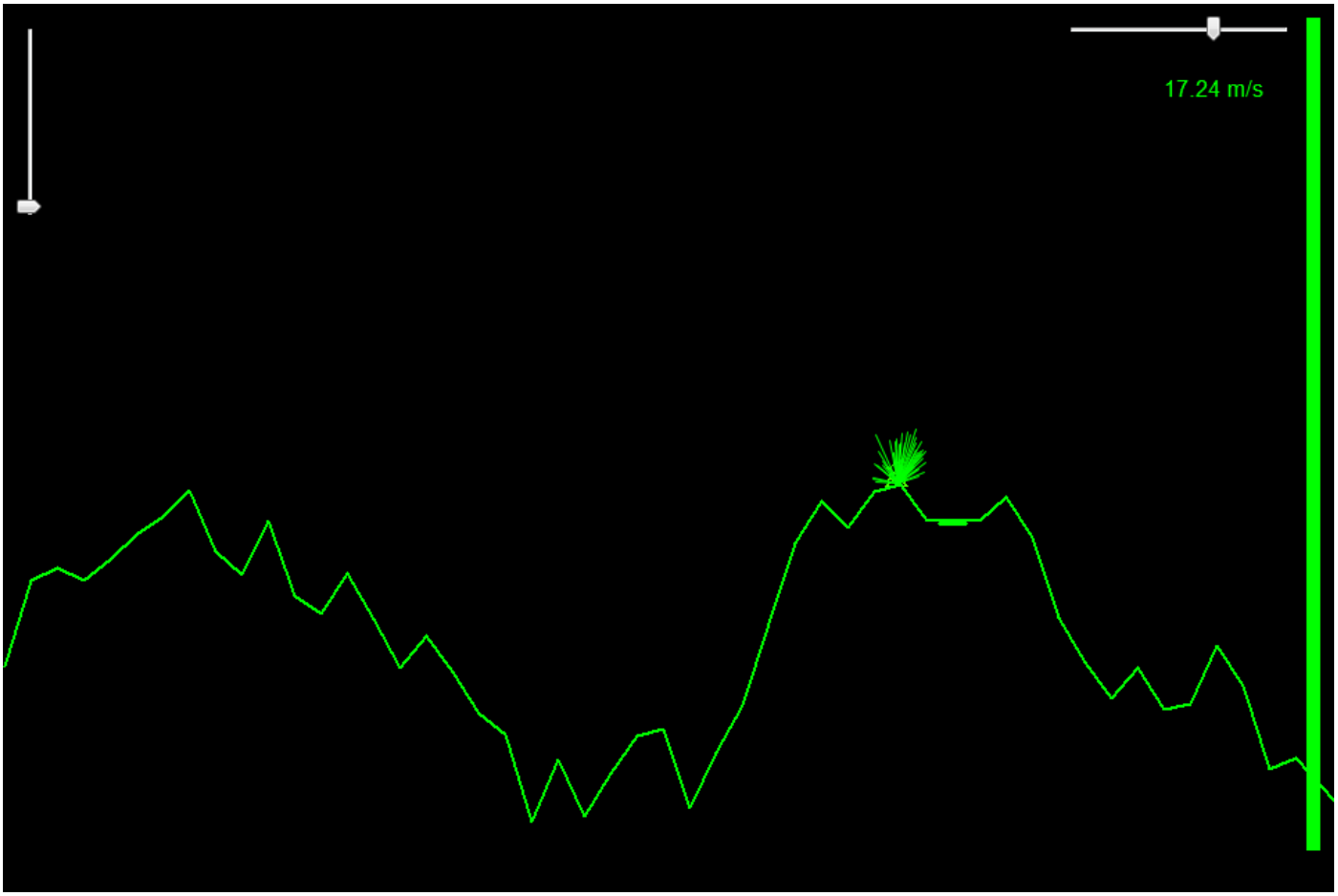
```
on GameOver
  show field "GameOver"
  if sScore is not among the lines of sScoreList then
    put sScore & cr after sScoreList
    sort sScoreList numeric descending
    put line 1 to 10 of sScoreList into field "TopScoreList"
    put sScoreList into URL("file:" & TopScorePath())
  end if
end GameOver
```

Tetris on LiveCode. Enjoy, chasing that ever higher score. At the risk of repeating one central idea from the previous games, how cool is it that with LiveCode a game like this can be coded in an afternoon?

Here are some suggestions for improving the game.

- Make the tetrominos fall faster as the score increases.
- Add a preview that shows the next tetromino that will appear.
- Allow anti-clockwise rotation of the tetrominos .

Blank Page



Oops! Yet another crash near the landing site.

Game 6

Lunar Lander

Another true classic. This was around before personal computers were invented. The very first versions were text only games that printed as text your height and velocity above the moon. Then you specified the amount of fuel you wanted to burn. The aim being make sure the velocity is small enough when reaching the surface so the LEM wouldn't crash. The mathematics that powered it was reasonably easy to understand and with a bit of practise you could master it with a safe landing every time.

The game here is a graphical Lunar Lander inspired on later versions from the 1970's that introduced vector styled graphics and 2 dimensions of movement.

In this game you will see:

- A random moonscape made with a single graphic object
- Scroll bars used to indicate a value and allow control by touch
- A simple animation for a crash landing
- Handling of more than one key pressed at the same time
- Multiple images used to animate the LEM

Below is the StartGame handler. This handler is called from the openCard handler and calls commands to initialise everything. Then ShowInstructions displays important information to the player. Originally this game was developed with a touch device in mind so while the instructions refer to keys, the game can be played by touch alone by using the scroll bars to control the thrust.

```
command StartGame
  lock screen
  CancelPendingMessage "GameLoop"
  InitializeLEM
  InitializeSurface
  InitializeFuelThrust
  InitializeStatus
  ShowInstructions
  unlock screen
```

```
end StartGame
```

CancelPendingMessage is removes any GameLoop messages in the message queue. As in previous games, this is a good idea when starting a game, because if there are messages pending that call your GameLoop (which can happen during development) if you don't first remove them the game animation will be unpredictable. Unlike previous games this action has been put into its own handler.

```
command CancelPendingMessage pMessage
  repeat for each line loopMsg in the pendingMessages
    if (item 3 of loopMsg) = pMessage then
      cancel (item 1 of loopMsg)
    end if
  end repeat
end CancelPendingMessage
```

InitializeLEM sets the size of the field used to display text and some commonly used values are put into variables on the card. It is more convenient to refer to sCardWidth instead of the width of this card. The LEM (the lunar lander) is actually 6 different images, only one of these being visible at a time. These images were created in a bitmap editor (Paint.net), each one showing a different combination of exhaust from the main and navigation thrusters. The LEM is then put at a random location at the top of the screen.

```
command InitializeLEM
  put "lem-d.png" into sVisibleLem

  put the width of this card into sCardWidth
  put the height of this card into sCardHeight
  put the width of image sVisibleLem into sLemWidth

  set the width of field "textDisplay" to round(sCardWidth * 0.7)
  set the height of field "textDisplay" to round(sCardHeight * 0.7)
  set the loc of field "textDisplay" to sCardWidth div 2,sCardHeight div 2
  hide field "textDisplay"

  put 2 * sLemWidth into sDoubleLemWidth
  put sLemWidth div 2 into sHalfLemWidth

  showimage "lem-d.png" to true
  hide image "lem-t.png"
  hide image "lem-l.png"
  hide image "lem-r.png"
  hide image "lem-tl.png"
  hide image "lem-tr.png"

  put random(round(sCardWidth - (sLemWidth * 4))) + (sLemWidth * 2) into sLemX
  put sLemWidth into sLemY
  put 0 into sLemVX
  put 1 into sLemVY
  set the loc of image sVisibleLem to round(sLemX),round(sLemY)
end InitializeLEM
```

The surface of the moon is drawn in InitializeSurface. The surface is a single LiveCode graphic control with the style of curve. This graphic is defined by a list of points (a pair of x,y values) that are then drawn. A curve can have any number of points, but unlike a polygon does not need to be closed.

Random numbers make a jagged landscape with some extra code making sure the actual landing area is flat. The sHeightArray stores the value of the height of the landscape at set intervals across the screen. It is this variable that is used to detect whether the LEM has reached the surface, or not.

```
command InitializeSurface
  local pointCount, pointList, pX, pY, pDX, pDY, prevY, platformIndex, platformHeight,
heightIndex, lemOffset

  put 10 into lemOffset
  put sCardWidth div sLemWidth into pointCount
  put sCardWidth / (pointCount * 3) into sHeightFactor
```

```
  put round(sCardHeight * 0.6) into platformHeight
  put random(pointCount - 4) + 2 into platformIndex

  put the height of this card * 0.6 + random(round(sCardHeight * 0.3)) into pY
  put pY into prevY

  put 0 into heightIndex
  put sCardWidth div pointCount into pDX
  put empty into pointList
  put 1 into pX
  repeat with loopIndex = 1 to pointCount
    put pX,pY & cr after pointList
    put pY - lemOffset into sHeightArray[heightIndex]
    add 1 to heightIndex
    if loopIndex = platformIndex then
      put pX + pDX div 2 into sPlatformX1
      put pY + 2 into sPlatformY1
    else if loopIndex = platformIndex + 1 then
      put pX + pDX div 2 into sPlatformX2
      put pY + 2 into sPlatformY2
    else
      add random(round(sCardHeight * 0.2)) - round(sCardHeight * 0.1) to pY
      if pY > (sCardHeight - 20) then put (sCardHeight - 20) -
                                                    round(sCardHeight * 0.1) into pY
      if pY < (sCardHeight div 2) then put sCardHeight div 2 +
                                                    round(sCardHeight * 0.1) into pY
      if abs(pY - prevY) < 4 then subtract 8 from pY
    end if
    put (pY - lemOffset + sHeightArray[heightIndex - 1]) div 2 into
                                                          sHeightArray[heightIndex]
    add 1 to heightIndex
    put pY - lemOffset into sHeightArray[heightIndex]
    add 1 to heightIndex

    put pY into prevY
    add pDX to pX
  end repeat
  put sCardWidth,pY & cr after pointList
  set the points of graphic "LunarSurface" to pointList
  set the points of graphic "Platform" to sPlatformX1,sPlatformY1 & cr &
                                                      sPlatformX2,sPlatformY2
  put sPlatformX1 + (sPlatformX2 - sPlatformX1) div 2 into sPlatformMidX
end InitializeSurface
```

InitializeFuelThrust sets up the scroll bars that indicate (and optionally control) the thrust of the LEM. The fuel gauge is two rectangle graphics. One that is unfilled, and the other a filled rectangle that changes in size as the fuel is consumed.

```
command InitializeFuelThrust
  set the startValue of scrollbar "Navigation" to kThrustLimit
  set the endValue of scrollbar "Navigation" to 0
  set the thumbPosition of scrollbar "Thrust" to 0

  set the startValue of scrollbar "Navigation" to -kNavLimit
  set the endValue of scrollbar "Navigation" to kNavLimit
  set the thumbPosition of scrollbar "Navigation" to 0

  put kMaxFuel into sFuel
  put sCardWidth - 20 into sFuelLeft
  put sCardWidth - 10 into sFuelRight
  set the rectangle of graphic "FuelBorder" to sFuelLeft,10,sFuelRight,10+kMaxFuel
  set the rectangle of graphic "Fuel" to the rectangle of graphic "FuelBorder"
  show graphic "Fuel"
end InitializeFuelThrust
```

InitializeStatus sets a few more variables and puts kReady into sLemStatus. The sLemStatus variable controls what is happening in the Lunar Lander. There are 5 main types of states. Ready to start, has flown out of bounds, has landed, is showing the score, and the game is over. This variable makes the flow of the game become like a state machine. Like in Pong, a variable and switch statement can simplify the coding for what happens at each stage of the game.

```
command InitializeStatus
  put sqrt(((sLemX - sPlatformMidX) * (sLemX - sPlatformMidX)) +
                ((sLemY - sPlatformY1) * (sLemY - sPlatformY1))) into sPlatformDistance
  put sqrt((sCardWidth * sCardWidth) + (sCardHeight * sCardHeight)) into sMaxDistance
  set the text of field "labeHighScore" to sHighScore
  set the text of field "labeSpeed" to "0.00 m/s"
  put kReady into sLemStatus
end InitializeStatus
```

Then ShowInstructions makes the text field visible and puts some text in it. Once the text is shown, the game waits for a mouseDown event on the field before starting the game loop.

```
command ShowInstructions
  lock screen
  show field "textDisplay"
  set the text of field "textDisplay" to "Land the Lunar Module as gently and
accurately as you can. Keep watch on your amount of fuel." & cr & cr & \
      "Press the Left and Right arrow keys to move left or right." & cr & cr & \
      "Press A and Z to control the thrust of the main retro rocket." & cr & cr & \
      "Touch here to begin."
  unlock screen
end ShowInstructions
```

In mouseDown, a check is made to see if sLemStatus is empty, and if it is kEndOfGame is put in it. This has been added because if in the LiveCode IDE, in the Script Editor Preferences the Variable preservation option is turned off, the sLemStatus may become empty if you are editing the code during development. If sLemStatus was empty setting it to kEndOfGame ensures that StartGame is called to properly initialize everything again.

```
on mouseDown
  if sLemStatus is empty then put kEndOfGame into sLemStatus
  switch the short name of the target
    case "textDisplay"
      switch sLemStatus
        case kReady
          StartDescent
          break
        case kShowScore
          ShowScore
          break
        case kEndOfGame
          StartGame
          break
      end switch
  end switch
end mouseDown
```

If the status is kReady, StartDescent is called to begin the game play. StartDescent is a very small handler that calls GameLoop. Its code could have been moved to one of the case statements in the switch statement of mouseDown, but having a separate handler makes the code self documenting. Also, if later more logic needs to be added to start the descent, the code in mouseDown is kept tidy.

```
on StartDescent
  hide field "textDisplay"
  put false into sGameOver
  GameLoop
end StartDescent
```

Next is the GameLoop. This calls HandleKeyboard, UpdateGraphics, CheckForLanding and UpdateSpeed. Then depending on whether the surface has been reached, CrashAnimation and ShowLandingOutcome are called. If the surface has not been reached GameLoop is called again in 2 ticks so the game runs at 30 fps.

```
command GameLoop
  HandleKeyboard
  lock screen
  UpdateGraphics
  UpdateVelocity
  CheckForLanding
  unlock screen
  if sGameOver then
    if sVelocity > 10 and sLemStatus <> kOutOfBounds then
      CrashAnimation
    end if
    ShowLandingOutcome
  else
    send "GameLoop" to me in 2 ticks
  end if
end GameLoop
```

HandleKeyboard uses the LiveCode keysDown function which is very useful in real-time games. It returns a list of all keys that are currently pressed down. This is ideal in a game where the player may be pressing more than one key at once. Four conditional statements change the thumbposition for the appropriate scroll bar, depending on the pressed key.

```
command HandleKeyboard
  local keyBuffer
  put the keysDown into keyBuffer
  if 65361 is among the items of keyBuffer then -- left
    if the thumbPosition of scrollbar "Navigation" > -kNavLimit then
      set the thumbPosition of scrollbar "Navigation" to the thumbPosition of
                                                          scrollbar "Navigation" - 1
    end if
  end if
  if 65363 is among the items of keyBuffer then -- right
    if the thumbPosition of scrollbar "Navigation" < kNavLimit then
      set the thumbPosition of scrollbar "Navigation" to the thumbPosition of
                                                          scrollbar "Navigation" + 1
    end if
  end if
  if 97 is among the items of keyBuffer then -- a
    if the thumbPosition of scrollbar "Thrust" < kThrustLimit then
      set the thumbPosition of scrollbar "Thrust" to the thumbPosition of
                                                              scrollbar "Thrust" + 1
    end if
  end if
  if 122 is among the items of keyBuffer then -- z
    if the thumbPosition of scrollbar "Thrust" > 0 then
      set the thumbPosition of scrollbar "Thrust" to the thumbPosition of
                                                              scrollbar "Thrust" - 1
    end if
  end if
end HandleKeyboard
```

UpdateGraphics checks the thumbPosition of the navigation and thrust scroll bars and then selects the image of the LEM that should be visible. There is a different image for each possible combination of thrust. For example, if the main retro rocket (the one underneath) is firing and the LEM is moving to the left (which means the thruster on the right side is firing) then lem-tr.png is made visible.

The horizontal and vertical position of the LEM is adjusted according to the velocity. Then the amount of fuel is reduced depending on the amount of thrust. None of the calculations reflect real world physics. All the number constant were figured out through experimentation to get satisfying game play that is not too easy or difficult.

```
command UpdateGraphics
  local thrust, navigation, velocity, visibleLem

  put "lem-d.png" into visibleLem
  if sFuel > 0 then
    put the thumbPosition of scrollbar "Navigation" div 5 into navigation
    add kNavForce * navigation to sLemVX
    --if abs(sLemVX) > 3 then set the thumbPosition of scrollbar "Navigation" to 0

    put the thumbPosition of scrollbar "Thrust" into thrust
    subtract kThrustForce * thrust from sLemVY
    if thrust > 0 then
      put "lem-t.png" into visibleLem
      if navigation <> 0 then
        if navigation < 0 then
          put "lem-tr.png" into visibleLem
        else
          put "lem-tl.png" into visibleLem
        end if
      end if
    else
      if navigation <> 0 then
        if navigation < 0 then
          put "lem-r.png" into visibleLem
        else
          put "lem-l.png" into visibleLem
        end if
      end if
    end if
  else
    set the thumbPosition of scrollbar "Navigation" to 0
    set the thumbPosition of scrollbar "Thrust" to 0
  end if
  add kGravity to sLemVY
  add sLemVX to sLemX
  add sLemVY to sLemY

  -- update lem image
  set the loc of image visibleLem to sLemX,sLemY
  if visibleLem <> sVisibleLem then
    show image visibleLem
    hide image sVisibleLem
    put visibleLem into sVisibleLem
  end if

  if navigation <> 0 then
    subtract abs(navigation) * 0.005 from sFuel
  end if
  if thrust <> 0 then
    subtract thrust * 0.05 from sFuel
  end if
  put max(sFuel, 0) into sFuel
  set the rectangle of graphic "Fuel" to sFuelLeft,round(10 + kMaxFuel -
                                                         sFuel),sFuelRight,10 + kMaxFuel
end UpdateGraphics
```

UpdateVelocity calculates the velocity of the LEM in m/s. As mentioned above, real physics is not used so kSpeedFactor is used to convert the velocity which is actually in pixel units into numbers that look like reasonable values for metres per second.

```
command UpdateVelocity
  put abs(sqrt((sLemVX * sLemVX) + (sLemVY * sLemVY)) * kSpeedFactor) into sVelocity
  set the numberFormat to "#.00"
  set the text of field "labeSpeed" to round(sVelocity, 2) & " m/s"
end UpdateVelocity
```

After updating the graphics and the velocity, CheckForLanding is called in the game loop. This compares the current vertical coordinate of the LEM with the contents of sHeightArray that was filled in InitializeSurface. If the current vertical coordinate is greater than the contents of sHeightArray, the LEM has reached the surface and the velocity is checked to see if it is a smooth or crash landing. If you fly off the screen then the game ends because you are out of bounds.

```
command CheckForLanding
  local platformCentre
  put empty into sLemStatus
  if sLemY > sHeightArray[round(sLemX / sHeightFactor)] then
    put true into sGameOver
    put sPlatformX1 + ((sPlatformX2 - sPlatformX1) div 2) into platformCentre

    if (sLemStatus is empty) and (abs(sLemX - platformCentre) < sHalfLemWidth) and
                                                              (sVelocity <= 5) then
      put kAccurateLanding into sLemStatus
    end if
    if (sLemStatus is empty) and (abs(sLemX - platformCentre) < sLemWidth) and
                                                            (sVelocity <= 10) then
      put kRoughLanding into sLemStatus
    end if
    if (sLemStatus is empty) and (abs(sLemX - platformCentre) < sDoubleLemWidth) and
                                                            (sVelocity <= 10) then
      put kOffTargetLanding into sLemStatus
    end if
    if (sLemStatus is empty) and (sVelocity <= 10) then
      put kWildernessLanding into sLemStatus
    end if
    if (sLemStatus is empty) and (sVelocity > 10) then
      put kCrashLanding into sLemStatus
    end if
    if (sLemStatus is empty) and (sVelocity > 15) then
      put kFatalLanding into sLemStatus
    end if
  else
    if (sLemY < -sDoubleLemWidth) or (sLemX < -sDoubleLemWidth) or
                  (sLemX > (sCardWidth + sDoubleLemWidth)) or sLemY > sCardHeight then
      put true into sGameOver
      put kOutOfBounds into sLemStatus
    end if
  end if
end CheckForLanding
```

If there is a crash landing CrashAnimation is called. This draws lines radially from the base of the LEM. The higher the velocity, the more lines drawn. These are deleted after 4 seconds.

```
command CrashAnimation
  local lunarY
  put sHeightArray[round(sLemX / sHeightFactor)] + sHalfLemWidth into lunarY

  reset templateGraphic
  set the style of templateGraphic to "line"
  set the penColor of templateGraphic to "green"
  set the pvTemp of templateGraphic to true
  repeat round(abs(sVelocity) * 3) times
    create graphic
    set the points of it to round(sLemX),round(lunarY),round(sLemX -20 +
                                                  random(40)),round(lunarY - random(40))
    wait 10 milliseconds
  end repeat
  wait 4 seconds
  hide image sVisibleLem
  repeat with loopIndex = the number of graphics down to 1
    if the pvTemp of graphic loopIndex then delete graphic loopIndex
  end repeat
end CrashAnimation
```

Then ShowLandingOutcome shows a short description about how well you landed.

```
command ShowLandingOutcome
  lock screen
  show field "textDisplay"
  switch sLemStatus
    case kOutOfBounds
      set the text of field "textDisplay" to "All contact lost. Mission failed."
      break
    case kAccurateLanding
      set the text of field "textDisplay" to "Touch-down on target. Perfect Mission."
      break
    case kRoughLanding
      set the text of field "textDisplay" to "On target, but rough touch-down.
Successful mission, hope nothing is damaged."
      break
    case kOffTargetLanding
      set the text of field "textDisplay" to "Successful touch-down, but missed the
target. Mission complete."
      break
    case kWildernessLanding
      set the text of field "textDisplay" to "Safe touch-down, but missed landing pad.
Will need recovery team."
      break
    case kCrashLanding
      set the text of field "textDisplay" to "Crash landing. There may be survivors,
but mission failed."
      break
    case kFatalLanding
      set the text of field "textDisplay" to "Crash landing, all contact lost. Mission
failed."
      break
  end switch
  put sLemStatus into sLandingStatus
  put kShowScore into sLemStatus
  unlock screen
end ShowLandingOutcome
```

After you click on the landing information, ShowScore is called to show your total score, and if necessary update the high score. Click on this and you are ready to try again.

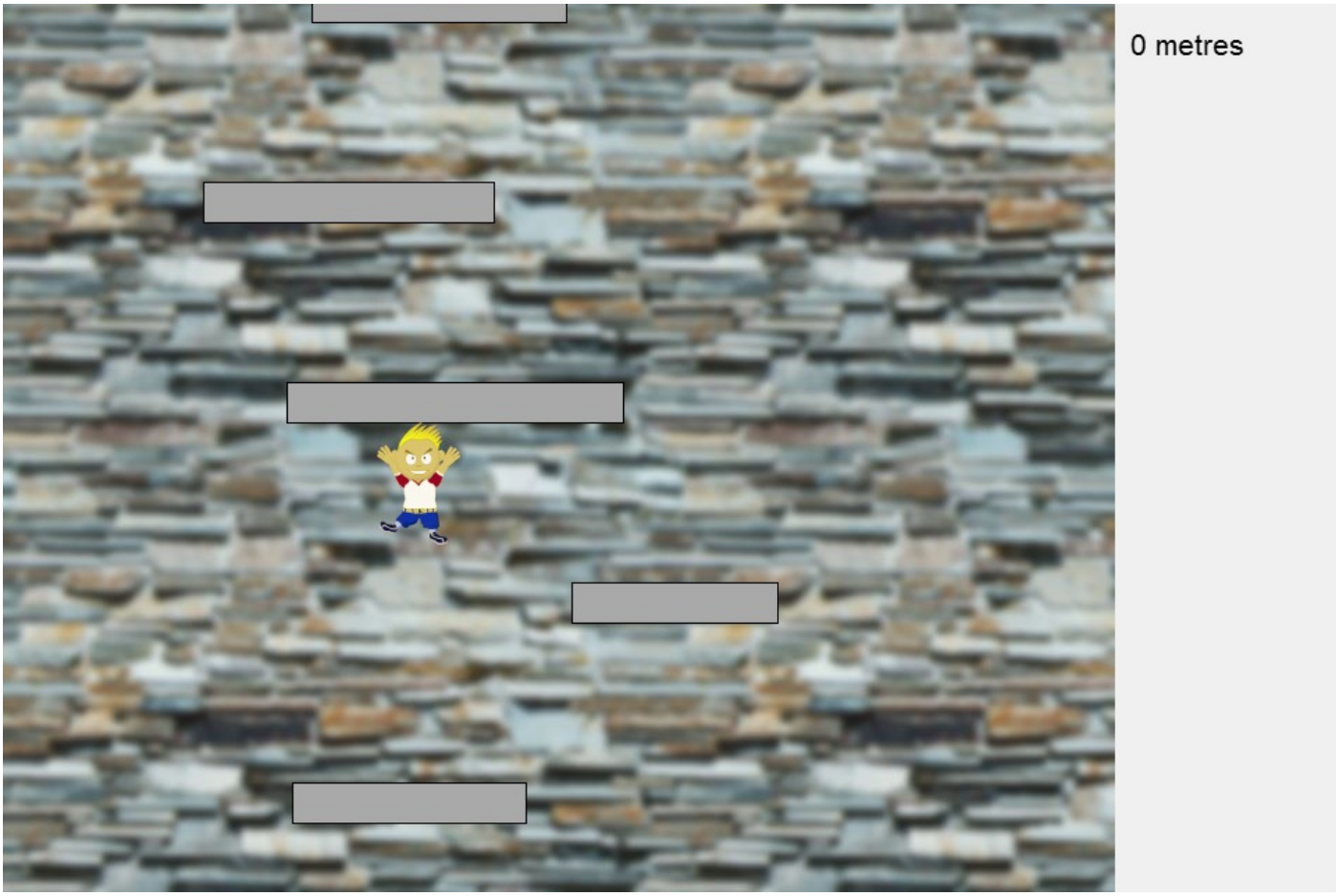
```
command ShowScore
  local distanceScore, speedScore, fuelScore, accuracyScore, totalScore
  set the tabStops of field "textDisplay" to 200

  if (sVelocity < 15) and (kOutOfBounds <> sLandingStatus) then
    put round((10 - sVelocity) * 100) into speedScore
    put round((round(sPlatformDistance) / sMaxDistance) * 500) into distanceScore
    put round((sFuel / kMaxFuel) * 500) into fuelScore
    if abs(sLemX - sPlatformMidX) < sDoubleLemWidth then
      put round(((sDoubleLemWidth - abs(sLemX - sPlatformMidX)) / sDoubleLemWidth) *
                                                                  1000) into accuracyScore
    else
      put 0 into accuracyScore
    end if
    if sVelocity > 10 then
      put 0 into speedScore
    end if
  else
    put 0 into speedScore
    put 0 into distanceScore
    put 0 into fuelScore
    put 0 into accuracyScore
  end if

  put distanceScore + speedScore + fuelScore + accuracyScore into totalScore
```

```
  set the text of field "textDisplay" to "Pilot Rating" & cr & \
      "Speed:" & tab & speedScore & cr & \
      "Accuracy:" & tab & accuracyScore & cr & \
      "Fuel Left bonus:" & tab & fuelScore & cr & \
      "Distance bonus:" & tab & distanceScore & cr & \
      "TOTAL SCORE:" & tab & totalScore
  if totalScore > sHighScore then
    put cr & cr & "New High Score." after field "textDisplay"
    put totalScore into sHighScore
  end if

  put kEndOfGame into sLemStatus
end ShowScore
```

That's it, another true classic game in little more than 400 lines of code. Will you safely land the LEM?

Here are some suggestions for improving the game.

- Make the jaggedness of the lunar surface more variable, i.e. sometimes smooth, other times rough.
- Add levels of difficulty where the amount of fuel varies.
- Make the thruster rockets have a bigger effect as the LEM burns fuel and gets lighter.



How many metres can Anime Boy jump?

Game 7

Tower Jump

Another arcade style game, which I call Tower Jump. Before the indie game scene really took off and became the cool label for individual and small teams of independent game developers, there was Free Lunch Design. In 2001, Johan Peitz and team made Icy Tower, a very fun addictive free game. Which according to Wikipedia is based on an earlier Linux game named Xjump or Falling Tower.

In this game you will see:

- An endless scrolling background
- A LiveCode scrolling group
- An animated character
- Collision detection with one way platforms

The StartGame handler calls ResetGame, InitializeGraphics, InitializeHero, InitializePlatforms and starts the game loop. ResetGame and InitializeGraphics deletes and re-creates graphics every time you run the game. This is done because during development it is not uncommon to end up with extra copies of the graphics. So to keep the stack tidy, deleting and making fresh graphics is done each time you play. But in a polished game, you would reorganise the code to minimise the number of objects created at runtime to speed up the loading of the game.

```
on StartGame
  ResetGame
  InitializeGraphics
  InitializePlatforms
  InitializeHero

  put 0 into sGameLoopCounter
  put false into sGameOver
  hide field "labeGameOver"
  GameLoop
end StartGame
```

ResetGame removes any messages in the message queue. As mentioned in almost every game in this book, this is a good idea when starting a game, because if there are pending messages that call your GameLoop (which can happen during development) if you don't first remove the messages, the animation will be unpredictable. Next the platforms from the previous run of the game are removed. Each platform graphic has a custom property named pvPlatform that is set to true in InitializePlatforms so only those graphics are deleted.

```
command ResetGame
  local pendMsg
  put the pendingmessages into pendMsg
  repeat for each line loopLine in pendMsg
    cancel item 1 of loopLine
  end repeat

  local blockID
  repeat with loopIndex = the number of graphics down to 1
    if the pvPlatform of graphic loopIndex then delete graphic loopIndex
  end repeat
end ResetGame
```

InitializeGraphics sets the templateGraphic so it is ready to create the graphic objects for the platforms. The group used for the scrolling wall background image is also set up. A LiveCode group is a powerful object. Put objects in a group, lock the size of the group so it is smaller than the area required for the objects and you now have objects that can scroll on screen with very little programming effort. Changing the two properties hscroll and vscroll of the group, scrolls all the objects in that group on screen.

To get best performance, the layerMode of the templateGraphic is set to dynamic and the layerMode of the group is set to scrolling. Finally, the acceleratedRendering of the stack is set to true.

```
command InitializeGraphics
  reset templateGraphic
  set the style of the templateGraphic to "rectangle"
  set the brushColor of the templateGraphic to "DarkGray"
  set the penColor of the templateGraphic to "black"
  set the filled of the templateGraphic to true
  set the layerMode of the templateGraphic to "dynamic"
  set the pvPlatform of the templateGraphic to true
  put kWallRepeatHeight into sWallOffset
  set the vscroll of group "Wall" to sWallOffset
  set the layerMode of group "Wall" to "scrolling"
  set the acceleratedRendering of this stack to true

  put the height of this card into sCardHeight
end InitializeGraphics
```

The jumping character (you the player) is set up in InitializeHero. There are 5 images for the hero. These were made in Anime Studio using "AnimeBoy" from the included Content Library. Each image has a layerMode of dynamic and variables on the card related to the hero are initialized.

```
command InitializeHero
  put 300 into sHeroX
  put 0 into sHeroY
  put 0 into sSpeedX
  put 0 into sSpeedY
  put 0 into sStandingCounter
  put true into sCanJump
  put kHeroJumping into sHeroState

  set loc of image "hero-1.png" to sHeroX,sHeroY
  set the layerMode of image "hero-1.png" to "dynamic"
  set the layer of image "hero-1.png" to "top"
  show image "hero-1.png"
  put "hero-1.png" into sHeroImage
```

```
  set the layerMode of image "hero-2.png" to "dynamic"
  set the layer of image "hero-2.png" to "top"
  hide image "hero-2.png"
  set the layerMode of image "hero-3.png" to "dynamic"
  set the layer of image "hero-3.png" to "top"
  hide image "hero-3.png"
  set the layerMode of image "hero-4.png" to "dynamic"
  set the layer of image "hero-4.png" to "top"
  hide image "hero-5.png"
  set the layerMode of image "hero-5.png" to "dynamic"
  set the layer of image "hero-5.png" to "top"
  hide image "hero-5.png"
end InitializeHero
```

The number of platforms created in InitializePlatforms is one more than can fit on screen. This is because each platform starts above the top edge of the screen so it scrolls into view, instead of suddenly appearing. CreatePlatform returns the short ID of the platform graphic that has a random position and width. These IDs are stored in the sPlatformList variable with one ID per line.

```
command InitializePlatforms
  local platformTop

  put empty into sPlatformList
  put empty into sCurrentPlatform
  put sCardHeight div kPlatformSpacing + 1 into sPlaformCount

  put kNewPlatformY into platformTop
  repeat sPlaformCount times
    put CreatePlatform(platformTop) & cr after sPlatformList
    add kPlatformSpacing to platformTop
  end repeat
end InitializePlatforms

function CreatePlatform pTop
  local platformID,platformWidth,platformLeft
  put 100 + random(300) into platformWidth
  put 100 + random(600 - platformWidth) into platformLeft
  create graphic
  put the short ID of it into platformID
  set the rect of the graphic ID platformID to platformLeft,pTop,platformLeft +
platformWidth,pTop+30
  return platformID
end CreatePlatform
```

After calling InitializePlatforms, StartGame then starts the game by calling GameLoop, after first setting sGameLoopCounter. This variable is later used in UpdatePlatforms to determine the right time to move the bottom platform back to the top, (after it has scrolled off screen). The GameLoop runs at 30 fps by sending a message to itself in 2 ticks. In LiveCode there are 60 ticks each second. The calls made within the loop are between a pair of lock screen and unlock screen statements to speed the animation.

```
on GameLoop
  lock screen
  add 1 to sGameLoopCounter
  MoveBackground
  MovePlatforms
  HandleKeyboard
  MoveHero
  CheckForCollision
  UpdatePlatforms
  unlock screen
  if sGameOver then
    show field "labeGameOver"
  else
    send "GameLoop" to me in 2 ticks
  end if
end GameLoop
```

MoveBackground is called each time round the game loop to move the image of the wall. The image is taller than the height of the stack to allow it to scroll down the screen without a gap appearing. Back in InitializeGraphics the vscroll property of the group containing the wall image is set to kWallRepeatHeight (364). This number is chosen because the wall image was prepared in a bitmap editor in such way that the brick pattern repeats every 364 pixels without an obvious seam. So what happens is that when the vscroll property reaches zero, it is set back to 364 but you cannot see the sudden jump because the visible section of the image looks exactly the same.

What this means is that the vscroll of the wall group repeatedly cycles like this 364,360,356,...,8,4,364,360,356,...8,4,364,.. which gives the illusion of a never ending wall that forever scrolls down the screen even though the bitmap itself is less than 1100 pixels in height.

```
command MoveBackground
  set the vscroll of group "Wall" to sWallOffset
  subtract kMoveDelta from sWallOffset
  if sWallOffset <= 0 then put kWallRepeatHeight into sWallOffset
end MoveBackground
```

The platforms are not in the wall group. Instead they are animated in MovePlatforms by moving them down the screen at the same speed as the wall by adding kMoveDelta (4) to the y coordinate of each platform.

```
command MovePlatforms
  local xy
  repeat for each line loopID in sPlatformList
    put the loc of graphic ID loopID into xy
    add kMoveDelta to item 2 of xy
    set the loc of graphic ID loopID to xy
  end repeat
end MovePlatforms
```

After moving the wall and platforms, HandleKeyboard is called to change the speed and direction of the hero depending on the list in the keysDown function. In games that are likely to have the player press more than one key at a time, this function is better than trying to keep track of each key press with the key and rawKey Up and Down events. The down side is that you need to add your own logic, like done here with the spacebar if you need to detected the specific event that is the equivalent of keyDown.

sHeroState keeps track of the state of the hero. Is the player standing, sliding or jumping? Checking the state when the spacebar is pressed, stops you from trying to jump when you are already in the air. The horizontal and vertical speeds are then adjusted for the jumping movement.

```
command HandleKeyboard
  local keyBuffer
  put the keysDown into keyBuffer
  if kLeftKey is among the items of keyBuffer then put -kHSpeed into sSpeedX
  if kRightKey is among the items of keyBuffer then put kHSpeed into sSpeedX
  if 32 is among the items of keyBuffer then
    if (sHeroState <> kHeroJumping) and sCanJump then
      put kHeroJumping into sHeroState
      put 0.7 * sSpeedX into sSpeedX
      put -50 into sSpeedY
    end if
    put false into sCanJump
  else
    put true into sCanJump
  end if
end HandleKeyboard
```

MoveHero then does the actual movement of the hero image. Depending on the state (standing, sliding or jumping) the sHeroX and sHeroY variables are adjusted by the horizontal and vertical speeds and the appropriate image is selected. The loc of the hero image is set to the new position. Lastly, if the hero drops off the card the game is over.

To give the hero just a little more character there is a fourth state kHeroImpatient where if the hero is standing for more than half a second, the right foot starts tapping to remind you to hurry up.

```
command MoveHero
  local heroImage
  add kMoveDelta to sHeroY
  put "hero-1.png" into heroImage
  switch sHeroState
    case kHeroStanding
      add 1 to sStandingCounter
      if sSpeedX <> 0 then put kHeroSliding into sHeroState
      if sStandingCounter > 15 then
        put 0 into sStandingCounter
        put kHeroImpatient into sHeroState
      end if
      break
    case kHeroImpatient
      add 1 to sStandingCounter
      if sSpeedX <> 0 then put kHeroSliding into sHeroState
      if sStandingCounter > 5 then put "hero-2.png" into heroImage
      if sStandingCounter > 10 then put 0 into sStandingCounter
      break
    case kHeroSliding
      put 0 into sStandingCounter
      if sSpeedX <> 0 then put kHeroSliding into sHeroState
      put kFriction * sSpeedX into sSpeedX
      add round(sSpeedX) to sHeroX
      if round(sSpeedX) = 0 then
        put 0 into sSpeedX
        put kHeroStanding into sHeroState
      else
        if sSpeedX < 0 then
          put "hero-3.png" into heroImage
        else
          put "hero-4.png" into heroImage
        end if
      end if
      break
    case kHeroJumping
      put 0 into sStandingCounter
      add kGravity to sSpeedY
      add sSpeedX to sHeroX
      add sSpeedY to sHeroY
      put "hero-5.png" into heroImage
      break
  end switch
  if heroImage <> sHeroImage then
    hide image sHeroImage
    put heroImage into sHeroImage
    show image sHeroImage
  end if
  set loc of image sHeroImage to sHeroX,sHeroY
  put sHeroY > sCardHeight into sGameOver
end MoveHero
```

In CheckForCollision, if the hero is standing there is nothing to do. When the hero is sliding the within function is checked to find out whether the hero has slipped of the edge of a platform. in that case, the state is changed to jumping. (Even though sliding off the edge of a platform is not really "jumping", for this code it is similar enough to use the same state.)

Next a check is made for a collision with any platform during the jump. This is more complex. Firstly, the platforms are "one way". What this means is that when moving upwards you can move through a platform. While on the way down the platforms are solid and there can be a collision. To handle this the vertical speed is first checked. If it is negative, you are going up the screen and no further checks are done.

When the hero is moving down, sSpeedY is positive, and checks are made. The LiveCode within function is used again, but the vertical speed of the falling hero is added to the bottom edge of the rectangle that is checked. This is necessary because the speed of the falling hero can result in a collision not being detected.

What can happen is the hero is above a platform in one pass of the game loop, and then when CheckForCollision is called in the next pass though GameLoop, the hero is below the platform. And so no collision is detected. In effect, the hero falls through the platform without ever touching it. This is a common case in platform games and needs to be considered when writing code. (Unless the speed of the hero is never larger than the thickness of the platforms and walls.)

Since this game is not about quantum physics this behaviour is not allowed, and so by increasing the size of the rectangle when checking for a collision, it is possible to detect when the hero has passed thought the platform. When that occurs, sHeroY is adjusted to put the image back on the platform that was collided with.

```
command CheckForCollision
  local collisionRect
  switch sHeroState
    case kHeroSliding
      if sHeroX,sHeroY+kHalfHeroHeight+1 is not within the rect of
                                                    graphic ID sCurrentPlatform then
        put kHeroJumping into sHeroState
      end if
      break
    case kHeroJumping
      if sSpeedY > 0 then
        repeat for each line loopID in sPlatformList
          put the rect of graphic ID loopID into collisionRect
          add round(sSpeedY) to item 4 of collisionRect
          if sHeroX,sHeroY+kHalfHeroHeight+1 is within collisionRect then
            if sSpeedX = 0 then
              put kHeroStanding into sHeroState
            else
              put kHeroSliding into sHeroState
            end if
            put 0 into sSpeedY
            put (item 2 of collisionRect) - kHalfHeroHeight into sHeroY
            set loc of image sHeroImage to sHeroX,sHeroY
            UpdateScore item 2 of collisionRect
            put loopID into sCurrentPlatform
            exit repeat
          end if
        end repeat
      end if
      break
  end switch
end CheckForCollision
```

UpdateScore is called from CheckForCollision because the score is updated whenever there is a collision. This means the score can get smaller if you fall off the current platform and fall onto a lower one. When you jump upwards the score increases by one metre for each platform that you go up.

```
on UpdateScore pTopOfPlatform
  if sCurrentPlatform is empty then
    put 0 into sCurrentHeight
  else
    add ((item 2 of the rect of graphic ID sCurrentPlatform) - pTopOfPlatform) div
                                                              50 to sCurrentHeight
  end if
  put sCurrentHeight & " metres" into field "labeScore"
end UpdateScore
```

Lastly in the game loop, UpdatePlatforms is called to check if the bottom platform has moved off the screen. If it has, then it is moved to the top above the visible screen with a new position and width.

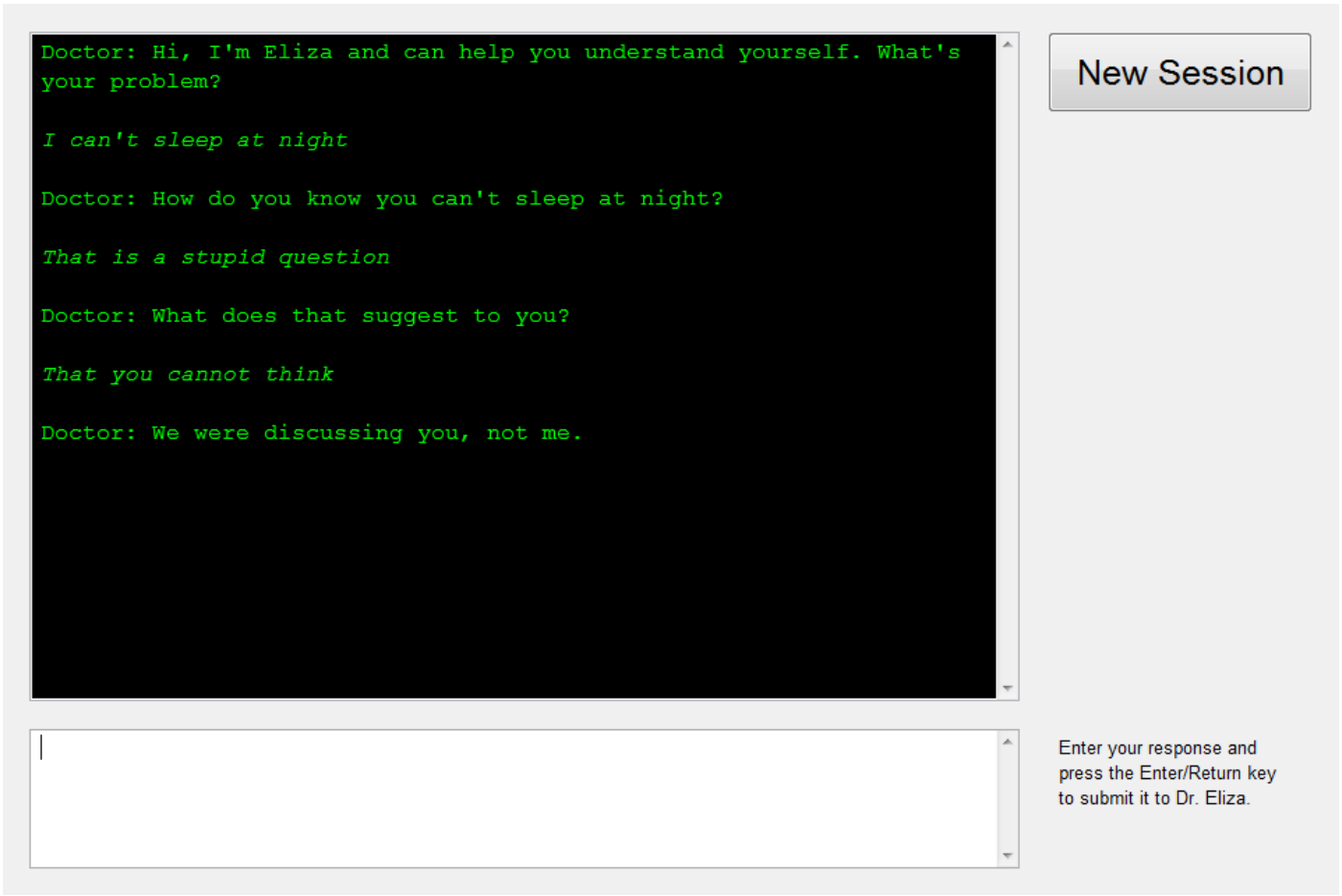
```
command UpdatePlatforms
  local platformID,platformWidth,platformLeft
  if sGameLoopCounter mod (kPlatformSpacing div kMoveDelta) = 0 then
    put the last line of sPlatformList into platformID
    delete the last line of sPlatformList
    put platformID & cr before sPlatformList
    put 100 + random(300) into platformWidth
    put 100 + random(600 - platformWidth) into platformLeft
    set the rect of the graphic ID platformID to
                platformLeft,kNewPlatformY,platformLeft + platformWidth,kNewPlatformY+30
  end if
end UpdatePlatforms
```

How many metres can you jump, and don't you want to have just one more go? A compelling game in just a few hundred lines.

An important feature in this game is that there isn't a button you need to click to play again. For a fast arcade game like this, you do not want to disrupt the flow of the player, by making him or her reach for the mouse to play again. Instead just tap the spacebar and immediately try to beat your previous score.

Here are some suggestions for improving the game.

- Make the platforms less random in size and position, i.e. make sure they are not too far apart.
- Change platform colour as you jump higher.
- Allow a number of lives that you can use up before the game is over.



Difficulty sleeping? Let the doctor help you.

Game 8

Doctor Eliza

Time for some more artificial intelligence (stupidity?), or A.I. for short. The program this time isn't really a game. Way, way back in the 1960's one of the first chatterbot programs was written. Called ELIZA it used simple text processing tricks in an attempt to carry its side of a human conversation. The text processing tricks are so simple that calling it artificial intelligence is laughable, but it can produce uncanny responses, which encourages the human user to give it more credit than is due. At the time it created quite a stir.

The original ELIZA could be programmed for different styles of conversations, with the Doctor style the most successful. Playing the role of a non-directive pyschologist/counsellor meant the program could produce vague and reflective comments, that could seem plausibly human, sometimes. Which resulted in some users asking to be left alone during their personal conversations with the "Doctor". (The version here is not as complex, and it is difficult to imagine this happening with it.)

In this game you will see:

- Sorting a list with one line of code
- How to scroll to the bottom of a text field
- The power of using a custom item delimiter
- A danger of Variable Preservation in the IDE

Because the code is so short, less than 200 lines including the text of all possible replies, I was struggling to find interesting items for the list above. This brevity is possible because of the high level of the LiveCode language, and so this chapter is shorter than the rest because LiveCode made writing the code way too easy. (Which is good.)

StartGame initializes the list of replies and the opposites that sometimes transform the text typed by the user into a form ready to be reflected back in the Doctor's reply. The memoOutput Text Field is filled with the introductory text and the keyboard focus set to memoInput, ready for the user to begin typing.

```
command StartGame
  InitializeReplies
  InitializeOpposites

  put "Doctor: Hi, I'm Eliza and can help you understand yourself. What's your
problem?" & cr & cr into field "memoOutput"
  put empty into field "memoInput"
  focus field "memoInput"
end StartGame
```

InitializeReplies fills the sReplyArray array with the Doctor's replies. The handler starts with the delete local sReplyArray line because if you have the LiveCode IDE Script Editor Preferences set with Variable Preservation turned on, and you are changing the keys used in sReplyArray, you may get unexpected bugs during development.

For example, in this line of code:

```
put "Are you saying no just to be negative?|You are being a bit negative.|Why not?|Are
you sure?|Why no?" into sReplyArray["no "]
```

earlier versions did not include the space after the "no " in the key. The space was added because if the user typed, "Is Eliza a noun?", the Doctor would reply with "Are you saying no just to be negative?". This happened because the letters *no* in noun gave a match. But after fixing the code, the problem continued to occur. Why? What happened is that Variable Preservation was on in the LiveCode IDE preferences and so there was still an element in the array with an index of "no".

This issue is something to be mindful of when developing anything in LiveCode. For some programs, keeping Variable Preservation turned on can be a time-saver, (you don't need to reinitialise a big variable every time you make a code change), but other times there can be unwanted side effects. If in doubt, deleting an array before filling it with values, is a good idea.

Once the array is filled, the keys of the array are sorted according to the number of words and put into sPhraseList. The sort command is one of the high level features of LiveCode that sometimes feels like cheating as a programmer. It does so much in one line of code. This is not a simple alphabetical sort either, but a sort based on the number of words in each line – all done with a single LiveCode statement!

```
command InitializeReplies
  delete local sReplyArray
  put "Don't you believe that I can #?|Perhaps you would like to be able to #?|You want
me to be able to #." into sReplyArray["can you"]
  put "Perhaps you don't want to #?|Do you want to be able to #?" into sReplyArray["can
i"]
  put "What makes you think that I am #?|Does it please you to believe I am #?|Perhaps
you would like to be #?|Do you sometimes wish you were #?" into sReplyArray["you are"]
  put "Don't you really #?|Why don't you #?|Do you wish to be able to #?|Does that
trouble you?" into sReplyArray["youre"]
  put "Tell me more about such feelings.|Do you often feel #?|Do you enjoy feeling #?"
into sReplyArray["i feel"]
  put "Do you really believe I don't #?|Perhaps in good time I will #.|Do you want me
to #?" into sReplyArray["why dont you"]
  put "Do you think you should be able to #?|Why can't you #?" into sReplyArray["why
cant I"]
  put "Why are you interested in whether or not I am #?|Would you prefer if I were not
#?|Perhaps in your fantasies I am #?" into sReplyArray["are you"]
  put "How do you know you can't #?|Have you tried?|Perhaps you can now do #." into
sReplyArray["i cant"]
  put "Did you come to me because you are #?|How long have you been #?|Do believe it is
normal to be #?|Do you enjoy being #?" into sReplyArray["i am"]
  put "We were discussing you, not me.|Oh, I #.|You're not really talking about me, are
you?" into sReplyArray["you"]
  put "What would it mean to you if you got #?|Why do you want #?|Suppose you soon got
#?|What if you never got #?|I sometimes also want #?" into sReplyArray["i want"]
```

```
  put "Why do you ask?|does that question interest you?|What answer would please you
the most?|What do you think?|Are such questions on your mind often?" into
sReplyArray["what"]
  put "What is it that you really want to know?|Have you asked anyone else?|Have you
asked such questions before?|What else comes to mind when you ask that?" into
sReplyArray["how"]
  put "Names don't interest me.|I don't care about names, please go on." into
sReplyArray["who"]
  put "Is that the real reason?|Don't any other reasons come to mind?|What other reason
might there be?|Does that reason explaining anything else?" into sReplyArray["cause"]
  put "Please don't apologise!|Apologies are not necessary.|What feelings do you have
when you apologise?|Don't be so defensive!" into sReplyArray["sorry"]
  put "What does that dream suggest to you?|Do you dream often?|What persons appear in
your dreams?|Are you disturbed by your dreams?" into sReplyArray["dream"]
  put "How do you do, please state your problem." into sReplyArray["hello"]
  put "You don't seem quite certain.|Why the uncertain tone?|Can't you be more
positive?|You aren't sure?|Don't you know?" into sReplyArray["maybe"]
  put "Are you saying no just to be negative?|You are being a bit negative.|Why
not?|Are you sure?|Why no?" into sReplyArray["no "]
  put "Why are you concerned about my #?|What about your own #?" into
sReplyArray["your"]
  put "Can you think of a specific example?|When?|What are you thinking of?|Really,
always?" into sReplyArray["always"]
  put "Do you really think so?|But you are not sure you #.|Do you doubt you #?" into
sReplyArray["think"]
  put "In what way?|What resemblance do you see?|What does the similarity suggest to
you?|What other connections do you see?|Could there really be some connection?|How?|You
seem quite positive." into sReplyArray["alike"]
  put "Are you sure?|I see.|I understand." into sReplyArray["yes"]
  put "Why do you bring up the topic of friends?|Do your friends worry you?|Do your
friends pick on you?|Are you sure you have any friends?|Do you impose on your
friends?|Perhaps your love for friends worries you." into sReplyArray["friend"]
  put "Do computers worry you?|Are you talking about me in particular?|Are you
frightened by machines?|Why do you mention computers?|What do you think machines have
to do with your problem?|Don't you think computers can help people?|What is it about
machines that worries you?" into sReplyArray["computer"]
  put "What does that suggest to you?|I see.|I'm not sure I understand you fully.|Come
come elucidate your thoughts.|Can you elaborate on that?|That is quite interesting."
into sReplyArray[kDontUnderstand]

  put the keys of sReplyArray into sPhraseList
  sort sPhraseList descending by the number of words of each
  repeat for each line loopKeyword in sPhraseList
    put 0 into sReplyIndex[loopKeyword]
  end repeat
end InitializeReplies
```

One of the tricks that Doctor uses to make it appear intelligent, is to use the user's own input as the basis of its replies. So if you type, for example, *I feel that you are not human like I am* the reply is *What makes you think that I am not human like you are*. InitializeOpposites initializes the sOppositeArray that transforms the last part of the phrase, "not human like I am" into "not human like you are."

```
command InitializeOpposites
  constant kOppositeList =
                          "are,am,were,was,you,I,your,my,I've,you've,I'm,you're,me,you"
  local pairIndex
  repeat with loopIndex = 1 to the number of items in kOppositeList - 1 step 2
    put item loopIndex+1 of kOppositeList into pairIndex
    replace "'" with empty in pairIndex
    put item loopIndex of kOppositeList into sOppositeArray[pairIndex]

    put item loopIndex of kOppositeList into pairIndex
    replace "'" with empty in pairIndex
    put item loopIndex+1 of kOppositeList into sOppositeArray[pairIndex]
  end repeat
end InitializeOpposites
```

AnalyseText is the engine of this program. There isn't a game loop, instead after you enter text, AnalyseText is called by handlers in the memoInput Text Field script to formulate an appropriate reply. The first loop is one way to remove extra white space from the text. The text is then appended to the output field that has a nice old green screen look that was set in the Property Inspector for the memoOutput control. The textStyle of the appended characters is set to italic.

After removing unwanted punctuation, another loop converts a small number of words into another, so they can be found in the keys of the replies in sPhraseList (which was made from the keys in sReplyArray).

Each line in the sPhraseList is looked for in the entered text. If none of the phrases are found, a non-committal reply is produced with GetNextReply. When a phrase is found, GetNextReply gets the appropriate reply. If the reply includes a #, all the words after the keyword in the user's text are substituted for the #.

Lastly, anomalies in the punctuation are removed and DisplayReply shows the reply.

```
command AnalyseText
  local textInput,foundPhrase,index,buffer,replySentence,replyWord,replyEnd,isQuestion
  put empty into textInput
  repeat for each word loopWord in field "memoInput"
    put loopWord & space after textInput
  end repeat
  put length(field "memoOutput") into index
  put textInput after field "memoOutput"
  set the textStyle of char index to -1 of field "memoOutput" to "italic"
  put cr & cr after field "memoOutput"

  replace "'" with empty in textInput
  replace "," with empty in textInput
  replace "?" with empty in textInput
  replace "!" with empty in textInput

  put empty into buffer
  repeat for each word loopWord in textInput
    put ProcessSynonym(loopWord) & space after buffer
  end repeat
  put buffer into textInput

  repeat for each line loopPhrase in sPhraseList
    if textInput contains loopPhrase then
      put loopPhrase into foundPhrase
      exit repeat
    end if
  end repeat
  if foundPhrase is empty then
    DisplayReply GetNextReply(kDontUnderstand)
  else
    put GetNextReply(foundPhrase) into replySentence
    if replySentence contains "#" then
      put offset(foundPhrase, textInput) into index
      repeat for each word loopWord in char index + length(foundPhrase) to -1
                                                                    of textInput
        put loopWord into buffer
        replace "'" with empty in buffer
        put sOppositeArray[buffer] into replyWord
        if replyWord is empty then
          put loopWord into replyWord
        end if
        put replyWord & space after replyEnd
      end repeat
      replace "#" with replyEnd in replySentence
    end if
    replace ".." with "." in replySentence
    replace ".?" with "?" in replySentence
    replace " ?" with "?" in replySentence
```

```
    DisplayReply replySentence
  end if
  put empty into field "memoInput"
  focus field "memoInput"
end AnalyseText
```

ProcessSynonym is simple, but could be expanded to handle many more words. The idea is that words that are similar to the keys of the sReplyArray array are converted, so an appropriate reply is more likely to be found.

```
function ProcessSynonym pWord
  switch pWord
    case "hi"
      put "hello" into pWord
      break
    case "im"
      put "i am" into pWord
      break
    case "when"
    case "why"
      put "how" into pWord
      break
  end switch
  return pWord
end ProcessSynonym
```

GetNextReply gets the next reply from sReplyArray. The sReplyIndex array keeps track of the reply last used and increments the element for the phrase so the replies are used in a cycle. This stops the same reply being used twice in a row. Although they will repeat when the cycle reaches the end. Normally, items in a LiveCode list are comma delimited, but since a reply can have commas, the | character is used as the delimiter. By setting the itemDelimiter property, the chunk expressions of LiveCode can be used with any arbitrary delimiter. (The tab character is another good alternative item delimiter for many types of data.)

```
function GetNextReply pPhrase
  add 1 to sReplyIndex[pPhrase]
  set itemDelimiter to "|"
  if sReplyIndex[pPhrase] > the number of items in sReplyArray[pPhrase] then
    put 1 into sReplyIndex[pPhrase]
  end if
  return item sReplyIndex[pPhrase] of sReplyArray[pPhrase]
end GetNextReply
```

DisplayReply puts the reply into the memoOutput Text Field one character at a time. A delay between each character is added to give the output a "human" characteristic. To make sure the text is visible, the LiveCode *select after last line* code positions the cursor after the text, to cause the field to scroll if the cursor (and the last line) is not visible.

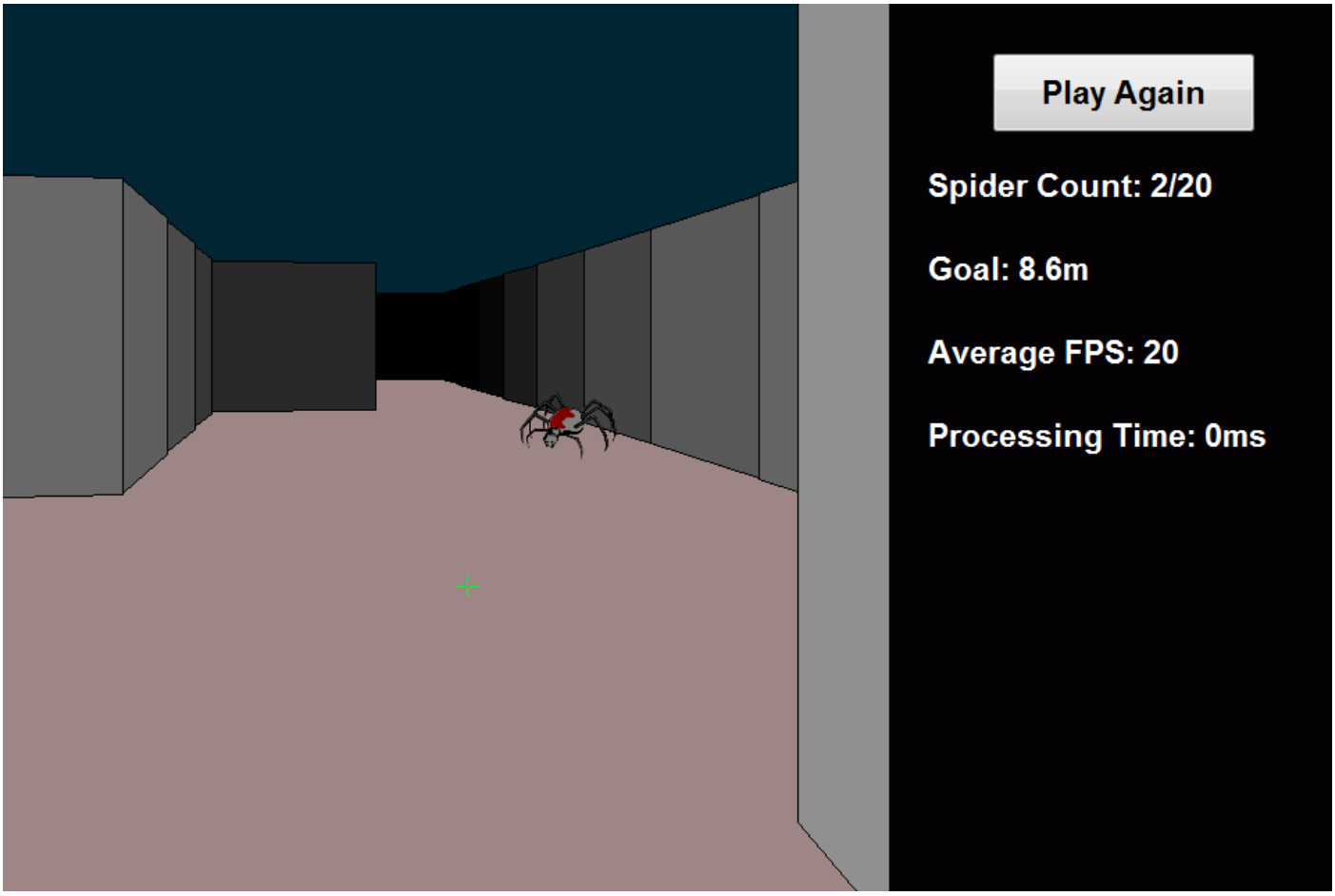
```
command DisplayReply pReply
  put "Doctor: " after field "memoOutput"
  set the textStyle of last line of field "memoOutput" to "plain"
  select after last line of field "memoOutput"
  wait 700 milliseconds
  repeat for each char loopCh in pReply
    put loopCh after field "memoOutput"
    wait random(50) milliseconds
    select after last line of field "memoOutput"
  end repeat
  put cr & cr after field "memoOutput"
  select after last line of field "memoOutput"
end DisplayReply
```

There you have it, a chatbot in 160 lines of LiveCode. Not the smartest Doctor on the planet, but always available.

Here are some suggestions for improving the program.

- Improve the vocabulary and responses.
- See if you can make the doctor chat about a different topic.
- Make the doctor remember what has been said so it can go back to a previous topic when there is no match in the replies.

Blank Page



You know, I'm glad those spiders don't come running towards me.

Game 9

Spider Hunt

Here is a game that started as an experiment with LiveCode. This code is not as polished as the other games in this book. As an exercise in optimising with LiveCode, a personal challenge was set of developing a 3D raycasting game using only the LiveCode engine. This means no externals (like Franklin 3D) or a library written in another language more suited to 3D games.

If you are thinking that pure LiveCode is not the best platform to choose when developing 3D games, you are right. As stated above, this was an experiment in finding ways of optimising the speed of graphics and code in LiveCode. In the process, if some fun can be had by creating the bare bones of an early 1990s first person game, why not?

The result of this experiment is LiveCode Spider Hunt. The concept puts you in a small building infested with overgrown spiders. Your goal is to kill all the spiders and to find the exit point. You have a can of bug spray. To keep track of how you are going, the status shows the number of dead spiders out of a total, and the distance to your goal.

In addition to this information, Spider Hunt shows the average frames per second (fps), and the average time in milliseconds (ms) required to update each frame. This status is updated every five seconds so this calculation does not affect the frame rate.

In this game you will see:

- A 3D representation of a room created with LiveCode graphic objects
- A 2D layer over the 3D rendering for the can of bug spray
- Objects in the rooms made from 2D bitmaps
- Automatic opening and closing doors
- A captured mouse cursor for controlling the view
- Frame rates of up to 20 fps on current hardware

Provided the processing time is under 50 ms, 20 fps is achieved. In Spider Hunt the fps is fixed to a maximum of 20 fps even if the hardware is capable of higher rates. On current hardware, achieving this is simple. In contrast, when run on an aging ThinkPad X100e Netbook from 2009 it would struggle to maintain half that.

Raycasting is a 3D technique first made popular in the 1990s in games like Ultima Underworld and Wolfenstein 3D. Then the processor of a typical home computer ran at around 66Mhz, in contrast to the 2GHz and more of today. Back then drawing an accurate 3D world like in games you see today just wasn't possible.

The mathematics for drawing an accurate 3D representation of a game world was known, but the time required meant those algorithms would not result in a usuable game for the hardware of the time. When the time taken to draw a single frame is measured in seconds, or much more, a real time first person game is not possible. Other game genres such as Adventures could have better 3D worlds, because all the image processing was done during development, for later display at runtime. (Myst is a classic game with this appoach.)

Raycasting is an algorithm that simplifies and reduces the number of calculations required to draw a 3D scene. It does this by placing restrictions on the type of world that can be represented. At the most basic, only interior rooms with walls from floor to ceiling at perpendicular angles are possible. To further simplify the processing and increase the speed, items in a room are not proper 3D models, but are instead 2D images.

Despite these limitations, and because there was nothing better, raycasting was used in many successful games. And if you have an idea that doesn't require realism, raycasting could still be used for an indie game today.

Spider Hunt is a first attempt at raycasting with LiveCode. It has problems and limitations even more restrictive than those early games. For example:

- Sometimes the spiders pop in and out view, or flicker
- Walls with an exposed corner do not render correctly at some viewing angles
- The walls are not textured

The first two problems could be solved with more work, but I suspect texturing the walls is simply asking too much from the LiveCode engine that is not designed for 3D effects.

While there are many implementations of the raycasting algorithm, most are not suited to an interpreted language like LiveCode. For example, an implementation written in C++ does not need to be as carefully optimised because it runs as compiled machine code.

Spider Hunt is based it on a simple raycasting demo written in Lua. This is by Andrew Burch and can be found here: http://developer.coronalabs.com/code/raycasting-engine. Since Lua is interpreted like LiveCode, that design is a more suitable basis for a LiveCode raycaster.

The central idea in raycasting is that the view of the world in front of the viewer, often referred to as the camera, is divided into a series of thin vertical slices going from left to right across the field of view. To make a 3D effect, the height of each slice is determined by how far away it is from the viewer.

To reduce the number of calculations, the world is divided into a grid. The grid can be thought of as tiles on the floor that are either empty, or contain a block that makes up part of a wall. This is the essence of basic raycasting, and by pre-calculating the trigonometric values needed, acceptable frame rates can be achieved.

If you are interested in the details of how raycasting works, these tutorials Ray-Casting Tutorial For Game Development And Other Purposes (http://www.permadi.com/tutorial/raycast/) and Lode's Computer Graphics Tutorial Raycasting (http://lodev.org/cgtutor/raycasting.html) will get you started.

As said before, LiveCode is not the best platform to use if you want to do 3D graphics without an external. A standard implementation of the raycasting algorithm, was not going to work fast enough. This meant an alternative approach was needed.

Instead of individually rendering thin slices of wall, a LiveCode graphic object with the polygon style is used to show entire blocks of wall, as large as each tile that makes up the grid. Once this change was made, which was complex to code, achieving a frame rate of 20 fps or more on current hardware was possible.

As an experiment, achieving empty rooms was satisfying but boring. To add interest a simple way of including bitmaps in the room was added. The size of the bitmap varies according to the distance from the viewer to produce a 3D effect. But since the bitmap is not a 3D model, the spiders in Spider Hunt always face the viewer. Then doors were added which were surprisingly easy to code, although they are not very realistic in operation.

Lastly, a way to kill the spiders was added, which uses a simple 2D layer over the raycasting. With a spray can bitmap and an animation for the cloud of bug spray. The cloud animation was made in Anime Studio which can produce this sort of animation in minutes. Spray the spider long enough and it disappears. (If you have trouble killing it, try moving so the cross hair is over the back of the abdomen.)

That's it. A LiveCode "game" based on raycasting. When running Spider Hunt, note that the mouse cursor is "captured" and made invisible, so you cannot leave the game window with the mouse. To get the mouse back, press the Esc key.

When examining this game in the LiveCode IDE, a warning about versions of LiveCode from 6.0 and newer. Close the Project Browser before testing games like this that frequently change the layer property of controls. If the Project Browser is open when layers are changing, your program will not run smoothly and the resulting jerky animation may mislead you to think that the LiveCode Engine is not up to the task.

Here are some suggestions for improving the game.

- Make the spiders move towards the viewer.
- Add a health bar that falls when a spider is too close.
- Change the spray can so it it has limited capacity. Maybe you need to find fresh cans to replace an empty one.